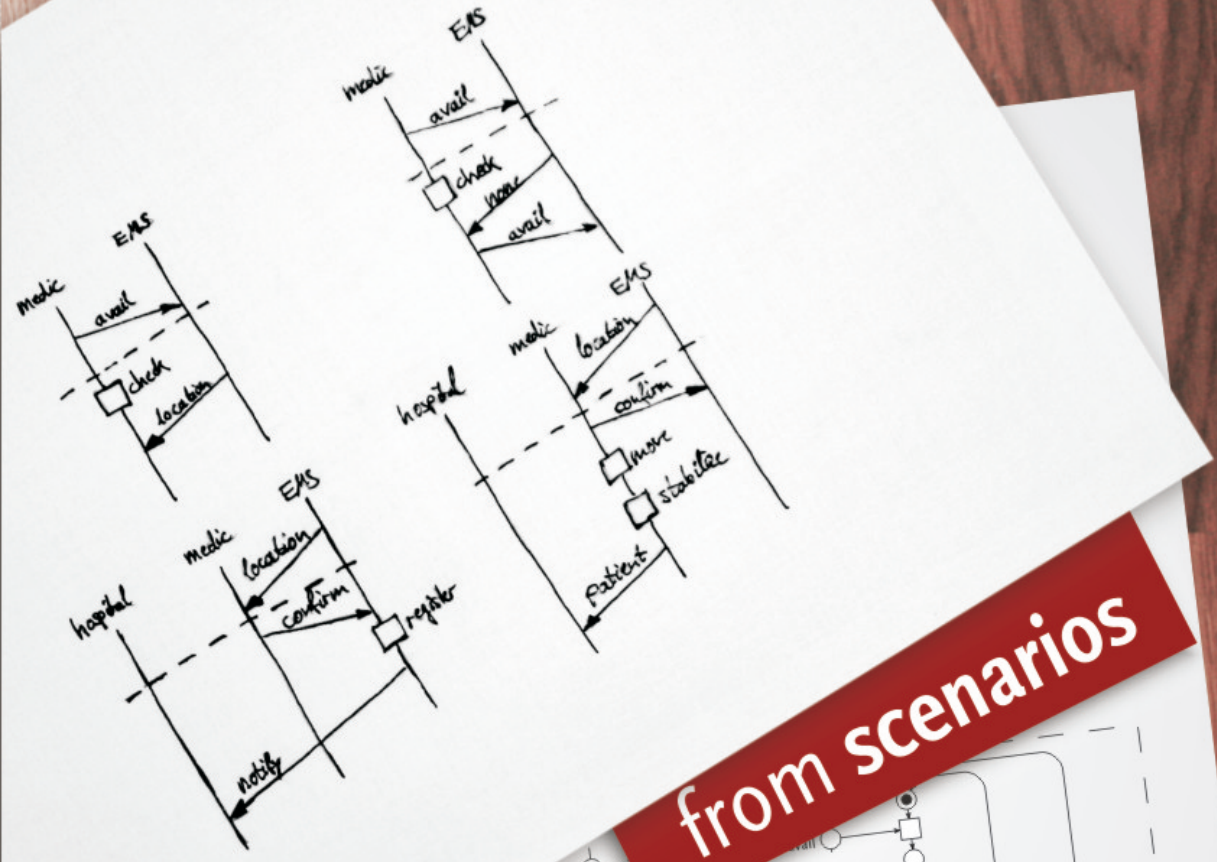
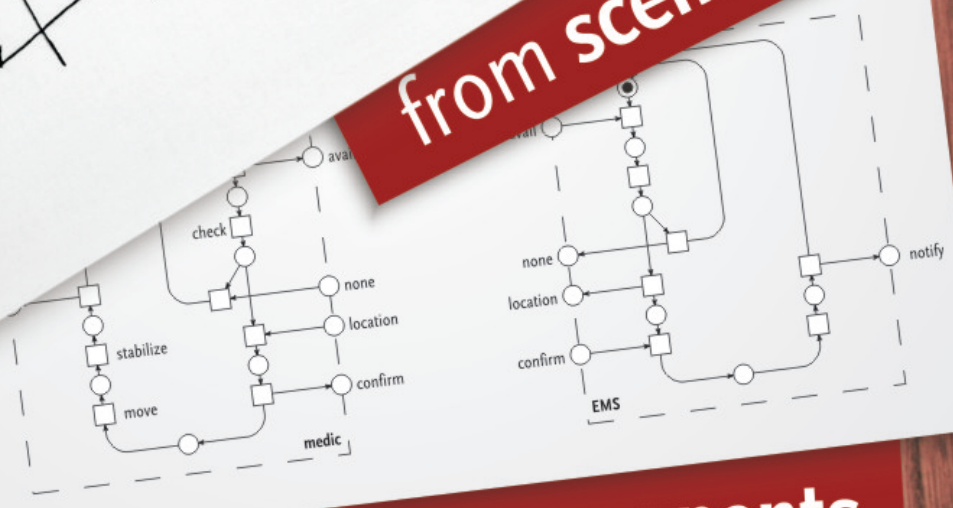
dirk fahland



from scenarios to components



From Scenarios to Components

Dirk Fahland

A library record is available from the Eindhoven University of Technology Library.

Fahland, Dirk

From Scenarios To Components / by Dirk Fahland
– Eindhoven: Technische Universiteit Eindhoven, 2010. – Proefschrift. –

ISBN 978-90-386-2319-1
NUR 993

Keywords: scenarios / distributed systems / verification / synthesis / Petri nets

SIKS Dissertation Series No. 2010-38

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis has been partially supported by the DFG-Graduiertenkolleg "METRIK" (GRK 1324).

Printed by University Press Facilities, Eindhoven.

From Scenarios to Components

Dissertation

zur
Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(*doctor rerum naturalium*, Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II der
Humboldt-Universität zu Berlin

im Rahmen einer Binationalen Promotion mit der
Technische Universiteit Eindhoven, Niederlande

von
Herrn Diplom-Informatiker

Dirk Fahland

geboren am 8. Dezember 1980 in Berlin

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Dr. h.c. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Peter Frensch

1. Gutachter	Prof. Dr. Wil M.P. van der Aalst
2. Gutachter	Prof. Dr. Wolfgang Reisig
3. Gutachter	Prof. Dr. Karsten Wolf
eingereicht am	30. Juli 2010
Tag der mündlichen Prüfung	27. September 2010

From Scenarios to Components

Proefschrift

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
maandag 27 september 2010 om 16.00 uur

door

Dirk Fahland

geboren te Berlijn, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. W.M.P. van der Aalst
en
Prof.Dr. W. Reisig

Copromotor:
Prof.Dr. K. Wolf

Abstract

Scenario-based modeling has evolved as an accepted paradigm for developing complex systems of various kinds. Its main purpose is to ensure that a system provides desired behavior to its users. A *scenario* is generally understood as a behavioral *requirement*, denoting a course of actions that shall *occur* in the system. A typical notation of a scenario is a Message Sequence Chart or, more general, a finite partial order of actions. A *specification* is a set of scenarios. Intuitively, a system *implements* a specification if all scenarios of the specification can occur in the system. The main challenge in this approach is to systematically *synthesize* from a given scenario-based specification state-based components which together implement the specification; preferably to be achieved automatically. A further challenge is to *analyze* scenarios to avoid erroneous specifications.

Existing scenario-based techniques exhibit a conceptual and formal gap between a scenario-based specification on the one hand and a state-based implementation on the other hand. This gap often renders synthesis surprisingly complex, and obscures the relationship between a specification and its implementation. Additionally, existing techniques for analyzing implementations cannot immediately be reused for analyzing specifications, and vice versa.

In this thesis, we introduce a semantic model for scenarios that seamlessly integrates scenario-based specifications and state-based implementations. We focus on modeling and analyzing the *control-flow* of systems. Technically, we use Petri nets together with the well established notion of *distributed runs* for (1) describing the semantics of scenarios, for (2) systematically constructing and analyzing a specification, and for (3) synthesizing an implementation from a given specification. An industrial case study shows the feasibility of our techniques.

Kurzfassung

Szenario-basiertes Modellieren hat sich zu einer akzeptierten Entwurfstechnik für komplexe Systeme verschiedenster Art entwickelt. Mit ihr soll sichergestellt werden, dass ein System sich wie gewünscht gegenüber seinen Nutzern verhält. Ein *Szenario* beschreibt im Allgemeinen eine *Anforderung* an das Verhalten des Systems als eine Folge von Aktionen, die im System *vorkommen* soll. Häufig wird ein Szenario als ein Message Sequence Chart oder allgemein als halbgeordnete Menge von Aktionen notiert. Eine *Spezifikation* ist eine Menge von Szenarien. Intuitiv *implementiert* ein System eine Spezifikation wenn alle Szenarien der Spezifikation im System vorkommen können. Die Hauptschwierigkeit dieses Ansatzes besteht darin, aus einer szenario-basierten Spezifikation systematisch zustands-basierte *Komponenten* zu *synthetisieren*, die die Spezifikation gemeinsam implementieren; idealerweise gelingt dies automatisch. Eine weitere Herausforderung ist die *Analyse* von Szenarien, um fehlerhafte Spezifikationen zu vermeiden.

Existierende szenario-basierte Techniken weisen eine konzeptuelle Lücke zwischen einer szenario-basierten Spezifikationen und einer zustands-basierten Implementation auf. Aufgrund dieser Lücke ist die Synthese überraschend komplex und die Beziehung zwischen Spezifikation und Implementation nur schwer nachvollziehbar. Darüberhinaus können Analysetechniken für eine Implementation nicht unmittelbar zur Analyse einer Spezifikation wiederverwendet werden und umgekehrt.

In diese Arbeit führen wir ein Verhaltensmodell für Szenarien ein, das szenario-basierte Spezifikationen und zustands-basierte Implementationen nahtlos integriert. Wir konzentrieren uns hierbei auf die Modellierung und die Analyse des *Kontrollflusses* von Systemen. Auf der technischen Ebene verwenden wir Petrinetze und ihre *verteilten Abläufe*, um (1) die Semantik von Szenarien zu beschreiben, (2) eine Spezifikation systematisch zu konstruieren und zu analysieren, sowie (3) eine Implementation aus einer Spezifikation zu synthetisieren. Wir stellen Ergebnisse aus einer industriellen Fallstudie vor, die die Tauglichkeit unserer Techniken in der Praxis belegen.

Contents

Part I.

Distributed Systems

1. Introduction

Designing a distributed system is a tedious and error-prone task as we lack intuition for how the behavior of the entire system emerges from the behavior of its components and their interaction. In the worst case, a system designer cannot tell whether a designed system satisfies all behavioral requirements she has in mind. This chapter introduces scenarios and the scenario-based approach for designing distributed systems. We identify the research problem of supporting the design process by a unified theory that relates a distributed system to its scenarios, and vice versa. This chapter closes with an overview of the main results of this thesis and an outline of the forthcoming chapters.

1.1. Designing Distributed Systems with Scenarios

Distributed Systems

Many of today's computer-based systems are built from smaller systems, called system *components*. Each component typically implements a certain functionality of the system by providing certain *actions*. The functionality of the entire system emerges from the actions of its components and their *interaction*. A component of a *distributed* system is capable of acting independently of other components — up to a point where two components have to interact to proceed, as sketched in Figure 1.1. The complexity of distributed systems does not primarily stem from complex computations or complex data, but from local interactions of components giving rise to *complex behavior*.

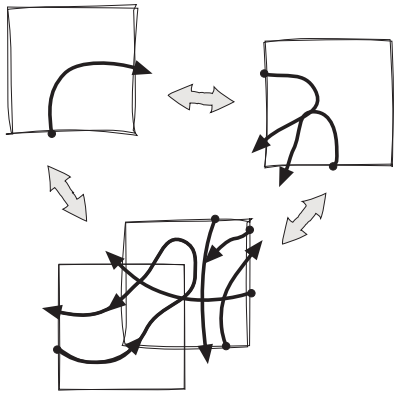


Figure 1.1. Component-centric view: a distributed system consists of several components that interact with each other. [55]

In principle, each component of a distributed system is easier to design and to maintain than the entire system. However, designing all components in a way that they compose to an intended system is tedious and error-prone. We lack intuition for how different actions of different components depend on and influence each other once the system reaches a certain level of complexity.

Typical examples of a distributed systems are the Internet or a telecommunication network. A less obvious example is a *disaster management process* which describes how several organizations cooperate to mitigate effects of natural and man-made catastrophic events. Supporting disaster management with information- and communication technology emerged as a research topic in recent years [77]. Systems of this kind are technically not much different from systems in other domains. Yet, two factors in disaster management render their design rather complex. First, a "component" of a disaster management process is an organizational entity, say a group of fire fighters or scientific experts. These groups are not isolated against each other but may *overlap* whenever they cooperate at the same location together to achieve a joint goal. For example, a group of fire fighters may cooperate with medics and engineers to rescue people from a collapsed

building. Second, a disaster management process faces many exceptional situations in which organizations deviate from their standard process. These deviations lead to complex behavior of the entire system [77].

Figure 1.4 on page 7 depicts a model of an actual disaster management process. The depicted Petri net describes a simplified version of a process run by the "Taskforce Earthquakes" of the German Research Center for Geosciences (GFZ). The main purpose of the task force is to coordinate the allocation of an interdisciplinary scientific-technical expert team after catastrophic earthquakes worldwide [42]. The model identifies three separate components, the larger component in the middle consists of two overlapping sub-components that cannot be separated due to a high degree of synchronization; the coloring distinguishes the actions of the different components.

It is virtually impossible to design systems of this kind by first identifying each component's actions and their relations, and then adjusting and refining all component interactions to form the intended system.

Scenarios

The *scenario-based approach* has evolved has a technique for describing *interaction* of system components. A *scenario* describes how a user interacts with the system or how different parts of the system interact with each other in a specific situation [8]. We may read a scenario as a "self-contained story" about several components interacting with each other for a specific purpose [55]. So the behavior of a distributed system may be described by its scenarios, as sketched in Figure 1.2.

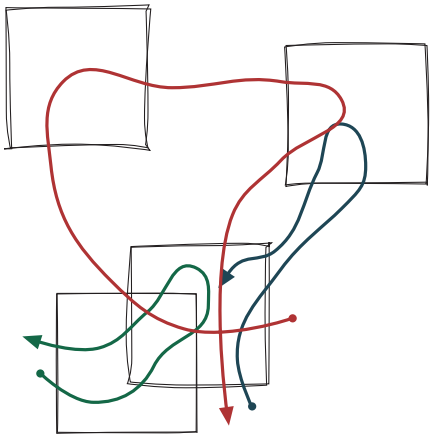


Figure 1.2. Scenario-centric view: each scenario describes a story involving several components. [55]

Figure 1.3 shows six scenarios of an actual disaster management process run by the "Taskforce Earthquakes" in a formal syntax. The scenarios describe how the Taskforce prepares a scientific mission after arriving at a disaster area. To begin a mission, the Taskforce has to get its equipment (eq) through customs, organize a

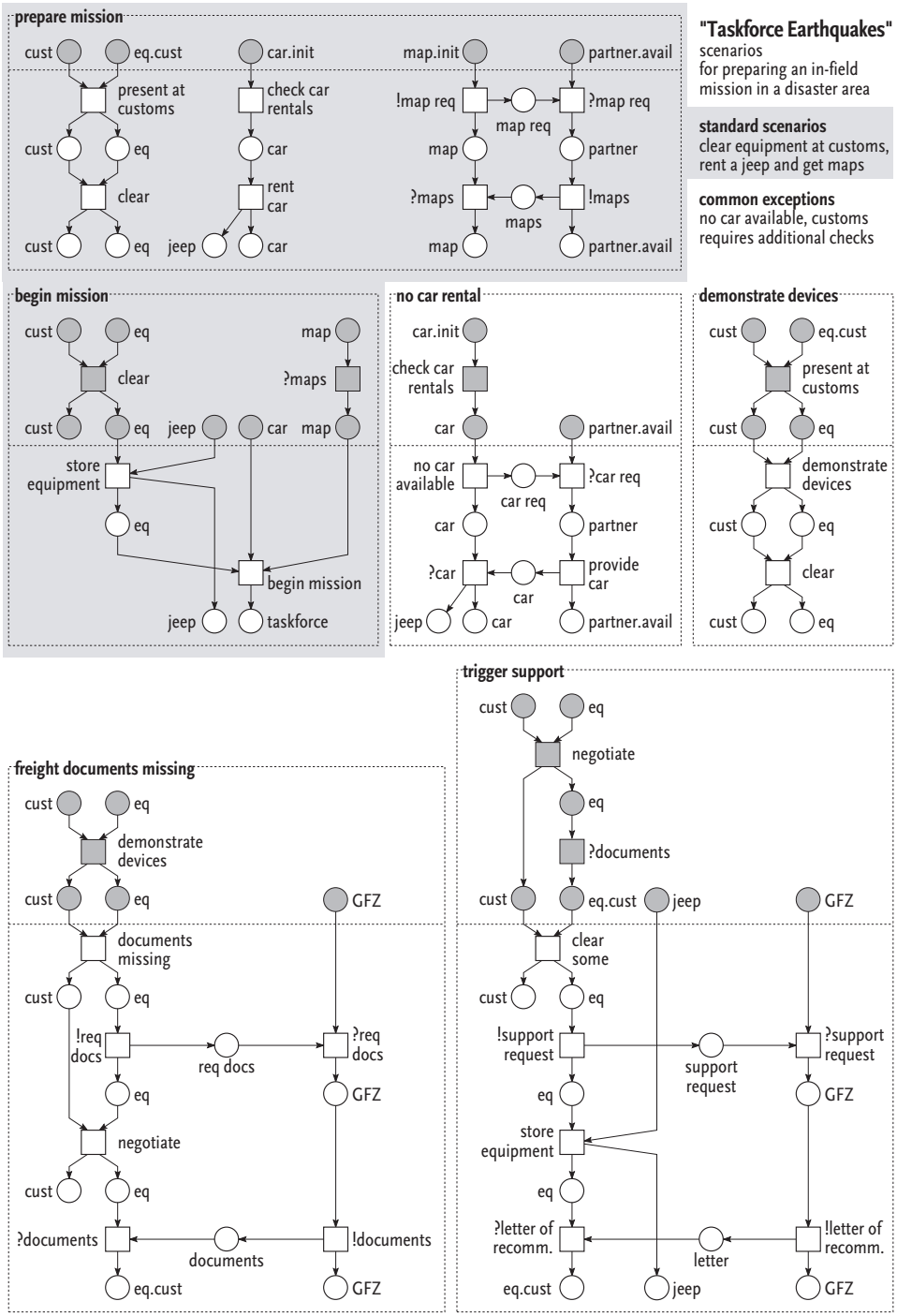


Figure 1.3. Scenarios of the "Taskforce Earthquake" for preparing an in-field mission in a disaster area.

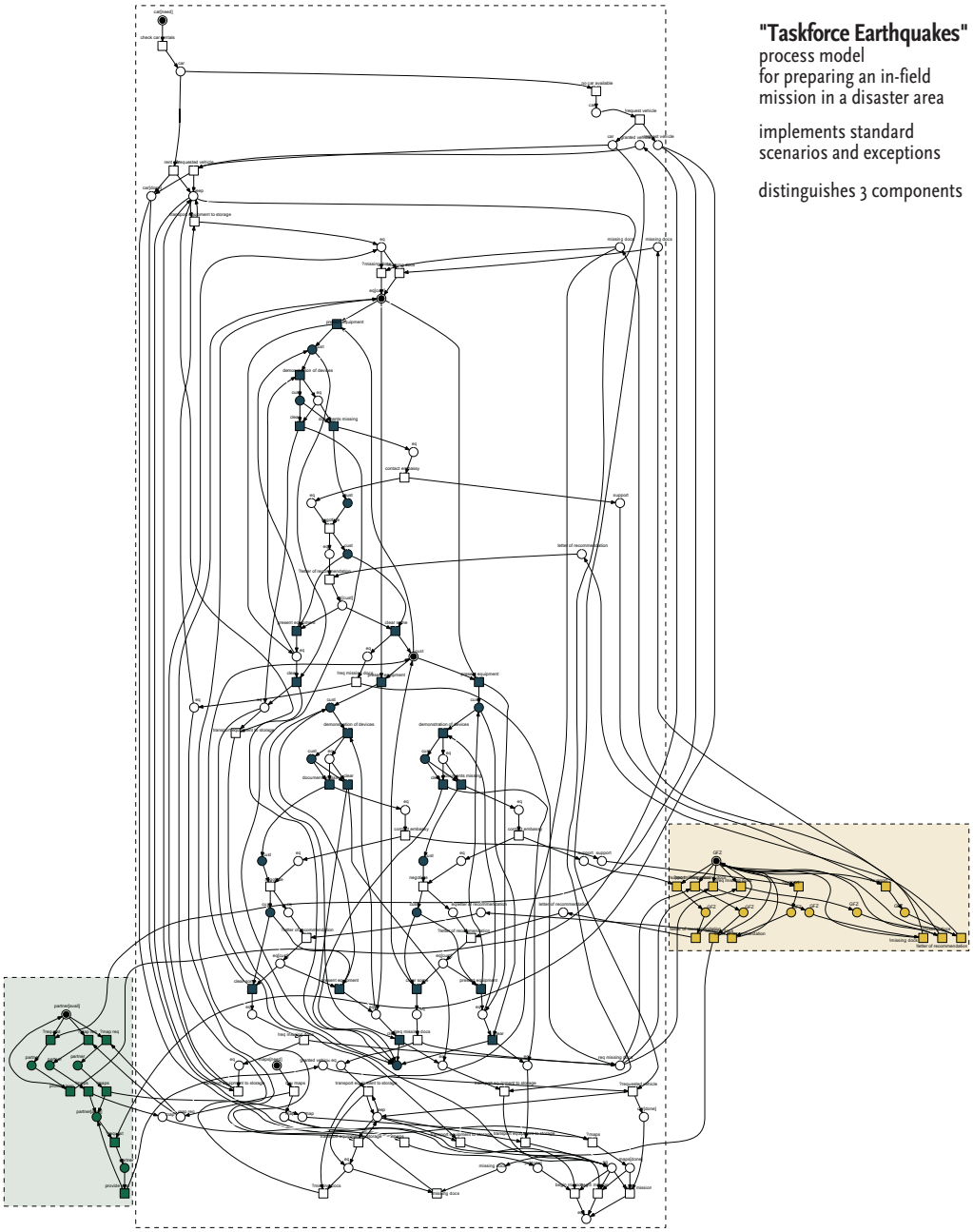


Figure 1.4. Process model of the "Taskforce Earthquake" for preparing an in-field mission in a disaster area. The left component describes the behavior of a local partner organization, the right one describes the behavior of the GFZ office in Germany, and the central component describes the cooperation of the Taskforce (white nodes) with local customs (dark nodes).

jeep, and obtain maps of the area. To achieve their goal, the Taskforce cooperates with customs (cust), a local partner organization, and the GFZ office in Germany. The standard process begins with scenario prepare mission. After the Taskforce cleared equipment at customs, organized a jeep, and obtained a map, they can store their equipment and begin the mission according to scenario begin mission.

The other scenarios describe exceptions that occur frequently. There may be no car available after checking car rentals, so the Taskforce has to organize a car together with a local partner organization (scenario no car rental). Customs may request to demonstrate devices such as seismic sensors before clearance. However, the process can become more involved in case freight documents are missing, which have then to be sent from the GFZ office in Germany while the Taskforce is negotiating with customs. In case customs only clears some of the equipment, the Taskforce may trigger support from the GFZ office in Germany in the form of an official letter of recommendation. While this letter is underway, the Taskforce may already store the cleared equipment. The model of the disaster management process shown in Figure 1.4 *implements* all scenarios of the specification in Figure 1.3.

To *implement* a scenario in a system, the involved system components provide all actions of the scenario so that these actions occur in the system's behavior in the order described in the scenario. A system typically implements several scenarios. At least one "good weather" scenario describes an intended use of the system. During that scenario, an exception may occur for various reasons. The user and the system components react accordingly giving rise to a "bad weather" scenario. Moreover, a distributed system may be used in more than one way defining several "good weather" scenarios and it may encounter various exceptions which leads to several "bad weather" scenarios. Finally, a distributed system is typically used by several users concurrently. Their scenarios (good and bad) depend on each other, interact, or overlap. A scenario-based *specification* is a set of scenarios that mutually depend on each other in some way. Yet, each individual scenario describes one story of the system involving several system components as illustrated in Figure 1.2 and shown by the example of the "Taskforce Earthquakes".

The scenarios of Figure 1.3 and the process model of Figure 1.4 equivalently describe the disaster management process of the "Taskforce Earthquakes": both describe the *same runs* and allow to take the *same decisions*. The process model consists of four components. The left component describes all behaviors of the local partner organization in this process, the right component all behaviors of the GFZ. The Taskforce and the local customs overlap in the central component; the custom's behavior is described by the dark nodes. Taskforce and customs cannot be separated easily because of a high degree of synchronization in their interaction. The scenarios of Figure 1.3 are easier to understand, to reason about, and to create compared to the intricate model of Figure 1.4 — although both describe exactly the same behavior. The preference for scenarios largely stems from a more explicit description of system behavior, which requires less effort to understand compared to a classical component-centric model.

In this thesis, we develop a *theory* that will allow us to reason about the behavior described by the scenarios of Figure 1.3 and the one described by the process model

of Figure 1.4 in a unified way. This theory will even allow us to automatically construct the model of Figure 1.4 from the scenarios of Figure 1.3.

1.2. Problem Statement and Research Goal

When designing a distributed system, it is relatively easy to let a system user tell a story about how she uses the system or how several system parts interact. A system designer rephrases each story as a scenario in a more or less formal notation. The difficulty of the scenario-based approach is to build a system that implements all scenarios. In this thesis, we focus on two design approaches.

First design approach: synthesize components from scenarios

The standard approach is to construct the components of the system such that each component integrates all scenarios in which it participates. The construction is difficult because of how scenarios and components relate to each other: One component usually participates in several scenarios and the same action of a component may occur in several scenarios. Thus, one action of a component participates in several scenarios. Conversely, several actions in several components are required to implement one scenario. These intricate relations render the construction of components from a given scenario-based specification a complex problem.

To implement a scenario-based specification, a system designer transforms the scenario-centric view of Figure 1.2 where "each story involves many components" into an *equivalent* component-centric view like in Figure 1.1 where "each component involves many stories" [25]. Solving this transformation for any given scenario-based specification and for any choice of components is a *synthesis problem* [99]. The synthesis problem from scenarios to any choice of components [58, 26] as illustrated in Figure 1.5 is the main problem addressed in this thesis.

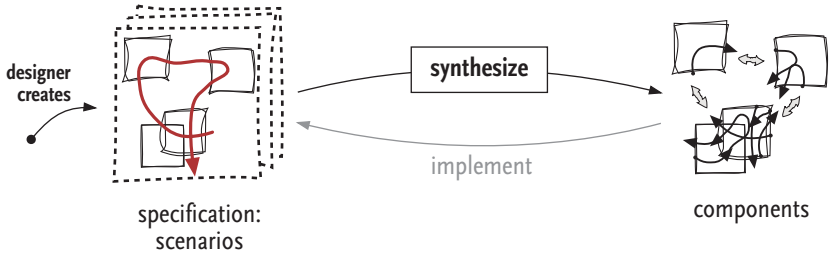


Figure 1.5. The synthesis problem.

Second design approach: execute scenarios

Synthesizing specific components from a scenario-based specification *statically* solves the problem of implementing all scenarios in a distributed system. Each

action is assigned to one specific component. If an action shall be assigned to another component, usually all components have to be synthesized again — even those that do not contain the re-assigned action.

For this reason, some domains favor a different design approach. For example, a workflow management system does not implement an action statically in a specific component. Rather, it *dynamically* assigns an action at run-time to a component depending on the component's availability [117]. A component to which an action is assigned *executes* this action. To this end, the system needs to know for each action under which conditions this action may occur. Whenever these conditions are met, the action is *enabled*. If an action is enabled, the system may trigger the action's execution. In other words, a system of this kind takes an *action-centric* view on system behavior rather than a component-centric view. In the action-centric view, system behavior follows only from the system's actions and their respective enabling conditions.

To dynamically implement a scenario-based specification in this way, a system designer transforms the scenario-centric view, where each scenario is a set of actions in a specific order, into an equivalent action-centric view. This transformation determines for each action occurring in the specified scenarios its enabling conditions. Finding a generic transformation for all scenario-based specifications amounts to defining *operational semantics* for scenarios. In effect, a scenario-based specification with operational semantics becomes an operational *system model* alleviating the need to synthesize specific components.

A solution to either design approach requires a *formal model* for system components, system behavior, scenarios, and their inter-relations. In this thesis, we restrict ourselves to specifying and synthesizing the *control-flow* of distributed systems. We abstract from data and assume system behavior to be time-invariant. Consequently, the results of this thesis only hold for the *behavioral* aspects of a distributed system.

State of the Art

In the past decades, many techniques for specifying systems with scenarios have been developed. Likewise, many synthesis algorithms have been proposed. Two surveys by Amyot and Eberlein al. [8] and by Liang et al. [80] provide an overview. In the following, we summarize the state of the art in (1) synthesizing components of distributed systems from scenarios and in (2) operational semantics of scenarios. We also consider the *hardness* of the synthesis problem in the respective techniques, and how *flexible* a system designer is in the process of specifying a system with a specific technique.

All scenario-based techniques for specifying behavior of *distributed* systems formalize a scenario as a *partial order* of actions and interactions. Additional concepts relate different scenarios to each other.

MSCs. A scenario-based specification may be just a *set* of scenarios in which each scenario is a partial order of actions and interactions. This most basic notion has been formalized in the industry standard of *Message Sequence Charts* (MSCs) [65].

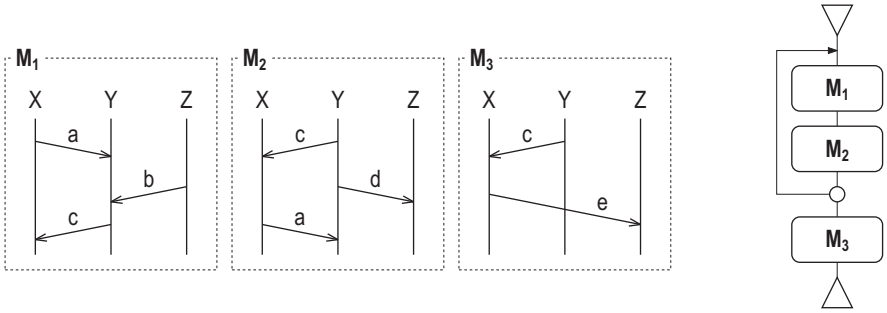


Figure 1.6. Three MSCs $\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3$ (left) and an HMSC (right).

Specifications of this kind have limited expressive power effectively denoting a set of sample runs from beginning to end [11, 25]. These techniques only qualify for specifying finite behavior and the synthesis problem is decidable. Several algorithms are available for synthesizing communicating automata [7] or Petri nets [12]. Figure 1.6 depicts three MSCs M_1, M_2, and M_3 which specify interactions between three components X, Y, and Z in terms of exchanged messages a-e. Each MSC describes one system run.

HMSCs. A second kind of specifications uses *global composition operators* over a set of scenarios to describe behavior of a distributed system. The composition operators of a specification explicitly relate different scenarios to each other and allow to describe infinite behavior. The most prominent techniques of this kind are *Hierarchical MSCs* (HMSCs) [65] and *UML activity diagrams* [95]. Figure 1.6 on the right depicts an HMSC over the three MSCs M_1, M_2, and M_3 . This HMSC describes that a system first exhibits the behavior of M_1 followed by M_2. Then the system may either return to the beginning of M_1 or exhibit M_3 and terminate.

Not every HMSC specification can be synthesized to any kind of implementation: the problem whether for a given HMSC specification H there exists a Petri net that exhibits the behavior of H is *undecidable* [26]. Synthesis of Petri nets succeeds for restricted classes of specifications [108, 13]. Several algorithms exist for synthesizing statecharts [79, 21, 129] or SDL models [85]. Regarding the design process, the global composition operators in hierarchical specifications limit the flexibility of these techniques. Generally, scenarios may not overlap unless dedicated operators are defined [72], which constrains the system designer how she may specify system behavior. Often, a hierarchical specification tends to consist of a large number of small scenarios that are composed in complex ways [113].

LSCs. In the third kind of specifications, each scenario distinguishes a *history* which *locally* describes *when* the scenario may occur. Relations between scenarios emerge from the relation between each scenario's history and all other scenarios. The first technique to formalize this concept were *Live Sequence Charts* (LSCs) which extend MSCs in several ways [25]. Since then, the notion of a history has been adopted by other techniques as well [45, 107]. Figure 1.7 depicts three

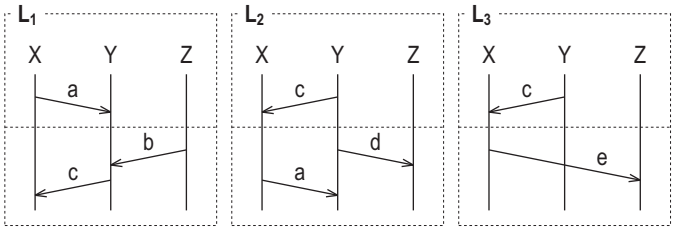


Figure 1.7. Three scenarios $\mathsf{L}_1, \mathsf{L}_2, \mathsf{L}_3$ that distinguish a history.

scenarios L_1, L_2, L_3 that distinguish a history. Intuitively, L_2 follows L_1 because message c of L_1 triggers an occurrence of L_2. For the same reason, message a of L_2 triggers an occurrence of L_1 which leads to cyclic behavior $\mathsf{L}_1\mathsf{L}_2\mathsf{L}_1\mathsf{L}_2\ldots$. Message c of L_1 also triggers L_3. Whether L_2 and L_3 occur alternatively or overlappingly depends on the technique. Compared to HMSCs, scenarios with a history can be used more flexibly in system design: a specification is just a set of scenarios that may overlap. Their interplay simply follows from each scenario's history.

Like for HMSCs, the synthesis problem from LSCs has no general solution: the problem whether a given LSC specification can be implemented in a given set of components is *undecidable* [18]. Several synthesis algorithms from LSCs to statecharts and to communicating automata have been proposed [49, 16, 19, 17, 53]. Due to undecidability, the synthesized system mostly requires some notion of centralized control or is not complete wrt. the specified behavior.

Relation between expressive power and synthesis result. Figure 1.8 qualitatively illustrates the relation between MSCs, HMSCs, and LSCs. The diagram at the top abstractly relates for existing synthesis algorithms which class of specifications can be synthesized to which choice of components.

The horizontal axis denotes the class of specifications a synthesis algorithm can take as input in terms of their *expressive power*. The higher the expressive power, the more system behavior can be specified and distinguished by this class of specifications. MSCs have the weakest expressive power as each MSC only describes a finite run. HMSCs and LSCs allow to specify arbitrary system behavior including infinite runs that satisfy various behavioral constraints.

The vertical axis denotes to which extent the output of a synthesis algorithm can be *distributed into any choice of components*. A system with complete synchronization of all components is positioned at the bottom of the axis. The more independent the system's components are and the less each component knows about the other components' states and their interaction, the higher the system is positioned on the axis. The least output of a synthesis algorithm defines a single component in which all actions are synchronized like in an automaton. Such systems can be synthesized from all three classes of specification techniques (MSCs, HMSCs, and LSCs). However, scenarios specify behavior between components of a distributed system. Preferably, *any choice of components* should be synthesizable.

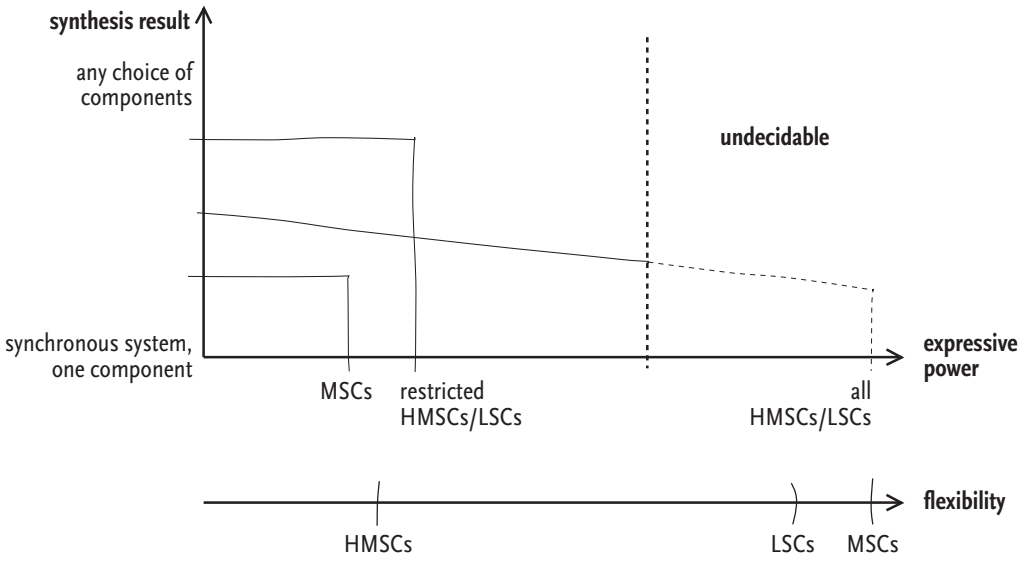


Figure 1.8. Qualitative illustration of flexibility and synthesis capabilities in the state of the art of scenario-based techniques.

The areas in the diagram indicate for which range of specifications which range of components can be synthesized, that is, the range of inputs and outputs of synthesis algorithms. Synthesis becomes undecidable for HMSCs and LSCs from a certain point onwards; moreover, some notions in LSCs diminish the ability to synthesize chosen components and enforce synchronization [26, 18]. Restricted classes of HMSCs and LSCs exhibit specific properties, which improves synthesis into chosen components [13, 17].

Flexibility. The bottom diagram of Figure 1.8 indicates the *flexibility* of a scenario-based technique. Flexibility expresses to which extent a technique requires the system designer to relate scenarios explicitly. In MSCs, scenarios are not related to each other at all, so specifications can be changed in a very flexible manner. In LSCs, a system designer only notes down a scenario's history to express when it may occur. HMSCs require to express relations between scenarios by global composition operators. When a system designer adds a scenario S to a specification, she has to put S in explicit relation to all existing scenarios. This may trigger further editing steps to maintain a syntactically valid specification. These issues do not arise in LSCs and MSCs.

A formal observation. Every synthesis algorithm from scenarios bridges a "conceptual gap" between the "one story for many components" view of a specification and the "one component for many stories" view of an implementation. This is the nature of the problem. However, the presented specification and synthesis

techniques also exhibit a *formal gap* between scenarios, system behavior, and implementation.

Scenario-based techniques are typically defined on three different formal notions for scenario, system behavior, and implementation. First, all techniques describe a scenario as a *partial order* of actions. HMSCs (and similar techniques) additionally define *composition operators* over scenarios. Second, *semantics* of (H)MSCs and LSCs are defined on *sequences* of actions. Third, an implementation is typically modeled as an *automaton*, a *statechart*, or a *Petri net*.

The state of the art in operational semantics of scenarios reveals a similar picture. Operational semantics of (H)MSCs are typically based on a translation to another formal model like process algebras [87] or Petri nets [93, 62] or use formal techniques like graph grammars [59]. Operational semantics of LSCs follow from an algorithm that *interprets* a given specification in terms of sequential runs [51].

These formal gaps between scenarios, system behavior, and implementation and render synthesis and operational semantics surprisingly involved whereas scenarios appear to be very intuitive.

There are several approaches where specification, behavior, and implementation are all based on the same formal model. Here, Desel at al. [13] possibly succeed farthest by formalizing scenarios and system behavior as partial orders of actions, and an implementation as a Petri net. Though, their approach uses global composition operators on scenarios similar to HMSCs, which renders the technique less flexible compared to LSCs.

Research goal

The goal of this thesis is to identify a *well-balanced position* between flexible and sufficiently expressive scenario-based techniques on one hand, and efficient synthesis and analysis techniques on the other hand. The specific aim is

- to *identify a class of scenario-based specifications* that is sufficiently expressive to specify the behavior of any kind of distributed systems in the flexible style of LSCs,
- to *develop operational semantics* for scenarios that allow to model distributed system with scenarios, and
- to *solve the synthesis problem*. Specifically, we shall define an algorithm that constructs for *every scenario-based specification* S system components that are specified in S so that the composed system implements S and is *minimal*, that is, exhibits as few additional behavior as possible.

The results of this thesis shall be supported by formal proofs which requires a corresponding *formal model* for scenarios. This model shall

- use as few technical notions as possible by formalizing scenarios, system behavior, and implementations with the *same technical notions*, and
- base on existing formal theories so that system design techniques for scenarios become available to state-based implementations, and

– analysis techniques for state-based implementations become available to scenario-based specifications.

1.3. Results of this Thesis

The thesis contributes a novel technique to specify the behavior of distributed systems with scenarios in the flexible style of LSCs, to model systems with scenarios by operational semantics, and to synthesize components from every scenario-based specification, except for those cases where synthesis is undecidable. The formal technique — which we call *oclets* — roots in Petri nets and LSCs. Oclets can be applied to practical problems: the oclets in Figure 1.3 specify an actual disaster management process, distinguishing standard scenarios from exceptional scenarios. The synthesis algorithm developed in this thesis constructs the minimal implementation on page 7 which distinguishes the different components of the disaster management process. Figure 1.9 sketches the results of this thesis, which we explain in more detail in the following.

A minimal model for scenarios

In this thesis, we identify a *kernel* of scenario-based specifications from the class of history-based specification such as LSCs. Critically reviewing existing scenario-based techniques, we identify a *minimal* set of notions that suffices for specifying the behavior of distributed systems with scenarios. It turns out that this minimal set of notions has not been formalized in existing scenario-based techniques yet.

Based on these findings, we develop *oclets* as a formal model for scenario. An oclet formalizes a scenario as an acylic Petri net that distinguishes a *history* similar to LSCs. A *specification* is a set of oclets. The specification on page 6 depicts several oclets. The semantics of oclets is given in terms of *distributed runs* by formal *declarative semantics*. In the style of LSCs, this semantics defines when a system behavior (a set of distributed runs) *satisfies* a specification. A system model like a Petri net *implements* a specification if its behavior satisfies the specification. The class of oclets is expressive enough to specify the behavior of *every* Petri net.

Oclets canonically extend to a unifying formal model for scenarios, system behavior, and implementations. We develop a *composition* operator on oclets which appends an oclet o_2 to another oclet o_1 if the history of o_2 matches the behavior in o_1. This notion allows to compose larger scenarios from smaller scenarios and to compose distributed runs from scenarios. A corresponding *decomposition* operator allows to decompose a scenario into smaller scenarios or into its single actions.

Using oclet composition, we construct the unique least set $R(O)$ of distributed runs that satisfies an oclet specification O. For some oclet specifications O there exists no implementation that exhibits exactly $R(O)$ — any implementation of O exhibits *more behaviors*. These additional behaviors are known as *implied scenarios*. We show that implied scenarios can be explained by occurrences of single actions. By first decomposing O into its single actions and then composing the single actions of O, we construct the unique behavior $\hat{R}(O)$ of a *minimal implementation*

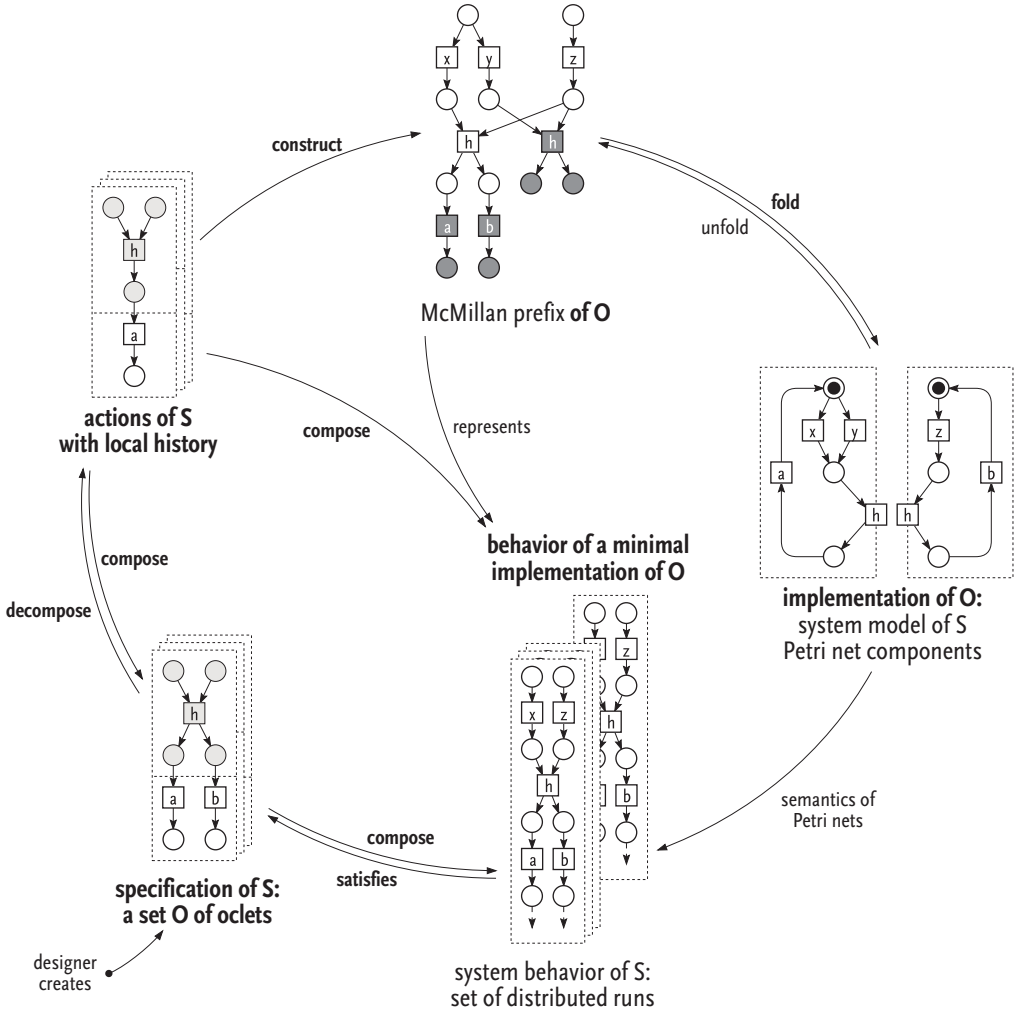


Figure 1.9. Overview on the results of this thesis.

of O. A synthesis algorithm has to construct a minimal implementation of O: a system that exhibits $\hat{R}(O)$.

Modeling systems with scenarios

The composition and decomposition operators on oclets yield canonical *operational semantics* for scenarios. Every scenario equivalently decomposes into its single actions with a *local history*. An action a of a scenario is *enabled* at a distributed run π if π ends with a's history. Composing run π with action a yields an occurrence of a. We show that these operational semantics construct for every oclet specification O exactly the behavior $\hat{R}(O)$ of a minimal implementation of O.

Because of these operational semantics, a system designer may *model* the behavior of a distributed system with scenarios. She simply describes all scenarios of the system; the operational semantics derive the system's behavior. Different scenarios may overlap. This allows for organizing system models *behaviorally*, for example, by distinguishing standard behavior and exceptions as shown in Figure 1.3. We show that oclets are more expressive than Petri nets and allow for more compact, but still intuitive system models as illustrated by Figures 1.3 and 1.4.

Operational semantics of oclets also technically generalize semantics of Petri nets. This allows us to generalize a symbolic state space construction technique from Petri nets to oclets: we adapt McMillan's unfolding approach [88] to represent the behaviors $\hat{R}(O)$ of a minimal implementation of an oclet specification O in a *finite* structure called the *McMillan prefix* of O. Technically, we extend McMillan's technique by the notion of a *history* and define an algorithm for constructing the McMillan prefix of O. This algorithm allows to *verify* behavioral properties of systems that are modeled with scenarios. We present experimental results confirming the feasibility of this technique in an industrial setting.

Synthesizing components from scenarios

Using the formal model of oclets, we contribute a novel algorithm for synthesizing components from a scenario-based specification. We first show that despite the restricted expressive power of oclets compared to LSCs, the synthesis problem has no general solution.

We then identify *boundedness* as a sufficient criterion for successful synthesis. Intuitively, an oclet specification O is bounded if a component sends only a bounded number of message to any other component before receiving a reply. In other words, the number of messages that may be in transit between components is bounded. We present an algorithm that *synthesizes for any bounded oclet specification* O Petri nets $\Sigma_1, \ldots, \Sigma_k$ which model the components described in O. We prove that, by construction, the composition $\Sigma_1 \oplus \ldots \oplus \Sigma_k$ is a *minimal* implementation of O. The synthesis algorithm is based on the McMillan prefix of O. With respect to the synthesis problem for scenarios, we contribute the following.

- Oclets are more flexible than HMSCs: oclets can be positioned between HMSCs and LSCs in terms of flexibility of specifying the behavior of distributed systems because of the notion of a history.
- Compared to existing synthesis algorithms, we identify a larger class of scenario-based specifications from which synthesis to Petri nets succeeds: oclets are more expressive than currently known sub-classes of HMSCs with successful synthesis to Petri nets [108, 13].
- The synthesis algorithm presented in this thesis does not restrict the shape or the independence of the synthesized components $\Sigma_1, \ldots, \Sigma_k$. Each component Σ_i is a Petri net and each bounded Petri net can be synthesized. Most important, the synthesis algorithm introduces no synchronization between the components except those described in the specification O.

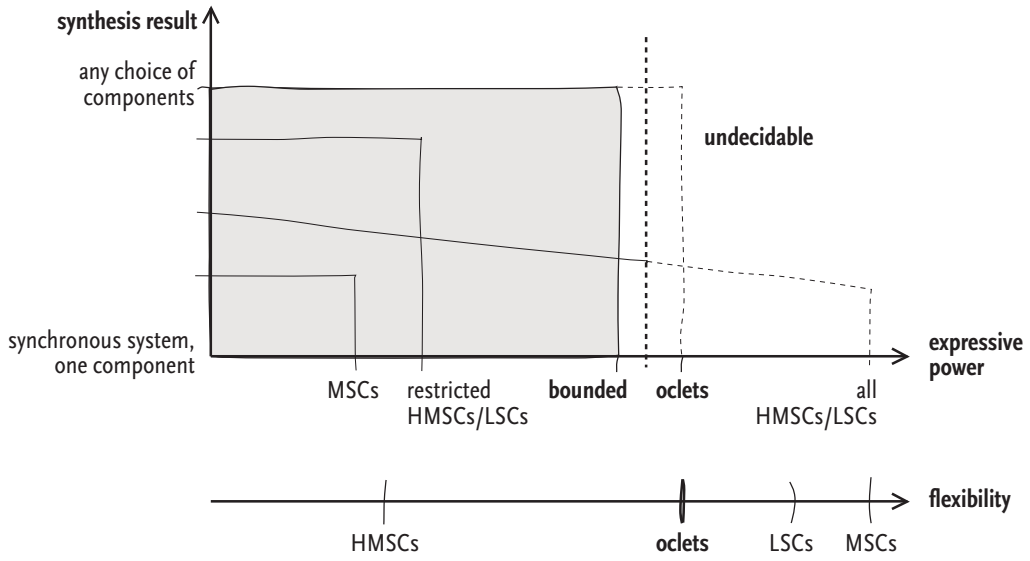


Figure 1.10. Qualitative illustration of the contribution of this thesis regarding flexibility and synthesis capabilities.

– Oclets embed Petri nets: for each Petri net exists an equivalent oclet specification. Consequently, our synthesis algorithm can synthesize every bounded Petri net and every bounded composition $\Sigma_1 \oplus \ldots \oplus \Sigma_k$ of Petri net components.

Altogether, oclets balance the trade-off between expressive power, flexibility, and synthesis capabilities as illustrated in Figure 1.10. The model of oclets uses the same technical notions from Petri nets to formalize scenarios, system behavior, and implementations. The result of this thesis have been implemented in our prototype tool GRETA that is available at `http://service-technology.org/greta`.

Road map

We conclude the introduction with a road map to this thesis as sketched in Figure 1.11.

Part I. After this introduction, Chapter 2 provides the formal background for this thesis. We introduce *Petri nets* as a formal model to describe distributed systems, and we develop the notion of a *distributed run* in the way required for the results of this thesis. Chapter 2 closes with the formal definitions of distributed runs of Petri nets.

Part II identifies a minimal class of scenario-based specifications that is expressive enough to specify the behavior of distributed systems in a flexible way. To this

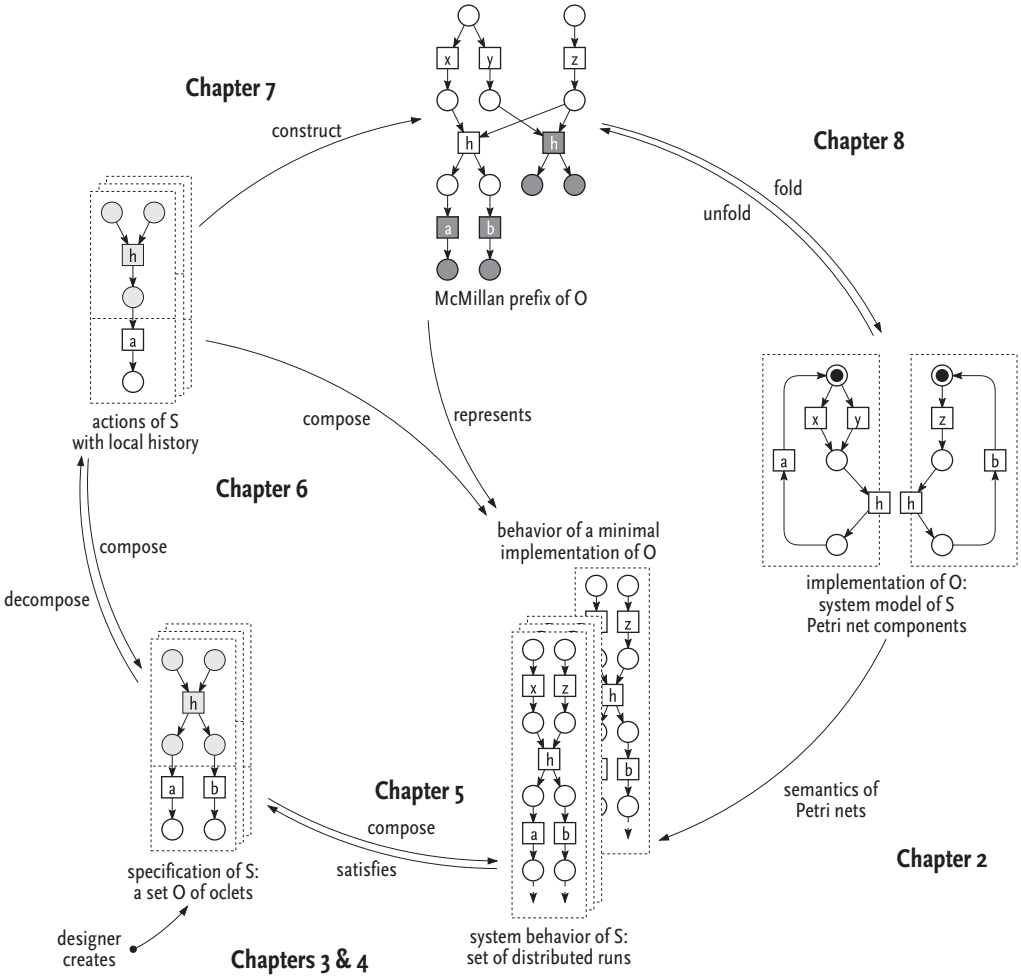


Figure 1.11. Road map to the following chapters.

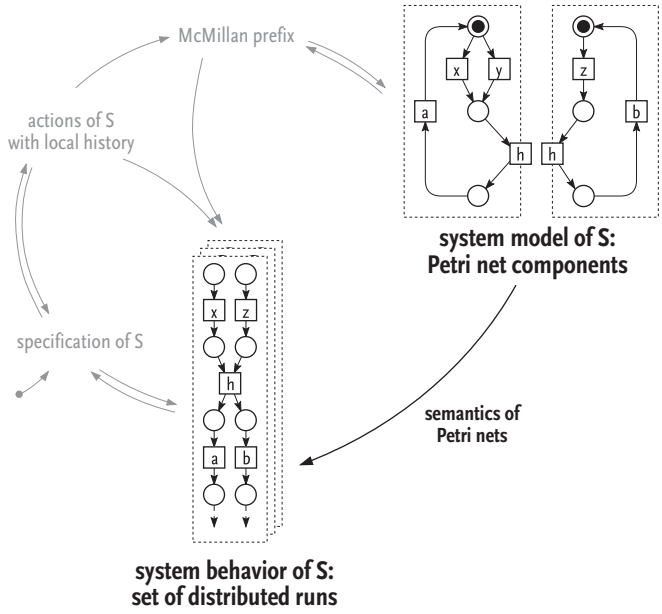
end, Chapter 3 critically reviews existing techniques for scenarios and identifies a *minimal* set of notions that suffice for specifying the behavior of distributed systems with scenarios. In Chapter 4, we formalize these minimal notions for scenarios in the model of *oclets* by defining a *declarative semantics* of scenario-based specifications in the style of LSCs. We discuss several properties of oclets, show how to increase their expressive power, and demonstrate how to specify system behavior with oclets.

Part III extends oclets to a technique for modeling system behavior with scenarios. In Chapter 5, we define a simple *behavior-preserving* composition operator on oclets. This composition operator canonically constructs for each oclet specification O the unique least set of runs that satisfies O. Chapter 6 introduces a

complimentary decomposition operator. We apply this decomposition operator to define operational semantics for scenario-based specifications in the style of Petri nets.

Part IV generalizes Petri net analysis techniques to the domain of scenario-based specifications and solves the synthesis problem. In Chapter 7, we generalize McMillan's technique of finite complete prefixes of unfoldings to compute the state space of an oclet specification O in a symbolic representation. This technique allows to verify behavioral properties of O which we demonstrate by experimental results. Chapter 8 extends the technique of Chapter 7 and defines an algorithm for synthesizing components from a scenario-based specification. We summarize the results of this thesis in the concluding Chapter 9 where we also discuss open problems and future research.

2. Background

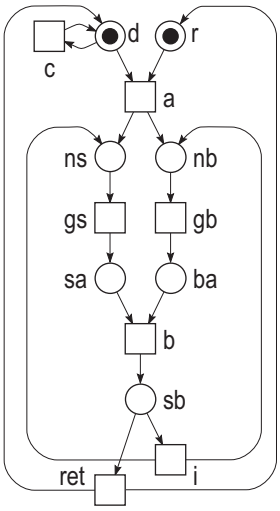


This chapter repeats some well-established notions for describing distributed systems and their behavior. We present the corresponding definitions, and formal notation to the necessary extent.

Section 2.1 presents Petri nets as a formalism for modeling distributed systems and their behavior. We define the semantics of Petri nets in terms of sequential runs. Section 2.2 takes a step back and considers sequential runs as a general *behavioral model* for describing system behavior independent of a specific system model. We explain the disadvantages of sequential runs for describing the behavior of distributed systems. Finally, Sections 2.3 and 2.4 introduce *distributed runs*, which we use as our behavioral model of distributed systems in all forthcoming chapters of this thesis.

2.1. Petri Nets

A *Petri net* is a state-based model of a discrete system. This model assumes that the system can reach a number of *local states* and can execute a number of *actions*; local states and actions depend on each other. Figure 2.1 depicts an example of such a system in the syntax of Petri nets.



The Petri net N models a flood protection process at a dike. Two dike workers are cooperating: The first worker, depicted on the left, is initially on a dike (d) while the second worker, depicted on the right, is initially ready (r). The first worker moves along the dike and checks the dike's condition (c). If he finds a leak, he alerts (a) the second worker and both workers start building a sand bag barrier. To this end, they need sand and bags (ns, nb) which they get independently of each other (gs, gb). Once sand and bags are available (sa, ba), the workers synchronize and build (b) a sand bag barrier (sb). If the workers realize that the current barrier is insufficient (i), they need more sand and bags. Otherwise, both workers return (ret) to their initial state.

Figure 2.1. A Petri net system $\Sigma = (N, m_0)$ modeling a flood protection process.

Syntax of Petri nets. A Petri net describes a system like the one in Figure 2.1 as a directed graph with two kinds of *nodes* called *places* and *transitions*. A transition describes an active system element, for instance, an action of the system, and is drawn as a box. A place describes a passive system element, for instance, a precondition for an action or a message channel, and is drawn as a circle or ellipse. An *arc* connects a transition with a place or vice versa; it describes a dependency between an active and a passive system element. Thus, the syntax of a Petri net formally reads as follows. (The formal definitions presented subsequently are based on a few standard notions that are listed in Appendix A.1 for completeness.)

Definition 2.1 (Petri net). A *Petri net* $N = (P, T, F)$ consists of two disjoint, *countable* sets P and T and a relation $F \subseteq (P \times T) \cup (T \times P)$. Each element of P is a *place*, each element of T is a *transition*; each element of F is an *arc* of the Petri net. ⌟

This definition of a Petri net permits that a net has infinitely (but countably) many places and transitions. We will consider infinite nets later. A Petri net which models a system is finite in any case.

We now fix some notions and conventions for Petri nets which we will use in all following chapters. Let $N = (P, T, F)$ be a Petri net.

1. We define $X_N := P \cup T$ and write $x \in X_N$ instead of $x \in P \cup T$; each $x \in X_N$ is a *node* of N.
2. The *pre-set* of $x \in X_N$ is $pre_N(x) := \{y \mid (y, x) \in F\}$; its *post-set* is $post_N(x) := \{y \mid (x, y) \in F\}$. If there is no confusion about N, we abbreviate ${}^\bullet x := pre_N(x)$ and $x^\bullet := post_N(x)$.
3. The *minimal* nodes of N are $\min N := \{x \in X_N \mid {}^\bullet x = \emptyset\}$; the *maximal* nodes of N are $\max N := \{x \in X_N \mid x^\bullet = \emptyset\}$.

A node in the pre-set of a transition is a *pre-place* of this transition. The notions of *post-place*, *pre-* and *post-transition* are defined correspondingly.

A Petri net may have infinitely many nodes. Nevertheless, we confine ourselves to Petri nets with finite pre-sets and finite post-sets. We further assume that a transition's pre-set and post-set are non-empty. The latter restriction rules out abnormal system behavior; it has no influence on the practical applicability of our results.

We write $N, M, \ldots$ for nets, possibly with indices, e.g., N', N_1. We refer to the parts of a net by P_N, T_N, etc. and abbreviate symbols like P_{N_1} to P_1.

In the following chapters, we compare and compose different Petri nets. To avoid technical difficulties, we strictly type places and transitions of nets. $\mathcal{P}$ denotes the universe of all places, and $\mathcal{T}$ the universe of all transitions of all nets. We assume $\mathcal{P} \cap \mathcal{T} = \emptyset$, and we write $\mathcal{X} = \mathcal{P} \cup \mathcal{T}$ for the universe of all nodes of all nets. For every place p (transition t) of every Petri net holds: $p \in \mathcal{P}$ ($t \in \mathcal{T}$).

Semantics of Petri nets. We now turn to the semantics of Petri nets. As said before, a Petri net N describes a distributed system S. The semantics of N formally describes the *behavior* of S. N describes a system state as a *marking*, which is a distribution of *tokens* on N's places. A token is drawn as a block dot inside a place. A token on a place describes, among others, a message in a channel or that a specific condition holds. For example, the places d and r in Figure 2.1 each have one token expressing that the first worker is on the dike (d) and that the second worker is ready (r). A place can have several tokens, which can be used to describe several messages in the same message channel.

Definition 2.2 (Marking, Petri net system). A marking of a Petri net N is a mapping $m : P_N \to \mathbb{N}$ that assigns each place $p \in P_N$ a non-negative number of tokens $m(p)$. If at least one token is assigned to a place p, i.e., $m(p) > 0$, then p is *marked*. A *Petri net system* $\Sigma = (N, m_0)$ is a Petri net N together with an *initial* marking m_0 of N. ⌟

A marking m of a finite Petri net N is frequently written as $[k_1 \cdot p_1, \ldots, k_n \cdot p_n]$ with p_i the places of N and $k_i = m(p_i)$. Nonmarked places ($m(p_i) = 0$) and factors $k_i = 1$ will be skipped. For example, the initial marking of N of Figure 2.1 is $[\mathsf{d}, \mathsf{r}]$.

The state of a system changes when an action of the system *occurs*; this yields the system's behavior. In a Petri net N, an occurrence of a transition t represents an occurrence of a system action. Technically, an occurrence of t "moves" tokens through the net. Thereby the directed arcs denote how each occurrence of t

depends on and influences tokens on places as follows: t is *enabled* at a marking m of N iff each *pre-place* of t has a token in m. If t is enabled at m, then t can *occur* by consuming a token from each pre-place of t and producing a token on each post-place of t. This results in a *successor marking* m_t and yields the *step* $m \xrightarrow{t} m_t$ of N.

Figure 2.2 illustrates the occurrence of a transition. The marking on the left enables transition t; an occurrence of t yields the successor marking depicted on the right.

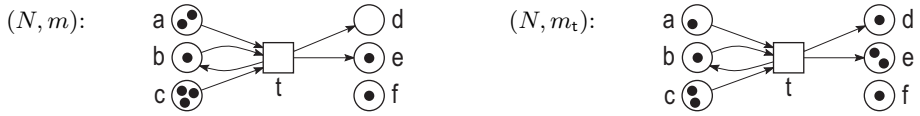


Figure 2.2. An occurrence of transition t in the marking $m = [2\mathsf{a}, \mathsf{b}, 3\mathsf{c}, \mathsf{e}, \mathsf{f}]$ (left) yields the successor marking $m_{\mathsf{t}} = [\mathsf{a}, \mathsf{b}, 2\mathsf{c}, \mathsf{d}, 2\mathsf{e}, \mathsf{f}]$ (right).

As usual, a (possibly empty) sequence of steps of N, $m_1 \xrightarrow{t_2} m_2 \xrightarrow{t_3} m_3 \xrightarrow{t_4} \ldots$, is a *sequential run* of N. For instance, the sequence

$$\begin{matrix}[\mathsf{d},\\ \mathsf{r}]\end{matrix} \xrightarrow{\mathsf{c}} \begin{matrix}[\mathsf{d},\\ \mathsf{r}]\end{matrix} \xrightarrow{\mathsf{a}} \begin{matrix}[\mathsf{nb},\\ \mathsf{ns}]\end{matrix} \xrightarrow{\mathsf{gb}} \begin{matrix}[\mathsf{ba},\\ \mathsf{ns}]\end{matrix} \xrightarrow{\mathsf{gs}} \begin{matrix}[\mathsf{ba},\\ \mathsf{sa}]\end{matrix} \xrightarrow{\mathsf{b}} [\mathsf{sb}] \xrightarrow{\mathsf{ret}} \begin{matrix}[\mathsf{d},\\ \mathsf{r}]\end{matrix} \tag{2.1}$$

is a sequential run of the Petri net N in Figure 2.1. Formally, the semantics of a Petri net system $\Sigma = (N, m_0)$ reads as follows.

Definition 2.3 (Sequential semantics of Petri nets). Let N be a Petri net and let m be a marking of N. Transition $t \in T_N$ is *enabled* at m iff $m(p) > 0$, for all $p \in {}^\bullet t$. If t is enabled at m, then an *occurrence* of t in m yields the *successor marking* m_t of N with

$$m_t(p) := \begin{cases} m(p) - 1, & \text{if } p \in {}^\bullet t \text{ and } p \notin t^\bullet \\ m(p) + 1, & \text{if } p \notin {}^\bullet t \text{ and } p \in t^\bullet \\ m(p), & \text{otherwise.} \end{cases}$$

The triple (m, t, m_t) is called *step* of N which we denote $m \xrightarrow{t} m_t$. A (possibly empty) sequence of steps of N, $m_1 \xrightarrow{t_2} m_2 \xrightarrow{t_3} m_3 \xrightarrow{t_4} \ldots$ is a *sequential run* of N. A marking m_n is *reachable* from a marking m_1 in N iff there exists a sequential run $m_1 \xrightarrow{t_2} m_2 \xrightarrow{t_3} \ldots \xrightarrow{t_n} m_n$ of N. The *sequential semantics* of a Petri net system $\Sigma = (N, m_0)$ is the set of all sequential runs of N that start in m_0. ⌟

Any prefix of a run of a Petri net N is also a run of N. Our example system in Figure 2.1 has infinitely many runs of infinite length, varying, for example, in the number of loop cycles after each occurrence of the alert transition a.

We use the following basic operations and relations on Petri nets in the forthcoming sections and chapters. These operations and relations are canonical extensions of corresponding notions from set algebra.

Definition 2.4 (Basic notions on Petri nets). Let N_1, N_2 be two Petri nets.

1. N_1 is a *sub-net* of N_2, denoted $N_1 \subseteq N_2$ iff $P_1 \subseteq P_2$, $T_1 \subseteq T_2$, and $F_1 \subseteq F_2$.
2. $N_1 \cup N_2 := (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2)$ denotes the *union* of N_1 and N_2,
3. $N_1 \cap N_2 := (P_1 \cap P_2, T_1 \cap T_2, F_1 \cap F_2)$ their *intersection*, and
4. $N_1 - N_2 := (P_1 \setminus P_2, T_1 \setminus T_2, F_1 \setminus F_2)$ their *difference*.
5. The *empty* Petri net is denoted $\emptyset := (\emptyset, \emptyset, \emptyset)$. ⌟

Describing components. A distributed system consists of several components that can interact with each other. Each system component has its own *local* state and the capability to *locally* step from one of its local states to another of its local states *independent* of the other system components, unless two components need to interact. Practically, any technical or organizational system that consists of two or more spatially disjoint components is distributed.

A Petri net N naturally describes a distributed system because an occurrence of a transition t of N only depends on its pre-places ${}^\bullet t$ and influences its post-places $t^\bullet$. For example, we can describe the flood alert process of Figure 2.1 also as a composition of the two components N_l and N_r on the left of Figure 2.3. The Petri net N_l describes the behavior of the first dike worker in our flood protection process, the net N_r the behavior of the second worker. The dashed lines indicate where N_l and N_r *synchronize* for interaction. We can compose N from N_l and N_r along their synchronized places and transitions. Intuitively, the composition of components is their union $N = N_l \cup N_r$ where N_l and N_r are disjoint except for the synchronized places and transitions.

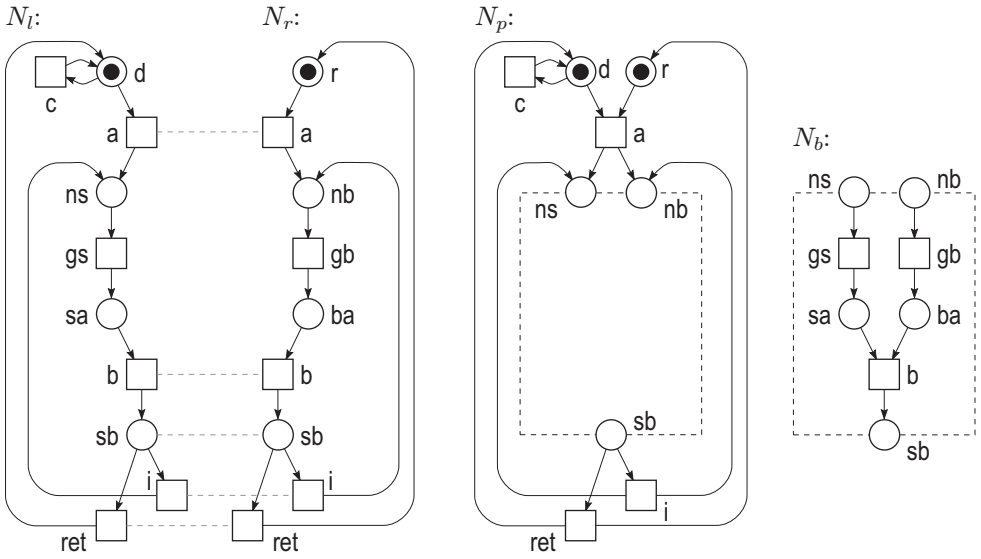


Figure 2.3. Two decompositions of the Petri net N of Fig. 2.1 into two sequential components N_l and N_r (left) and into two asynchronous components N_p and N_b (right).

We may also choose other components for N. For example, the components N_p and N_b on the right of Figure 2.3 describe how the dike workers prepare building the sand bag barrier and actually build the barrier, respectively. N_p and N_b interact *asynchronously*: both nets only share the places ns, nb, and sb. Their composition $N_p \cup N_b = N$ also describes the entire process of Figure 2.1. Which choice of components is appropriate to describe a given system depends on various factors. We return to this topic in Chapter 8.

2.2. Sequential Runs

Sequential runs as defined for Petri nets in the preceding section are a *behavioral model* of discrete systems. A sequential run $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$ of a system is a qualitative abstraction of an actual evolution of the system over time. A sequential run simply enumerates the visited states $s_0, s_1, s_2, \ldots$ and the action occurrences $a_1, a_2, \ldots$ regardless of how long the system resides in a state or how long it takes to move from one state to another state. A sequential run can also be just a sequence of states $\langle s_0, s_1, s_2, \ldots \rangle$ or actions $\langle a_1, a_2, \ldots \rangle$, depending on which phenomena one wants to study. In a sequential run, the same state can be visited multiple times. Correspondingly, the same action can occur multiple times.

A state s_i of a sequential run subsumes all relevant state information of the entire system at a given moment in time; in other words, s_i is a *global state*. Correspondingly, an action a_{i+1} moves the entire system from one global state s_i to another global state s_{i+1}; in other words, $s_i \xrightarrow{a_{i+1}} s_{i+1}$ is a *global step*.

Assume we describe each possible execution of a system *Sys* as a sequential run. The set of all these runs describes the complete *system behavior* of *Sys*. Although we might not know the concrete system, we can study its behavioral properties by its set of sequential runs. For this reason, sequential runs are a *behavioral model* for discrete systems.

Sequential runs are accepted as a behavioral model for discrete systems and have been studied extensively using many different formalisms. While two different formalisms may describe the same system in different ways using different concepts, their sets of sequential runs may be identical or isomorphic. In this sense, the behavioral model of sequential runs serves as the common denominator between these formalism. To abstract from any specific formalism is used to describe a system, we conceptually describe a system's behavior by its set of (sequential) runs. This set describes all possible executions of the system and may contain infinitely many infinite runs.

Sequential Runs and Distributed Systems The behavioral model of sequential runs using global states and global steps naturally suits systems that are also executed sequentially. But this model has disadvantages for describing the behavior of *distributed systems*. The behavior of a distributed system emerges from the behavior of its components and their interaction. This emergence of behavior is rather difficult to describe with sequential runs. We explain some phenomena of sequential runs that arise in distributed systems in the following.

We consider the two components N_l and N_r of Figure 2.3 which describe the behavior of the left and the right dike worker of the flood alert process, respectively. Each net has purely sequential behavior. The following is a sequential run of N_l:

$$[\mathsf{d}] \xrightarrow{\mathsf{c}} [\mathsf{d}] \xrightarrow{\mathsf{a}} [\mathsf{ns}] \xrightarrow{\mathsf{gs}} [\mathsf{sa}] \xrightarrow{\mathsf{b}} [\mathsf{sb}] \xrightarrow{\mathsf{ret}} [\mathsf{d}] \ . \tag{2.2}$$

A sequential run of N_r is

$$[\mathsf{r}] \xrightarrow{\mathsf{a}} [\mathsf{nb}] \xrightarrow{\mathsf{gb}} [\mathsf{ba}] \xrightarrow{\mathsf{b}} [\mathsf{sb}] \xrightarrow{\mathsf{ret}} [\mathsf{r}] \ . \tag{2.3}$$

Each of these runs describes how the *local state* of each dike worker changes as each worker's actions occur. We obtain the entire system N by composing N_l and N_r along their joint places and transitions. The entire system N exhibits the behavior of N_l and N_r together, for instance the run (2.1) which we repeat here:

$$\begin{bmatrix}\mathsf{d},\\ \mathsf{r}\end{bmatrix} \xrightarrow{\mathsf{c}} \begin{bmatrix}\mathsf{d},\\ \mathsf{r}\end{bmatrix} \xrightarrow{\mathsf{a}} \begin{bmatrix}\mathsf{nb},\\ \mathsf{ns}\end{bmatrix} \xrightarrow{\mathsf{gb}} \begin{bmatrix}\mathsf{ba},\\ \mathsf{ns}\end{bmatrix} \xrightarrow{\mathsf{gs}} \begin{bmatrix}\mathsf{ba},\\ \mathsf{sa}\end{bmatrix} \xrightarrow{\mathsf{b}} [\mathsf{sb}] \xrightarrow{\mathsf{ret}} \begin{bmatrix}\mathsf{d},\\ \mathsf{r}\end{bmatrix} \ .$$

We can obtain the runs (2.2) and (2.3) from (2.1) by hiding the upper and lower part of the visited markings of (2.1), respectively. Though, the run (2.1) does not follow directly from the runs (2.2) and (2.3).

For instance, the step $[\mathsf{d},\mathsf{r}] \xrightarrow{\mathsf{c}} [\mathsf{c},\mathsf{r}]$ of (2.1) occurs in (2.2) as the step $[\mathsf{d}] \xrightarrow{\mathsf{c}} [\mathsf{d}]$ but it does not occur in (2.3). The same observation can be made on the step $[\mathsf{ba},\mathsf{ns}] \xrightarrow{\mathsf{gs}} [\mathsf{ba},\mathsf{sa}]$ of run (2.1). In both cases, N makes a step due to an occurrence of a transition of N_l while all tokens on places of N_r remained untouched. This phenomenon of repeating a local state of a component in a run of a distributed system when another local state changes is called *stuttering* [114, p.438].

Another phenomenon is that the run (2.1) puts the occurrences of gb and gs in a specific order that does not follow from the runs (2.3) and (2.2). Transitions gb and gs have disjoint presets. Thus an occurrence of gb does not depend on an occurrence of gs, and vice versa. Both transitions are *independent*. For this reason, we may also observe gb and gs occurring in the reverse order compared to run (2.1):

$$\begin{bmatrix}\mathsf{d},\\ \mathsf{r}\end{bmatrix} \xrightarrow{\mathsf{c}} \begin{bmatrix}\mathsf{d},\\ \mathsf{r}\end{bmatrix} \xrightarrow{\mathsf{a}} \begin{bmatrix}\mathsf{nb},\\ \mathsf{ns}\end{bmatrix} \xrightarrow{\mathsf{gs}} \begin{bmatrix}\mathsf{nb},\\ \mathsf{sa}\end{bmatrix} \xrightarrow{\mathsf{gb}} \begin{bmatrix}\mathsf{ba},\\ \mathsf{sa}\end{bmatrix} \xrightarrow{\mathsf{b}} [\mathsf{sb}] \xrightarrow{\mathsf{ret}} \begin{bmatrix}\mathsf{d},\\ \mathsf{r}\end{bmatrix} \ . \tag{2.4}$$

This phenomenon is called *interleaving* of steps because of independent transitions [114, p.435]. The runs (2.1) and (2.4) are two different interleavings of the sequential runs (2.3) and (2.2). Yet, (2.1) and (2.4) represent the *same* distributed behavior.

Working with different representations of the same behavior raises a complexity problem: the number of different interleavings is exponential in the number of pairwise independent transitions. The Petri net in Figure 2.4 illustrates this exponential growth. The transitions $\mathsf{t}_1, \ldots, \mathsf{t}_n$ are pairwise independent. Thus, the net has the following behavior: first s occurs, followed by $\mathsf{t}_1, \ldots, \mathsf{t}_n$ in arbitrary order and finally e occurs. There are $n!$ different sequential orders of $\mathsf{t}_1, \ldots, \mathsf{t}_n$,

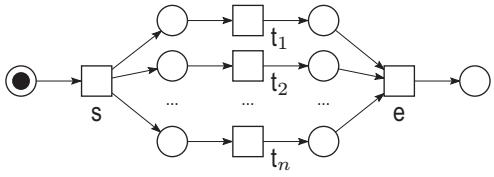


Figure 2.4. The transition $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are pairwise independent; the net has $n!$ different sequential runs.

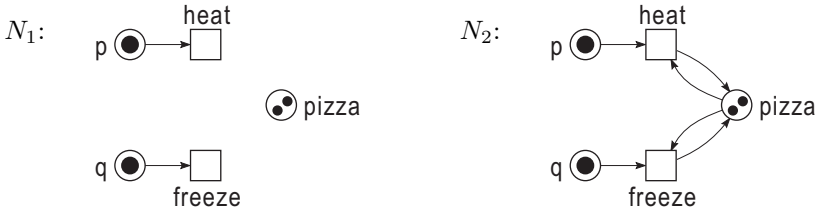


Figure 2.5. Both nets have the same sequential runs, but transitions heat and freeze depend on pizza in the left net and are independent in the right net.

thus the net has $n!$-many different sequential runs that express the *same* behavior.

Moreover, sequential runs cannot distinguish certain kinds of systems. We adapted the following example from [27]. The two nets in Figure 2.5 have the same sequential runs $[\mathsf{p}, \mathsf{q}, 2 \cdot \mathsf{pizza}] \xrightarrow{\mathsf{heat}} [\mathsf{q}, 2 \cdot \mathsf{pizza}] \xrightarrow{\mathsf{freeze}} [2 \cdot \mathsf{pizza}]$ and $[\mathsf{p}, \mathsf{q}, 2 \cdot \mathsf{pizza}] \xrightarrow{\mathsf{freeze}} [\mathsf{p}, 2 \cdot \mathsf{pizza}] \xrightarrow{\mathsf{heat}} [2 \cdot \mathsf{pizza}]$. Though, the nets N_1 and N_2 obviously describes different systems. In N_2, transitions heat and freeze depend on place pizza which is not the case in N_1. Thus, the pizzas in the left net are not involved in occurrences of heat and freeze and both transition occur *independently*. In the right net in turn, the occurrences of heat and freeze are ordered via place pizza. This difference is not represented in the sequential runs. In other terms, the behavioral model of sequential runs does not represent *causal dependencies* (or the lack thereof) between transitions and places.

Altogether, stuttering and interleaving arise because a sequential run describes behavior in terms of global states and global steps. Moreover, causal dependencies between different steps vanish because a sequential run orders states and steps by time. These phenomena give rise to study the behavior of a distributed system at a different level of granularity using *local states* and *local steps* ordered by *causality*. This leads to the notion of a *distributed run*.

2.3. Distributed Runs

This section introduces the behavioral model of distributed runs, also known as *partially ordered* runs. Distributed runs are a behavioral model that specifically suits for describing the behavior of distributed systems [104, 48, 30, 67]. In this

behavioral model, a distributed run represents a possible execution of a distributed system. The complete behavior of a system is described by its *set of distributed runs*.

2.3.1. Distributed Runs in a Nutshell

The idea. Before introducing a formal notion of distributed runs, we present all relevant concepts by the help of an example. Figure 2.6 depicts a distributed run of the Petri net of Figure 2.1. We use the same graphical notation as for Petri nets, but their interpretation is different.

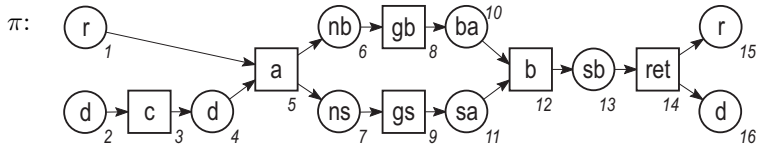


Figure 2.6. A distributed run of the Petri net N of Fig. 2.1.

The distributed run π in Figure 2.6 consists of *events* (drawn as boxes) and *conditions* (drawn as circles). Each event and each condition has a *label* which is inscribed. Further, we attached an identifying number to each node of π. A condition with label p represents a token on place p. An event with label t represents an occurrence of transition t. The incoming (outgoing) arcs of an event denote which tokens the event consumes (produces). A distributed run usually contains several events and conditions with the same label; for instance, the conditions 2, 4, and 16 with label d in Figure 2.6. The distributed run π represents the same behavior as the sequential runs (2.1) and (2.4).

Depending on the level of detail of the observations, we may denote a distributed run also as a *partial order of events*. The Hasse diagram in Figure 2.7 denotes the run of Figure 2.1 in this way. Each arc in the diagram denotes a *direct successor* relation between two events whilst all transitive relations are omitted. Which representation is more appropriate depends on which phenomena one wants to study.

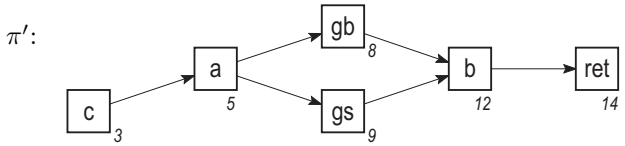


Figure 2.7. The distributed run π' represents the run π of Fig. 2.6 as a partial order of events.

Comparison to sequential runs. Compared to a sequential run, a distributed run describes system behavior in terms of local observations: a condition describes

that a *local state* is being visited whereas an event describes that an action *occurs locally* having only local effects.

In the example of Figure 2.6, we can decompose the distributed run π of the entire system N into smaller distributed runs π_l and π_r of the system's components N_l and N_r as shown in Figure 2.8.

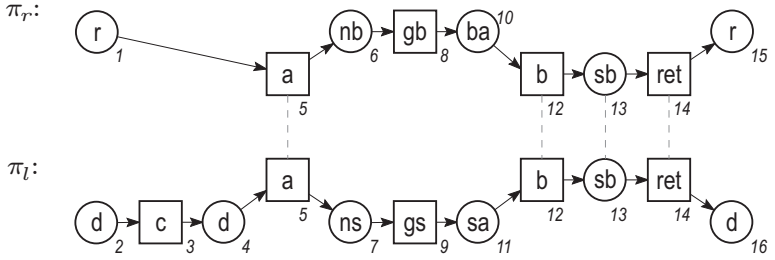


Figure 2.8. A decomposition of the distributed run π of Fig. 2.6 into distributed runs π_l and π_r of the nets N_l and N_r of Fig. 2.3.

In each run π_l and π_r, the conditions and events are sequentially ordered: both runs are sequential. Every sequential run can be expressed as a special distributed run. For example, π_l corresponds to the sequential run (2.2) and π_r corresponds to (2.3) of Section 2.2. Unlike for sequential runs, the run π of the entire system N of Figure 2.1 can be composed from the runs π_l and π_r of its parts N_l and N_r of Figure 2.3. Composing π_l and π_r along their joint nodes preserves the locality of their events and conditions.

Stuttering and interleaving, which we can observe in sequential runs, do not occur in distributed runs. For instance, the events 8 (gb) and 9 (gs) remain local in π. Thus, the "local steps" $[\mathsf{nb}] \xrightarrow{\mathsf{gb}} [\mathsf{ba}]$ from π_r and $[\mathsf{ns}] \xrightarrow{\mathsf{gs}} [\mathsf{sa}]$ from π_l also occur in π. Thereby the events 8 and 9 are unordered, or *concurrent*, in π. Altogether, the conditions and events of π are only *partially ordered*.

Because distributed runs do not need to interleave behavior, system behavior can be represented much more succinctly compared to sequential runs. The distributed run in Figure 2.9 is the only (maximal) distributed run of the net of Figure 2.4; it represent all $n!$ sequential runs of this net. We benefit from the succinctness of the representation in definitions and algorithms in the following chapters.

Finally, distributed runs make the *causal dependencies* of a system explicit. The run π in Figure 2.10 is the only distributed run of the net N_1 of Figure 2.5. In contrast, N_2 of Figure 2.5 has several fundamentally different distributed runs. For instance, π_1 produces one warm pizza, π_2 produces one frozen pizza, and π_3 produces one warm and one frozen pizza. Although all runs reach the same global state $[2 \cdot \mathsf{pizza}]$, they differ by the *history* of the pizzas and can be interpreted differently. The arcs in the different runs make explicit which token (i.e., pizza) was involved in which event. The sequential runs of N_2 do not distinguish these different behaviors. This ability of distinguishing different behaviors wrt. causality makes distributed runs a very expressive behavioral model. This expressive power and the notion of a history will be exploited from Chapter 3 on.

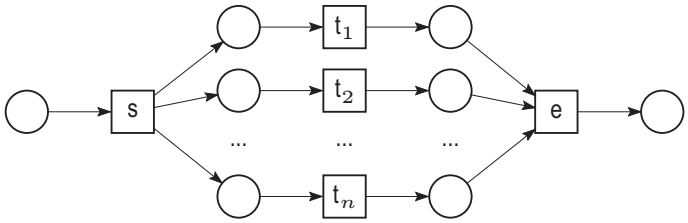


Figure 2.9. The only distributed run of the net in Fig. 2.4 represent $n!$ different sequential runs.

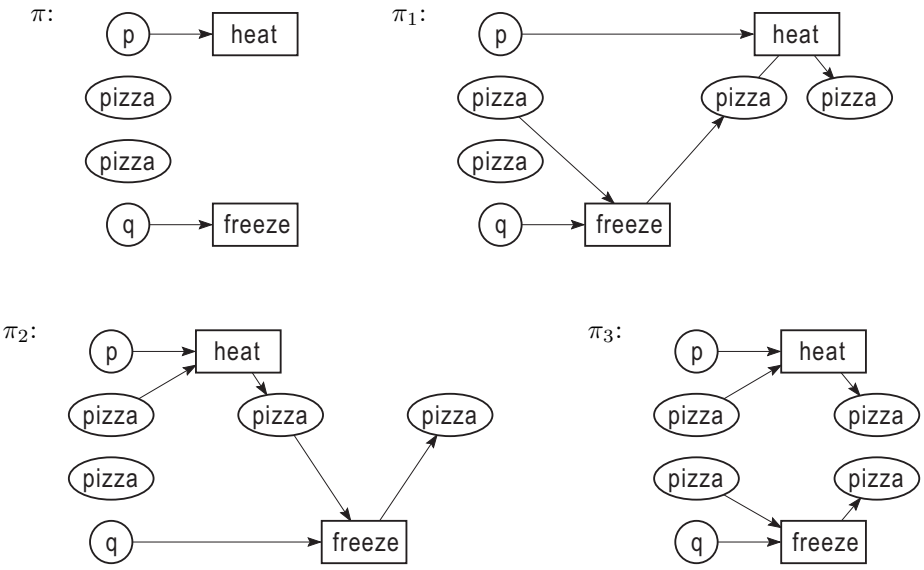


Figure 2.10. Different distributed runs of the nets N_1 and N_2 of Fig. 2.5.

This concludes our informal introduction to distributed runs. The following sections present a formal model of distributed runs that we use in the remainder of this thesis.

2.3.2. Conditions and Events

We want to use distributed runs as a general behavioral model for distributed systems. In the preceding example, a distributed run represents system behavior by ordering conditions and events in a specific way. In that example, a condition represents a local state of a system component whereas an event represents a local step of one system component or an interaction of several system components. We presented these notions by referring to places and transitions of a concrete Petri net. But our behavioral model of distributed runs shall be generally applicable to several formalisms which use other notions than Petri net places and transitions.

In this section, we provide a more general interpretation of events and conditions *which is independent of a concrete system model in a specific formalism.* The idea is to interpret events and conditions as *observations* on the actual system.

The behavioral model of sequential runs assumes an observer who distinguishes different states of the system. The observer detects when the system visits a specific state. Further, he detects when the system changes its state which he observes as a step. In correspondence with sequential runs, we assume that an observer makes the following observations on a distributed system.

- The observer can distinguish different local states of the system. Moreover, he can name a characteristic property that distinguishes every local state from all other local states, such as that a specific resource is available or that some variable has a specific value. In this thesis, we assume an observer who distinguishes only finitely many local states. Let $\mathcal{L}_B$ denote the set of all names of observable local states.
- A *condition* is the observation that a specific local state is being visited viz. the observation that the local state's characteristic property holds.
- The observer can distinguish different local actions of the system. Moreover, he can name a characteristic property that distinguishes every local action from all other local actions, such as sending a specific message or evaluating a query. Corresponding to names of states, $\mathcal{L}_E$ denotes the finite set of all names of observable actions.
- An *event* is an observable change in the system due to an occurrence of an action.

In analogy to a sequential run $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots$ we assume that observations alternate between events and conditions as shown in the examples of Section 2.3.1. The rationale is that every local action observably influences one or more local states. Thus, every event is directly preceded and succeeded by one or more conditions. Conversely, no local state changes without an action occurring: a local state is always visited or left via a local action. Thus, every condition is directly preceded and directly succeeded by at most one event, respectively. An in-depth discussion of this conceptualization of events and conditions is given by Holt [63].

2.3.3. Formal Definitions of Distributed Runs

With our conceptual background on events and conditions, we now provide a formal model for distributed runs. Our formal model is based on Engelfriet [35] and serves as a basis for the following chapters. We proceed as follows. Firstly, as we explained by our examples in the preceding section, a distributed run is a *partially ordered set* of events and conditions. Secondly, conditions and events alternate in a run. We formally represent this bipartite structure as a special Petri net called *causal net*. Finally, each event and each condition of a distributed run is *labelled* with a name as discussed in Section 2.3.2.

Let X be an arbitrary set. A *partial order* over X is a binary relation $\leq \subseteq X \times X$ that is reflexive, transitive, and anti-symmetric; see Definition A.4 in Appendix A.4.

To model distributed runs, we use partial orders where each $x \in X$ has only finitely many predecessors. This restriction corresponds to the idea that each event in a run is reached after finitely many predecessor events, i.e., a run has a finite history.

Definition 2.5 (Predecessor, successor, finitely preceded). Let X be a set, let $\leq$ be a partial order over X. The set $x{\downarrow_{\leq}} := \{y \mid y \leq x\}$ is the set of all (transitive) *predecessors* of $x \in X$. The (transitive) *successors* of $x \in X$ are $x{\uparrow_{\leq}} := \{y \mid x \leq y\}$. $\leq$ is *finitely preceded* iff each $x \in X$ has only finitely many predecessors. ⌟

We are interested in very specific partial orders over events and conditions. As described in Section 2.3.2, we consider events and conditions to alternate. Further, a condition has at most one predecessor event and at most one successor event. Such a structure is formalized in Petri net theory as a causal net.

Definition 2.6 (Causal net). A *causal net* $\pi = (B, E, F)$ is a Petri net where

1. the reflexive-transitive closure F^* of F is a partial order over $B \cup E$,
2. F^* is finitely preceded, and
3. each $b \in B$ has at most one pre-node and at most one post-node. ⌟

The elements of B are called *conditions*, the elements of E are called *events*. Like for Petri nets, we write $X_\pi = B \cup E$ for the *nodes* of π. The first requirement of Def. 2.6 is equivalent to π containing no cycles along the arcs F.

To fully describe a distributed run, we label its events and conditions. According to Section 2.3.2, an event (condition) represents an occurrence of a local action (a visit of a local state). Thus, we label each event with the name of the occurring local action and each condition with the name of the visited local state. We consider only runs over a finite set of *names* $\mathcal{L} = \mathcal{L}_E \cup \mathcal{L}_B$ partitioned into the names $\mathcal{L}_E$ of local actions and the names $\mathcal{L}_B$ of local states. Nevertheless, a distributed run may be infinite; in this case one or more names occur infinitely often in the run. Technically, a distributed run is just a *labeled* causal net.

Definition 2.7 (Distributed run). A *distributed run* over $\mathcal{L}$ is a causal net $\pi = (B, E, F, \ell)$ together with a *labeling* $\ell : B \cup E \to \mathcal{L}$ s.t. $\ell(b) \in \mathcal{L}_B$, for all $b \in B$ and $\ell(e) \in \mathcal{L}_E$, for all $e \in E$. We write $\varepsilon := (\emptyset, \emptyset, \emptyset, \emptyset)$ for the *empty* distributed run that contains no events and no conditions. ⌟

We have seen several examples of distributed runs in Section 2.3.1. A distributed run may be infinite. It formally describes an observed succession of events and conditions. The observations are limited to those names in $\mathcal{L}$. If the events and conditions in a distributed run π are ordered totally then π is a sequential run. In this case, each condition describes a global state and each event describes a global step.

Intermission: causal nets vs. partial orders of events

A distributed run is often formalized as a *labeled partial order* (LPO) of events that contains no conditions.[1] This section briefly relates labeled partial orders to causal nets.

Every causal net π canonically projects onto an LPO r_π by simply removing from π all conditions while preserving the partial order. Conversely, every LPO r canonically extends to a causal net π_r by placing a condition on every arc between two events in the Hasse diagram of r.[1]

However, these formalizations of distributed runs are not equivalent. For example, we may project in Figure 2.11 the causal net π onto the partial order r_π; the canonical extension of r_π to the causal net π' lacks some information compared to π.

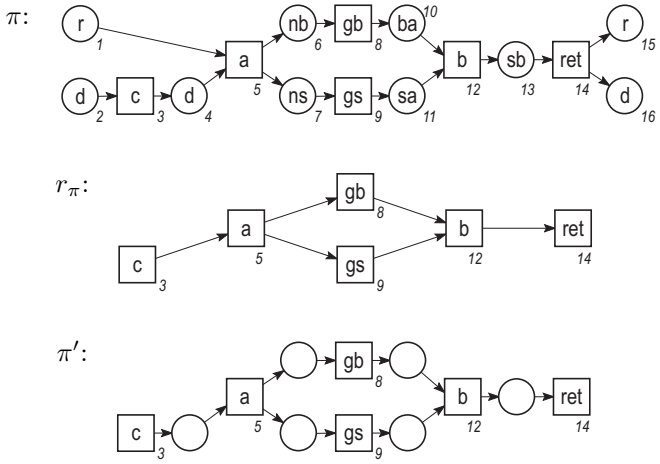


Figure 2.11. The causal net π can be projected onto the partially ordered set of events r_π; canonically extending r_π to a causal net yields π'.

A more subtle yet important difference between these notions is depicted in Figure 2.12. In the projection of the causal net π onto the LPO r_π all events are ordered sequentially. The dependency from a to c is transitive in r_π. Thus, b occurs between a and c as depicted on the right. The corresponding causal net π' is different from π, but only in terms of conditions. The order of events is the same in all four representations.

These examples show already that causal nets are more expressive than partially ordered sets of events. The canonical extension to causal nets neither defines labels of conditions nor pre-conditions (post-conditions) of events that have no predecessor (successor). Hence partial orders of events do not allow to structure states, i.e., they provide no information that a local state is visited. In this thesis, we will specifically rely on post-conditions of events that have no successor event. For this reason, we use the more general model of causal nets for representing

[1] See Appendix A.4 for corresponding formal definitions.

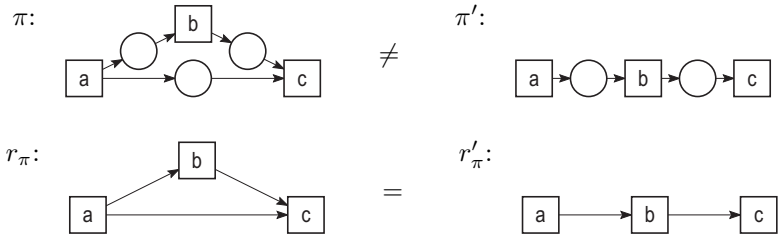


Figure 2.12. Labeled partial order do not preserve local states.

distributed runs. Nevertheless, we will omit conditions in a graphical representation of a run whenever they are not necessary for studying a specific phenomenon.

2.3.4. Basic Operations and Relations on Distributed Runs

This section defines some fundamental relations and operations on distributed runs. As a foundation, we inherit the basic operations and relations on Petri nets from Section 2.1. Specifically, $\min \pi$ ($\max \pi$) denotes the set of minimal (maximal) nodes of a run π that have no predecessor (successor). By B_π, E_π, and F_π we refer to π's conditions, events, and arcs, respectively. Likewise the notions $\pi_1 \subseteq \pi_2$ (π_1 contained in π_2), $\pi_1 \cap \pi_2$ (intersection of runs), and $\pi_1 \cup \pi_2$ (union of runs) holds for distributed runs. The forthcoming definitions exploit the specific structure of distributed runs.

None of our operations will change an event's or condition's name. To simplify our definitions we assume that the name of an event (condition) is an inherent property of this event (condition). We assume that for any two distributed runs $\pi_1 = (B_1, E_1, F_1, \ell_1)$ and $\pi_2 = (B_2, E_2, F_2, \ell_2)$ holds: if $x \in (X_1 \cap X_2)$, then $\ell_1(x) = \ell_2(x)$. Technically, we assume a *universal labeling* function ℓ that assigns each event or condition x of a distributed run "its" label $\ell(x) \in \mathcal{L}$ according to Definition 2.7. If confusion is safely avoided, we write for a distributed run over $\mathcal{L}$ simply the underlying causal net $\pi = (B, E, F)$ as a shorthand. The canonical labeling of π is the restriction $\ell|_{(B \cup E)}$ of ℓ to the conditions and events of π.

Dependency and Concurrency

A distributed run puts events and conditions in a specific partial order. This partial order has a reasonable interpretation in terms of causality. An arc (x, y) in a distributed run denotes a *direct causal dependency* from x to y. Causality is transitive: if there is a path along the arcs of run from x to y, then y *causally depends* on x. If there is no path, then x and y are *concurrent*.

Definition 2.8 (Depends on, concurrent, co-set). Let π be a distributed run and let $x, y \in X_\pi$ be two nodes of π. The node x (causally) *depends* on y in π, denoted $y \leq_\pi x$, iff $(y, x) \in F_\pi^*$. The nodes x and y are *concurrent*, denoted $x \,||_\pi\, y$, iff neither x depends on y nor y depends on x. A set $Y \subseteq X_\pi$ of pair-wise concurrent nodes is a *co-set*. ⌟

For example, in the run π in Figure 2.13, condition 15 depends on conditions 1 and 2 and is concurrent to condition 16. Using dependency and concurrency, we can reconstruct the notions of global state and global step from events and conditions.

Cuts and Configurations

As said in Section 2.2, a sequential run r of a system explicitly describes the global states which the system visits as r occurs. Moreover, all global steps that precede a global state s in r are necessary to reach s in r. There are two corresponding notions for distributed runs.

In a distributed run π, a "global state" consists of a set of local states that can be visited "together". In our formalization, a global state in π is described as a maximal co-set B of conditions, called a *cut*. Correspondingly, all events C that precede B are necessary in π to reach B. These events C form a *configuration*. More precisely, any set of events of π that contains all its predecessor events wrt. $\leq$ is a configuration of π and every finite configuration of π reaches a cut of π.

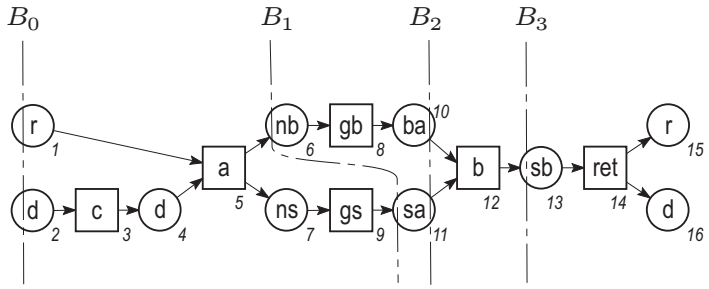


Figure 2.13. The distributed run π from Fig. 2.6 with some cuts.

Figure 2.13 illustrates cuts and configurations. The set $B_1 = \{6, 11\}$ of conditions is a cut representing the global state $[\mathsf{nb}, \mathsf{sa}]$. In contrast, the conditions $\{6, 10\}$ do not constitute a cut because 10 depends on 6. The configuration that precedes the cut B_2 is the set of events $C_1 = \{3, 5, 9\}$ representing occurrences of actions c, a, and gs. Event 8 represents an occurrence of action gb. Adding event 8 to configuration C_1 yields configuration $C_2 = \{3, 5, 9, 8\}$. C_2 reaches the cut $B_2 = \{10, 11\}$ which represents the global state $[\mathsf{ba}, \mathsf{sa}]$. In other words, adding event 8 to configuration C_1 corresponds to the global step $[\mathsf{nb}, \mathsf{sa}] \xrightarrow{\mathsf{gb}} [\mathsf{ba}, \mathsf{sa}]$. The formal definitions are as follows.

Definition 2.9 (Cut, configuration). Let π be a distributed run over $\mathcal{L}$. A maximal co-set B of conditions in π is a *cut* of π. A set C of events in π is a *configuration* of π iff for each event $e \in C$ and each $e' \in E_\pi$ with $e' \leq_\pi e$ holds $e' \in C$.

For a given cut B of π, the configuration that leads to B is $C_B := \{e \in E_\pi \mid \exists b \in B : e \leq_\pi b\}$. Conversely, the cut of π reached by a configuration C of π is $Cut(C) := (\min \pi \cup C^\bullet) \setminus {}^\bullet C$. ⌟

A configuration can be infinite. An infinite configuration does not lead to a cut. For instance, every distributed run has a maximal configuration which is the set of all its events.

Prefixes and Continuations

One of the most important relations on distributed runs is the prefix relation $\sqsubseteq$. Informally, a prefix $\pi \sqsubseteq \rho$ of a run ρ is an initial part of ρ s.t. each event and each condition of ρ either lies in π or causally follows after π. We are only interested in those prefixes where each event e of π also has all post-conditions $e^\bullet$ in π.

Definition 2.10 (Prefix). Let π and ρ be distributed runs over $\mathcal{L}$. The run π is a *prefix* of ρ, denoted $\pi \sqsubseteq \rho$, iff (1) $X_\pi \subseteq X_\rho$, (2) $\leq_\pi = \leq_\rho|_{(X_\pi \times X_\pi)}$ (the causal relations coincide in the nodes in π), (3) $\forall x \in X_\pi : x{\downarrow_\rho} \subseteq X_\pi$ (in ρ, a node of π has only predecessors from π), and (4) for each event $e \in E_\pi$ holds $post_\pi(e) = post_\rho(e)$. If π is a prefix of ρ, then ρ is a *continuation* of π. We call π a *technical* prefix of ρ, denoted $\pi \sqsubseteq_{\mathbf{t}} \rho$, iff π and ρ satisfy conditions (1)-(3). ⌟

Figure 2.14 depicts some examples: the runs π_1 and π_3 are prefixes of π; π_2 is only a technical prefix of π because the post-condition 7 of event 5 is missing; π_4 is *not* a (technical) prefix of π because condition 1 precedes event 5 but is not part of π_4.

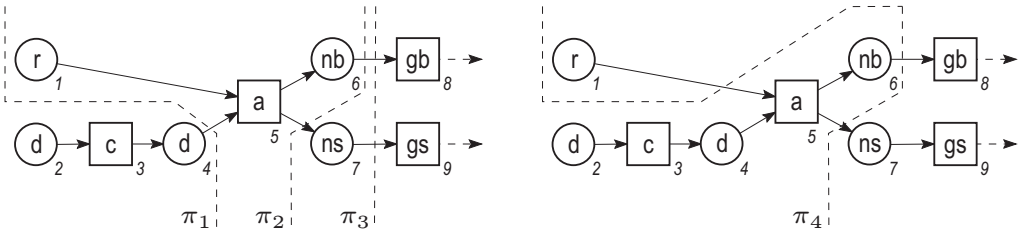


Figure 2.14. Some prefixes π_1-π_3 of the run π from Fig. 2.6, the run π_4 is not a prefix of π.

The following lemma simplifies some proofs in the forthcoming chapters; its proof is given in Appendix A.5.

Lemma 2.11: *Let π and ρ be distributed runs over $\mathcal{L}$. The following notions are equivalent:*

- $\pi \sqsubseteq_{\mathbf{t}} \rho$.
- $X_\pi \subseteq X_\rho$ *and* $F_\pi = F_\rho|_{X_\pi \times X_\pi}$ *and* $(x \in X_\pi \wedge (x, y) \in F_\rho) \Rightarrow y \in X_\pi$, *for all* $x, y \in X_\rho$.
- $X_\pi \subseteq X_\rho$ *and* $F_\pi = F_\rho|_{X_\rho \times X_\pi}$. ⋆

Corollary 2.12: *For any two distributed runs π, ρ over $\mathcal{L}$ holds: if $\pi \sqsubseteq \rho$ then $\pi \subseteq \rho$.* ⋆

Every distributed run canonically induces its set of complete prefixes; this notion lifts to sets of distributed runs.

Definition 2.13 (Prefix-closure, prefix-closed). For a set R of distributed runs, its *prefix-closure* is the set $\mathit{Prefix}(R) := \{\pi \mid \rho \in R, \pi \sqsubseteq \rho\}$; R is *prefix-closed* iff $\mathit{Prefix}(R) = R$. ⌟

A set of nodes Y of a distributed run π induces the prefix $\pi[Y]$ that consists of Y and all nodes that precede Y in π.

Definition 2.14 (Induced prefix). Let ρ be a distributed run. Let $Y \subseteq X_\rho$. The *Y-induced prefix* of ρ is the run $\pi = (B_\pi, E_\pi, F_\pi)$ with $X_\pi := \bigcup_{y \in Y} y{\downarrow_\rho}$; $B_\pi = B_\rho \cap X_\pi$; $E_\pi = E_\rho \cap X_\pi$; $F_\pi = F_\rho|_{X_\pi \times X_\pi}$. We write $\rho[Y] := \pi$. ⌟

For example, the conditions 6 and 7 in Figure 2.14 induce the prefix π_3 on the left. Lemma A.7 in Appendix A.5 states some technical properties of induced prefixes that will be used in some proofs later in this thesis.

Partial distributed runs

Besides prefixes and continuations of runs, we will also reason about arbitrary "parts" of a run. We call such a "part" a *partial* run. A partial run is simply sub-net of a distributed run. For example, the run π_l and π_r of Figure 2.8 are partial runs of π of Figure 2.6. Also, every prefix of a run is a partial run.

Definition 2.15 (Partial distributed run). Let π and ρ be distributed runs over $\mathcal{L}$. The run π is a *partial run* of ρ iff $\pi \subseteq \rho$. ⌟

One could think of a more restrictive notion of partial runs such as requiring π to be connected. However, Definition 2.15 suffices our needs in this thesis.

In the following, we will also use the notion of a "partial run" as an attribute of a distributed run: we call a distributed run π a *partial run* when we are looking for some, yet unknown, distributed run ρ s.t. π is a partial run of ρ. This attribute "π is a partial run" emphasizes that π does not describe a complete observation of occurrences of actions, but only some part of a larger observation. In Chapter 5, we shall compose distributed runs from partial runs.

2.4. Distributed Runs of Petri Nets

The preceding section introduced distributed runs to generally describe the behavior of a distributed system. This section defines the distributed runs of a Petri net system Σ. The distributed runs of Σ describe the behavior of the system that is modeled by Σ. Originally, distributed runs of Petri nets have been introduced as "processes", see [47, 102, 14]. The formal model presented here is based on [35, 14].

Let $\Sigma = (N, m_0)$ be a Petri net system, which describes some system S. The observable local actions and local states of S are described by the places and transitions of N: the transitions of N constitute the names $\mathcal{L}_E := T_N$ of events and the places of N constitute the names $\mathcal{L}_B := P_N$ of conditions. In a distributed run of Σ, we interpret events and conditions as follows:

1. a condition b with name $\ell(b) = p$ represents an occurrence of a token on place p,
2. an event e with name $\ell(e) = t$ represents an occurrence of transition t.

Because a transition t consumes tokens from ${}^\bullet t$ and produces tokens in $t^\bullet$, the events and conditions of a distributed run of Σ are ordered in a specific way. For any event e that represents an occurrence of t,

1. the pre-conditions ${}^\bullet e$ represent tokens on ${}^\bullet t$, and
2. the post-conditions $e^\bullet$ represent tokens on $t^\bullet$.

Like every sequential run of Σ, a distributed run π of Σ begins in the initial marking m_0. Because each condition represents a token on a place, the collection of *minimal* conditions of π which have no pre-event, i.e., $\min \pi$, together represent m_0. The following axiomatic definition states these properties formally.

Definition 2.16 (Distributed runs of a Petri net system, [35]). Let $\Sigma = (N, m_0)$ be a Petri net system. A distributed run $\pi = (B, E, F)$ over the nodes X_N of N is a distributed run of Σ iff the following properties hold:

1. $\ell(b) \in P_N$ for each $b \in B$, $\ell(e) \in T_N$ for each $e \in E$,
2. $\min \pi \subseteq B$ and for all $p \in P$ holds $m_0(p) = |\{b \in \min \pi \mid \ell(b) = p\}|$, and
3. for each $e \in E$ with $\ell(e) = t$ the labeling ℓ bijectively maps ${}^\bullet e$ to ${}^\bullet t$ and $e^\bullet$ to $t^\bullet$. ⌟

We have already seen several examples of distributed runs of a Petri net system in the preceding section, e.g., the run of Figure 2.13 is a run of the system of Figure 2.1. There, the cut B_0 corresponds to the initial marking of $m_0 = [\mathsf{d}, \mathsf{r}]$ of Figure 2.1, the cut B_1 corresponds to marking $[\mathsf{nb}, \mathsf{sa}]$, and B_2 to marking $[\mathsf{ba}, \mathsf{sa}]$. Event 8 of π represents an occurrence of transition gb consuming a token from place nb and producing token on place ba. So the cuts B_1 and B_2 and event 8 together correspond to the global step $[\mathsf{nb}, \mathsf{sa}] \xrightarrow{\mathsf{gb}} [\mathsf{ba}, \mathsf{sa}]$ of the Petri net system of Figure 2.1.

This correspondence between cuts and global states as well as configurations and steps has already been discussed for distributed runs in general in Section 2.3.4. It also holds for any Petri net system $\Sigma = (N, m_0)$. Let B be a cut of a distributed run π of Σ. Each condition $b \in B$ represents a token on place $\ell(b)$. We can sum up these tokens to the marking m_B of N with $m_B(p) = |\{b \in B \mid \ell(b) = p\}|$ for all $p \in P_N$. This leads to the following correspondence between the distributed and the sequential runs of a Petri net system.

Lemma 2.17 (Distributed runs capture sequential runs, Theorem 3.4.3 in [14]):
Let $\Sigma = (N, m_0)$ be a Petri net system. Then the following properties hold:

1. *The marking m of N is reachable from m_0 iff there exists a distributed run π of Σ that has a cut B s.t. $m_B = m$.*

2. *Let m_1 be a reachable marking of Σ. Σ has a step $m_1 \xrightarrow{t} m_2$ iff there exists a distributed run π of Σ with two cuts B_1 and B_2 s.t. $m_{B_1} = m_1$ and $m_{B_2} = m_2$ and for the corresponding configurations C_{B_1} and C_{B_2} holds $C_{B_2} \setminus C_{B_1} = \{e\}$ with $\ell(e) = t$.* ⋆

From this lemma follows immediately that every distributed run π of a Petri net system Σ represents several sequential runs.

Corollary 2.18: *Let $\Sigma = (N, m_0)$ be a Petri net system. A distributed run π is a run of Σ iff there exists a sequence $B_0, B_1, B_2, \ldots$ of cuts of π s.t. $m_{B_0} = m_0$ and for all $i \geq 0$ holds $m_{B_i} \xrightarrow{t} m_{B_{i+1}}$ is a step of N with $C_{B_{i+1}} \setminus C_{B_i} = \{e_i\}$ and $\ell(e_i) = t$.* ⋆

Corollary 2.19: *Let Σ be a Petri net system. The set of distributed runs of Σ is prefix-closed.* ⋆

2.4.1. A Constructive Definition

Definition 2.16 simply characterizes all distributed runs of a given Petri net system by a few axioms. But we can also *construct* the distributed runs of a Petri net system. We will generalize this approach in the following chapters to construct distributed runs from scenarios.

The constructive approach considers a distributed run to be composed of "atomic" distributed runs which we call *local steps*. A local step consists of exactly one event with its pre- and post-conditions. Figure 2.15 depicts the local steps of our running example π; every local step is a partial run of π. We can think of the local steps as those "puzzle pieces" the run π consists of. The constructive semantics *composes* a run from local steps.

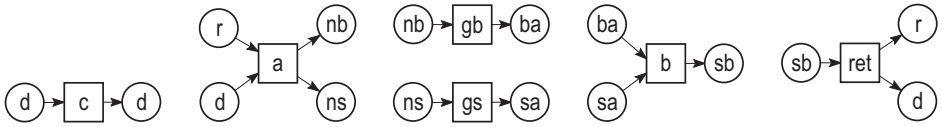


Figure 2.15. The local steps of π from Fig. 2.6.

In a distributed run of a Petri net system a local step describes how an occurrence of a transition consumes and produces tokens. Thereby, the structure and labeling of a local step follows from the *structure* of the Petri net as illustrated in Figure 2.16.

Definition 2.20 (Local step). Let N be a Petri net, let $t \in T_N$. A *local step* of t (or *t-step* for short) is a distributed run A consisting of exactly one *event* e and its pre- and post-conditions s.t.

1. e represents t,
2. ${}^\bullet e$ represents tokens on ${}^\bullet t$, and
3. $e^\bullet$ represents tokens on $t^\bullet$. ⌟

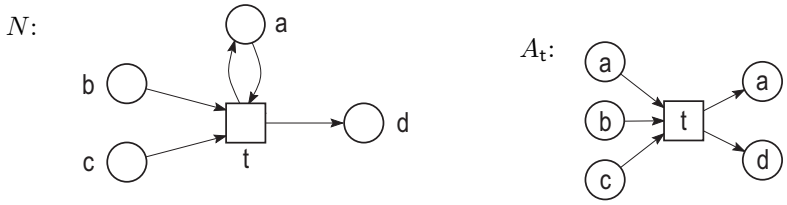
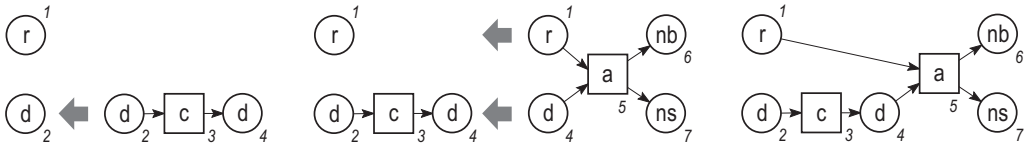


Figure 2.16. A net N and a local step A_t of N that represents a firing of transition t. Place a is pre-place *and* post-place of t; the local step A_t unfolds this cycle in the net structure.

As a convention we refer to the event of a local step A by e_A. *Technically, each transition t yields infinitely many isomorphic t-steps.*

We construct a distributed run of a Petri net system by composing (copies of) local steps in an acyclic manner. Beginning with a set of conditions that represents the initial marking, we append a fresh copy of a local step to a distributed run which results in a longer distributed run. Figure 2.17 depicts how a prefix of the run π of our flood alert example is constructed from the local steps of the Petri net in Figure 2.1.



(a) run π_0 and a local step A_c of transition c

(b) composed run $\pi_0 \rhd A_\mathsf{c}$ and a local step A_a of transition a

(c) composed run $(\pi_0 \rhd A_\mathsf{c}) \rhd A_\mathsf{a}$

Figure 2.17. Composing a distributed run of N in Fig. 2.1 from local steps of transitions of N.

On the formal side, we first adapt the notion of an enabled transition to local steps. We say that a local step A is *enabled* at a distributed run π iff the maximal conditions of π include the preconditions of A, i.e., ${}^\bullet e_A \subseteq \mathit{Cut}(\pi)$. We then may *continue* π with A by composition, i.e., the union $\pi \cup A$. Here, we assume without loss of generality that A and π are disjoint except for ${}^\bullet e_A$, i.e., $X_\pi \cap X_A = {}^\bullet e_A$. This property ensures that we *append* A to π. If A does not satisfy this property, we choose an isomorphic copy of A that does. We write $\pi \rhd A$ for the continuation of π with A.

For instance, the local step A_c of c in Figure 2.17(a) is enabled at the run π_0; appending A_c to π_0 yields the run $\pi_0 \rhd A_\mathsf{c}$ in (b). Now, the local step A_a of Figure 2.17(b) is enabled, appending A_a yields the run $(\pi_0 \rhd A_\mathsf{c}) \rhd A_\mathsf{a}$ in (c).

Definition 2.21 (Construction of distributed runs of a Petri net system). Let $\Sigma = (N, m_0)$ be a Petri net system. The set $R(\Sigma)$ of distributed runs that can be constructed from Σ is inductively defined.

1. A distributed run $\pi_0 = (B_0, \emptyset, \emptyset)$ with $m_{B_0} = m_0$ is the *initial distributed run* of Σ; $\pi_0 \in R(\Sigma)$.
2. Let $\pi \in R(\Sigma)$, let $t \in T_N$ and let A be a t-step. The t-step A is *enabled* at the end of π iff ${}^\bullet e_A \subseteq \max \pi$ and $X_\pi \cap X_A = {}^\bullet e_A$. If A is enabled, then the run $\pi \rhd A := \pi \cup A \in R(\Sigma)$. ⌟

In the induction step, the run π is a prefix of the continued run $\pi \rhd A$; see Definition 2.10. Every run constructed in this way is a distributed run of Σ and every run of Σ can be constructed from local steps.

Lemma 2.22 (Equality of axiomatic and constructive semantics, Theorem 3.5.3 in [14]): *Let Σ be a Petri net system. $R(\Sigma)$ of Def. 2.21 is the set of all distributed runs of Σ characterized by Definition 2.16.* ⋆

From Def. 2.21 and this lemma follows that every sequential run of a Petri net system $\Sigma = (N, m_0)$ induces a corresponding distributed run. Let $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \ldots$ be a sequential run of Σ. Then there exists for each $i > 0$ a t_i-step A_i and a sequence of distributed runs $\pi_0, \pi_1, \pi_2, \ldots$ with:

- π_0 represents m_0 as in Definition 2.21, and
- for each each $i \geq 0$ holds: A_{i+1} is enabled at π_i and $\pi_{i+1} = \pi_i \rhd A_{i+1}$.

Each run $\pi_i, i \geq 0$ is a distributed run of Σ.

2.4.2. Labeled Petri nets and their distributed runs

After introducing Petri nets and their distributed runs, we generalize both notions to *labeled* Petri nets.

Definition 2.23 (Labeled Petri net). Let $\mathcal{L} = \mathcal{L}_B \cup \mathcal{L}_E$ be a set of names $\mathcal{L}_B$ of local states and names $\mathcal{L}_E$ of actions. A *labeled* Petri net over $\mathcal{L}$ is a Petri net $N = (P, T, F, \ell)$ with a labeling $\ell : P \cup T \to \mathcal{L}$ s.t. $\ell(p) \in \mathcal{L}_B$, for all $p \in P$ and $\ell(t) \in \mathcal{L}_E$, for all $t \in T$. ⌟

For instance, every distributed run over $\mathcal{L}$ is a labeled Petri net over $\mathcal{L}$. Like in distributed runs, several nodes of a labeled Petri net can have the same label. A system designer can use the labeling to express, for instance, that two different transitions of the net describe the same action in different contexts, or that the same local state can be entered in different contexts. A labeled net $N = (P, T, F, \ell)$ where ℓ is the identity on $P \cup T$ is *unlabeled*, i.e., an unlabeled net is a "normal" Petri net having the technical overhead of ℓ.

The distributed runs of a labeled Petri net N are straight forward. Instead of labeling each event e with a transition t of N, each event is labeled with the label $\ell_N(t)$ of the respective transition; correspondingly for conditions and places.

Definition 2.24 (Distributed runs of a labeled Petri net). Let $\Sigma = (N, m_0)$ be a Petri net system of a labeled net $N = (P_N, T_N, F_N, \ell_N)$ over $\mathcal{L}$. Let $\Sigma' = (N', m_0), N' = (P_N, T_N, F_N)$ be the underlying unlabeled Petri net system. The *set of distributed runs* of Σ is the set $R(\Sigma) := \{(B, E, F, \ell_N \circ \ell) \mid (B, E, F, \ell) \in R(\Sigma')\}$. ⌟

Labeled nets are strictly more expressive than unlabeled nets; see [37] for a survey. This notion concludes our introduction to modeling the behavior of a distributed system.

2.5. Concluding Remarks

This chapter introduced the behavioral model of *distributed runs* to precisely describe the behavior of a distributed system. We defined distributed runs independently of a specific formal system model. This allows us to consider any set of distributed runs to be the behavior of some system. Our approach is technically based on labeled causal nets from Petri net theory. This formalization

- is more expressive than sequential runs because it distinguishes independence of actions from arbitrary orders of actions,
- is more expressive than labeled partial orders because it includes information about visited states in a run, and
- generalizes the true concurrency semantics of Petri nets to arbitrary systems, both in the axiomatic and in the constructive definition.

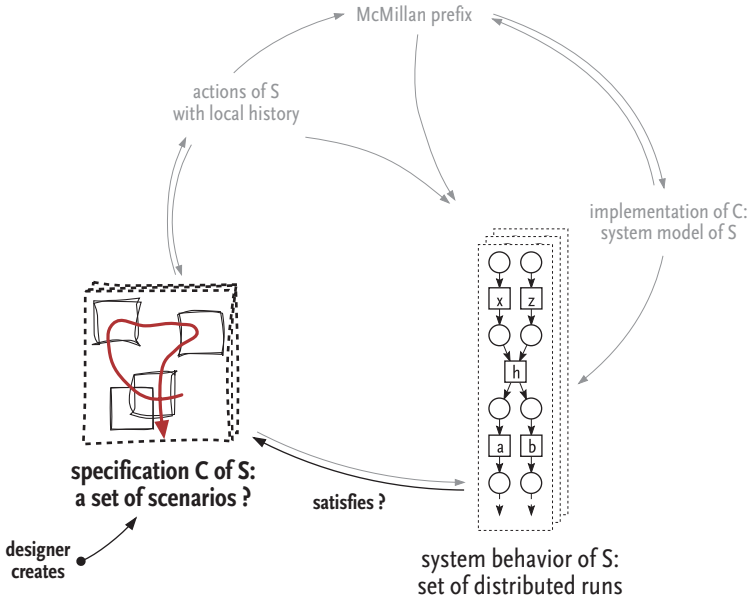
We shall use this general behavioral model of distributed systems in the following chapters to discuss and define a true concurrency semantics of *scenarios*. Many notions introduced in this chapter turn out to be important:

- A *partial distributed run*, i.e., a part of a run, corresponds to the notion of a scenario.
- The idea of a *history* that we observed in the pizza example of Figure 2.10 will help us to relate different scenarios to each other.
- We shall combine the idea of a history and the *constructive semantics* of Petri nets in Chapter 5 where we construct distributed runs from scenarios.
- That a distributed run succinctly represents a set of sequential runs shall help us in Chapter 8 to efficiently synthesize a Petri net from scenarios. The synthesized net will be a *labeled* Petri net.
- A Petri net is a natural model of a distributed system, which consists of several *components*. We consider in Chapter 8 how to synthesize components from scenarios.

Part II.

A Minimal Class of Scenario-Based Specifications

3. Scenarios



This chapter reviews the notion of a scenario and existing scenario-based techniques regarding their expressive power. We then characterize a novel *kernel* of scenario-based specifications that balances the trade-off between expressive power and synthesis capabilities. In this kernel (1) a scenario describes a *partial run* of the system, (2) a scenario distinguishes a *history*, and (3) a system behavior satisfies a scenario if every run that ends with a scenario's history has a continuation with the entire scenario. This kernel has not been identified before. It allows to specify the complete behavior of a distributed system based on a *minimal* set of notions for scenarios *and* yields a more general solution to the synthesis problem, as we show later in this thesis. We formalize the kernel in Chapter 4 when we introduce *oclets*.

3.1. System Model, Specification, and Scenario

The preceding chapter introduced *Petri nets* as a formal technique to model a distributed system S. In this technique, a system designer first creates a Petri net system Σ that describes the local states and local actions of S. Σ itself may be structured to describe the *components* of S. The semantics of Σ define the set of *distributed runs* of Σ. These runs describe the behavior exhibited by S as sketched in Figure 3.1. In other words, Σ is a *system model* of S: it serves as a "blue print" for building S, and it allows a system designer to *formally* reason about the behavior of S.

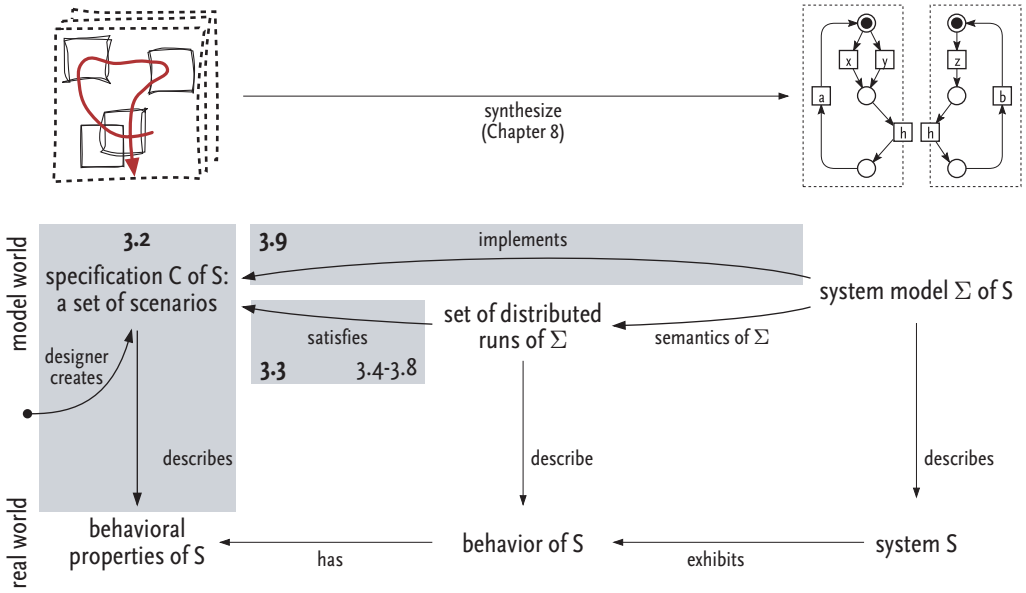


Figure 3.1. The relation between scenarios, system model, and system behavior.

A distributed system S may exhibit complex behavior because of intricate interactions between its components, as we discussed in Chapter 1. To ensure that the system S behaves as intended, the system designer creates a *specification* C which describes behavioral properties that S should exhibit. If the distributed runs of Σ *satisfy* C, then S has these intended properties — under the assumption that Σ and C appropriately describe S and its intended properties, respectively. In this case, we call S (or Σ) an *implementation* of C.

In the scenario-based approach, a specification C is a *set of scenarios*. A scenario describes how system components of S interact in a specific situation, technically expressed as a partial order of actions. Intuitively, the distributed runs of Σ *satisfy* C if each scenario in C *occurs* in these runs.

A system designer usually has to formally prove that Σ implements C. Though, a system designer may pursue a different approach. After describing all intended properties of S in C, the designer uses a *synthesis* technique to automatically

construct a system model Σ that implements C. Synthesis from scenarios is a non-trivial problem as discussed in Chapter 1. We will develop a synthesis technique from scenarios in Chapter 8. But first and foremost, we need to understand more precisely how scenarios specify system behavior.

The problem: how much expressive power is enough?

The scenario-based approach has evolved to an accepted technique for specifying behavior of distributed systems, and several notions of scenarios with different expressive power have been developed [8, 80]. High expressive power is desirable for describing behavioral properties but it renders the synthesis from scenarios infeasible in general [26, 18]. For this reason, existing synthesis algorithms restrict the class of scenario-based specifications [13] or constrain the synthesis result [17, 18].

This thesis aims at balancing the trade-off between expressive power and synthesis capabilities. In other words, we want to identify a *kernel* of scenario-based specifications with limited expressive power so that synthesis becomes feasible. Yet, this kernel shall be sufficiently expressive to specify *all* behavior of any distributed system we want to design. In addition, we are interested in a *flexible* specification technique which does not constrain the system designer when creating the specification. To find such a kernel, we answer the following question.

> *Which notions are necessary for specifying the complete behavior of a distributed system with scenarios?*

We proceed as follows. We review established scenario-based techniques and notions for scenarios wrt. their expressive power and flexibility, and we specifically investigate their underlying assumptions. Section 3.2 recalls established notations for scenarios and Section 3.3 recalls their intuitive semantics and discusses deficits. Based on these insights, we propose a kernel of scenario-based specification that generalizes ideas of *Live Sequence Charts* (LSCs) [25]. This kernel is based on the following three simple ideas: (1) a scenario describes a *partial run* of the system, (2) a scenario distinguishes a *history*, and (3) *a system behavior satisfies a scenario if every run that ends with the scenario's history has a continuation with the entire scenario.*

We develop this kernel systematically from Section 3.4 onwards by reviewing established scenario-based techniques in more detail. We highlight design decisions that have been made in each technique regarding how to specify system behavior with scenarios. We collect and explore alternative design decisions to finally identify a *minimal set of notions* for scenarios which amount to the proposed kernel.

The concluding Section 3.9 summarizes our insights and discusses when a system model implements a scenario-based specification by an example. The kernel of scenario-based specification identified in this chapter will be formalized in Chapter 4 in the model of *oclets*.

3.2. Notations for Scenarios

This and the following section recall syntax and semantics of scenarios at an informal level. The following running example supports our explanations.

We want to specify behavior in an emergency management procedure among a medic and a hospital. This procedure is supported by an Emergency Management System (EMS) which keeps track of pending emergencies. The medic and the hospital interact with the EMS as follows:[1]

1. The medic notifies the EMS that he is available for handling a medical emergency and then checks his equipment. Meanwhile, the EMS sends the medic the location of a pending emergency.
2. Alternatively, if there is no pending emergency, the EMS sends a respective reply to the medic after receiving the medic's message.
3. To handle an emergency, a medic confirms his current task to the EMS, moves to the location of the emergency, stabilizes the patient, and transports the patient to the hospital.
4. When the medic confirms his current task to the EMS, the EMS registers the confirmation, and notifies the hospital about the incoming patient.

Natural language. Some scenario-based techniques like *RATS* [32] already consider each of the above sentences 1.-4. as one scenario given in an informal notation. Typically, a scenario "specifies actions performed by the user and by the system as well as the communication in between them" [32, p.104]. Various techniques propose guidelines and formal grammars to describe natural language scenarios in a more structured way, e.g., [32, 106, 2]. All these techniques guide the system designer in stating explicitly for each scenario which actions occur in which order under which conditions.

Message Sequence Charts. To avoid ambiguities that come with natural language descriptions, most scenario-based techniques use a formal, usually graphical syntax. Figure 3.2 depicts two scenarios in the graphical syntax of *Message Sequence Charts* (MSCs). This syntax has become industry standard for formally denoting scenarios of distributed systems [65]. Various other techniques adapted and extended the MSC syntax, most notably *UML sequence diagrams* [95], *Live Sequence Charts* (LSCs) [25], and several MSC variants [45, 107, 72]. MSC M_1 in Figure 3.2 describes the first scenario of our example and MSC M_2 describes the second one.

An MSC describes a scenario as a partial order of actions of the system components. An arrow describes an asynchronous message exchange; each exchange consists of a *send* action and a *receive* action; the receive action is implicitly denoted by the arrow's head, the corresponding send action by the arrow's foot. A box describes an *internal* non-communicating action. Each action (sending,

[1] This example does not necessarily reflect an actual emergency management process and may be too simplistic to capture actual requirements. We use this example to illustrate concepts of scenarios in a more tangible setting.

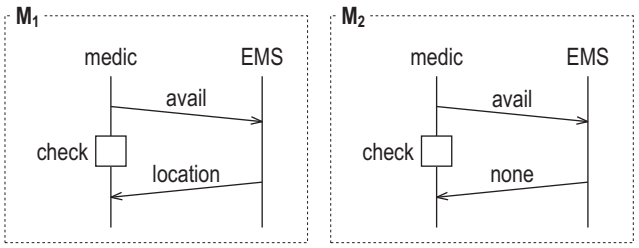


Figure 3.2. Two example scenarios M_1 and M_2 in MSC notation.

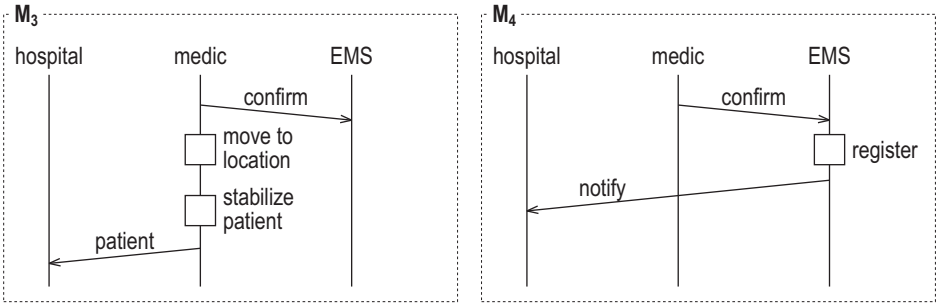


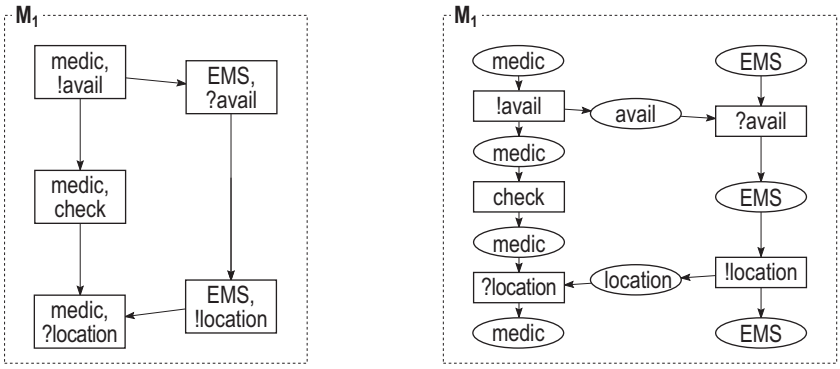
Figure 3.3. Two example scenarios M_3 and M_4 in MSC notation.

receiving, or internal) is assigned to a component[2]. The actions of a component are ordered along its vertical axis from top to bottom; the actions of the entire scenario are partially ordered.

In our example, scenario M_1 formally describes the following behavior: the medic first asynchronously sends an avail message to the EMS (that he is available for handling a medical emergency case) and then performs an internal check (of his equipment). After receiving the avail message, the EMS sends a location message (with the location of an emergency) to the medic. Figure 3.4(a), explicitly shows the partial order of actions of M_1 [58]. Action check of the medic is concurrent to the actions of the EMS. M_2 formally describes the second scenario from the beginning of this section. It differs from M_1 by letting the EMS send a none message instead of a location message. The MSCs M_3 and M_4 of Figure 3.3 complete our example: they formally describe the third and fourth natural language scenario from the beginning of this section.

Scenarios M_1 and M_2 as well as M_3 and M_4 partially *overlap*. This kind of overlap can also be observed in the textual description of our example, though it is possibly harder to spot. The scenario-based approach does not constrain how different scenarios syntactically relate to each other — except by fixing a set of actions that occur in the scenarios. That a system designer freely chooses size and shape of *each* scenario is inherent to the approach and its acceptance. Each

[2] A component occurring in an MSC is called 'instance' (of that component) in the MSC standard [65].



(a) scenario M_1 as a labeled partial order (b) scenario M_1 as a finite distributed run

Figure 3.4. Other notations for scenario M_1.

scenario usually describes some "self-contained story" of the system leading from an initial "triggering" situation to some "goal" situation involving all components that are relevant in this story; see [25] or [55]. This self-contained understanding makes each scenario a *local statement* that a system designer can understand, interpret and discuss with others in isolation. For this reason, particularly early stages of system design benefit from scenarios [8].

Other formal notions. There are other formal notations for scenarios that are closely related to MSCs:

- The MSC notation defines a special class of *labeled partial orders* of actions as depicted in Figure 3.4(a). Other approaches consider any labeled partial order of actions as a scenario [12].
- An MSC can be translated to a *Petri net*, i.e., a distributed run as defined in Chapter 2 [69]. Figure 3.4.(b) depicts scenario M_1 in this notation. But Petri nets in general allow for a more flexible notation of scenarios beyond the constraints of MSCs; this topic is discussed in [8].
- Like Petri nets, also other state-based techniques have been extended to note down scenarios, for instance using *statecharts* [46] or *UML activity diagrams* [8, 95].

In all these approaches, a scenario describes a finite course of actions involving several components of a distributed system. Other approaches use scenarios to describe the interaction behavior of a single component with its environment, e.g., *Chisel diagrams* [3] and *scenarios trees* [64]. Further notations such as *UML use case diagrams* [95] and *use case maps* [22] complement these detailed action-based descriptions of behavior: they provide a high-level view on all system components but abstract from specific orderings of actions.

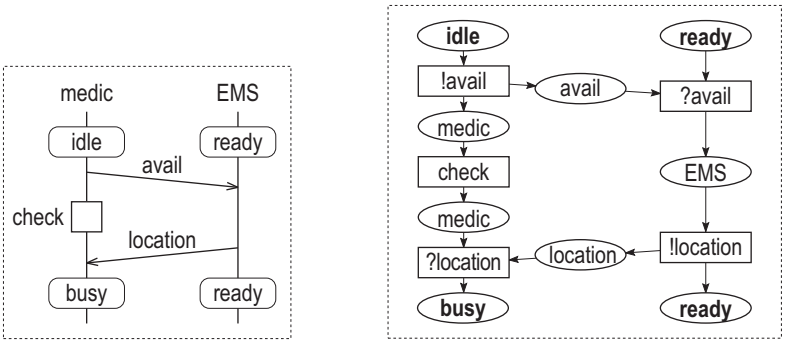


Figure 3.5. A scenario with conditions in MSC notation and a possible representation as a distributed run.

Specifying state information. In some cases, a system designer not only wants to specify which actions occur in a specific order, but also wants to express that the system reaches a specific situation. MSCs provide *setting conditions* to describe that some or all participants of a scenario are in a specific state: "Setting conditions define the actual system state of the [components] that share the condition." [65, p.43]. An MSC setting condition A on component C denotes that C visits the local state A. Thus, an MSC setting condition corresponds to a condition of distributed runs as introduced in Section 2.3.2.

For example, the scenario in Figure 3.5 distinguishes different local states. A condition in the MSC representation is drawn as a rounded rectangle. A possible translation to distributed runs is shown on the right. This scenario describes that whenever the medic is idle and the EMS is ready, the given course of actions may occur, and after that, the medic is in a different state busy while the EMS is ready again. Compared to M_1 in Figure 3.2, the scenario in Figure 3.5 states more precisely how the system shall evolve.

Conclusion: available options and decision. MSC-based notations and closely related techniques have become the most accepted approach to note down scenarios of distributed systems; see [8] for an in-depth evaluation. These techniques address the central problem of describing how several components interact with each other by telling "self-contained stories" about all components that are involved in this story.

In the remainder of this chapter, we use the MSC notation to discuss the *semantics* of scenarios; we eventually use a different notation like partial orders of actions or distributed runs to study a more subtle aspect. We shall see from the following discussion of the semantics of scenarios that — on a technical level — distributed runs as introduced in Chapter 2 are the most precise notation for scenarios. For this reason we use distributed runs from Chapter 4 onwards to note down scenarios.

3.3. Intuitive Semantics of Scenarios

Having introduced syntax of scenarios we now turn to their semantics. The usual and most widely accepted semantics of a scenario is the following: *a scenario describes a possible course of actions that the system shall exhibit* [110, p.384]. This section explains this intuitive semantics of scenarios in more detail. We then recall in which respect this intuitive semantics is insufficient for specifying system behavior. This insufficiency raises a few canonical questions about how scenarios relate to system behavior and to each other. We finally propose a novel semantics of scenarios that answers these questions with a minimal set of notions.

3.3.1. Semantics of a single scenario

The semantics of a scenario is usually defined in terms of a set of runs that *satisfies* the scenario. Intuitively, a set of runs *satisfies* a scenario if the scenario *occurs* in one or more of these runs. This is the case when the scenario's actions occur in a run in the same order as in the scenario. Figure 3.6 depicts an example in the notation of distributes runs: scenario M_1 (Fig. 3.4(b)) occurs in the distributed run in Figure 3.6. The run can have additional actions before, concurrent to, or after the occurrence of the scenario, but there may be no actions in between two subsequent actions of the scenario, e.g., [87, 61, 69]. Section 3.8 discusses more general notions of when "a scenario occurs in a run".

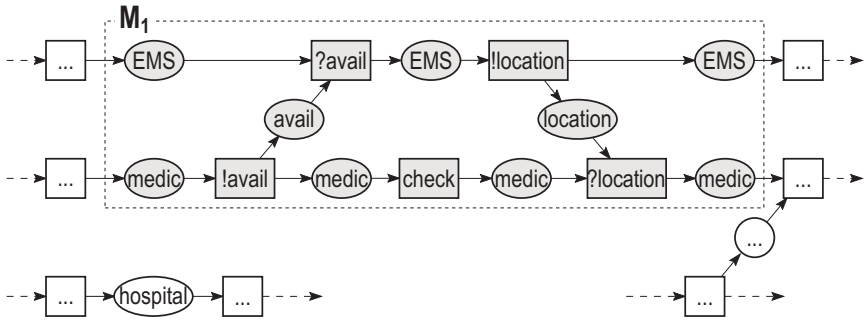
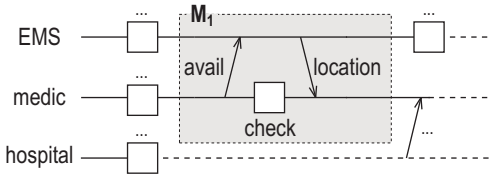
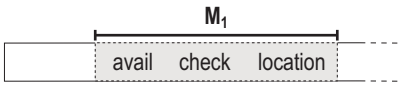


Figure 3.6. Scenario M_1 occurs in the run, the events and conditions of M_1 are highlighted.

The Petri net notation of distributed runs requires moderate graphical overhead for representing and discussing the semantics of scenarios. The syntax of MSC represents more succinctly the same relation between runs and scenarios as shown Figure 3.7(a). Because MSCs canonically translate to the formal model of distributed runs [69], we use the MSC syntax also for concisely noting down distributed runs in this chapter. Eventually, we use the even more abstract notation of Figure 3.7(b) to illustrate an argument.



(a) occurrence of M_1 in MSC notation



(b) abstract notation

Figure 3.7. An occurrence of M_1 of Fig. 3.2 in more succinct notations.

3.3.2. Semantics of a specification

The semantics of a scenario illustrated in Figure 3.6 and 3.7 is the generally agreed on meaning of a single scenario in all techniques discussed in this chapter — though technical details vary. In contrast, the interpretation of a specification which is a set of scenarios is not immediately clear. Regarding the interpretation of an MSC specification, Abdallah and Leue ask in [11, p.98]: "Does it describe *all* behaviors of a system or does it describe a set of *sample* behaviors of a system?" Damm and Harel answer this question in [25] with "it depends." They found that a system designer typically begins specifying a system by sample behavior, which they call the *existential view* of scenarios. As the specification matures in the design process the system designer can state more precisely which behavior the system shall only exhibit; all other behavior is excluded. This view is called the *universal view*. The view on the specification changes from existential to universal during system design.

We consider both existential *and* universal view in this thesis. This chapter primarily discusses the semantics of specifications in the existential view. Firstly, because this view is conceptually simpler; a set R of runs *satisfies* a specification C iff R satisfies each scenario of C, i.e., occurs in a run of R. Secondly, the existential view highlights the idea that a scenario describes a course of actions which a system designer expects to see in the system — regardless of what other behavior may occur. Finally, we shall see in Chapter 5 that the universal view can be obtained by restricting the existential view in a natural way.

Design decision: consider existential and universal view

In the *existential view*, for instance, the set $\{\pi_1, \pi_2\}$ of runs in Figure 3.8 satisfies the specification $\{\mathsf{M}_1, \ldots, \mathsf{M}_4\}$ from Figures 3.2 and 3.3. The first run π_1 describes a typical complete handling of one emergency. It contains occurrences of actions

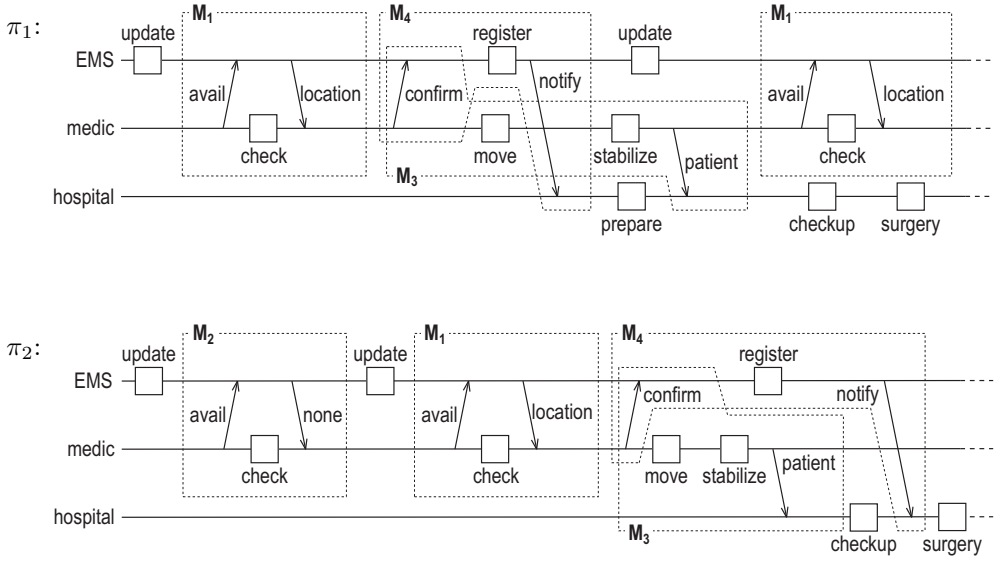


Figure 3.8. The set $\{\pi_1, \pi_2\}$ satisfies the specification $\{\mathsf{M}_1, \ldots, \mathsf{M}_4\}$.

that are not mentioned in the scenarios. Also, scenario M_2 is missing in π_1; it occurs in π_2 where the EMS learns about an emergency only later in the process. Each scenario occurs at least once in these runs, some like M_1 even occur several times in a single run. Some scenarios occur overlappingly like M_3 and M_4 on the message confirm.

Whether $\{\pi_1, \pi_2\}$ satisfies $\{\mathsf{M}_1, \ldots, \mathsf{M}_4\}$ in the *universal view* cannot be answered with the intuitive semantics of scenarios. One could argue, for instance, that action update at the EMS is not mentioned in any of M_1-M_4. In other words, the runs $\{\pi_1, \pi_2\}$ do not only express behavior described by M_1-M_4. Though, there are contrasting views [25]. Answering this question precisely calls for a number of *design decisions* on the semantics of scenarios.

3.3.3. Open questions and open design decisions

According to the intuitive semantics of scenarios, a set of runs satisfies a scenario-based specification if each scenario occurs somewhere in a run. It is well-known that this semantics is not expressive enough for specifying interesting system behavior [11, 25]. For instance, our pragmatic understanding of the specification $\{\mathsf{M}_1, \ldots, \mathsf{M}_4\}$ in our running example raises two questions about the semantics of scenarios which are not answered by their intuitive semantics.

Ordering scenarios. According to the pragmatics, scenarios M_1 and M_2 denote possible *alternatives* whereas scenario M_3 should occur *after* M_1 but *not* after M_2. This kind of *ordering information* is not expressed in M_1-M_3. So we have to ask: *when* does an occurrence of a scenario make sense?

Completeness. Assume we know when an occurrence of a scenario makes sense. Does this mean that the complete scenario has to occur if and whenever it makes sense to occur?

All scenario-based techniques answer these two questions by *providing at least on additional notion that increases the expressive power of scenarios.* In the light of this observation we ask a more general question: which notions are necessary for specifying the complete behavior of a distributed system with scenarios?

Necessary notions for scenarios

In the remainder of this chapter, we will systematically develop an answer to this question. Our answer then leads to the following *novel* semantics of scenarios.

> (1) Like in LSCs [25], a scenario distinguishes a finite prefix as a history. (2) A set of distributed runs satisfies a scenario if whenever a run ends with the scenario's history, then there exists a run that continues with the entire scenario. (3) A set of distributed runs satisfies a specification if it satisfies each scenario.

Figure 3.9(a) shows a variant of the scenarios M_1 and M_2 of our example where the message avail is distinguished as a history. Figure 3.9(b) depicts an *execution tree*, which represents two distributed runs π_1 and π_2. These runs satisfy scenario M_1' but not M_2'. The set $\{\pi_1, \pi_2, \pi_3, \pi_4, \ldots\}$ of runs indicated in (c) satisfies both M_1' and M_2'. In other words, $\{\pi_1, \pi_2, \pi_3, \pi_4, \ldots\}$ satisfies the specification $\{\mathsf{M}_1', \mathsf{M}_2'\}$.

Reviewing design decisions of scenario-based techniques

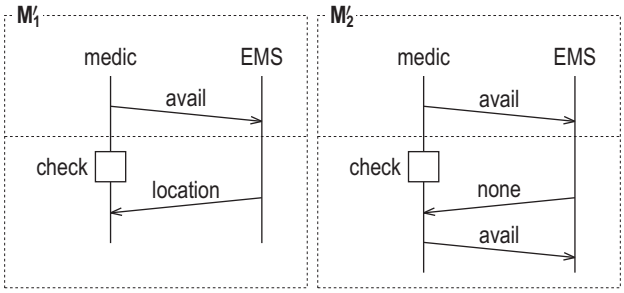
We now sketch the road map that leads to this semantics of scenarios. The already convinced reader may continue in Section 3.9. Our aim is to identify a kernel of scenario-based specifications that, on one hand, is sufficiently expressive to describe all behaviors of a distributed system, and on the other hand uses only those notions that are necessary for this expressive power. We reach our aim by systematically reviewing established scenario-based techniques. We specifically collect and explore *design decisions* that affect how a scenario relates to system behavior and to other scenarios. We proceed as follows.

First, we discuss *underlying assumptions* of the intuitive semantics of scenarios in Section 3.4. To adapt these assumptions to more expressive semantics of scenarios, we then investigate the following questions.

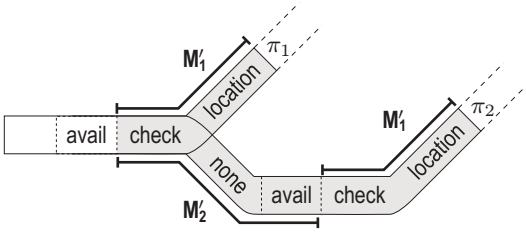
Alternative scenarios. Are two scenarios possible *alternatives* or does each scenario describe behavior that the system *must follow*?

Overlapping scenarios. Are two scenarios allowed to *overlap* or do they have to occur disjointly?

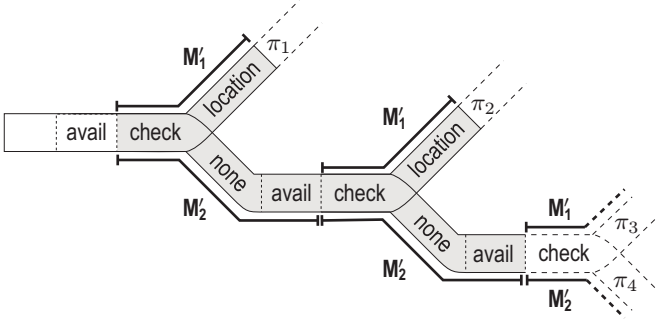
Answer to these questions are strongly related to the questions how to express ordering of scenarios (discussed in Section 3.5) and how to specify all behavior of a distributed system with scenarios (discussed in Section 3.6). We then turn our attention to the following two subtle aspects of scenarios.



(a) two scenarios with a history



(b) the runs $\{\pi_1, \pi_2\}$ satisfy M'_1 but not M'_2



(c) the runs $\{\pi_1, \pi_2, \pi_3, \pi_4, \ldots\}$ satisfy the specification $\{\mathsf{M}'_1, \mathsf{M}'_2\}$

Figure 3.9. A novel semantics of scenarios.

Initial behavior. How does a scenario-based specification describe *non-empty* behavior? How does it describe an *initial state*? Especially the synthesis problem is sensitive to this question which we discuss in Section 3.7.

Occurrences of a scenario. When does a scenario *occur* in a run? The current discussion presumes a specific, though natural, notion of when a scenario occurs in a run. Section 3.8 presents other notions.

For each of these questions we identify several answers in the respective sections. We discuss the impact of each answer on the semantics of scenarios by the help of examples. In the light of all available options we then make some design decisions which leads to the novel semantics of scenarios that we just presented. Section 3.9 summarizes our semantics. In Chapter 4, we formalize our design decisions in the model of *oclets*.

3.4. Underlying Assumptions of the Intuitive Semantics

In the following we highlight the formal concepts and underlying assumptions of the intuitive semantics of scenarios presented in Section 3.3.

A scenario is an existential statement. In its simplest interpretation, a set of (distributed) runs *satisfies* a scenario only if *there exists* a run where the scenario occurs. This is the case when the scenario's actions occur also in the run in the described order. Hence, a system *implements* a scenario-based specification if its set of distributed runs satisfies each scenario. An implementation may have runs that are not described by any of the scenarios at all, that is, the system behavior does not have to be "covered" by the scenarios. This interpretation corresponds to the existential view on scenarios [25].

A scenario is a partial statement. The example runs π_1 and π_2 already highlight an important aspect of scenarios: a scenario is inherently a *partial statement* on behavior, see [31, p.1113] and [8, p.63]. It does not describe a complete run from the start to its end but only a partial run as defined in Section 2.3.3. A run may contain actions that precede the scenario's actions as well as actions that follow the scenario's actions. A scenario occurs in a run if the run *somewhere* has the same order of actions as the scenario.

Two scenarios may be alternatives. Let us consider our example scenarios M_1 and M_2 of Figure 3.2 in more detail. Both scenarios start the same way, but differ in their outcome: in case of scenario M_1, the EMS knows about a medical emergency and sends its location to the medic; in case of scenario M_2, the EMS sends that there is no emergency. Pragmatically, either M_1 or M_2 should occur in a concrete run but not both together. Here, two different scenarios describe *possible alternative* courses of actions in the system. M_1 and M_2 do occur as alternatives in the example runs π_1 and π_2. Both runs start the same way until message avail was received and diverge afterwards.

Two scenarios may overlap. Scenarios M_3 and M_4 of our example illustrate another important aspect of how scenarios relate to each other. In the runs π_1 and π_2 in Figure 3.8, both scenarios occur *overlappingly* on the message confirm in the *same* run. M_3 and M_4 differ from M_1 and M_2 in the sense that M_3 and M_4 *do not* diverge from the perspective of the involved components. For the EMS, scenario M_3 does not describe how to continue *after* message confirm occurred, so the EMS may follow scenario M_4 and send notify to the hospital. Correspondingly, the medic may follow M_3 and eventually send the patient to the hospital — in the same run. The hospital indeed has to *choose*. Though only about the order of receiving sent messages. Once the hospital receives one message (e.g., notify) it chooses to follow the respective scenario (e.g., M_4). After completing its share of M_4 the hospital follows M_3 by receiving message patient.

Design decisions. This example illustrates an intriguing aspect of the semantics of scenarios: whether two scenarios occur as alternatives or overlappingly depends on the *inner structure* of the scenarios; that is, whether the two scenarios describe behavior that may occur together or not. Altogether, the following two interrelated design decisions arise.

- May two scenarios describe *alternative* behavior or do they always describe behavior of the same run? We encode this design decision in the following as [2Alt = yes/no].
- May two scenarios describe *overlapping* behavior or do they always occur disjointly? [2Overlap = yes/no].

The intuitive semantics of scenarios allows for answering both questions with yes for the following reason. An occurrence of a scenario in a run does not exclude an occurrence of another alternative scenario in the system. Two alternative scenarios simply occur in different alternative runs. The underlying assumption that supports this interpretation of alternative scenarios is that each scenario describes a "self-contained story" of the system. Two stories that cannot occur at the same time occur alternatively, and each story remains "self-contained." Technically, each scenario remains a *local* statement that is interpreted without referring to other scenarios. At the same time, two stories such as M_3 and M_4, which do not contradict about their courses of actions, may occur overlappingly. Again, each scenario remains a *local* statement that is interpreted without referring to other scenarios.

Design decision: Two scenarios may be alternatives [2Alt = yes], two scenarios may overlap [2Overlap = yes].

There are other interpretations of scenarios in this respect. Particularly *universal scenarios* in LSCs allow to note down scenarios that assume [2Alt = no, 2Overlap = yes]. With two scenarios of this kind, a system designer describes different aspects of the *same* run by two different scenarios [25]. The opposite choice [2Alt = yes, 2Overlap = no] treats any two scenarios as alternatives.

In this thesis we favor the assumption [2Alt = yes, 2Overlap = yes] of the intuitive semantics of scenarios. By this assumption, we can formally describe a scenario as a *partial distributed run* as stated in Definition 2.15. In the intuitive semantics, a scenario describes a partial run which the designer expects in the system: a system satisfies a scenario only if the scenario occurs as a partial run of some system run. The following two sections consider how to express ordering of scenarios and how to specify *all* runs of a system.

3.5. Ordering Scenarios

The pragmatics of the running example in this chapter suggests to express ordering information between different scenarios of a specification. For example, scenario M_3 occurs *after* scenario M_1 but *not after* scenario M_2. This section reviews how existing approaches express ordering of scenarios.

3.5.1. Existing approach to express ordering

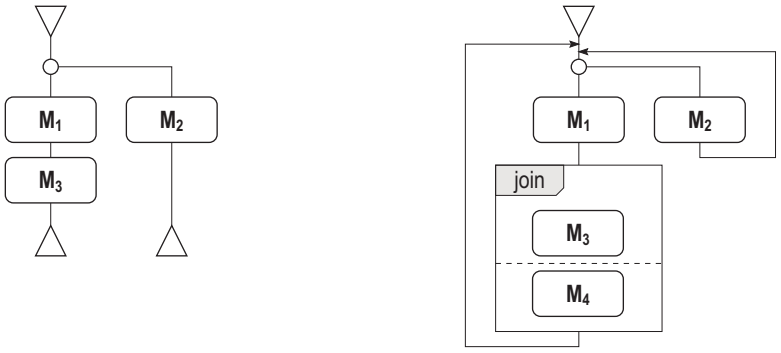
Complete runs

The most basic techniques assume that each scenario of a specification describes *a complete run* from the system's initial state to some final state [12, 5]. Accordingly, two different scenarios describe alternative runs. Although this approach is technically simple it is also quite restricted. A specification consists of finitely many finite scenarios (if it is meant to be noted down by a system designer). Thus, only a very limited class of systems may be specified in this approach. For this reason, two other approaches have been developed which we review in the following.

Expressions over scenarios

Most scenario-based specification techniques introduce *composition operators* to put scenarios in a specific order. Typically, these operators are *sequential-* and *parallel composition*, *alternative choice*, and *finite iteration*. A specification then is no longer a set of scenarios, but an expression over scenario. For example, in the approach of Desel et al. [13], the expression $(\mathsf{M}_1;\mathsf{M}_3)+\mathsf{M}_2$ composes M_1 sequentially with M_3 in alternative to M_2. In MSCs, the composition of scenarios is expressed by *Hierarchical MSCs* or *High-Level MSCs* (HMSCs) which are accompanied by a graphical notation [65]. Figure 3.10(a) depicts the HMSC that expresses $(\mathsf{M}_1;\mathsf{M}_3)+\mathsf{M}_2$. Both specifications reflect our pragmatic understanding of M_1-M_3 of Figures 3.2 and 3.3.

An HMSC is a graph. Its nodes are either MSCs or operators like choice. An edge from one node to another denotes sequential composition. A path through an HMSC denotes a valid composition of all MSCs that lie on this path. Thus, an HMSC H (or an expression over scenarios) describes a set of larger scenarios. An implementation of H exhibits all scenarios described by H.



(a) M_1 followed by M_3 is alternative to M_2

(b) iterate: M_1 followed by M_3 and M_4 is alternative to M_2

Figure 3.10. Specification of the medical emergency example in HMSCs.

Specifying the runs π_1 and π_2 in Figure 3.8 with HMSCs is more involved: scenario M_1 occurs several times, and scenarios M_3 *and* M_4 *overlap* in both runs. An overlapping composition of scenarios requires a non-standard operator like join introduced in [72]. A cycle in an HMSC expresses iterations over (compositions of) scenarios. For example, the HMSC in Figure 3.10(b) specifies two cycles: (1) M_1 followed by $\mathsf{join}(M_3, M_4)$, and (2) M_2. The entire HMSC specifies that these two cycles are iterated infinitely often in an arbitrary order. The specified behavior includes, among others, the runs π_1 and π_2 (assuming that all actions like update which do not occur in M_1-M_4 are "invisible"); [67, 72] provide details.

Advantages and Disadvantages. These two examples already illustrate the advantages and disadvantages of the compositional approach. On one hand, an HMSC provides a global overview of how the different scenarios relate to each other. On the other hand, the exact semantics of a single scenario follows only from this global view on the entire specification. When the scenario occurs and how often it does can only be understood from the complete expression or the complete graph. Thus, a specification is no longer a collection of "self-contained stories" about the system.

The sequential nature of the HMSC graph suggests a global coordination of scenarios that does not exist in distributed systems. In this respect, Genest et al. state: "HMSCs have several drawbacks, such as the difficulty to express concurrency between two independent threads, due to the sequential control of the graph. The result is that many systems are hard to model using HMSCs" [45, p.196].

The sequential nature of HMSCs also interferes with moments of choice. The HMSC in Figure 3.10(a) suggests that the choice between M_1 or M_2 is made before either scenario begins. But according to M_1 and M_2 in Figure 3.2 this choice has to be made only after message avail was received. To overcome this deficit, MSCs introduce several dedicated operators for precisely specifying the moment

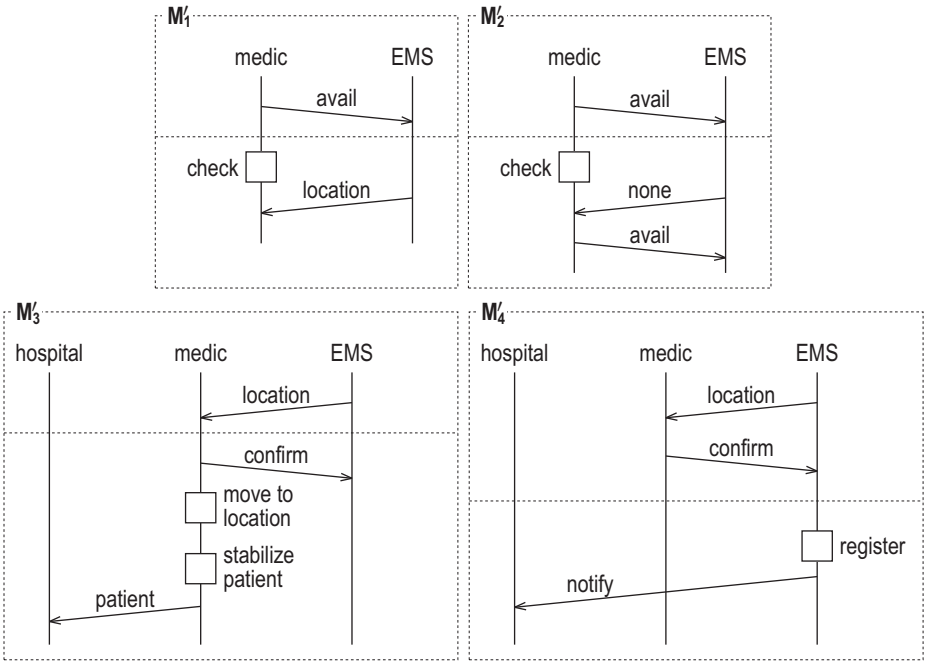


Figure 3.11. Specification of the medical emergency example by scenarios with history.

of choice [87]. Uchitel et al. observe: "scenario composition can lead to a large number of very short scenarios that must be composed in complex ways to describe the system's overall behavior" [113, p.99].

Such complex specifications are syntactically hard to maintain: including a new scenario always requires to extend the graph (or expression) in a syntactically correct way. Depending on the kind of change, this may entail complex restructuring of the entire specification [41]. Ren et al. [105] provide an illustrative example.

The history of a scenario

Several other techniques like Live Sequence Charts (LSCs) [25] follow a different approach to express when a scenario may occur: a prefix of a scenario is distinguished as its *history*. Intuitively, if a system exhibits behavior that is described in the scenario's history, then the system has reached a situation where an occurrence of the entire scenario makes sense.

In our medical emergency example, scenario M_3 of Figure 3.3 only makes sense when the medic knows about the location of the emergency. By receiving a location message from the EMS the medic reaches a situation where this prerequisite of M_3 is satisfied. Figure 3.11 depicts among others the scenario M_3 with a corresponding history as scenario M'_3.

The notion of a scenario's history has been proposed first in LSCs where the history is called "pre-chart" [25]; Triggered MSCs [107] and Template MSCs [45] embrace this concept as well. Other techniques use related concepts [31, 32].

The consequences of a history of a scenario are the following. A scenario with a history remains a "self-contained story" about the system; it can be understood in isolation. Genest et al. [45] relate scenarios with a history to *assume-guarantee techniques*. The history denotes an assumption, the remainder of the scenario is guaranteed to occur whenever the assumption holds.

Further, a specification is still a set of scenarios. Thus, changing a specification remains simple: modify a single scenario, or add or remove a scenario. This technique simplifies the creation of a specification compared to HMSCs: "a system description is most easily obtained combining [scenarios] for collections of directly interacting processes, and superimposing assume-guarantee patterns that further constrain interactions between individual scenarios" [45, p.196].

Conclusion and design decision: a scenario has a history

To summarize, existing scenario-based techniques propose three notions for expressing the ordering of scenarios: (1) each scenario scenario describes a complete run of the system, (2) global composition operators express ordering of scenarios, or (3) a local history describes when the scenario make sense. We encode this design decision by [Ordering = complete/composition/history].

While complete scenarios are limited in their expressive power, global composition over scenarios no longer follows the idea that a specification is a collection of self-contained stories about the system. In comparison, the notion of a history still fosters the metaphor of a story as in the intuitive semantics of scenarios. As a consequence, a system designer may focus on describing the system's individual stories ("*What* happens in the system?") rather than how the stories relate to each other ("*How* does it happen in the system?"). For these reasons, we adopt scenarios with a history for specifying system behavior.

Design decision: a scenario has a history [Ordering = history]

3.5.2. Semantics of scenarios with history

Next, we discuss the semantics of a scenario with a history. Intuitively, the "local" scenarios M'_1-M'_4 in Figure 3.11 specify the same behavior as the "global" HMSC in Figure 3.10(b). Specifically scenarios M'_1 and M'_2 should occur as *alternatives* if we recall our earlier design decision [2Alt = yes] in Section 3.4. Surprisingly, this is not the case in existing techniques because these generally assume [2Alt = no] as we explain in the following.

Existing semantics

LSCs [25], Template MSCs [45], and Triggered MSCs [107] employ the notion of a scenario's history and propose an "if-then" interpretation of scenarios: if the

scenario's history occurs in a run, then the run continues with the entire scenario. This implication distinguishes a scenario's history from the rest of the scenario which we call its *contribution*.

More precisely, the semantics of a scenario $S = (hist_S, con_S)$ in these techniques is the following. We consider a run ρ containing an occurrence of a S. This occurrence of S divides ρ into a prefix π of ρ ending with the history of S and a suffix that begins with the contribution of S as shown in Figure 3.12. We therefore say that π *ends with* S's history and that ρ *continues* π with S's contribution.

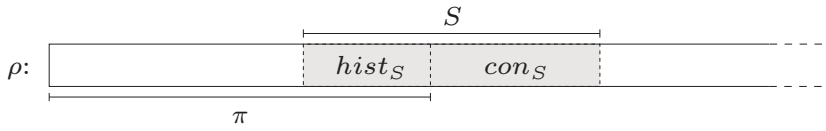


Figure 3.12. The run π ends with the history of scenario S, the run ρ continues π with the contribution of S.

In existing techniques, a set of (sequential or distributed) runs R satisfies the scenario S iff the following property holds:

> for each prefix π of a run ρ in R holds, if π ends with the history of S, then the run ρ continues π with the contribution of S. (3.1)

The concrete formal definitions for "ends with" and "continues with" vary, but this is the conceptual kernel of the semantics [25, 45, 107].

The following simple example shows that this interpretation does not allow to specify *alternative* scenarios. The scenarios M_1' and M_2' in Figure 3.11 share the same history but contribute different courses of actions. In LSCs, Triggered MSCs, and Template MSCs, M_1' and M_2' contradict each other. In each of these techniques holds: if avail occurs in a run, then the entire scenario M_1' also occurs in this run. For the same reason, M_2' has to occur after avail. However, in a single run message avail is either followed by location or by none, but not by both. This contradiction between pragmatics and formal semantics arises because the semantics (3.1) assumes [2Alt = no] in contrast to the intuitive semantics of scenarios in Section 3.4.

To specify alternative behavior, existing techniques with history like LSCs or Triggered MSCs introduce an *explicit choice operator*. The choice operator corresponds to the composition operators in HMSCs. For example, the LSC in Figure 3.13 expresses that location, and none followed by avail occur in alternative runs after avail and check. This solution introduces an additional notation and an additional semantic concept to scenarios with history. One scenario no longer describes one local story, but several stories that branch and merge again. If there are many alternative scenarios with different points of decision, such scenarios can become quite complex and less flexible to handle. To avoid unnecessary complexity and preserve flexibility, we propose a novel interpretation of scenarios with a history by assuming [2Alt = yes] like for the intuitive semantics of scenarios.

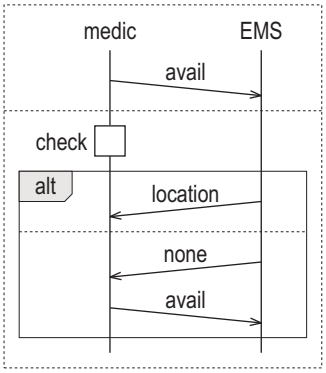
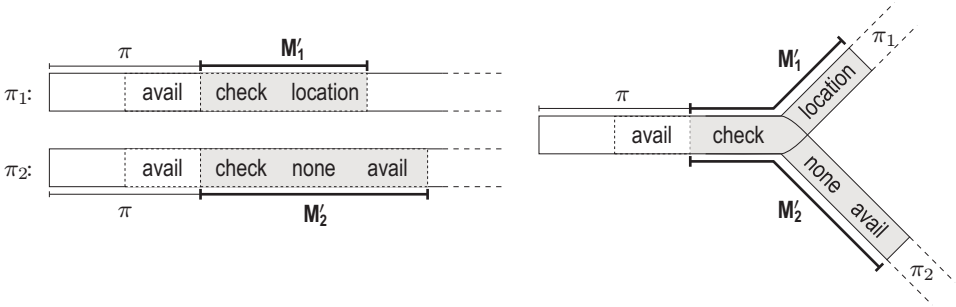


Figure 3.13. A Live Sequence Chart; the 'alt' operator expresses a choice between the upper and the lower sub-scenario.



(a) Representation as runs. (b) Representation as execution tree.

Figure 3.14. The set $\{\pi_1, \pi_2\}$ satisfies the scenarios M'_1 and M'_2 of Fig. 3.11.

A different interpretation of scenarios with history

By the pragmatics of our example, M'_1 and M'_2 should occur in *alternative* runs that branch after the history occurred. Figure 3.14 depicts this behavior as an *execution tree* that represents the two runs π_1 and π_2. In this tree, the runs branch after the occurrence of check. Run π_1 continues along scenario M'_1. Correspondingly, the alternative run π_2 continues along M'_2. In this execution tree, *both* scenarios occur after their history (message avail) occurred which satisfies M'_1 and M'_2. This way, two scenarios with the same history occur in alternative continuations of the same distributed run. The LSC in Figure 3.13 is satisfied by this execution tree.

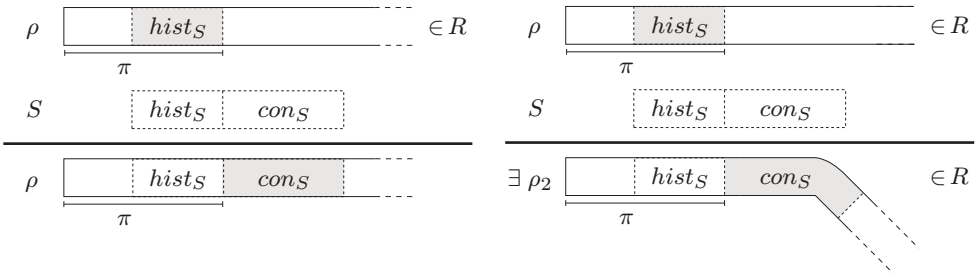
The singleton set $\{\pi_1\}$ should not satisfy M'_2 because there is no run that continues with M'_2 after its history occurred. In contrast, any set of runs where message avail never occurs may satisfy M'_1 and M'_2: because their history never occurs, there is no situation where either scenario is expected.

To capture this intuitive understanding of alternative scenarios with a history, we propose the following interpretation. We say that a set R of (distributed) runs is *closed under continuations* with a scenario S iff

> for each prefix π of a run ρ in R, if π ends with the history of S, then *there exists* a run ρ_2 in R that continues π with the contribution of S. (3.2)

For example, the tree in Figure 3.14(b) is closed under continuations with M'_1 and M'_2.

Definition (3.1) differs from *closed under continuations* (3.2) by the existential quantifier. Thus, (3.2) is a simple branching time property that is evaluated on *sets* of runs (or execution trees) whereas (3.1) is a linear time property that is evaluated on single runs. Figure 3.15 compares these two definitions graphically.



(a) classical semantics (3.1) of a scenario S (b) closed under continuations (3.2) with S

Figure 3.15. Comparison of the semantics (3.1) and (3.2) of a scenario S.

The difference between *closed under continuations* (3.2) and the semantics of Template MSC [45] is actually more subtle. A Template MSC is in principle a logical implication of the form $S = (hist \Rightarrow (con_1 \vee \ldots \vee con_n))$, i.e., different scenarios with the same history are connected via logical disjunction. However, Template MSCs evaluate scenario S only over single runs as in Figure 3.15 on the left. Thus, whenever the history *hist* occurred, it is sufficient when one of the contributions $con_i, i = 1, \ldots, n$ occurs in the run. There is no guarantee that, say, con_2 ever occurs as long as con_1 does. In terms of our example $\{\mathsf{M}'_1, \mathsf{M}'_2\}$ from Figure 3.11, the single run $\{\pi_1\}$ from Figure 3.14(a) satisfies $\{\mathsf{M}'_1, \mathsf{M}'_2\}$ according to Template MSCs. The proposed notion of *closed under continuations* (3.2) demands that both M'_1 and M'_2 occur as alternatives, i.e., only the tree of Figure 3.14(b) satisfies $\{\mathsf{M}'_1, \mathsf{M}'_2\}$.

Design decision: two scenarios may be alternatives

Closed under continuations (3.2) is consistent with the intuitive semantics of scenarios. Two scenarios with the same history may describe possible alternatives. Further, two scenarios may overlap as discussed in Section 3.4. Thus, *closed under continuations* generalizes the intuitive semantics of scenarios as it makes the same

underlying assumptions [2Alt = yes, 2Overlap = yes]. Consequently, a system designer can understand each scenario in isolation without relating it to other scenarios. A scenario with a history is an *expected partial run with a condition*. This leads to the following canonical semantics of scenarios:

> a set R of distributed runs *satisfies* a scenario S iff R is closed under continuations with S; R satisfies a scenario-based specification iff R satisfies each scenario of the specification. (3.3)

In this semantics the scenarios M_1'-M_4' of Figure 3.11 specify the behavior of the global HMSC in Figure 3.10(b). The next section shows that this semantics of scenarios (3.3) uses *a minimal set of notions* to specify the *complete* behavior of a distributed system. If we removed any notion from (3.3), then we could not specify the complete behavior of a distributed system with scenarios.

Design decision: Two scenarios with the same history may be alternatives [2Alt = yes], two scenarios may overlap [2Overlap = yes]

3.6. Specifying Complete System Behavior with Scenarios

In the preceding section, we adopted the notion of a history to express when an occurrence of a scenario makes sense. We then derived a novel, though canonical, semantics of scenarios (3.3) which makes an implicit assumption about how often a scenario occurs in the specified system. Namely, *whenever* the scenario's history occurred. In this section, we stress this assumption by considering how a system designer specifies the *complete* behavior of a distributed system as demanded by the *universal view* on scenarios, which we discussed in Section 3.3.

3.6.1. Completeness with respect to expected behavior

A system designer uses scenarios to specify system behavior which she *expects* to see in the system. That is, she thinks of a set R of partial runs that the system shall exhibit. A system is *complete* wrt. R if every expected run in R is also a run of the system. If the system does not exhibit one or more runs in R, then it is *incomplete*. Beyond these desired runs, a system may exhibit any behavior.

For instance, according to the pragmatics of our running example a medic should be able to ask more than once for a pending emergency. Pragmatically, a system designer may expect scenario M_1' of Figure 3.11 to occur in a system behavior as often as the medic wants and emergencies are pending. A system in which this is not possible is incomplete wrt. this expected behavior.

Completeness wrt. expected runs is a *behavioral property* of a system as sketched in Figure 3.1. In the following, we first show that completeness wrt. expected runs is an elementary behavioral property for the design of distributed systems. Afterwards, we consider how to describe this property with scenarios.

3.6.2. Completeness requires universal quantification and branching

The notion of *closed under continuations* proposes that a scenario occurs *whenever* its history occurred. The following example illustrates that this definition is just sufficient to specify the complete behavior of a *distributed* system. So far, we did not pay attention to this fact. To illustrate the specific challenges in distributed systems, we consider another example: the *crosstalk algorithm*.

Example: the crosstalk algorithm

The crosstalk algorithm describes a symmetric interaction protocol among two components named l and r. The standard interaction is fairly simple: component l requests some resources from component r; r then sends the requested resources to l; l finally returns the resources to r. In the same way, r requests and uses resources from l. The components l and r work autonomously and interact asynchronously [104]. The scenarios L and R in Figure 3.16 describe this interaction behavior of l and r in MSC syntax. The third interaction scenario in this protocol is that l and r mutually request resources at the same time. Assuming that resources are limited, a component that requests resources for itself cannot provide resources to another component if requested. Thus, l denies r's request and r denies l's request. The *crosstalk scenario* C in Figure 3.16 describes this interaction. The behavior specified in $\{L, R, C\}$ describes the crosstalk algorithm [104]. Such behavior can in principle also be observed in reality. For instance, in case of two working groups that usually work autonomously, but may from time to time request help to solve a specific task.

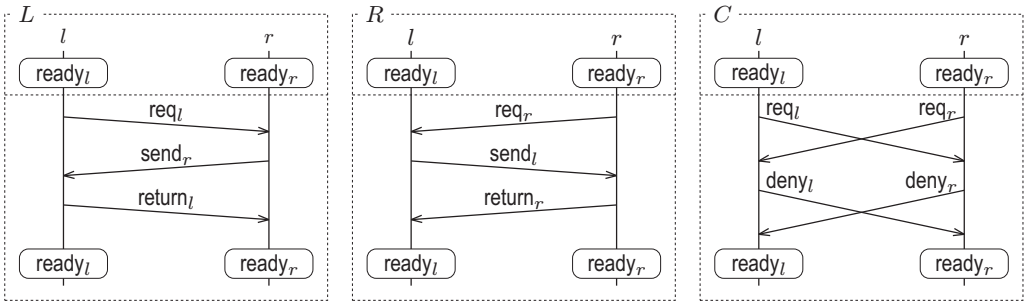


Figure 3.16. The three scenarios of the crosstalk algorithm.

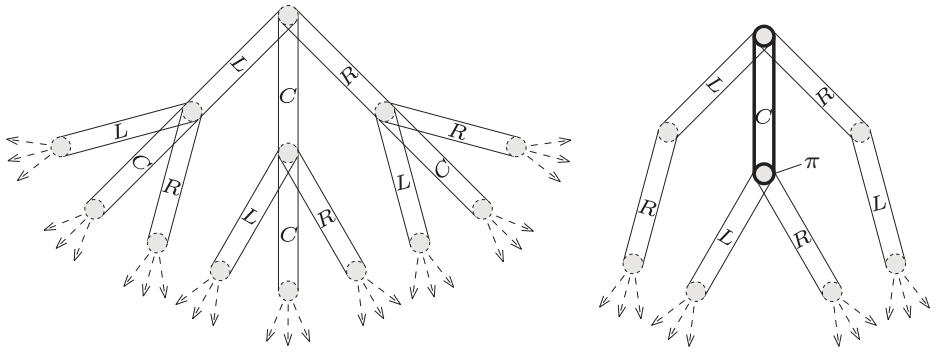
In contrast to all preceding examples the scenarios L and R describe behavior where neither component initiates all interactions. Instead either component may initiate an interaction on its own, and the other component reacts. This locality of choice for initiating an interaction is inherent in all distributed systems.

By example, l initiates scenario L by sending a request to r. The decision to send message req_l depends only on l and its local state. In turn, receiving l's request message req_l involves only r and its local state. Hence, the corresponding send and receive actions are *local* actions as defined in Chapter 2. Their locality allows that both l and r independently decide to send mutual requests in the

same situation: l sends a req_l message, and concurrently r sends a req_r message. Although l decided to initiate scenario L and r decided to initiate R, together they initiate the crosstalk scenario C. That means scenario C cannot be avoided in a distributed system because participants initiate their behavior *locally*.

Complete behavior of the crosstalk algorithm

The crosstalk algorithm illustrates when a system is *complete* wrt. expected behavior. The complete behavior of the crosstalk algorithm follows from a simple observation: whenever l and r are ready l may locally request resources which causes an occurrence of scenario L. Correspondingly, r may locally trigger an occurrence of R, and an occurrence of scenario C cannot be avoided as third alternative. At the end of each scenario, l and r are ready again. Thus, the *complete* behaviors exhibited by the crosstalk algorithm are all infinite arbitrary concatenations of the three scenarios L, R and C, i.e., the infinite distributed runs $(L|R|C)^\omega$ [104, 72]. Figure 3.17(a) indicates the corresponding infinite execution tree T. Each grey circle denotes an occurrence of the mutual precondition of L, R and C.



(a) complete execution tree T (b) incomplete execution tree T'

Figure 3.17. Infinite execution tree of the crosstalk algorithm.

We now consider the tree T as a *behavioral property* that a system shall have, as discussed in Section 3.6.1. More precisely, we want to specify a system S that is complete wrt. T, that is, each run of T is also a run of S. A system that exhibits less behavior, for instance, only the execution tree T' in Figure 3.17(b) is *incomplete* wrt. T.

A distributed system that is incomplete wrt. some expected behaviors may be inherently wrong. For instance, the run π consisting of one occurrence of C in T' does not continue with the crosstalk scenario C. Pragmatically, this means l and r globally coordinate at the end of π to avoid crosstalk. But this coordination would violate the assumption that each component l and r *locally* decides to send a request whenever it "wants". The same argument applies to runs in T' that do not continue with scenario L or R. Thus, a system that is incomplete wrt.

the entire tree T either does not implement the crosstalk algorithm or is not a distributed system where each component acts locally.

Closed under continuations expresses completeness

The notion of *closed under continuations* (3.2) allows to describe the behavioral property of "completeness wrt. expected runs" with scenarios: the execution tree T of the crosstalk algorithm is closed under continuations with L, R, and C as proposed in Definition (3.2). In other words, in a system behavior which is closed under continuations each scenario begins locally without any hidden global coordination. For this reason, the notion of *closed under continuations* suffices to specify complete system behavior.

A simple canonical extension allows to specify *all* system behavior as demanded by the *universal view* on scenarios [25]. The execution tree T (i.e., the behavior of the crosstalk algorithm) is the *least* system behavior where each scenario L, R and C occurs at least once, and is closed under continuations with L, R and C. This property makes *closed under continuations* a fundamental notion to specify system behavior with scenarios.

3.6.3. Conclusion and design decisions: necessary notions for scenarios

Design decision: whenever the history occurred

We now have enough arguments to justify an answer to the question "how often a scenario has to occur?" The two canonical answers are "at least once" and "whenever its history occurred" encoded as [HowOften = once/whenever].

The preceding discussion justifies the choice [HowOften = whenever] which we already made in the definition of *closed under continuations* (3.2). If we assumed the alternative choice [HowOften = once], the semantics of scenarios would be too weak to specify all system behavior. But this property also depends on the choice that two scenarios may be alternatives [2Alt = yes]; the specification $\{L, R, C\}$ is contradictory under the assumption [2Alt = no].

Design decision: a scenario occurs whenever its history occurred [HowOften = whenever]

A minimal set of notions

The decisive contribution of the proposed semantics (3.3) of scenarios is that (3.3) uses a *minimal set of notions* compared to existing techniques: it only requires the notion of *history* and *closed under continuations* (3.2). All other techniques discussed here require at least one other notion to specify complete system behavior, for instance an explicit choice operator, which expresses alternatives. We will show in Chapter 4 that this semantics of scenarios suffices for specifying distributed systems in an intuitive way. There, the various notions are formalized in terms of *oclets*.

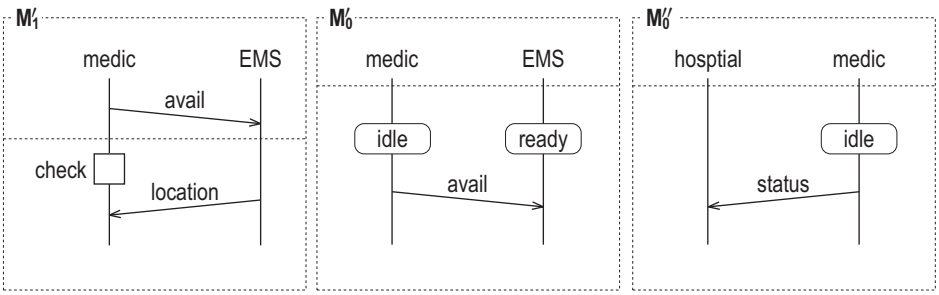


Figure 3.18. The scenario M'_0 and M''_0 each have an empty history. They specify alternative initial runs.

3.7. Non-Empty Behavior and Initial States

Scenario-based techniques which order occurrences of scenarios by their history face a subtle, but important, problem. According to the proposed semantics (3.3) every scenario-based specification is trivially satisfied by the *empty system behavior*: whenever a scenario's history occurs, there is a continuation with this scenario. With the aim of a minimal set of notions in mind, we explore three solutions to the problem.

3.7.1. Specifying non-empty behavior

Existential scenarios. LSCs ensure non-empty system behavior by the help of *existential scenarios*. An existential scenario has to occur at least once in a system behavior to satisfy the specification [25].

Initial runs. An alternative solution emerges when we think of how a system designer may specify *the beginning* of a run of the system. In Section 3.5, we decided to distinguish for each scenario a history that describes when the scenario may occur. Thus, a scenario like M'_1 in Figure 3.18 may only occur *after* some other behavior has occurred, e.g., message avail. At the beginning of a run, *no behavior has occurred yet*. We can express this by an *empty history* as in scenario M'_0 in Figure 3.18.

If we read an empty history as "no behavior has occurred yet", then the definition of *closed under continuations* (3.2) yields the following semantics of a scenario S with an empty history:

> whenever no behavior has occurred yet (i.e., at the beginning of a run), there exists a continuation ρ with scenario S. (3.4)

In other words, a scenario with an empty history describes an *initial run*. A system behavior would satisfy such a scenario S if S occurs as a prefix of some run.

Definition (3.4) allows to specify *alternative initial runs*: two scenarios with the same history may be alternatives according to Section 3.5.2. For instance, M'_0 and

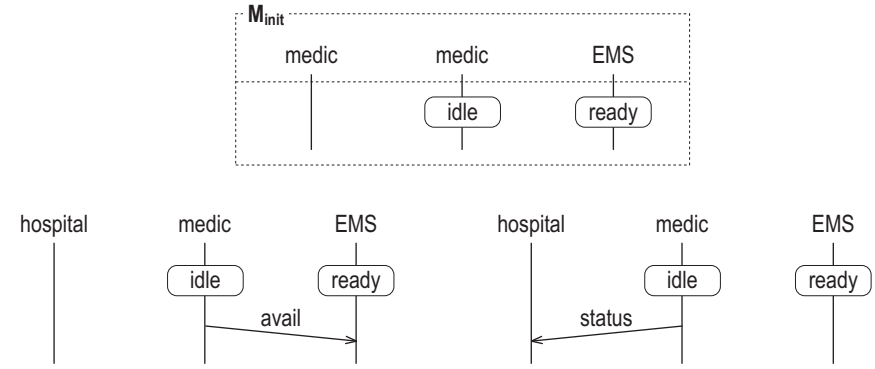


Figure 3.19. Scenario $\mathsf{M}_{\mathsf{init}}$ specifies the least common initial state of $\{\mathsf{M}_0', \mathsf{M}_0'', \mathsf{M}_1'\}$. This initial state can be continued with M_0' and with M_0'' in alternative initial runs.

M_0'' in Figure 3.18 describe two alternative initial runs. Any system behavior that satisfies the specification $\{\mathsf{M}_0', \mathsf{M}_0''\}$ gives the medic the initial choice to either send an avail message to the EMS or a status message to the hospital.

A unique initial state. Considering M_0' and M_0'' closely we see that their interpretation as initial runs permits a system to have *different initial states*. Scenario M_0' implies the initial state [idle, ready] whereas scenario M_0'' implies the initial state [hospital, idle]. There is no reason to prefer any of these initial states over the other. Specifying multiple initial states poses a problem for synthesizing an implementation from a scenario-based specification because a classical system model usually describes a *unique initial state*.

We could require that all scenarios with empty history denote the same initial state. The disadvantage of this requirement is that it restricts scenario-based specifications to certain sets of scenarios. Another approach is to derive the *least common initial state* of a scenario-based specification — simply by taking the union of all initial states in the specification. The least common initial state is unique.

For example, the scenario $\mathsf{M}_{\mathsf{init}}$ in Figure 3.19 is the least common initial state of the specification $\{\mathsf{M}_0', \mathsf{M}_0'', \mathsf{M}_1'\}$ of Figure 3.18. $\mathsf{M}_{\mathsf{init}}$ can be continued to a complete occurrence of M_0' and M_0'', respectively, as illustrated in Figure 3.19.

3.7.2. Design decision: initial runs

This section discussed three possibilities for specifying non-empty system behavior with scenarios: existential scenarios, initial runs, and a unique initial state; encoded as [NonEmpty = existential/initial run/initial state].

Assuming that each specification describes a unique initial state would restrict the notion of a specification. For instance, the set $\{\mathsf{M}_0', \mathsf{M}_0'', \mathsf{M}_1'\}$ of scenarios of Figure 3.18 would not be a specification. In contrast, the notions of existential scenarios and initial runs impose no restrictions. On the one hand, assuming initial runs is stricter than assuming existential scenarios because initial runs require

that a scenario occurs at a specific location. On the other hand, the semantics of initial runs (3.4) specializes *closed under continuations* (3.2) where the empty history describes that no behavior has occurred yet.

With the aim of identifying a minimal set of notions for scenarios, we choose initial runs to specify nonempty system behavior, [NonEmpty = initial run]. We will construct the least common initial state of a specification when it comes to synthesizing an implementation from a scenario-based specification.

Design decision: Initial runs specify nonempty behavior [NonEmpty = initial run].

3.8. Occurrences of Scenarios

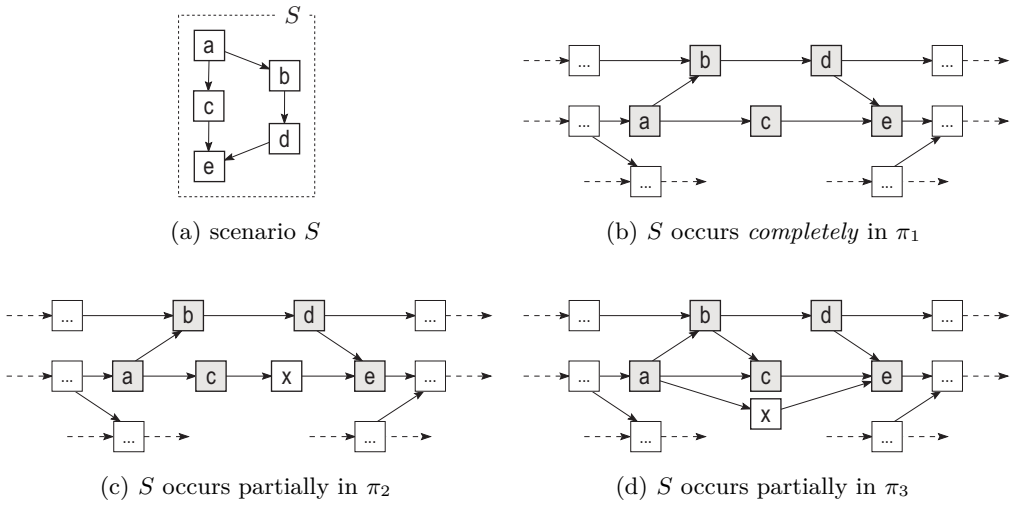
The preceding sections discussed syntax and semantics of scenarios in existing techniques at the conceptual level. The discussion has led us to a minimal set of notions for the semantics of scenarios which we formalize in the next chapter. The most basic notion is when a "scenario *occurs* in a run." In this section, we discuss different notions of occurrence of a scenario wrt. its influence on the relation between specification and satisfying behavior. Our formal model for scenarios in the next chapter is based on our findings in this discussion.

3.8.1. Different notions of occurrences

The main characteristic of a scenario is that it is a *partial* description of system behavior as discussed in Section 3.3 and in literature, e.g., [31, 8]. So what exactly is "partially" stated in a scenario? Certainly, the point *where* the scenario may occur in a run. Moreover, at a specific occurrence of a scenario, the run may contain further events than stated in the scenario which leads to different notions of occurrences. We distinguish *complete* occurrences of a scenario from *partial* occurrences of a scenario. Up to now, we considered only complete occurrences of a scenario in a run. Figure 3.20 illustrates several examples in the syntax of labeled partial orders; see also Figure 3.4(a).

An occurrence of a scenario in a distributed run is *partial* if the run contains (1) one or more occurrences of actions *between* two subsequent actions of the scenario or (2) if an action in the run has a predecessor (or successor) that is not described in the scenario. For instance, the occurrence of S is partial in Figure 3.20(c) because the additional action x occurs between c and e. The occurrence of S is partial in Figure 3.20.(d) because the additional action x succeeds a and precedes e. In contrast, any two subsequent actions of a complete scenario also occur subsequently in the run without any additional predecessors or successors.

If the semantics of scenarios allows partial occurrences of scenarios, then there are *more* runs that satisfy a specification. Simply because there are more runs where a scenario occurs. For instance, by allowing partial occurrences, scenario S of Figure 3.20 occurs in π_1, π_2, and π_3.



(a) scenario S

(b) S occurs *completely* in π_1

(c) S occurs partially in π_2

(d) S occurs partially in π_3

Figure 3.20. Different interpretations for how scenario S occurs in a distributed run.

Partial occurrences of scenarios in existing techniques. Existing scenario-based techniques allow to parameterize a scenario wrt. its occurrences, for instance, by allowing partial occurrence for some scenarios and requiring complete occurrences for others. Moreover, they provide more refined notions of partial occurrences. A system designer could, for instance, require that only between some actions of a scenario another action may occur.

LSCs provide the notion of an action that is *invisible* to a scenario; invisible actions may occur arbitrarily between two actions of a scenario. A system designer can define the set of invisible actions for each scenario separately if desired [25]. Template MSCs [45] follow the reverse approach. A partial occurrence is explicitly allowed by marking a region between two actions in which a specified set of actions is allowed to occur.

The following sections discuss how the different notions of occurrences influence the relation between a specification and system behavior. We finally make a design decision about whether our formal model for scenarios considers only complete occurrences of scenarios or also partial occurrence; [Occurrence = complete/partial].

3.8.2. Complete Occurrences: Syntax and Semantics Correspond Smoothly

First, we discuss some properties of complete occurrences. For every complete occurrence of a scenario in a distributed run, we observe a smooth correspondence between a scenario-based specification (which is syntax) and satisfying behavior (which is semantics). In terms of graphs, a complete occurrence of a scenario is a *connected* sub-graph in a run which we formalized as a *partial run* in Definition 2.15.

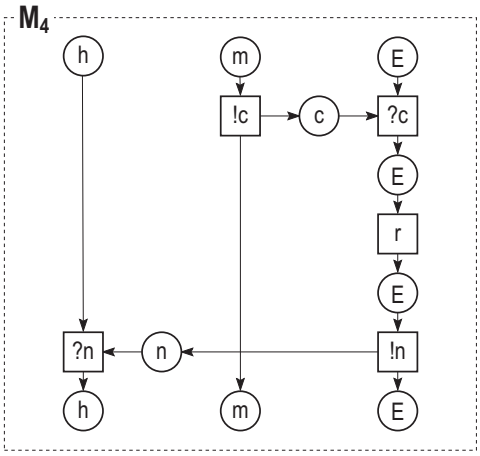


Figure 3.21. Scenario M_4 of Fig. 3.3 in Petri net notation.

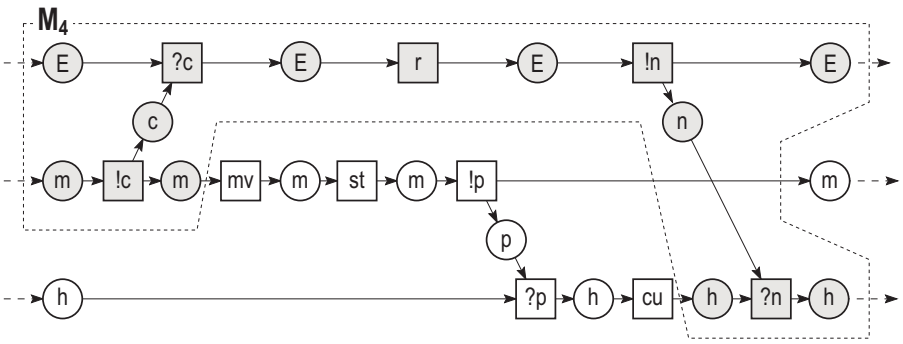


Figure 3.22. A part of the run π_2 of Fig. 3.8 in Petri net notation.

For example, the scenarios M_1-M_4 of the medical emergency specification of Figures 3.2 and 3.3 occur as partial runs in the distributed runs π_1 and π_2 of Figure 3.8. Figures 3.21 and 3.22 highlight such a complete occurrence of scenario M_4 in π_2 in the syntax of Petri nets.

This direct relation between the syntax of scenarios and their semantics allows to use the *same* technical notions to formalize scenarios and system behavior. Building syntax and semantics on the same technical notions is one goal of this thesis. It should simplify formal definitions and may provide a more intuitive understanding of the specified behavior. We shall follow this approach in Chapter 4.

A phenomenon of distributed runs. The notion of a complete occurrence of a scenario is genuine to distributed runs. Complete occurrences cannot be defined in the same way on sequential runs. We briefly compare how a sequential run and how a

distributed run reflect a complete occurrence of a scenario. A possible sequential observation of the partial run of Figure 3.22 is the sequence

$$\ldots, \underline{!c}, mv, \underline{?c}, st, \underline{f}, !p, \underline{!n}, ?p, cu, \underline{?n}. \tag{3.5}$$

The scenario M_4 occurs in π_2 (grey shaded) as well as in (3.5) (underlined), but in (3.5) its actions occur interleaved with other actions of the run. More sharply, M_4 does not occur as a connected sub-sequence in (3.5). There is actually no sequentialization of π_2 in which M_4 occurs as a connected sub-sequence because of the actions mv, st, etc. A system designer who wants to relate a sequential observation of π_2 to the scenario M_4 has to identify several sub-sequences in the run and relate these to each other. In contrast, the complete occurrence of scenario M_4 as a partial distributed run (i.e., as connected sub-graph) in π_2 of Figure 3.8 is obvious.

Towards a simple formal model for scenarios. The gap between syntax of scenarios and sequential runs indicates one reason why formal semantics of scenarios and solutions of the synthesis problem are technically involved. Sequential runs generally "break" an occurrence of a scenario into several non-connected parts. This way, an occurrence of a scenario is no longer a local phenomenon but requires global observations on the sequential run.

In contrast, a complete occurrence of a scenario in a distributed run is a local phenomenon: all events and conditions of the scenario occur next to each other in the run. We therefore expect a more intuitive understanding of the semantics of scenarios on distributed runs. Likewise, a corresponding formal semantics may benefit from scenarios occurring locally as partial distributed runs.

3.8.3. Partial occurrences of scenarios

After presenting some advantages of complete occurrences of scenarios in distributed runs, we now consider how partial occurrences influence the semantics of scenarios.

Scenarios are possible alternatives. We stated in Sections 3.3 and 3.5 that two different scenarios with the same history describe alternative courses of actions. The scenarios M_1 and M_2 in Figure 3.2 are an example. Consider the distributed run π in Figure 3.23. If we assume partial occurrences, the both scenarios M_1 and M_2 occur *overlappingly* in the run π on the message avail and the action check. The behavior $\{\pi\}$ alone satisfies the specification $\{M_1, M_2\}$ under the assumption of partial occurrences.

The problem here is that location and none occur in alternative scenarios M_1 and M_2, but location and none directly succeed each other in π. Thus, we lose the ability to describe two alternative courses of actions — unless we introduce another concept like an explicit, global notion of choice between scenarios [107, 65].

We could restrict the notion of partial occurrences by the help of invisible actions. An action is *invisible* wrt. a specification if the action does not occur in any of the

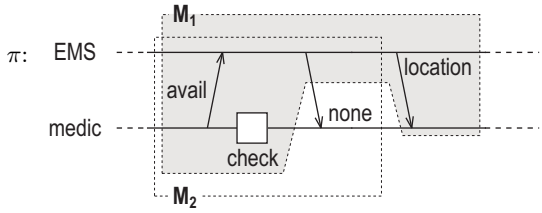


Figure 3.23. Allowing partial occurrences, the alternative scenarios M_1 and M_2 of Fig. 3.11 may occur in one run overlappingly.

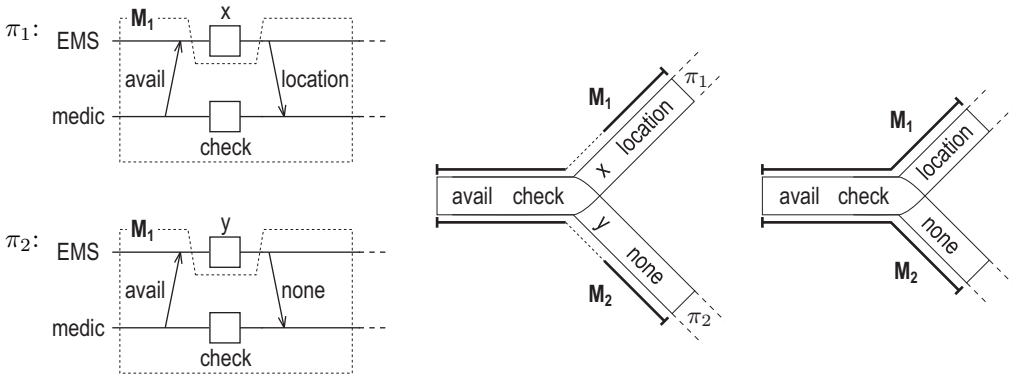


Figure 3.24. Partial occurrences cannot distinguish non-deterministic choices. The tree in the middle and on the right are not bisimilar, but cannot be distinguished by partial occurrences of M_1 and M_2.

scenarios of the specification. In the restricted notion, only invisible actions may occur between two actions of a scenario. In this interpretation, M_1 does *not occur* in π in Figure 3.23, because none is visible. Thus, M_1 and M_2 may occur only in alternative runs or one after the other.

Moment of choice. This proposed restricted notion of partial occurrences is still problematic. Consider the runs π_1 and π_2 in Figure 3.24 and their corresponding execution tree (middle). The runs π_1 and π_2 branch directly after message avail. The choice between both runs is made by an occurrence of either x or y after avail occurred. Actions x and y are invisible to the scenarios M_1 and M_2. This may contradict the intuitive understanding of M_1 and M_2 which only branch before message location of none is sent as illustrated by the right-most tree of Figure 3.14.

This phenomenon is well-understood in the notion of *(branching) bisimulation* [119]. Intuitively, two systems are (branching) bisimilar if every (visible) step that the first system makes can be mimicked by an equivalent (visible) step in the second system, and vice versa. Specifically, if the first system makes a choice, then any equivalent (visible) step in the second step yields the same choice. The two trees in Figure 3.24 are a classical example for two system that are *not* for

branching bisimilar. Though both trees satisfy the specification $\{\mathsf{M}_1, \mathsf{M}_2\}$ under partial occurrences.

3.8.4. Design decision: complete occurrences

Taking these observations together, we conclude that partial occurrences introduce unobservable non-determinism. This certainly makes semantics of scenarios and reasoning about systems more involved. For this reason, we decide to formally define semantics of scenarios in Chapter 4 with complete occurrences of scenarios and consider partial occurrences as an extension.

Design decision: complete occurrences of scenarios [Occurrence = complete]

Though, partial occurrences are in line with the notion of a scenario's history as discussed in Section 3.5 in the following sense. A system designer may specify a partial history of a scenario containing only those actions that suffice for an occurrence of the entire scenario. The history may omit other actions that are irrelevant for the scenario's occurrence.

3.9. Conclusion: Specification and Implementation

In this chapter, we gave an introduction to the scenario-based approach for specifying behavior of distributed systems. We discussed syntax and semantics of the state of the art in scenario-based techniques. Section 3.3 made several underlying assumptions about the semantics of scenarios explicit as *open design decisions*. Exploring these design decisions, we identified a *minimal set of notions* that are necessary to specify all bebaviors of a distributed system in Sections 3.4-3.6. We found that for specifying all system behavior, a system designer has to describe alternative scenarios. Existing techniques always require additional syntactic concepts on top of the basic scenario notation to specify alternatives.

Based on our findings, we identified a class of scenarios that is conceptually simple and allows for specifying system behavior in an intuitive way. In this class, a scenario has a *history* which describes when the scenario may occur. A specification is a set of scenarios. A set of distributed runs *satisfies* a specification iff whenever the history of a scenario S occurs in a run, then *there exists* a continuation of this run with the entire scenario S. The crosstalk example in Section 3.6.2 demonstrated that the notions required by this semantics are *minimal* and *sufficient* for specifying all behaviors of a distributed system. With the synthesis problem in mind, we decided to specify *initial runs* by scenarios with *empty history* in Section 3.7. Section 3.8 discussed several notions of occurrences of scenarios and how these influence the semantics. We finally decided to consider only *complete occurrences* of scenarios. Altogether, the following design decisions were made: [Occurrence = complete, Ordering = history, HowOften = whenever, 2Alt = yes, 2Overlap = yes, NonEmpty = initial run].

Concluding example. The following example summarizes all notions and insights of this chapter. To this end, we relate a scenario-based specification Med to an *implementation*: a system model is an implementation of Medif its behavior satisfies Med.

The specification Med consists of the scenarios M_0', M_1', and M_2' of Figure 3.25. They formally express the first and the second natural language scenario from our running example in Section 3.2. Scenario M_0' specifies an initial run which ensures that the medic sends message avail at least once when he is idle according to our design decision [NonEmpty = initial run] in Section 3.7.

The Petri net system Σ_1 in Figure 3.26 implements the specification Med under our chosen assumptions [2Alt = yes, Occurrence = complete, HowOften = whenever]. Scenario M_0' occurs as initial run of Σ_1, and the behavior of Σ_1 is closed under continuations with M_1' and M_2'. Further, each scenario only occurs completely in the runs of Σ. The left part of Σ_1 contains all actions and states of the medic, the right part of the net contains all actions and states of the EMS. The three places avail, none, and location describe the message channels in the system.

In contrast to Σ_1, the Petri net system Σ_2 in Figure 3.27 does *not* implement Med: Σ_2's behavior is closed under continuations with M_1' but not with M_2'. When M_2' occurs for the first time, the token from place p is consumed and not put back. Thus, transition !none is not enabled after ?avail occurs the second time, which means M_2' cannot occur *whenever* its history occurred. In contrast, M_1' can occur even after the token from p is consumed. Thus, Σ_2 implements M_1' but not M_2'.

In these examples, system model, system behavior, and specification correspond to each other: for instance, we find scenario M_1' as a substructure of Σ_1 of Figure 3.26, consisting of transitions !avail, ?avail, check, !location, ?location, and adjacent places.

The next two examples highlight the importance of the assumption of complete occurrences of scenarios [Occurrence = complete] which we made in Section 3.8. In the following, we assume [Occurrence = partial]. Figure 3.28 depicts a "typical" implementation of Med. The system is more detailed than the scenarios. For instance, it extends the EMS by a component that evaluates a query. The result

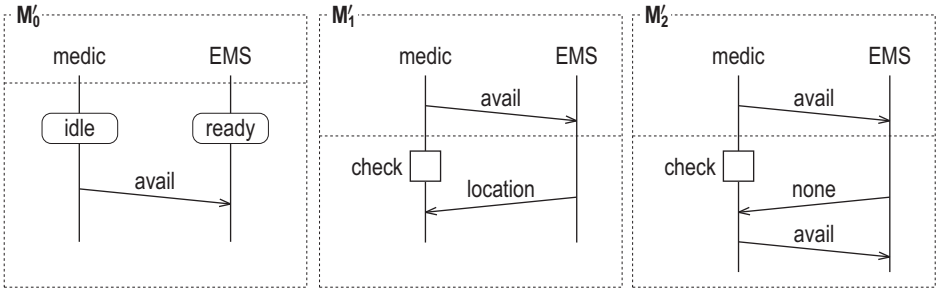


Figure 3.25. Sample specification Med = $\{\mathsf{M}_0', \mathsf{M}_1', \mathsf{M}_2'\}$. A medic queries an emergency management system for a pending medical emergency.

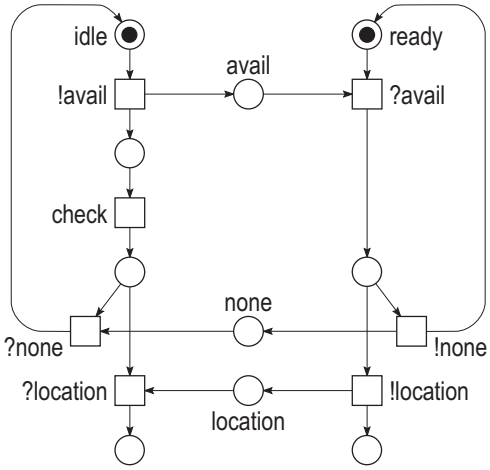


Figure 3.26. Petri net system Σ_1 implements Med.

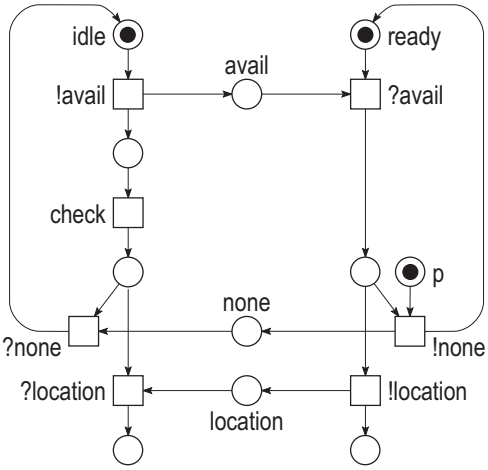


Figure 3.27. Petri net system Σ_2 implements M_1' but not $M_2' \in$ Med.

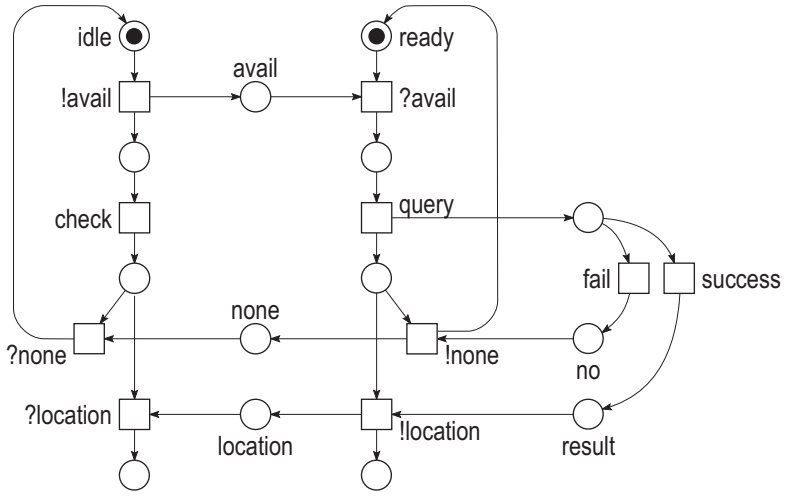


Figure 3.28. Petri net system Σ_3 is a "typical" implementation of Med, assuming [Occurrence = partial].

of the query determines which message the EMS sends back to the medic. Strictly speaking, Σ_3 implements Med only if we assume [Occurrence = partial]. Scenarios M_1' and M_2' only occur partial in Σ_3 because the EMS implements transition query, fail, and success between ?avail, !none, and !location. The systems Σ_1 and Σ_3 are branching bisimilar if we assume all transitions which do not appear in the specification Med to be invisible.

The last example in this conclusion illustrates that partial occurrences of scenarios yield unpredictable results. If [Occurrence = partial] holds, then the Petri

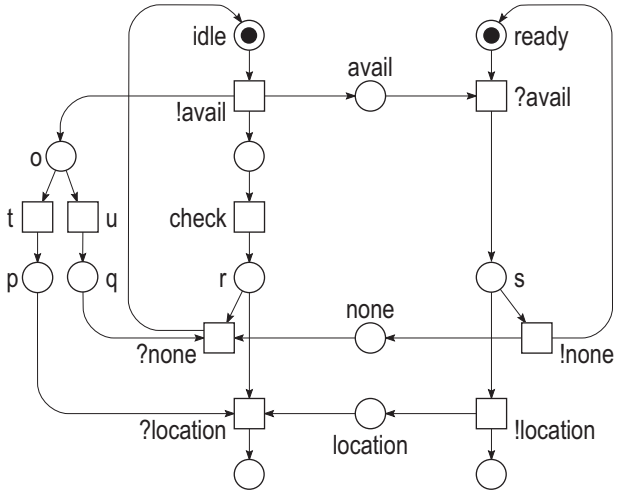
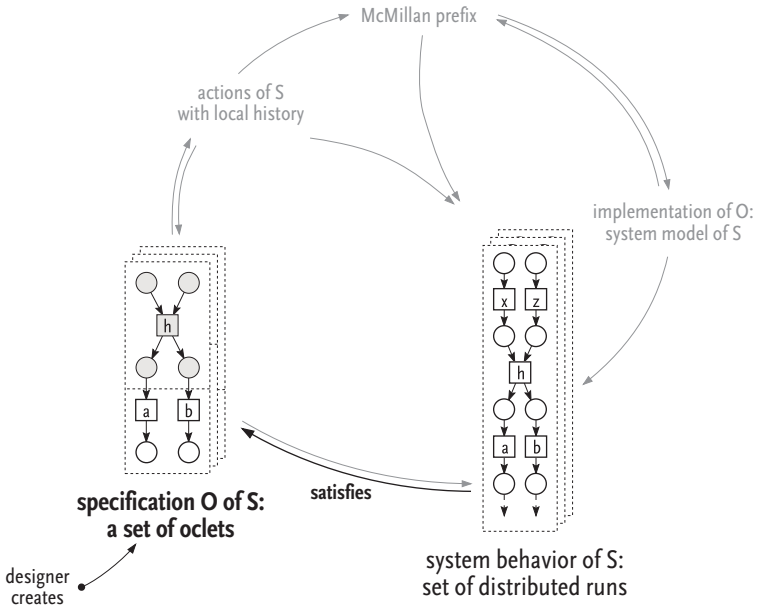


Figure 3.29. Petri net system Σ_4 implements Med assuming [Occurrence = partial]; Σ_4 has an undesired deadlock.

net system Σ_4 in Figure 3.29 does implement the specification Med. For instance, whenever the history of M_1' occurs, Σ_4 has a run that continues with entire M_1'. But, this system has an undesired deadlock: the marking $[\mathsf{o}, \mathsf{r}, \mathsf{s}]$ is reachable in Σ_4; the steps $[\mathsf{o}, \mathsf{r}, \mathsf{s}] \xrightarrow{\mathsf{t}} [\mathsf{q}, \mathsf{r}, \mathsf{s}] \xrightarrow{!\mathsf{none}} [\mathsf{q}, \mathsf{r}, \mathsf{none}]$ reach a deadlock — the local choices t of the medic and !none of the EMS do not correspond to each other. As discussed in Section 3.8, partial occurrences of scenarios do not distinguish the moment of choice. In Σ_4, the choice whether the medic follows scenario M_1' or M_2' is made by transitions t and u outside of M_1' and M_2'. The systems Σ_4 and Σ_1 are *not* branching bisimilar, unlike Σ_3 and Σ_1. This example justifies again our choice for complete occurrences of scenarios [Occurrence = complete].

The next steps. The next chapter formalizes the minimal notions of scenarios identified in this chapter in the model of *oclets*. Oclets constitute a kernel of scenario-based specifications with history. They essentially solve the first research problem of this thesis: to identify a minimal and sufficiently expressive class of scenario-based specifications. With oclets, we shall then address the second and the third research problem: to define operational semantics for scenarios (Chapter 5) and to solve the synthesis problem (Chapter 7).

4. Oclets



The preceding chapter reviewed the scenario-based approach for specifying behaviors of a distributed system S. We identified a kernel of scenarios that uses a minimal set of notions and is expressive enough to specify all behaviors of S. This chapter defines *oclets* as a formal model for the identified kernel. Oclets provide scenarios intuitive and well-defined semantics in the style of Live Sequence Charts. This formal model serves as a foundation for solving the challenges of the scenario-based approach in the subsequent chapters.

4.1. About this Chapter

This chapter introduces a novel formal model for scenario-based specifications which we call *oclets*. Oclets formalize our findings about the semantics of scenarios of Chapter 3 in terms of *distributed runs* of Chapter 2. We proceed as sketched in Figure 4.1. Section 4.2 introduces the syntax of oclets.

1. An oclet *formalizes* a scenario as a distributed run with a distinguished *history*. An *oclet specification* is a finite set of oclets.

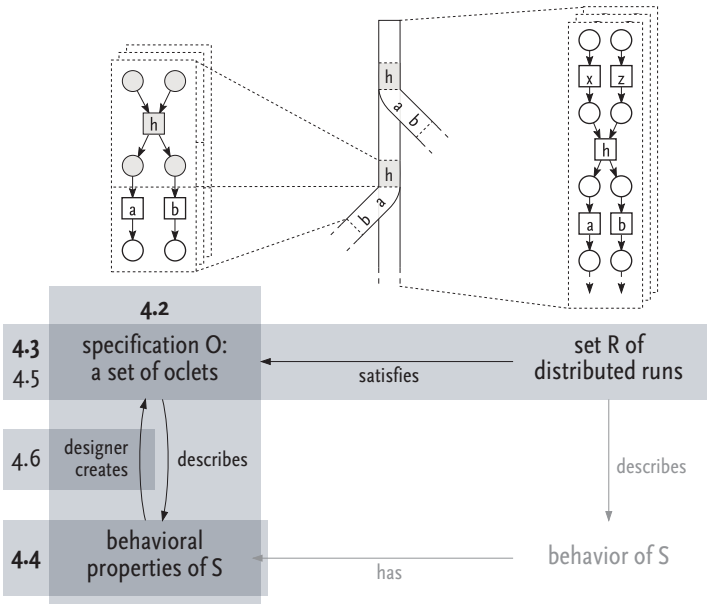


Figure 4.1. A set O of oclets specifies behavioral properties of a distributed system S.

An oclet specification O describes *behavioral properties* of a distributed system S and a *set R of distributed runs* describes the behavior of S. The *semantics* of oclets defines when R has all properties described by O:

2. A set R of distributed runs *satisfies* an oclet o if o occurs "often enough" in R: to each run π in R ending with o's history exists a continuation of π with o in R.
3. R *satisfies* an oclet specification O if R satisfies each oclet in O.

We developed this semantics in Chapter 3. Section 4.3 formalizes these notions in classical *declarative semantics* for oclets, which is standard for specification techniques. We then prove in Section 4.4 several basic properties of oclets and examine which behavioral properties of S can be described with oclets. Most importantly, an oclet specification O has a *unique minimal* set of runs that satisfies O. Moreover, we show that a set R of distributed runs satisfies a *composition*

$O_1 \cup O_2$ of two oclet specifications O_1 and O_2 if and only if R satisfies both O_1 and O_2. Section 4.5 shows how to extend the semantics of oclets towards a more expressive scenario-based technique. We demonstrate how to specify a system S using oclets in Section 4.6 by an example. Section 4.7 concludes this chapter and discusses how oclets relate to other formal models for scenarios.

We choose to formalize oclets using distributed runs. This choice makes oclets a technique where scenarios, system behavior, and implementation are all based on the same technical notions: Petri nets. Because of this technical convergence, we may develop a solution to the synthesis problem with a minimal technical overhead in the forthcoming chapters.

4.2. Syntax of Oclets

We now present our formal model for scenarios. This section introduces the syntax of oclets, the next section their semantics. The syntax constrains how a system designer notes down scenarios and specifications. In the spirit of Chapter 3, we formalize a scenario as an *oclet* which is a finite distributed run with a distinguished history. Note that the history may be empty to specify initial runs as decided in Section 3.7.

Definition 4.1 (History, Oclet). Let π be a distributed run. A *history* of π is a prefix $hist \sqsubseteq \pi$ that is either empty (i.e., $hist = \varepsilon$) or contains all minimal nodes of π (i.e., $\min hist = \min \pi$).

An *oclet* $o = (\pi_o, hist_o)$ is a distributed run π_o with a history $hist_o$ of π_o. We also call $hist_o$ the *history* of o, and $con_o := \pi_o - hist_o$ the *contribution* of o. ⌟

An oclet is *finite* if its underlying distributed run is finite. Intuitively, an *oclet specification* O is a finite set of finite oclets. However, we will provide a slightly different formalization for technical reasons in Section 4.3. The term "oclet" refers to the idea of describing a fragment of behavior that *occurs* repeatedly in the specified system.

We use the graphical syntax of Petri nets to depict an oclet. The oclet $\mathsf{o_1}$ on the left-hand side of Figure 4.2 describes the first scenario of our medical emergency example of Chapter 3. The vertical dashed line distinguishes the history (left, grey-shaded) from the contribution (right). $\mathsf{o_1}$ describes an interaction of two components E and m: after events $\mathsf{!av}$ and $\mathsf{?av}$ of the history occurred, the system may continue with events ch, $\mathsf{!loc}$, and $\mathsf{?loc}$ of the contribution.

Histories distinguish two kinds of oclets either having an empty history or a non-empty history. An oclet with empty history is called *ε-oclet*. For the other kind, we simply write "oclet *with* history". The oclet $\mathsf{o_1}$ of Figure 4.2 is a typical oclet with history. Figure 4.3 depicts two ε-oclets which specify the interaction behavior of each component E and m from oclet $\mathsf{o_1}$ in isolation.

An ε-oclet o describes that the system may continue with o's contribution *when no behavior has occurred yet*, that is, how to start a run. Thus, ε-oclets allow to specify that the system does something at all as discussed in Section 3.7.

A non-empty history is a special prefix which contains *all* minimal nodes of the oclet, i.e., $\min hist_o = \min \pi_o$. The rationale for this notion is that a history of an

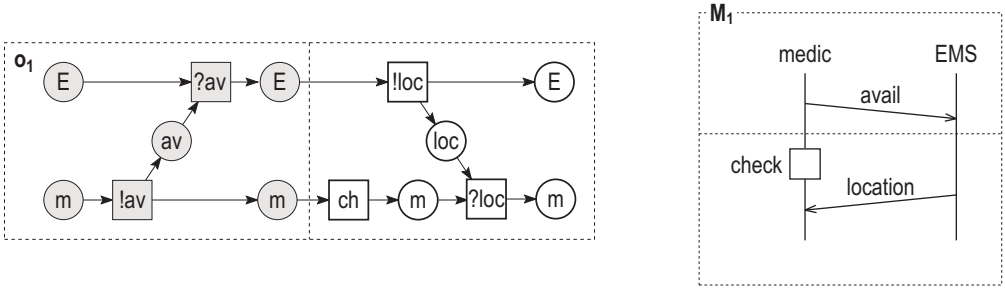


Figure 4.2. An example oclet o_1 in Petri net notation and in MSC notation, formalizing the first scenario of the medical emergency system of Chapter 3.

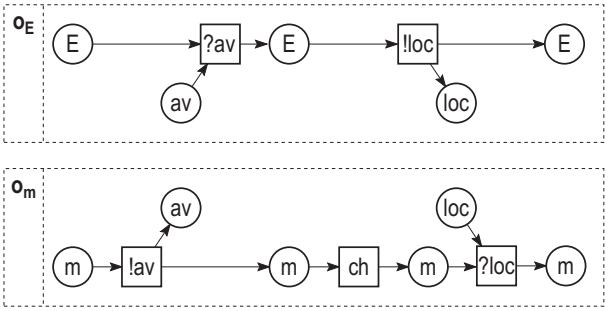


Figure 4.3. Two oclets with empty history.

oclet *entirely precedes* the oclet. Figure 4.4 depicts a scenario that violates this property: action !loc is *concurrent* to the scenario's history and, therefore, could occur concurrently to or before the history. The restriction $\min hist_o = \min \pi_o$ of Definition 4.1 forbids such non-causal descriptions of behavior.

Comparison to the syntax of MSCs. The graphical syntax of oclets is slightly redundant. Usually, a system designer only describes a course of events, not of events and conditions, for instance using MSC notation as shown on the right-hand side of Figure 4.2. Though, we can read an MSC as a short-hand for an oclet: the oclet's

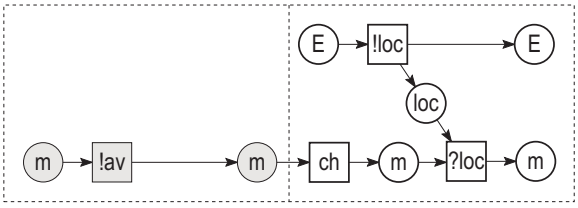


Figure 4.4. Not an oclet: the contribution contains a node without predecessor (condition E).

Figure 4.5. Two oclets F_1 and F_2 specifying behavior of the flood alert process of Sect. 2.1. Oclets do not require to assign each event to a specific component.

conditions follow canonically from the MSC notation by placing a corresponding condition between two events, and at the beginning and end of the oclet [69].

In direct comparison, the syntax of oclets is less restrictive than the syntax of MSCs. For instance, the oclets F_1 and F_2 in Figure 4.5 specify some behavior of our flood alert process from Figure 2.1 in Section 2.1. The condition sb (sand bags placed) in F_2 is not associated to a specific component whereas MSCs require all events and conditions to be associated to one (or more) specific components as discussed in Section 3.2. Though, a system designer can express components in oclets if she wants as illustrated by oclet o_1 of Figure 4.2. In the forthcoming sections we employ the notation that is most appropriate for studying a specific aspect of oclets at a time.

4.3. Semantics of Oclets

With the formal syntax of oclets at hand, we now define their semantics. The semantics of oclets formally defines a *satisfies* relation between system behavior and oclet specifications. The behavior of a system is described by its set of distributed runs as introduced in Section 2.3.

4.3.1. The approach

We first formalize when an oclet *occurs* in a distributed run, and when a (prefix of a) run *ends with* an oclet's history. Both notions together yield when a run *continues* with an oclet. Likewise, we formalize when an oclet occurs as a *prefix* of a distributed run.

A set R of runs *satisfies* an oclet o with history iff for each prefix π of a run in R that ends with the o's history, there exists a run ρ in R that continues π with o. To ensure that R is non-empty, we specify *initial runs* with ε-oclets. A set R of runs *satisfies* an ε-oclet o iff o occurs as a *prefix* of some run in R. Altogether, a set R of runs satisfies an oclet specification O iff R satisfies each oclet in O.

4.3.2. Oclets occur in distributed runs

Our most basic definition formally expresses when an oclet o *occurs* in a distributed run ρ. This is the case when ρ contains o's events and conditions in the same order as in o. However, an oclet may occur several times in a distributed run as well as

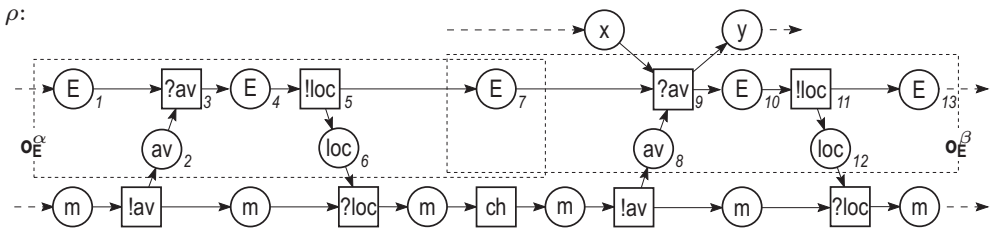


Figure 4.6. Oclet $\mathsf{o_m}$ occurs in π twice.

in several distributed runs. For example, oclet $\mathsf{o_E}$ of Figure 4.3 occurs twice in the run ρ of Figure 4.6 as indicated. We have to distinguishing these occurrences of $\mathsf{o_E}$. This technical problem will reappear frequently throughout this thesis. We solve it using the following technique.

We canonically consider for each oclet o its class $[o]$ of isomorphic oclets. Two different oclets $o', o'' \in [o]$ describe different *occurrences* of oclet o. We determine a specific occurrence $o' \in [o]$ by referring to its identifying isomorphism $\alpha : o \to o'$. We call the isomorphism α the *location* of the occurrence o' of o and write o^α as a shorthand.

For instance, the run ρ of Figure 4.6 contains two occurrence $\mathsf{o_E}^\alpha$ and $\mathsf{o_E}^\beta$ of $\mathsf{o_E}$ at locations α and β as indicated. Both occurrences overlap on condition 7 with label m. In contrast, oclet $\mathsf{o_m}$ of Figure 4.3 does not occur in ρ because the events of $\mathsf{o_m}$ occur in ρ in a different order (i.e., the arcs of $\mathsf{o_m}$ are not preserved).

Distinguishing different occurrences

Subsequently, we present the formal definitions to distinguish different occurrences of oclets. These notions follow canonically from isomorphisms on distributed runs.[1] The aim of the following definition is to distinguish two isomorphic runs π', π'' which *occur* as sub-nets of a distributed run ρ.

Definition 4.2 (Occurrence, location). Let π be a distributed run.

1. An *occurrence* of π is a distributed run that is isomorphic to π; $[\pi]$ denotes the class of all isomorphic occurrences of π. We write π^α for the occurrence of π that is defined by the isomorphism $\alpha : \pi \to \pi^\alpha$. α is the *location* of π^α.
2. Let ρ be a distributed run. π *occurs* in ρ iff π is a sub-net of ρ, i.e., $\pi \subseteq \rho$. We say that π *occurs* in ρ *at location* α, iff the occurrence $\pi^\alpha \in [\pi]$ occurs in ρ, i.e., $\pi^\alpha \subseteq \rho$. ⌟

The notion of an occurrence of a distributed run canonically extends to oclets as follows. Let $o = (\pi_o, hist_o)$ be an oclet. Every occurrence $\pi_o^\alpha \in [\pi_o]$ induces the occurrence $hist_o^\alpha \in [hist_o]$ by restricting α to the nodes of $hist_o$. The *oclet class* $[o]$ defines the oclets $[o] := \{o^\alpha \mid o^\alpha = (\pi_o^\alpha, hist_o^\alpha), \pi_o^\alpha \in [\pi]\}$. We call each $o^\alpha \in [o]$ an *occurrence* of o, and we read o as the *canonic representative* of the oclet class

[1] Isomorphisms on Petri nets and distributed are defined formally in Appendix A.3.

$[o]$. To simplify notation, we abbreviate $\forall \pi^\alpha \in [\pi]$ by $\forall \pi^\alpha$ and $\exists \pi^\alpha \in [\pi]$ by $\exists \pi^\alpha$. Correspondingly, for $\forall o^\alpha$ and $\exists o^\alpha$.

With these notions, any formal definition for an oclet o canonically extends to each occurrence o^α of o.

Complete occurrences of an oclet

Intuitively, an oclet $o = (\pi_o, hist_o)$ *occurs* in a distributed run ρ if its underlying run π_o occurs in ρ as a sub-net, i.e., $\pi_o \subseteq \rho$. This notion canonically extends to all occurrences o^α of o.

In the following, we consider a stricter notion of occurrences of oclets as discussed in Section 3.8. An oclet o *occurs complete* in a run ρ if additionally each event of o's contribution has the same pre- and post-conditions in o and in ρ. To lighten notation, we abbreviate by $pre_o(x) := pre_{\pi_o}(x)$, and $post_o(x) := post_{\pi_o}(x)$ the pre-set, and the post-set of a node x in an oclet $o = (\pi_o, hist_o)$, respectively.

Definition 4.3 (Complete occurrence of an oclet). Let ρ be a distributed run. An oclet $o = (\pi_o, hist_o)$ *occurs complete* in ρ, written $o \subseteq_{\mathbf{c}} \rho$, iff $\pi_o \subseteq \rho$ and

$$\forall e \in E_{con_o} : pre_o(e) = pre_\rho(e) \wedge post_o(e) = post_\rho(e). \tag{4.1}$$

⌟

The notion of a complete occurrence of an oclet o canonically extends to each occurrence o^α in the oclet class of o. Following the approach of Section 4.3.1, we say that o *occurs in* ρ *at location* α iff $o^\alpha \subseteq_{\mathbf{c}} \rho$. The notation $o^\alpha \subseteq_{\mathbf{c}} \rho$ compactly expresses that for the oclet $o' = (\pi'_o, hist'_o) \in [o]$ with the isomorphism $\alpha : o \to o'$ holds: $\pi'_o \subseteq \rho$ and $\forall e \in E'_{con_o} : pre'_o(e) = pre_\rho(e) \wedge post'_o(e) = post_\rho(e)$.

Figure 4.6 illustrates the notion of a complete occurrence of an oclet in a run. The first occurrence $\mathsf{o}_\mathsf{E}^\alpha$ of oclet o_E in ρ is complete whereas the second occurrence $\mathsf{o}_\mathsf{E}^\beta$ of o_E in ρ is not complete. The completeness condition (4.1) restricts only events of the contribution; a run may contain more pre- and post-conditions of an event than described in an oclet's history. This corresponds to the idea that a history describes a sufficient precondition for an occurrence of an oclet.

4.3.3. Semantics of oclets with history

We now have all formal tools to define the semantics of oclets with history. Each occurrence of an oclet o in a run ρ induces a prefix π of ρ that *ends with* o's history. The rest of ρ *begins with* o's contribution as sketched in Figure 4.7. We say that ρ *continues* π with o. The corresponding formal definitions are straight forward.

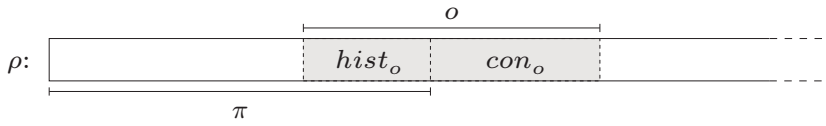


Figure 4.7. The run π ends with the history of oclet o, the run ρ continues π with o.

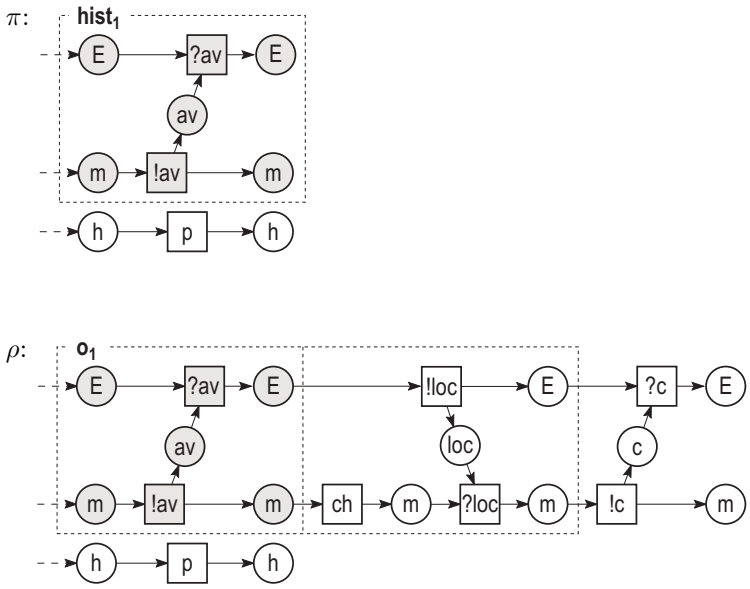


Figure 4.8. The run π ends with the history of oclet o_1 of Fig. 4.2; the run ρ continues π with o_1.

Definition 4.4 (Ends with and begins with). Let ρ be a distributed run, let π be a nonempty finite distributed run.

1. ρ *ends with* π, denoted $\rho \xrightarrow{\pi}]$, iff $\pi \subseteq \rho$ and $\max \pi \subseteq \max \rho$.
2. ρ *begins with* π, denoted $[\xrightarrow{\pi} \rho$, iff $\pi \subseteq \rho$ and $\min \pi \subseteq \min \rho$. ⌟

Definition 4.5 (Continue a run with an oclet). Let $o = (\pi_o, hist_o)$ be an oclet with history and let π and ρ be distributed runs. The run ρ *continues* π with o, written $\pi \xrightarrow{o} \rho$, iff $\pi \sqsubset \rho$ (π is a prefix of ρ) and $o \subseteq_{\mathbf{c}} \rho$ (o occurs complete in ρ) and $\pi \xrightarrow{hist_o}]$ (π ends with o's history). ⌟

For example, the run π in Figure 4.8 ends with the history hist_1 of oclet o_1 of Figure 4.2. In contrast, the second run ρ does not end with hist_1: hist_1 does occur in ρ but the maximal conditions of hist_1 have successors in ρ. The run ρ continues the run π with oclet o_1. We can interpret the difference between π and ρ also from a different angle: although π and ρ have maximal conditions with labels E and m, both runs differ in which histories occur at their respective ends.

The notions *ends with*, *begins with*, and *continues with* canonically extend to all occurrences of an oclet and its history. We say that ρ ends with (begins with) π *at location* α iff ρ ends with (begins with) π^α, written $\rho \xrightarrow{\pi^\alpha}]$ ($[\xrightarrow{\pi^\alpha} \rho$). Further, ρ continues π with o *at location* α iff ρ continues π with o^α; we write $\pi \xrightarrow{o^\alpha} \rho$ in this case.

Closed under continuations. The discussion on the semantics of scenarios in Section 3.5.2 identified *closed under continuations* (3.2) as a minimal notion for specifying the complete behavior of a distributed system with scenarios. There, we defined that a set R of distributed runs is closed under continuations with a scenario S iff

> for each prefix π of a run in R, if π ends with $hist_S$,
> then *there exists* a run ρ in R that continues π with S.

We directly formalize this sentence for oclet — but we have to add one technical detail. This sentence only implicitly refers to concrete occurrences of a scenario; the formal definition for oclets has to be explicit here. The formal definition relates each occurrence $hist_o^\beta$ of an oclet's history at the end of π to a respective occurrence of an entire oclet o^α at the same location $hist_o^\alpha = hist_o^\beta$. Thus, on a technical level, *closed under continuations* relates a set R of distributed runs to *all* occurrences $[o]$ of an oclet o. For this reason, we formally define the semantics of oclets on oclet *classes*.

Definition 4.6 (Semantics of oclets with history). Let $[o]$ be the class of an oclet $o = (\pi_o, hist_o)$ with history. A set R of distributed runs *satisfies* $[o]$, written $R \models [o]$, iff

$$\forall \pi \in Prefix(R)\ \forall hist_o^\beta\colon \\ \pi \xrightarrow{hist_o{}^\beta}] \Rightarrow \exists \rho \in R\ \exists o^\alpha : (hist_o^\alpha = hist_o^\beta \wedge \pi \xrightarrow{o^\alpha} \rho). \tag{4.2}$$

⌟

The semantics of oclets with history formally captures the notion of *closed under continuations* (3.2) from Section 3.5.2.

The two runs ρ_1 and ρ_2 in Figure 4.9 illustrate this definition. The set $\{\rho_1, \rho_2\}$ satisfies $\mathsf{o_1}$ whereas the singleton set $\{\rho_1\}$ does *not* satisfy $\mathsf{o_1}$. The run ρ_1 contains two occurrences of $\mathsf{hist_1}$; each occurrence induces a prefix of ρ_1 that ends with $\mathsf{hist_1}$. ρ_1 continues the first occurrence $\mathsf{hist_1^\alpha}$ with the entire oclet $\mathsf{o_1}$; ρ_1 *does not* continue the second occurrence $\mathsf{hist_1^\beta}$ with $\mathsf{o_1}$. This example also shows the importance of the technical condition $hist_o^\alpha = hist_o^\beta$ in (4.2). By this condition, we do not consider $\mathsf{o_1^\alpha}$ to continue $\mathsf{hist_1^\beta}$. To satisfy $\mathsf{o_1}$ we need the run ρ_2 which continues $\mathsf{hist_1^\beta}$ with $\mathsf{o_1^\beta}$.

4.3.4. Semantics of oclets with empty history

We introduced oclets with empty history (ε-oclets) to let a system designer specify that a system exhibits *at least one run*, and to provide a starting point for scenario-based behavior. In the following, we indicate an ε-oclet by index ε. Intuitively, a set R of distributed runs *satisfies* an ε-oclet o_ε if R contains a run where o_ε occurs as a *prefix*.

We adapt the prefix relation $\sqsubseteq$ from distributed runs: an ε-oclet $o_\varepsilon = (\pi_o, \varepsilon)$ *occurs as a prefix* of a distributed run ρ, written $o_\varepsilon \sqsubseteq \rho$, iff $\pi_o \sqsubseteq \rho$. Correspondingly, $o_\varepsilon^\alpha \sqsubseteq \rho$ expresses $\pi_o^\alpha \sqsubseteq \rho$, for each occurrence $o_\varepsilon^\alpha = (\pi_o^\alpha, \varepsilon) \in [o_\varepsilon]$.

In correspondence to the semantics of oclets with history, the semantics of ε-oclets is defined on oclet classes, because we only need some occurrence $o^\alpha \in [o]$ to be a prefix of a run.

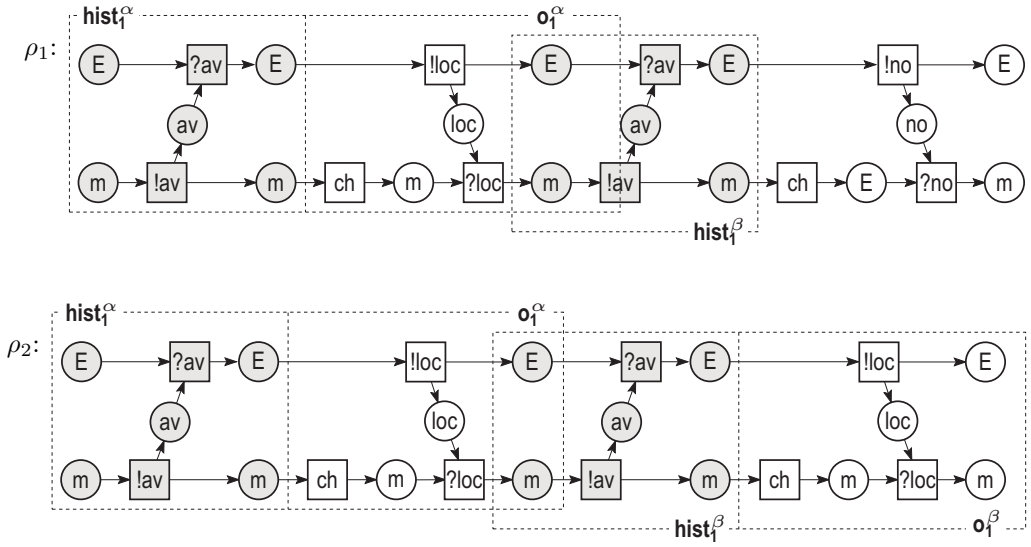


Figure 4.9. The singleton set $\{\rho_1\}$ is not closed under continuations with oclet o_1 of Fig. 4.2; the set $\{\rho_1, \rho_2\}$ is closed under continuations with o_1.

Definition 4.7 (Semantics of ε-oclets). Let $[o_\varepsilon]$ be the oclet class of an ε-oclet o_ε. A set R of distributed runs *satisfies* $[o_\varepsilon]$, written $R \models [o_\varepsilon]$, iff there exists a run $\rho \in R$ in which o_ε occurs as a prefix, i.e., $\exists o_\varepsilon^\alpha : o_\varepsilon^\alpha \sqsubseteq \rho$. ⌟

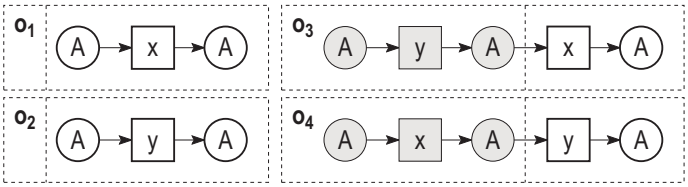
Figure 4.10 illustrates this definition by a technical example. Any set of runs that contains the run ρ_1 satisfies o_1; any set of runs that contains ρ_2 satisfies o_2. The singleton set $\{\rho_1\}$ satisfies o_1 and does not satisfy o_2 because o_2 does not occur as a prefix of ρ_2. The symmetric argument holds for $\{\rho_2\}$. Thus, only the set $\{\rho_1, \rho_2\}$ satisfies o_1 and o_2. Put differently, two different ε-oclets describe alternative initial runs.

4.3.5. Semantics of oclet specifications

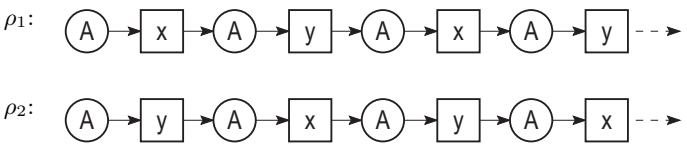
The semantics of an oclet specification O follows canonically from the preceding definitions. Intuitively, O is a set of oclets and a set R of runs satisfies O if and only if R satisfies each $o \in O$.

Yet, we have seen that we need to distinguish different occurrences $o^\alpha, o^\beta, \ldots$ of an oclet o in a run. For this reason, we read each specific oclet $o, o^\alpha, o^\beta, \ldots$ as a *semantic notion* which describes specific events and conditions. Their oclet class $[o]$ abstracts from these events and conditions and merely describes a *behavioral pattern*. Thus, an oclet class $[o]$ is a *syntactic description* of a scenario and a specification is a set of oclet *classes*.

Definition 4.8 (Oclet specification). An *oclet specification* is a finite set $O = \{[o_1], [o_2], \ldots, [o_n]\}$ of oclet classes of finite oclets $o_1, o_2, \ldots, o_n$. ⌟



(a) oclet specification $\mathsf{O}_{\mathsf{init}} = \{\mathsf{o}_1, \ldots, \mathsf{o}_4\}$



(b) set $\mathsf{R} = \{\rho_1, \rho_2\}$ of runs

Figure 4.10. The set R of runs satisfies o_1 and o_2.

Definition 4.9 (Semantics of an oclet specification). Let O be an oclet specification. R *satisfies* O iff R satisfies each $[o] \in O$. The *semantics* of O is the set $\mathcal{R}(O) = \{R \mid R \text{ a set of distributed runs that satisfies } O\}$. ⌟

For example, the set R of runs in Figure 4.10 satisfies the oclet specification $\{\mathsf{o}_1, \ldots, \mathsf{o}_4\}$. Though, there are many other sets of runs that satisfy $\{\mathsf{o}_1, \ldots, \mathsf{o}_4\}$. Definition 4.9 declares a relation between sets of distributed runs and oclet specifications. Any set R of runs that is *closed under continuations*[2] with all oclets in O with history and contains each ε-oclet in O as a prefix. Beyond these requirements, R may describe any other behavior. Simplifying the informal language, we say that R is a *satisfying behavior for* O. Further, we abbreviate $\mathcal{R}(\{[o]\})$ by $\mathcal{R}([o])$ for singleton oclet specifications.

A system S *implements* a scenario-based specification if the runs R exhibited by S satisfy the specification. In the model of oclets, a Petri net takes the role of S.

Definition 4.10 (Implementation of an oclet specification). Let Σ be a labeled Petri net system, and let O be an oclet specification. The system Σ *implements* O iff $R(\Sigma) \in \mathcal{R}(O)$. ⌟

Definition 4.10 concludes the formal model of oclets.

4.4. Basic Properties of Oclets

The preceding sections formalized the kernel of scenarios identified in Chapter 3 in the formal model of oclets. We may now investigate properties of this kernel using oclets. Most importantly, we show that

[2] We will eventually refer to the behavioral property "closed under continuations with o" (3.2) instead of the formal "satisfies o" in informal text to illustrate an argument about a set of runs.

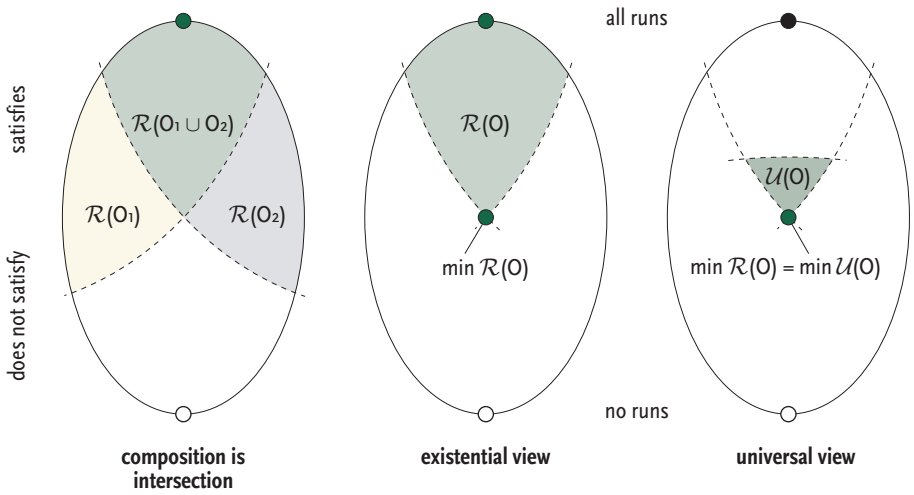


Figure 4.11. Illustration of the basic properties of oclets.

every oclet specification O has a unique minimal set $\min \mathcal{R}(O)$ *of runs that satisfies O.*

We also investigate how oclets express *safety* and *liveness* properties and how these are composed. We begin by showing that oclets describe complex behavioral properties as an *intersection* of basic properties. Specifically, each single oclet describes a liveness properties which contrasts standard specification and proof techniques, cf. [1]. Then, we show that the semantics $\mathcal{R}(O)$ of an oclet specification O formalizes the *existential* view on scenarios in which a specification describes sample behaviors. The contrasting view is the *universal* view in which O describes all system behaviors as discussed in Section 3.3. We show that we obtain by the universal- from the existential via by intersection with a *canonical* safety property which we define in Section 4.4.5. Figure 4.11 sketches these properties.

The existence of a minimal satisfying behavior $\min \mathcal{R}(O)$ and the notion of a universal view form the basis for synthesizing an implementation from an oclet specification.

4.4.1. Composition is intersection

A specification is a set of oclets. Thus we can compose two specifications O_1 and O_2 by joining them to $O_1 \cup O_2$. The following corollary is an immediate consequence of Definition 4.9. It tells us that composing oclet specifications is also a monotonous operation on system behavior: a set R of distributed runs satisfies the composition $O_1 \cup O_2$ iff R satisfies both O_1 and O_2. Thus, the semantics of oclets is *compositional* by *intersection*. Figure 4.11 (left) illustrates this property.

Corollary 4.11: *Let O_1 and O_2 be two oclet specifications. Then the following property holds:* $\mathcal{R}(O_1 \cup O_2) = \mathcal{R}(O_1) \cap \mathcal{R}(O_2)$. ⋆

Proof. Let $R \in \mathcal{R}(O_1 \cup O_2)$. By Def. 4.9 holds: $R \models [o]$, for all $[o] \in O_1 \cup O_2$. This is equivalent to: $R \models [o]$, for all $[o] \in O_1$, and $R \models [o]$, for all $[o] \in O_2$. This is equivalent to: $R \in \mathcal{R}(O_1)$ and $R \in \mathcal{R}(O_2)$ (by Def. 4.9), which is equivalent to $R \in \mathcal{R}(O_1) \cap \mathcal{R}(O_2)$. □

According to this corollary, a system designer can build up a specification oclet by oclet. Each new oclet introduces a new requirement (e.g., to close behavior under continuations with this oclet) which rules out more sets of distributed runs (those which are not closed under continuations with the new oclet).

4.4.2. Oclets are consistent

One of the basic questions for a specification is whether it can be satisfied at all or whether its requirements are contradictory. A specification is *consistent* iff there is a set of runs that satisfies the specification; otherwise the specification is *inconsistent*.

In the model of oclets, the question can be formulated as follows: given an oclet specification O, is $\mathcal{R}(O) \neq \emptyset$? If this holds true, then O is consistent. The following lemma tells that *every* oclet specification is consistent.

Lemma 4.12: *Let O be an oclet specification. Then $\mathcal{R}(O) \neq \emptyset$.* ⋆

Proof. We show that the set $\mathfrak{R}$ of all distributed runs is contained in $\mathcal{R}(O)$. Let $[o]$ be a class of some oclet $o = (\pi_o, hist_o)$. We first show that $\mathfrak{R} \in \mathcal{R}([o])$ and then generalize to $\mathcal{R}(O)$. There are two cases.

$[hist_o = \varepsilon]$ $\mathfrak{R} \in \mathcal{R}([o])$ holds only if there exists a run $\rho \in \mathfrak{R}$ and an occurrence $o^\alpha \in [o]$ with $o^\alpha \sqsubseteq \rho$ (by Def. 4.9 and Def. 4.7). This is trivially true by $\mathfrak{R}$ being the set of all distributed runs.

$[hist_o \neq \varepsilon]$ Let $\pi \in \mathit{Prefix}(\mathfrak{R})$ with $\pi \xrightarrow{hist_o{}^\alpha}]$ for some occurrence $hist_o^\alpha \in [hist_o]$. Then there exists an occurrence $o^\alpha \in [o]$ and run $\rho \in \mathfrak{R}$ with $\pi \xrightarrow{o^\alpha} \rho$, by $\mathfrak{R}$ being the set of all distributed runs. Thus $\mathfrak{R} \models [o]$ by Def. 4.6 which is equivalent to $\mathfrak{R} \in \mathcal{R}([o])$.

Thus for each $[o] \in O$ holds $\mathfrak{R} \in \mathcal{R}([o])$, and Cor. 4.11 implies $\mathfrak{R} \in \mathcal{R}(O)$. □

This lemma confirms that each oclet desctibes *liveness* property: "something good eventually happens" [4]. We discuss this observation in more detail in Section 4.4.6. For the moment, Lemma 4.12 indicates that oclets serves the central purpose of scenarios: to specify *expected* behavior of distributed systems as discussed in Chapter 3.

Non-empty system behavior

The proof of Lemma 4.12 shows that an oclet specification O is trivially satisfied by the set $\mathfrak{R}$ of all distributed runs. How about the other trivial solution, the *empty* set of runs? According to the following lemma, only an oclet specification *without* ε-oclet is trivially satisfied by the empty behavior.

Lemma 4.13: *Let O be an oclet specification. There exists $[o] \in O$ with an empty history $hist_o = \varepsilon$ iff $\emptyset \notin \mathcal{R}(O)$.* ⋆

Proof. We first show the proposition for single oclet classes and then apply Corollary 4.11 to lift it to oclet specifications. Let $[o]$ be the class of an oclet $o = (\pi_o, hist_o)$. We distinguish two cases.

$[hist_o \neq \varepsilon]$. We have to show $\emptyset \in \mathcal{R}([o])$. By Def. 4.9 and Def. 4.6, we have to show that (4.2) holds on the empty set $R = \emptyset$. Formula (4.2) trivially evaluates to true because $R = \emptyset$ implies $Prefix(R) = \emptyset$. Thus $\emptyset \in \mathcal{R}([o])$.

$[hist_o = \varepsilon]$. From Def. 4.9 and Def. 4.7 follows that $R \in \mathcal{R}([o])$ iff there exists an oclet $o^\alpha \in [o]$ and a run $\rho \in R$ with $o^\alpha \sqsubseteq \rho$. Thus $\emptyset \notin \mathcal{R}([o])$.

Now, let O be an oclet specification. By Corollary 4.11 holds: there exists $[o] \in O$ with $hist_o = \varepsilon$ iff $\emptyset \notin \mathcal{R}(O)$. □

4.4.3. Canonical satisfying behavior

A declarative semantics, like the semantics for oclets, defines a *relation* between system behavior and specifications. Thus, there are many sets of distributed runs that satisfy a specification. Immediately, the question arises whether an oclet specification O has a *canonical* system behavior $R(O)$ that satisfies O. A canonical behavior $R(O)$ is significant for synthesizing a system model from O. The synthesized system model should preferably exhibit the canonical behavior $R(O)$ and as few additional behaviors as possible.

We know from Lemma 4.12 that the set $\mathfrak{R}$ of all runs trivially satisfies O. In this section, we show that every oclet specification O has a *unique minimal* set $\min \mathcal{R}(O)$ of runs that satisfies O.

Minimal satisfying behavior

A system S_1 exhibits less behavior than a system S_2 if S_1 exhibits less distributed runs than S_1. In that respect, a set R_1 of distributed runs describes *less behavior* than a set R_2 of distributed runs iff R_1 contains less (prefixes of) runs than R_2, i.e., $Prefix(R_A) \subset Prefix(R_B)$. Thus, we can semantically characterize the unique least set of runs that satisfies an oclet specification as follows.

Definition 4.14 (Minimal satisfying behavior). Let R_1 and R_2 be two sets of distributed runs; R_1 *denotes less behavior* than R_2 iff $Prefix(R_1) \subset Prefix(R_2)$.

The *unique minimal* behavior that satisfies an oclet specification O is $\min \mathcal{R}(O) := R_{\min}$ with $R_{\min} \in \mathcal{R}(O)$ s.t. for all $R' \in \mathcal{R}(O) : Prefix(R') \not\subset Prefix(R_{\min})$. ⌟

Theorem 4.15 (Minimal behavior of an oclet specification). *Let O be an oclet specification. Then $\min \mathcal{R}(O)$ is well-defined (up to isomorphism and prefixes).* ⋆

We prove Theorem 4.15 by the help of the following Lemma.

Lemma 4.16: *Let O be an oclet specification and let $R_1, R_2 \in \mathcal{R}(O)$. Then $R_1 \cap R_2 \models [o_\varepsilon]$, for each ε-oclet $[o_\varepsilon] \in O$.* ⋆

Proof. From $R_1 \in \mathcal{R}(O)$ follows: $\exists o_\varepsilon^\alpha = (\pi_o^\alpha, \varepsilon) : \pi_o^\alpha \in \mathit{Prefix}(R_1)$, by Def. 4.7. Correspondingly, $\exists o_\varepsilon^\beta = (\pi_o^\beta, \varepsilon) : \pi_o^\beta \in \mathit{Prefix}(R_2)$. We distinguish R_1 and R_2 only up to isomorphism. So, w.l.o.g. holds $\pi_o^\alpha = \pi_o^\beta \in \mathit{Prefix}(R_1 \cap R_2)$, which implies $R_1 \cap R_2 \models [o_\varepsilon]$ by Def. 4.7. □

Proof (of Theorem 4.15). $\min \mathcal{R}(O)$ exists because each oclet specification has at least one satisfying behavior by Lemma 4.12.

To show that $\min \mathcal{R}(O)$ is unique, we have to show the following: For any two sets $R_1, R_2 \in \mathcal{R}(O)$, if $\forall R' \in \mathcal{R}(O) : \mathit{Prefix}(R') \not\subset \mathit{Prefix}(R_i), i = 1, 2$, then $R_1 = R_2$ (up to isomorphism and up to prefixes, i.e., we show $\mathit{Prefix}(R_1) = \mathit{Prefix}(R_2)$).

If O contains no ε-oclet, then $R_1 = R_2 = \emptyset$ by Lemma 4.13, and the proposition holds. For the case that O contains an ε-oclet, we prove the proposition by contradiction. Assume $R_1 \neq R_2$; we distinguish three cases:

$[R_1 \subset R_2]$ By assumption $R_1 \in \mathcal{R}(O)$. But this contradicts the assumption that there is no $R' \in \mathcal{R}(O)$ with $R' \subset R_2$.

$[R_1 \cap R_2 = \emptyset]$ From Lem. 4.16 follows: $R_1 \cap R_2 \models [o_\varepsilon]$, for each ε-oclet $[o_\varepsilon] \in O$. Thus, Lem. 4.13 implies $R_1 \cap R_2 \neq \emptyset$. Contradiction.

$[R_1 \cap R_2 \neq \emptyset$, but $R_1 \not\subset R_2]$ Let $R_{12} = R_1 \cap R_2$ and let $\rho^* \in R_2 \setminus R_1$ be a run missing in R_1. We consider the set $R_2^- := R_2 \setminus \{\rho_2 \in R_2 \mid \rho^* \sqsubseteq \rho_2\}$ of runs where ρ^* and all its continuations are removed from R_2; $R_{12} \subseteq R_2^-$ by construction. There are two cases:

- $R_2^- \in \mathcal{R}(O)$. Additionally, $R_2^- \subset R_2$ holds by construction. But $R_2^- \in \mathcal{R}(O)$ and $R_2^- \subset R_2$ together contradict the lemma's assumption.
- $R_2^- \notin \mathcal{R}(O)$. The violation of the semantics of O cannot come from an ε-oclet because R_{12} satisfies each ε-oclet of O, by Lem. 4.16. Thus, there exists $[o] \in O$ with history and a run $\pi \in \mathit{Prefix}(R_2^-)$ s.t. $\pi \xrightarrow{o^\alpha} \rho^*$ and $\rho^* \notin R_2^-$, for some occurrence $o^\alpha \in [o]$.

  Claim: $\pi \in \mathit{Prefix}(R_{12})$. Firstly, $\pi \sqsubseteq \rho^*$ because $\pi \xrightarrow{o^\alpha} \rho^*$. Thus, if $\pi \notin \mathit{Prefix}(R_{12})$ and $\pi \in R_2^-$, then $\pi \notin R_1$. So, we could remove π from R_2 instead of ρ^* and obtain a smaller R_2^-. Iterating this procedure eventually leads to a prefix $\pi \in \mathit{Prefix}(R_{12}) \subseteq \mathit{Prefix}(R_1) \subseteq \mathit{Prefix}(R_1)$.

  By construction holds: $\pi \in \mathit{Prefix}(R_1)$, $\pi \xrightarrow{o^\alpha} \rho^*$, and $\rho^* \notin R_1$. Thus, R_1 does not satisfy $[o]$, by Def. 4.6. Hence, $R_1 \notin \mathcal{R}(O)$ which contradicts the lemma's assumption.

We conclude $R_1 = R_2$. Hence, $\min \mathcal{R}(O)$ is unique and well-defined. □

Theorem 4.15 primarily holds because of the semantics of ε-oclets (Def. 4.7). If we dropped this assumption, then there were oclet specifications O which had several minimal satisfying behaviors that are incomparable. Thus, Theorem 4.15 supports our design decision from Section 3.7 to interpret oclets with empty history as initial runs.

4.4.4. Existential view vs. universal view

Having established some basic formal properties of the semantics of oclets, we now consider which behavioral properties may be described by an oclet specification. Following Section 3.3, a system designer can read an oclet specification in two ways. (1) A specification describes *sample behaviors* of the system; this view is called the *existential* view on scenarios. (2) The contrasting view is the *universal* view in which a specification is assumed to describe *all behaviors* of the system [25].

The semantics of oclets (Def. 4.9) correspond to the existential view, as follows. Consider a set R of runs that satisfies a composition $O_1 \cup O_2$ of two oclet specifications O_1 and O_2. According to Corollary 4.11, R satisfies all oclets in O_1 and in O_2. Thus, R also satisfies only O_1. From the perspective of O_1, all runs in R that continue with oclets in $O_2 \setminus O_1$ are not necessary to satisfy O_1. So, R satisfies O_1 although it contains more behaviors than described by O_1. If R is the behavior of some system S, then O_1 describes only sample behaviors of S but not *all* behaviors of S.

The existential view follows from an *open world assumption*: a set of runs that satisfies an oclet specification O can have additional runs beyond those described by O. Allowing additional runs is necessary for composing two specifications to $O = O_1 \cup O_2$ as described by Corollary 4.11.

In contrast, a *closed world assumption* implies that a set R of runs satisfies an oclet specification O iff R contains only those behaviors that are described by O and no additional behavior. Under a closed world assumption, if R satisfies O_1, then R has only runs that continue with oclets in O_1 and no run that continues with an oclet in $O_2 \setminus O_1$. Consequently, there exists no set of runs that satisfies O_1 and O_2 if O_1 and O_2 are different.

The closed world assumption corresponds to the universal view where a specification describes all possible system behaviors — the system must not exhibit behaviors that are not described in the specification.

4.4.5. Formal Definition of the Universal View

This section formalizes the *universal view* on scenarios in terms of oclets. In the universal view, a specification is assumed to describe *all* system behaviors. This view is important for synthesizing an implementation from a specification. Damm and Harel gave the following informal definition of the universal view in terms of runs and scenarios:

> once a run of the system has reached a point where the [scenario] applies, designers expect that from now on, regardless of possible ways the system may continue its run, the behavior specified in the [scenario] should always be exhibited. [25, p.50]

Oclets propose a different interpretation of a scenario for which we find the following definition of the universal view:

> for every run of the system and every point of this run, the designers expect the system to continue its run *only* with oclets of which *the history has occurred*.

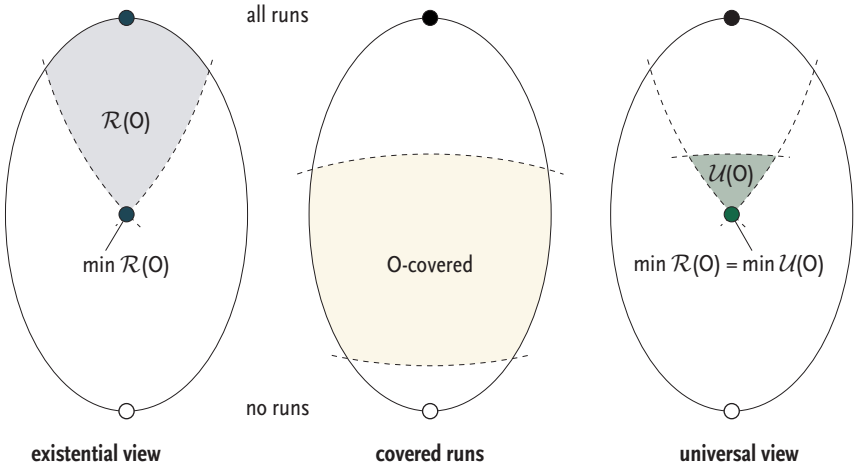


Figure 4.12. The universal view is the intersection of the existential view with oclet-covered runs.

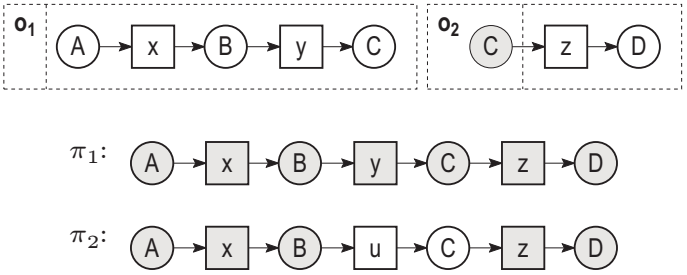


Figure 4.13. Oclet-covered behaviors: the set $\{\pi_1\}$ is covered by $\{o_1, o_2\}$, whereas the set $\{\pi_1, \pi_2\}$ is *not* covered by $\{o_1, o_2\}$.

Formally, the universal view $\mathcal{U}(O)$ of an oclet specification O constrains the semantics $\mathcal{R}(O)$ of O: each run *only* continues according to oclets of which the history has occurred — we call these runs *oclet-covered*. Specifically, an ε-oclet only occurs after its empty history occurred: at the beginning of a run.

Figure 4.13 illustrates how oclet-covered behavior restricts the semantics of oclets. The depicted set $\{\pi_1, \pi_2\}$ of runs satisfies the oclet specification $\{o_1, o_2\}$ but is *not* covered by $\{o_1, o_2\}$. Neither o_1 nor o_2 describes an occurrence of u and its post-condition C in π_2. The subset $\{\pi_1\}$ satisfies $\{o_1, o_2\}$ *and* is covered by $\{o_1, o_2\}$.

The intersection of the existential view $\mathcal{R}(O)$ with oclet-covered and oclet-initial runs defines the universal view $\mathcal{U}(O)$ on scenarios as illustrated in Figure 4.12.

The technical definitions

If an oclet specification O is meant to describe *all* system behavior, then each event of a satisfying behavior occurs due to an oclet $o \in O$.

This notion corresponds to the *axiomatic characterization* of distributed runs of a Petri net system where each event occurs due to a transition of the system. Definition 2.16 stated that a distributed run ρ is a run of a Petri net system (N, m_0) iff (1) the minimal conditions of ρ represent the initial marking m_0, and (2) each event e of ρ represents an occurrence of a transition $t = \ell(e)$.

The technical definition for oclet-covered behavior is slightly different because an oclet describes a larger piece of behavior. (1) A node x at the beginning of a run ρ occurs because an ε-oclet o which contains x and occurs as a prefix of ρ. (2) Each node x "in the middle" of ρ is part of an occurrence o^α in ρ of some oclet o with history. The formal definition involves a small technicality: the same node x may occur in different runs, however it is sufficient if x is covered in one run.

Definition 4.17 (Covered by oclets). Let O be an oclet specification. Let R be a set distributed runs.

Let $\rho \in R$. A node $x \in X_\rho$ is *covered by oclet class* $[o] \in O$ *in* ρ iff there exists a contributed node $y \in X_{con_o}$ and an occurrence $o^\alpha \in [o]$ at location $\alpha : o \to o^\alpha$ s.t.

- if $hist_o \neq \varepsilon$, then $o^\alpha \subseteq_{\mathbf{c}} \rho$ and $\alpha(y) = x$ (o occurs complete in ρ at location α s.t. node y of o contributes node x), and
- if $hist_o = \varepsilon$, then $o^\alpha \sqsubseteq \rho$ and $\alpha(y) = x$ (ε-oclet o occurs as a prefix of ρ at location α s.t. node y of o contributes node x).

Node $x \in X_\rho$ is *O-covered in* ρ iff x is covered by $[o]$ in ρ, for some $[o] \in O$.

Let $X_R := \bigcup_{\rho \in R} X_\rho$ be all nodes (events and conditions) of R; a node $x \in X_R$ may occur in several runs of R. The set R of runs is *O-covered* iff for each $x \in X_R$ exists a run ρ s.t. x is O-covered in ρ. The *O-covered behaviors* are $\mathcal{R}^{\mathrm{cov}}(O) = \{R \mid R \text{ a set of distributed runs that is } O\text{-covered}\}$. ⌟

The example in Figure 4.14 illustrates this definition including its technical aspects. All nodes of π_1 are covered by oclet $\mathsf{o_1}$ or by oclet $\mathsf{o_2}$. Nodes $\mathsf{A}, \mathsf{x}, \mathsf{B}$ of π_2 and of π_3 are covered by $\mathsf{o_1}$ because they are also contained in π_1. Nodes z and D of π_2 are covered by $\mathsf{o_2}$. Nodes v and E are covered by $\mathsf{o_3}$. Only nodes u and C of π_2 are not covered by any of the oclets. Thus, $\{\pi_1, \pi_3\}$ is $\{\mathsf{o_1}, \mathsf{o_2}, \mathsf{o_3}\}$-covered.

The universal view for oclets is the intersection of the existential view (i.e., the semantics of oclets) with oclet-covered behaviors.

Definition 4.18 (Universal view). Let O be an oclet specification. The *semantics of O in the universal view* is $\mathcal{U}(O) := \mathcal{R}(O) \cap \mathcal{R}^{\mathrm{cov}}(O)$. ⌟

It is easy to see that the minimal set of runs $\min \mathcal{R}(O)$ that satisfies an oclet specification O is also minimal in the universal view: $\min \mathcal{R}(O) = \min \mathcal{U}(O)$. By construction holds $\min \mathcal{U}(O) \in \mathcal{R}(O)$. The converse $\min \mathcal{R}(O) \in \mathcal{U}(O)$ holds as well. If this was not the case, then $\min \mathcal{R}(O)$ contained a run ρ and a node x that was not covered by some oclet in O. Thus, ρ could be shortened by removing x and all its predecessors. The resulting set of runs would still satisfy O which

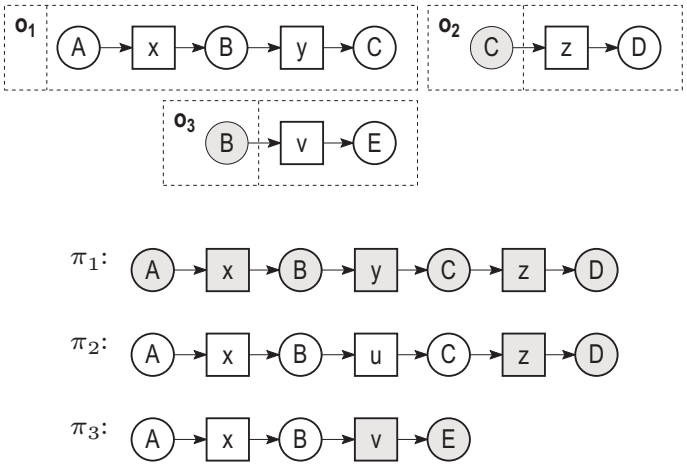


Figure 4.14. Oclet-covered behaviors: the set $\{\pi_1, \pi_3\}$ of runs is $\{o_1, o_2, o_3\}$-covered, the set $\{\pi_1, \pi_2, \pi_3\}$ is not $\{o_1, o_2, o_3\}$-covered.

violates the minimality of $\min \mathcal{R}(O)$. Thus, the uniqueness of $\min \mathcal{R}(O)$ implies $\min \mathcal{R}(O) = \min \mathcal{U}(O)$.

4.4.6. Oclets as a specification technique

This section classifies oclets as a *specification technique*. We relate the results of the preceding sections on composition, consistency, and existential and universal view to *safety* and *liveness* properties and their composition.

Alpern and Schneider [4] have shown that every behavioral property P is an intersection $P = S \cap L$ of a safety property S and a liveness property L. A safety property S denotes that "something bad never happens" whereas a liveness property L denotes that "something good eventually happens". Together with a result of Plotkin follows that S and L themselves are composed from smaller safety properties $S = S_1 \cap \ldots \cap S_n$ and liveness properties $L = L_1 \cap \ldots \cap L_m$ by *intersection*. Abadi and Lamport developed this idea systematically in temporal logics [1, 76].

By their declarative semantics, an oclet specification O describes a *behavioral property* $\mathcal{R}(O)$ in the following sense: any system Σ that exhibits behaviors $R(\Sigma)$ contained in $\mathcal{R}(O)$, i.e., $R(\Sigma) \in \mathcal{R}(O)$, has the behavioral property of exhibiting initial runs and continuations as specified in O. In other words, $\mathcal{R}(O)$ denotes all systems that have this property.

In correspondence to Alpern and Schneider, oclets express complex behavioral properties by interaction (Cor. 4.11). More precisely, an oclet specification $O = \{[o_1], \ldots, [o_m]\}$ describes the behavioral property $\mathcal{R}(O) = \mathcal{R}([o_1]) \cap \ldots \cap \mathcal{R}([o_m])$. This composition principle is based on an open world assumption: any behavior that is not specified is *allowed* to occur as discussed in Section 4.4.4. Lamport

developed this composition principle and the open world assumption for his Temporal Logic of Actions [76], which is based on sequential runs.

From Lemma 4.12 follows that every oclet class $[o]$ describes a liveness property $\mathcal{R}([o])$. This liveness property expresses that $[o]$ will occur — at the beginning of a run or when its history occurred.

In the existential view, an oclet specification O describes the behavioral property $\mathcal{R}(O) = \mathcal{R}([o_1]) \cap \ldots \cap \mathcal{R}([o_m])$ which is a liveness property. The universal view adds the safety property $\mathcal{R}^{\text{cov}}(O)$ defining the behavioral property $\mathcal{U}(O) = \mathcal{R}(O) \cap \mathcal{R}^{\text{cov}}(O)$.

To conclude, the model of oclets allows to specify system behavior as an intersection of liveness and safety properties. However, oclets specify system behavior *primarily* with liveness properties; the safety property follows canonically. In contrast, classical techniques specify system behavior primarily with safety properties [1]. To let an oclet specification O describe an interesting behavioral property, O has to be considered in its universal view. We return to this aspect in Chapter 5.

The liveness properties described by oclets indicate a feasible expressive power expressive: oclets describe expected system behaviors. In other words, the complexity of an oclet specification stems from descriptions of how different components interact with each other — and nothing else. This property shall help us solving the synthesis problem at its very core: to transform the scenario-based view where "one story involves many components" into a component-based view where "one component involves many stories". We prove in Chapter 5 that oclet specifications are expressive enough to specify every Petri net.

4.5. Extending Oclets

Up to now, we introduced the formal model of oclets and studied its basic properties. This section presents two basic extension to oclets. These extensions will not be studied in the following chapters. We primarily show how oclets can be extended in non-trivial ways to a more expressive formal model for scenarios.

4.5.1. Oclets and progress

This first extension to oclets allows to distinguish *intermediate* runs from *accepting* runs of a specified system. We first consider how Petri nets address the problem and propose an extension to oclets afterwards.

Accepting behavior

Let us first consider the set of distributed runs of a Petri net system Σ as defined in Section 2.4. For each run $\rho \in R(\Sigma)$, also each prefix $\pi \sqsubseteq \rho$ is a run of Σ. The set $R(\Sigma)$ of runs of a Petri net system is prefix-closed by Corollary 2.19. Thus, each distributed run of a Petri net system simply describes a partial order of local steps that the system can exhibit, or, in other words, a *possible* system behavior.

There is another interpretation of a set R of distributed runs: each run $\rho \in R$ describes an *accepting* behavior of the system. If ρ is finite, then the system *terminates* at the end of ρ and, in principle, may remain there forever. Any prefix π of ρ that is not in a run in R is *not* accepting, i.e., the system may not terminate at the end of π and has to *progress* until acceptance. If ρ is infinite, then the system does not terminate. In this interpretation, R is not prefix-closed in general.

Figure 4.15. The set $\mathsf{R}_{\mathsf{acc}} = \{\rho_1, \rho_2\}$ of runs is not prefix-closed.

For example, we could read the set $\mathsf{R}_{\mathsf{acc}}$ of distributed runs of Figure 4.15 as follows: the system may terminate after x occurred and remain there forever. If z occurs, then the system has to progress with y before it may terminate again. So, how could a system designer specify the behavior $\mathsf{R}_{\mathsf{acc}}$?

Oclets does not distinguish accepting runs from non-accepting runs. Technically, the semantics of oclets does not distinguish whether a set R of runs contains all prefixes of R, or only some specific prefixes of R, or only the maximal runs of R. According to the following lemma, we can take any set R of runs and add to or remove from R any prefix of a run in R; the resulting set of runs is equivalent wrt. the semantics of oclets.

Lemma 4.19: *Let O be an oclet specification. Let R be a set of distributed runs. Let $\rho \in R$ and $\pi \sqsubseteq \rho$ with $\pi \neq \rho$. Then $R \in \mathcal{R}(O)$ iff $(R \cup \{\pi\}) \in \mathcal{R}(O)$ iff $(R \setminus \{\pi\}) \in \mathcal{R}(O)$.* ⋆

Proof. By compositionality of the semantics of oclets (Cor. 4.11), it is sufficient to prove the proposition for $O = \{\ [o]\ \}$.

[$hist_o = \varepsilon$] If $R \notin \mathcal{R}([o])$, then there exists no run $\rho \in R$ an no oclet $o^\alpha \in [o]$ with $o^\alpha \sqsubseteq \rho$. Thus, there exists no $\pi \subseteq_{\mathbf{c}} \rho$ with $o^\alpha \sqsubseteq \pi$. Thus, $(R \cup \{\pi\}) \notin \mathcal{R}([o])$ and $(R \setminus \{\pi\}) \notin \mathcal{R}([o])$.

If $R \in \mathcal{R}([o])$, then there exist a run $\rho \in R$ and an oclet $o^\alpha \in [o]$ s.t. $o^\alpha \sqsubseteq \rho$; let π be as assumed. Firstly, $\rho \in R \setminus \{\pi\}$, hence $R \setminus \{\pi\} \models [o]$ by Def. 4.7, which implies $R \setminus \{\pi\} \in \mathcal{R}([o])$ by Def. 4.9. Secondly, $R \cup \{\pi\} \models [o]$ and $R \cup \{\pi\} \in \mathcal{R}([o])$, by the same arguments.

[$hist_o \neq \varepsilon$] Let $\rho \in R$ be a run an $\pi \sqsubseteq \rho$ be a prefix s.t. $\pi \neq \rho$. Firstly, from $\pi \sqsubseteq \rho$ follows $\mathit{Prefix}(R) = \mathit{Prefix}(R \cup \{\pi\}) = \mathit{Prefix}(R \setminus \{\pi\})$. Secondly, for any $\pi_2 \in \mathit{Prefix}(R)$, if $\pi_2 \xrightarrow{o^\alpha} \pi$, then $\pi_2 \xrightarrow{o^\alpha} \rho$, for all oclets $o^\alpha \in [o]$, by $\pi \sqsubseteq \rho$. Thus, $R \models [o]$ iff $(R \cup \{\pi\}) \models [o]$ iff $(R \setminus \{\pi\}) \models [o]$ by Def. 4.6. Thus $R \in \mathcal{R}([o])$ iff $(R \cup \{\pi\}) \in \mathcal{R}([o])$ iff $(R \setminus \{\pi\}) \in \mathcal{R}([o])$ by Def. 4.9. □

From this lemma, we conclude that *oclets do not describe accepting behavior*. This property of oclets has the following two implications. Firstly, if we are *not interested* in distinguishing accepting from progressing behavior, then we

may consider *canonical* sets of runs as satisfying behavior. For instance, only prefix-closed sets or only sets of maximal runs.

Secondly, because oclets do not describe accepting bebavior, their semantics are compatible with any kind of additional notion that does. Thus, a system designer can freely choose how to distinguish accepting from progressing behavior. The subsequent section presents the notion of *progress* as an example for distinguishing accepting runs from progressing runs.

Progress

The notion of *progress* in Petri nets distinguishes runs where the system may terminate from runs where the system has to progress. The principal idea of progress is the following: as long as some action of the system is enabled, the action *will occur*, i.e., no action remains enabled forever. By requiring progress only for selected actions, a system may stop only when non-progressing actions are enabled. We will generalize progress from Petri nets to oclets to show one possibility for distinguishing accepting from progressing runs.

Progress is a requirement on how system behavior advances wrt. single actions. Intuitively, a system behavior *respects progress* of an action a iff

> whenever the preconditions of a all hold, then either a will eventually occur or some competing action b will occur, affecting some of a's preconditions. [103, p.582] (4.3)

In other terms: action a does not remain enabled forever in this run. The formal definition for Petri nets reads as follows.

Definition 4.20 (Progress). Let $\Sigma = (N, m_0)$ be a Petri net system, let π be a distributed run of Σ. The run π *neglects progress* of transition $t \in T_N$ iff the marking $m_{\max \pi}$, denoted by the cut $\max \pi$, enables t.

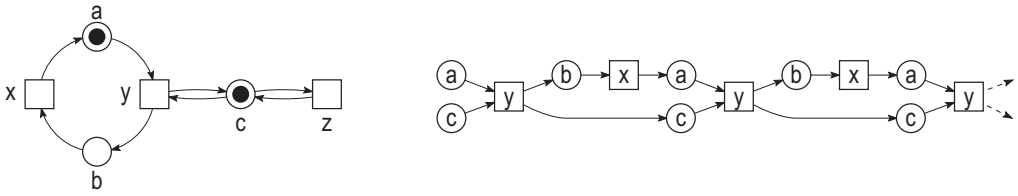
Let $T^p \subseteq T_N$ be a set of transitions of N. The run π *respects progress* of T^p iff π does not neglect progress for any $t \in T^p$. ⌟

The notion of progress is closely related to the notion of *weak fairness* [78] (originally called justice). Yet, there is a subtle difference between both notions which we illustrate in the following. Intuitively, a run *neglects weak fairness* of an action a if a is enabled from some point in the run but never occurs.

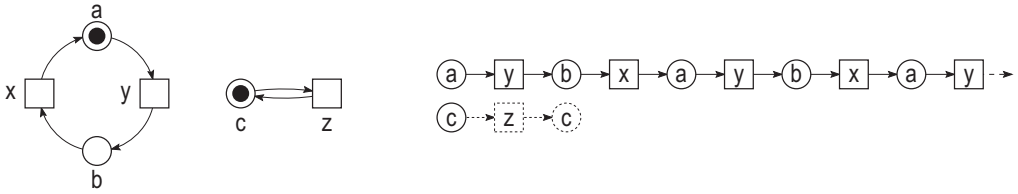
Definition 4.21 (Weak fairness). Let $\Sigma = (N, m_0)$ be a Petri net system, let π be a distributed run of Σ. The run ρ *neglects weak fairness* of transition $t \in T_N$ iff there exists a prefix $\pi \sqsubseteq \rho$ s.t. each cut B of the suffix $(\rho - \pi)$ represents a marking m_B that enables t, and t does not occur in $(\rho - \pi)$.

Let $T^w \subseteq T_N$ be a set of transitions of N. The run π *respects weak fairness* of T^w iff π does not neglect weak fairness for any $t \in T^w$. ⌟

Progress and weak fairness differ slightly on the model of distributed runs. The Petri net system Σ_1 in Figure 4.16 has, among others, the depicted infinite distributed run ρ_1. ρ_1 neglects weak fairness wrt. transition z: z is enabled at



(a) a Petri net system Σ_1 and a distributed run ρ_1 of Σ_1



(b) a Petri net system Σ_2 and a distributed run ρ_2 of Σ_2

Figure 4.16. The runs ρ_1 and ρ_2 neglect weak fairness of transition z. Only run ρ_2 neglects progress of transition z.

every marking of ρ_1 but it never occurs. In contrast, ρ_1 respects progress wrt. transition z because its pre-condition c is affected infinitely often by the alternative transition y. The run ρ_2 of Σ_2 neglects weak fairness and progress wrt. z likewise. Specifically, we could extend ρ_2 by an occurrence of z as indicated by the dashed event. This difference between ρ_1 and ρ_2 can only be observed on distributed runs because both ρ_1 and ρ_2 describe the same sequential runs.

Progress for oclets

We canonically generalize progress of actions to progress of oclets as follows: a system behavior *respects progress* of an oclet o iff

> whenever o's history occurs in a run, then the run either continues with o or it continues with some alternative course of actions. (4.4)

In other terms: no run ends in the middle of an oclet. Figure 4.17 illustrates this notion on our running example of oclet $\mathsf{o_1}$ from the medical emergency specification. Oclet $\mathsf{o_1}$ occurs in the distributed run ρ as indicated by the shaded nodes. The prefixes π_1, π_4, and π_5 of ρ respect progress of oclet $\mathsf{o_1}$; a prefix either ends before the entire history of $\mathsf{o_1}$ occurs (π_1), or the prefix ends right with the oclet (π_4) or after the oclet occurred (π_5). In contrast, the prefixes π_2 and π_3 *neglect progress* of $\mathsf{o_1}$; they either end with $\mathsf{o_1}$'s history (π_2), or when already some part of $\mathsf{o_1}$'s contribution occurred in the run, but not yet all of $\mathsf{o_1}$ (π_3). The formal definition reads as follows.

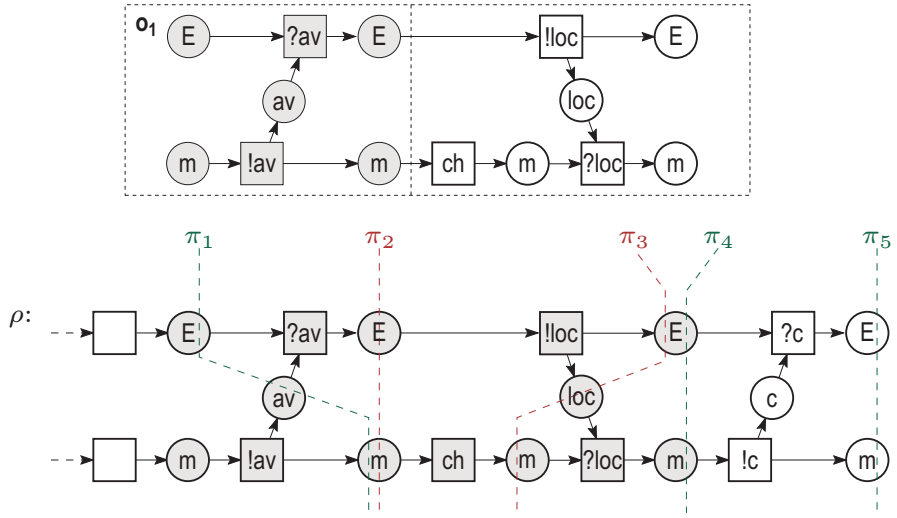


Figure 4.17. The prefixes π_2, π_3 neglect progress of oclet o_1; the prefixes π_1, π_4, π_5 respect progress of o_1.

Figure 4.18. The run π respects progress of oclet o, but $\{\pi\}$ is not closed under continuations with o.

Definition 4.22 (Respect progress of oclets). Let $[o]$ be the class of an oclet $o = (\pi_o, hist_o)$. The *intermediate prefixes* of o are $int(o) := \{\pi^* \mid hist_o \sqsubseteq \pi^* \sqsubseteq \pi_o, \varepsilon \neq \pi^* \neq \pi_o\}$.

A distributed run ρ *neglects progress* of $[o]$ iff there exists an occurrence π^α of an intermediate prefix $\pi \in int(o)$ s.t. ρ ends with π^α. The run ρ *respects progress* of $[o]$ iff it does not neglect progress of $[o]$. A set R of distributed runs *respects progress* of $[o]$ iff each $\rho \in R$ respects progress of $[o]$. ⌟

The notion of progress is independent from the semantics of oclets. A system behavior may satisfy an oclet specification, but it might not respect progress of all oclets. For example, the runs $\{\pi_1, \ldots, \pi_5\}$ in Figure 4.17 satisfy o_1 although π_2 and π_3 neglect progress of o_1. More general, according to Lemma 4.19, for every set R of runs that satisfies an oclet specification O, also $Prefix(R)$ satisfies O. But $Prefix(R)$ usually neglects progress of $[o] \in O$ because some prefix $\pi \in Prefix(R)$ ends with o's history.

Conversely, a system behavior that respects progress of all oclets does not necessarily satisfy these oclets. For example, the set of runs $\{\pi\}$ in Figure 4.18 respects progress of oclet o, but $\{\pi\}$ does not satisfy o.

Thus, a system designer who wants to specify system behavior that respects progress of an oclet has to state this explicitly.

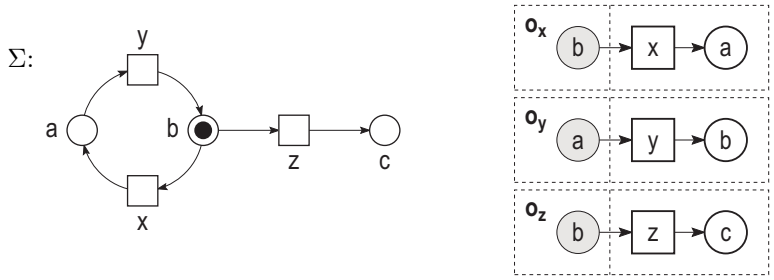


Figure 4.19. A Petri net system Σ and three oclets o_x, o_y, o_z.

Closed under continuations complements progress

The preceding examples and proofs did show that *closed under continuations* (3.2) and *progress* (4.4) are technically independent notions. Though, both notions serve a similar purpose: to specify in which way a system may or has to continue a run. The following example shows that both notions complement each other.

Consider the Petri net system Σ and the three oclets o_x, o_y, o_z in Fig. 4.19. We consider two different oclet specifications $O_{xy} = \{o_x, o_y\}$ and $O_{yz} = \{o_y, o_z\}$, and we compare

1. the runs of Σ that respect progress of O_{xy} and of O_{yz}, respectively, to
2. the runs of Σ satisfy O_{xy} and O_{yz}, respectively.

The runs of Σ that respect progress of the oclets in O_{xy} also respect progress of the oclets in O_{yz}, and vice versa. These are all runs that do not stop in marking $[a]$ (y is enabled) or marking $[b]$ (x and z are enabled). More precisely, the finite runs $(xy)^*z = \{z, xyz, xyxyz, \ldots\}$ and the infinite run $(xy)^\omega$ respect progress; all other runs of Σ neglect progress of the oclets in O_{xy} or O_{yz}, respectively.

In this example, we see that it does not matter whether we require progress for oclet o_x (viz. action x) or for oclet o_z (viz. action z). The simple reason is that progress for o_x or o_z only requires to consume tokens from ${}^\bullet x = {}^\bullet z = \{b\}$. Progress is "not interested" in occurrences of x or z specifically.

Our notion of *closed under continuations* (3.2) fills this gap conceptually: whenever an oclet's history (viz. the preconditions of all actions) occurs in a run, there exists a continuation of this run with this oclet (viz. all actions of the oclet). Hence, the semantics of oclets, which formalize *closed under continuations* distinguish behavior regarding which actions are *available* to continue a specific run.

In our example, the runs $(xy)^*z$ satisfy o_y and o_z but not o_x. Thus, $(xy)^*z$ satisfies O_{yz}, but not O_{xy}. However, the runs $(xy)^*z$ respect progress of O_{xy} and of O_{yz}.

Conversely, the singleton set $\{(xy)^\omega\}$ satisfies O_{xy}, but not O_{yz}. Again, $\{(xy)^\omega\}$ respects progress of O_{xy} and of O_{yz}. Moreover, all runs $(xy)^*z$ and $(xy)^\omega$ together satisfy $O_{xy} \cup O_{yz}$ likewise.

In this sense, *closed under continuations* (3.2) is in line with the intuitive idea of progress (4.3). Progress requires that "either a will occur, or some competing

action b will occur." *Closed under continuations* emphasizes that a is indeed possible. It complements progress of action a by demanding an occurrence of this a.

4.5.2. Extending oclets to invisible actions

Section 4.4.4 already discussed that the existential view of oclets assumes an open world where an implementation may exhibit more behavior than described in the specification. This open world assumption is common to scenario-based techniques with history [25, 45, 107]. However, these techniques have a more liberal open world assumption compared to oclets.

Recall: complete and partial occurrences

Following the extensive discussion on occurrences of scenarios in Section 3.8, we based the formal model of oclets on *complete* occurrences. An implementation may not introduce an action causally between two subsequent actions of a scenario. Only by this strict assumption, the model of oclets guarantees that an implementation does not introduce erroneous behavior which the specification cannot 'see'.

LSCs [25] and Template MSCs [45] are more liberal here and allow for *partial* occurrences of scenarios. To this end, LSCs distinguish *visible* and *invisible* actions: any finite number of invisible actions may occur between two subsequent actions of a scenario. Template MSCs provide *gaps* between two subsequent actions: within a gap additional unspecified behavior may occur.

In the following, we discuss how to generalize the formal model of oclets wrt. invisible actions.

Invisible actions and oclets

The notion of a *complete occurrence* of an oclet o (Definition 4.3) can be generalized to a *partial occurrence* as follows. Here is a possible definition that allows for *invisible* actions between actions of o.

Definition 4.23 (Partial occurrence of an oclet). Let $o = (\pi_o, hist_o)$ be an oclet and let ρ be a distributed run. Let $Vis \subseteq \mathcal{L}$ be a set of visible names containing all names that occur in o. Oclet o *occurs partial* in ρ wrt. Vis iff $X_o \subseteq X_\rho$ s.t. for all $x, y \in X_o$ holds

1. $x \leq_o y$ iff $x \leq_\rho y$, and
2. if $(x, y) \in F_o$ then there exists no $z \in X_\rho$ with $x \leq_\rho z \leq_\rho y$ and $\ell(z) \in Vis$⌟

Partial occurrences of oclets differ from complete occurrences (Definition 4.3) in three aspects. Firstly, the direct successor relationships, defined by the arcs of π_o are no longer preserved in a partial occurrence; instead the oclet's partial order $\leq_o$ must be preserved equivalently. Secondly, the pre- and post-set of events can change arbitrarily. Thirdly, the additional requirement Definition 4.23-2 ensures that the run contains no visible event or condition between two nodes of the

oclet. This requirement is necessary to guarantee that alternative oclets occur in alternative runs as discussed in Section 3.8.

Invisible actions and implementations

We did not consider partial occurrences of oclets in our formal model because *invisible actions*, which are introduced by partial occurrences, may introduce unobservable non-determinism. Specifically, two non-branching bisimilar systems Σ_1 and Σ_2 may satisfy the same specification O if partial occurrences of oclets are allowed; Section 3.9 presented a corresponding example.

Though, the notion of an *implementation* of an oclet specification can be generalized to allow for invisible actions in a safe way. We defined that a Petri net system Σ *implements* an oclet specification iff the runs of Σ satisfy O, i.e., $R(\Sigma) = R \in \mathcal{R}(O)$ (Def. 4.10). To allow for occurrences of invisible actions in Σ, we no longer require equality $R(\Sigma) = R$ but only an equivalence $R(\Sigma) \sim R$ wrt. visible actions; this equivalence has to preserve the branching behavior of R.

Vogler adapts in [125] the classical notion of branching bisimulation [119] and defines when two event structures are *history-preserving bisimilar* wrt. a set $Vis \subseteq \mathcal{L}$ of visible actions. Vogler's notion [125, Def.6.3.6] can be adapted to define when two sets R_1 and R_2 of distributed runs are *history-preserving bisimilar* wrt. *Vis* in our setting.

Intuitively, R_1 and R_2 are history-preserving bisimilar wrt. *Vis* iff (1) the projection of R_1 to the visible actions *Vis* is isomorphic to the projection of R_2 to *Vis*, and (2) each choice between two runs in R_1 has a corresponding choice between two equivalent runs in R_2, and vice versa. Appendix A.6 provides formal details.

This notion of history-preserving bisimulation allows to generalize the notion of an implementation (Def. 4.10) as follows.

Definition 4.24 (Implementation wrt. visible actions). Let O be an oclet specification. Let $Vis \subseteq \mathcal{L}$ be the names of actions and conditions occurring in O. A labeled Petri net system Σ *implements* an oclet specification O wrt. *Vis* iff there exists $R \in \mathcal{R}(O)$ s.t. R and $R(\Sigma)$ are history-preserving bisimilar wrt. *Vis*. ⌟

If Σ implements O wrt. *Vis*, then the oclets in O no longer occur complete in the behavior of Σ as defined in Definition 4.3. Instead, invisible actions may occur between two subsequent actions of an oclet in O. Thus, the notion of implementation wrt. visible events corresponds to the notion of partial occurrences of an oclet given in Definition 4.23.

4.6. Example for Specifying with Oclets

The first two sections of this chapter introduced syntax and semantics of oclets in a formal model based on distributed runs. The third section presented some basic properties of oclets and the fourth section introduced two extensions for oclets. In the following, we show by an example how a system designer may specify behavior of distributed systems with oclets. We return to our running example of

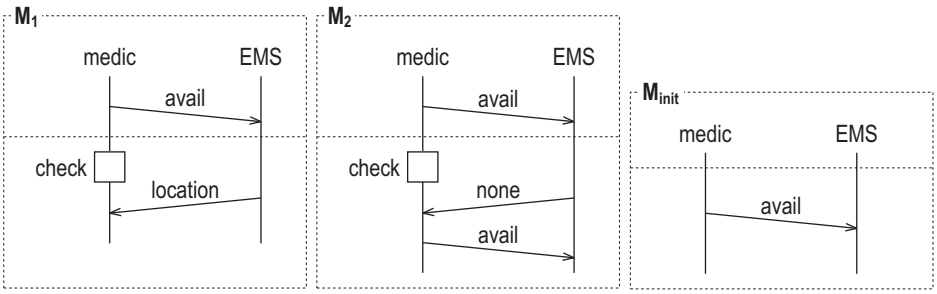


Figure 4.20. Oclet specification $\mathsf{O}_{\mathsf{contact}} = \{\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_{\mathsf{init}}\}$ of the medical emergency example (Sect. 3.2) in MSC notation.

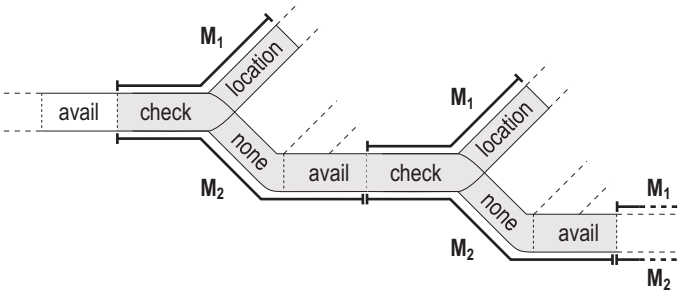


Figure 4.21. Behavior that satisfies $\mathsf{O}_{12} = \{\mathsf{M}_1, \mathsf{M}_2\}$.

the medical emergency specification from Chapter 3. Oclets are noted down in the syntax of MSCs for a more succinct representation.

4.6.1. Creating a well-designed specification

Figure 4.20 depicts the oclets M_1 and M_2 which specify how an Emergency Management System (EMS) reacts when a medic is available to handle a medical emergency. We begin with the oclet specification $\mathsf{O}_{12} = \{\mathsf{M}_1, \mathsf{M}_2\}$. Figure 4.21 depicts an execution tree, representing a set of distributed runs that satisfies O_{12}; the dashed branches indicate that a satisfying behavior may have more runs than those depicted in M_1 and M_2.

The labeled Petri net system $\Sigma_{\mathsf{contact}}$ in Figure 4.22 implements O_{12}. The set of distributed runs $R(\Sigma_1)$ corresponds to the tree in Figure 4.21.

Although $R(\Sigma_1)$ implements O_{12}, the oclet specification O_{12} is ill-designed: there are also systems that implement O_{12} but where neither oclet M_1 nor oclet M_2 occurs. For instance, all systems in which a message avail is never sent and so the history of M_1 and M_2 never occurs. Specifically, the empty system behavior satisfies O_{12} by Lemma 4.13, because O_{12} contains no ε-oclet. To improve the specification, we add oclet $\mathsf{M}_{\mathsf{init}}$ of Figure 4.20 to O_{12}; the resulting specification is $\mathsf{O}_{\mathsf{contact}} = \{\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_{\mathsf{init}}\}$. The new oclet $\mathsf{M}_{\mathsf{init}}$ specifies that message avail is indeed exchanged between medic and EMS initially.

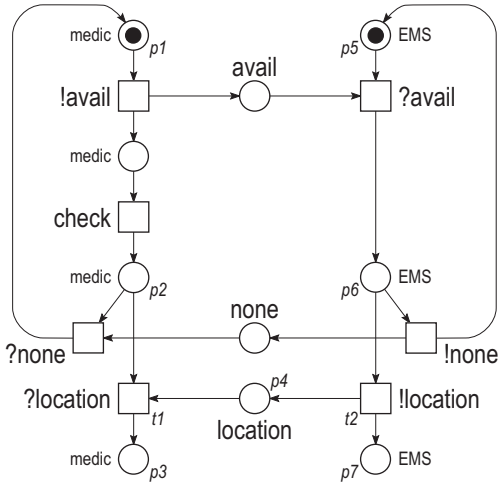


Figure 4.22. The Petri net system $\Sigma_{\mathsf{contact}}$ implements the oclet specification $\mathsf{O}_{\mathsf{contact}}$ of Fig. 4.20.

Now, every behavior R that satisfies $\mathsf{M}_{\mathsf{init}}$ contains occurrences of all oclets in $\mathsf{O}_{\mathsf{contact}}$. Every satisfying behavior R has a run where $\mathsf{M}_{\mathsf{init}}$ occurs, which is identical with an occurrence avail (i.e., the history of M_1 and M_2). Because R satisfies M_1 and M_2, both oclets occur in R after as well.

The system $\Sigma_{\mathsf{contact}}$ of Figure 4.22 implements $\mathsf{O}_{\mathsf{contact}}$. Further, the runs $R(\Sigma_{\mathsf{contact}})$ are minimal wrt. $\mathsf{O}_{\mathsf{contact}}$, and $\mathsf{O}_{\mathsf{contact}}$-covered. By the structural correspondence between the syntax of Petri nets and their distributed runs, we can also find the oclets of $\mathsf{O}_{\mathsf{contact}}$ as sub-nets in $\Sigma_{\mathsf{contact}}$. For example, M_1 occurs in $\Sigma_{\mathsf{contact}}$, beginning at the places p1, p5 and ending at the places p3, p7.

Finally, the system $\Sigma_{\mathsf{contact}}$ can be decomposed into two asynchronously communicating components Σ_{medic} and Σ_{EMS}. The component Σ_{medic} consists of all places labeled medic, their pre- and post-transitions, and the places avail, none and location as interface; Σ_{EMS} consists of the all other places and transitions plus the interface avail, none and location.

4.6.2. Composing specifications

In the second part of the example, we show how to compose oclet specifications, and how this affects their implementations.

Our second specification describes how a medic treats a patient in case of an emergency. Figure 4.23 depicts the corresponding oclets M_3 and M_4. Among others, the singleton set of distributed runs depicted in Figure 4.24 satisfies the specification $\mathsf{O}_{\mathsf{treat}} = \{\mathsf{M}_3, \mathsf{M}_4\}$. Specifically, oclets M_3 and M_4 *occur overlappingly* in Figure 4.24 as follows. After location and confirm occurred, oclet M_4 has to occur. Oclet M_3 allows any behavior for the EMS after confirm occurred, and M_4 allows any behavior for the medic after confirm occurred. Thus, the medic's and the hospital's actions can occur in the same run. By the same arguments, M_3

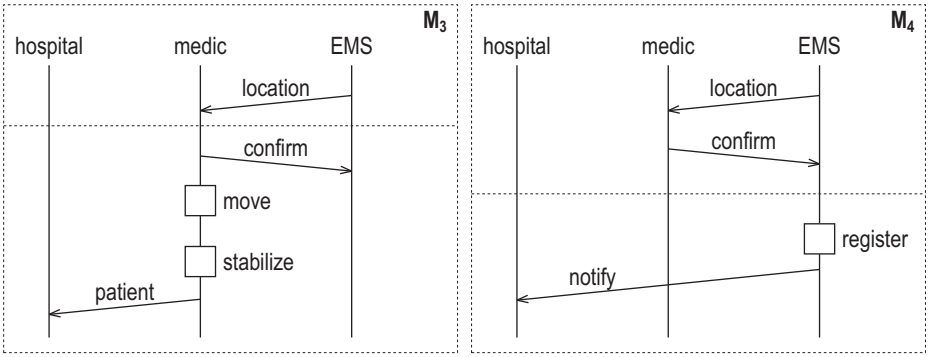


Figure 4.23. Oclet specification $\mathsf{O}_{\mathsf{treat}} = \{\mathsf{M}_3, \mathsf{M}_4\}$ of the medical emergency example (Sect. 3.2) in MSC notation.

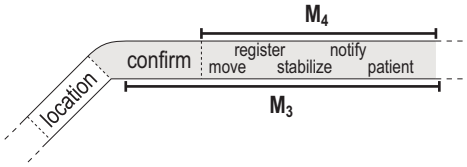


Figure 4.24. Behavior that satisfies $\mathsf{O}_{\mathsf{treat}} = \{\mathsf{M}_3, \mathsf{M}_4\}$.

allows any behavior for the hospital before and after patient and M_4 allows any behavior for the hospital before and after notify. Thus, the hospital may receive these messages in an arbitrary order in the same run as well. The Petri net system Σ_{treat} of Figure 4.25 implements $\mathsf{O}_{\mathsf{treat}}$.

We now compose the two specifications $\mathsf{O}_{\mathsf{contact}}$ and $\mathsf{O}_{\mathsf{treat}}$ to the specification $\mathsf{O}_{\mathsf{med}} = \mathsf{O}_{\mathsf{contact}} \cup \mathsf{O}_{\mathsf{treat}}$. According to Corollary 4.11, an implementation of $\mathsf{O}_{\mathsf{med}}$ has to satisfy all behavioral requirements of $\mathsf{O}_{\mathsf{contact}}$ and $\mathsf{O}_{\mathsf{treat}}$. The execution tree of Figure 4.26 satisfies $\mathsf{O}_{\mathsf{med}}$. The composed specification imposes more requirements to an implementation: neither $\Sigma_{\mathsf{contact}}$ nor Σ_{treat} implements $\mathsf{O}_{\mathsf{med}}$.

$\mathsf{O}_{\mathsf{med}}$ specifies one complete handling of a medical emergency. The upper-most branch of the tree of Figure 4.26 describes the following situation: after the hospital has been notified and the patient arrived, no further action of the hospital, the medic, or the EMS is required; this situations corresponds to the marking $[\mathsf{hospital}, \mathsf{p1}, \mathsf{p5}]$ of Σ_{treat} in Figure 4.25 which is a deadlock.

To specify that the medic should send a message to the EMS once he is available again, we have to add another oclet like M_5 of Figure 4.27. Let $\mathsf{O}_{\mathsf{repeat}} = \{\mathsf{M}_5\}$ denote this specification. The composed specification $\mathsf{O}_{\mathsf{med}+} = \mathsf{O}_{\mathsf{med}} \cup \mathsf{O}_{\mathsf{repeat}}$ requires that a message avail occurs after M_4 occurred; this message avail triggers scenarios M_1 and M_2 again. Thus, a behavior that satisfies $\mathsf{O}_{\mathsf{med}+}$ has infinitely many infinite runs.

We have already seen that neither $\Sigma_{\mathsf{contact}}$ nor Σ_{treat} implements $\mathsf{O}_{\mathsf{med}}$ or $\mathsf{O}_{\mathsf{med}+}$. But we can *compose* both Petri net systems to an implementation of $\mathsf{O}_{\mathsf{med}+}$:

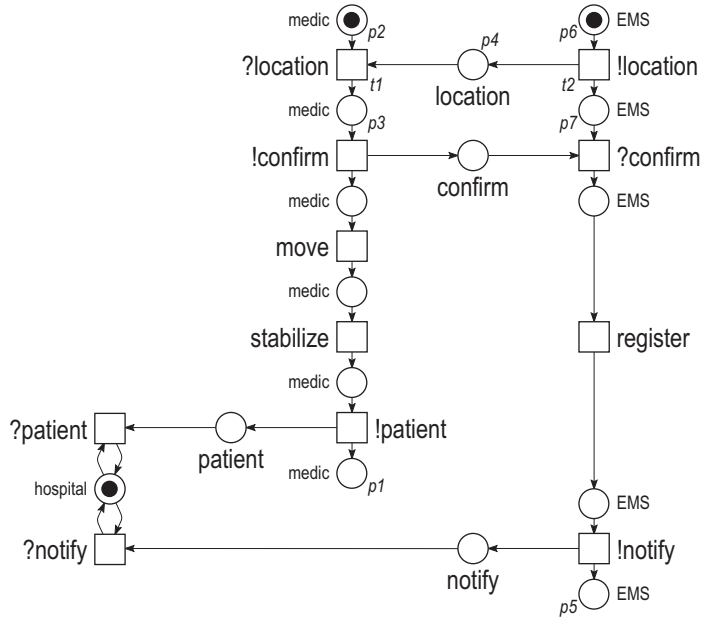


Figure 4.25. The Petri net system Σ_{treat} implements the oclet specification $\mathsf{O}_{\mathsf{treat}}$ of Fig. 4.23.

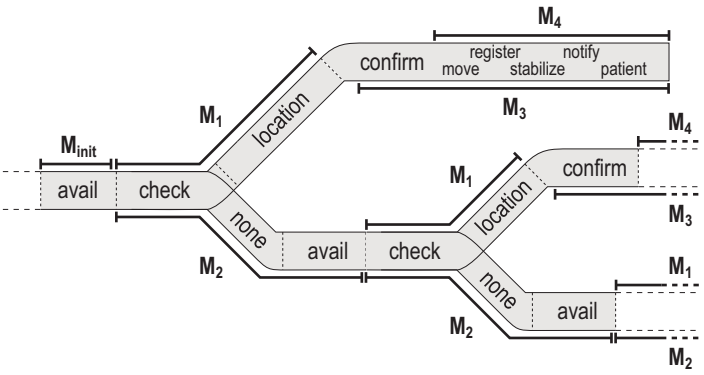


Figure 4.26. Behavior that satisfies $\mathsf{O}_{\mathsf{med}}$.

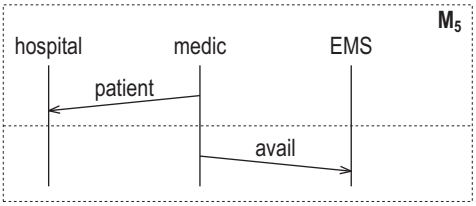


Figure 4.27. Oclet M_5 in MSC notation.

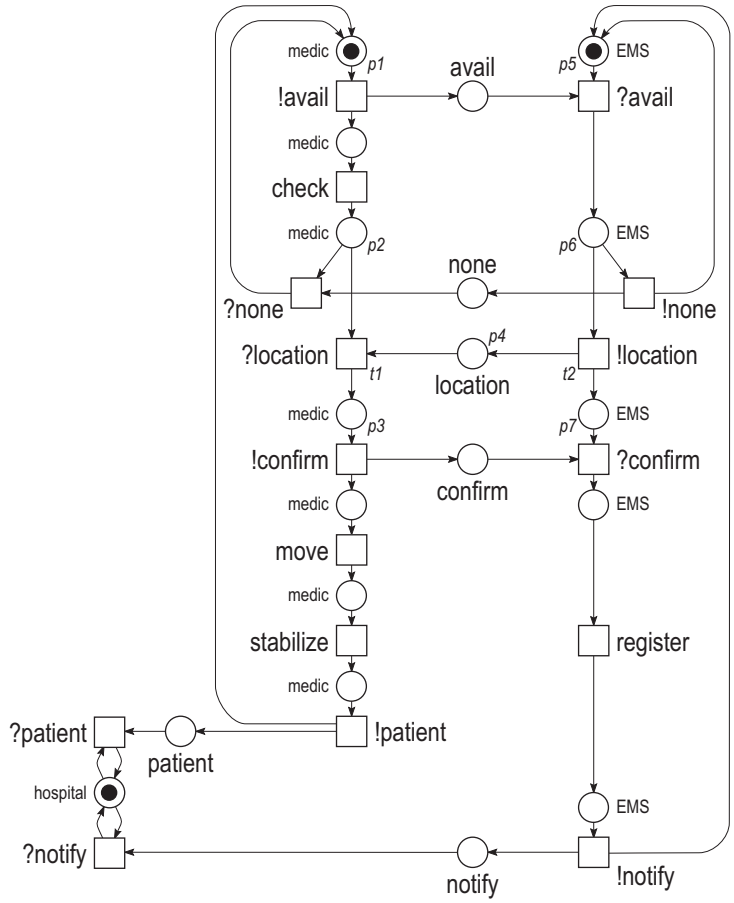


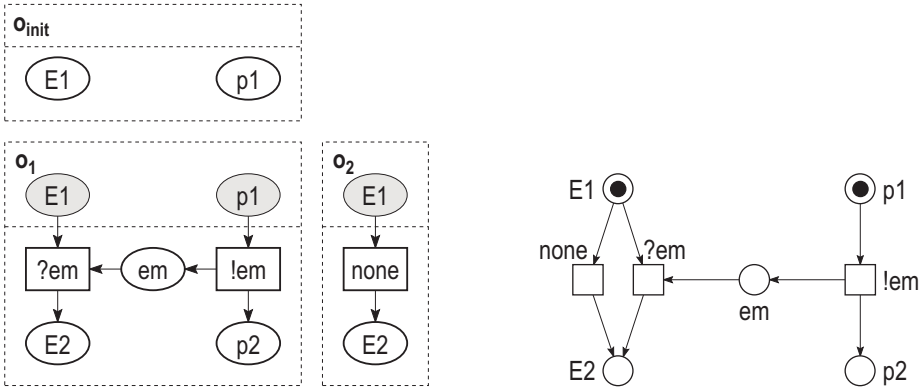
Figure 4.28. The Petri net system $\Sigma_{\mathsf{med+}}$ implements $\mathsf{O}_{\mathsf{med+}} = \{\mathsf{M}_{\mathsf{init}}, \mathsf{M}_1, \ldots, \mathsf{M}_5\}$.

1. Assume that the underlying nets N_{contact} and N_{treat} of both Petri net systems overlap on the places p1-p7 and transitions t1, t2, and are disjoint otherwise; see Figures 4.22 and 4.25.
2. Composing $\Sigma_{\mathsf{contact}}$ and Σ_{treat} yields the system $\Sigma_{\mathsf{med+}} = (N_{\mathsf{med+}}, m_0)$ where $N_{\mathsf{med+}} = N_{\mathsf{contact}} \cup N_{\mathsf{treat}}$ and $m_0 := [\mathsf{p1}, \mathsf{p5}]$.

Figure 4.28 depicts the composed Petri net system $\Sigma_{\mathsf{med+}}$; the system implements specification $\mathsf{O}_{\mathsf{med+}}$. The behavior of $\Sigma_{\mathsf{med+}}$ is $\mathsf{O}_{\mathsf{med+}}$-covered and minimal wrt. $\mathsf{O}_{\mathsf{med+}}$. By the open world assumption of oclets, the system $\Sigma_{\mathsf{med+}}$ also implements each of the original smaller specifications $\mathsf{O}_{\mathsf{contact}}$ and $\mathsf{O}_{\mathsf{treat}}$. Though, the behavior of $\Sigma_{\mathsf{med+}}$ is neither minimal wrt. the smaller specifications nor it is covered by either one.

We conclude this section by briefly comparing the composition of Petri net systems to the composition of oclet specifications in this example.

1. We composed $\mathsf{O}_{\mathsf{contact}}$, $\mathsf{O}_{\mathsf{treat}}$, and $\mathsf{O}_{\mathsf{repeat}}$ to $\mathsf{O}_{\mathsf{med+}}$, and



(a) specification $\mathsf{O}_{\mathsf{conf}}$ (b) implementation Σ_{conf}

Figure 4.29. Oclets specify causality: Petri net system Σ_{conf} implements $\mathsf{O}_{\mathsf{conf}}$.

2. we composed $\Sigma_{\mathsf{contact}}$ and Σ_{treat} to $\Sigma_{\mathsf{med+}}$.

Composing the systems $\Sigma_{\mathsf{contact}}$ and Σ_{treat} is not canonic. Which places and which transitions are identical in the nets and which are different does not follow from the nets' structure alone. Moreover, the initial marking of the composed system does not follow from $\Sigma_{\mathsf{contact}}$ and Σ_{treat} — even if we know which places are identical. Technically, we could have come up with a completely different composition that does not implement $\mathsf{O}_{\mathsf{med}}$.

In contrast, composing the oclet specifications $\mathsf{O}_{\mathsf{contact}}$, $\mathsf{O}_{\mathsf{treat}}$, and $\mathsf{O}_{\mathsf{repeat}}$ to $\mathsf{O}_{\mathsf{med+}}$ was straight forward. Each oclet describes a certain behavioral property; the behavior of an implementation has to satisfy all described behavioral properties. Here, it turns out very useful that the composition of two oclet specifications is the intersection of their satisfying behaviors; see Corollary 4.11.

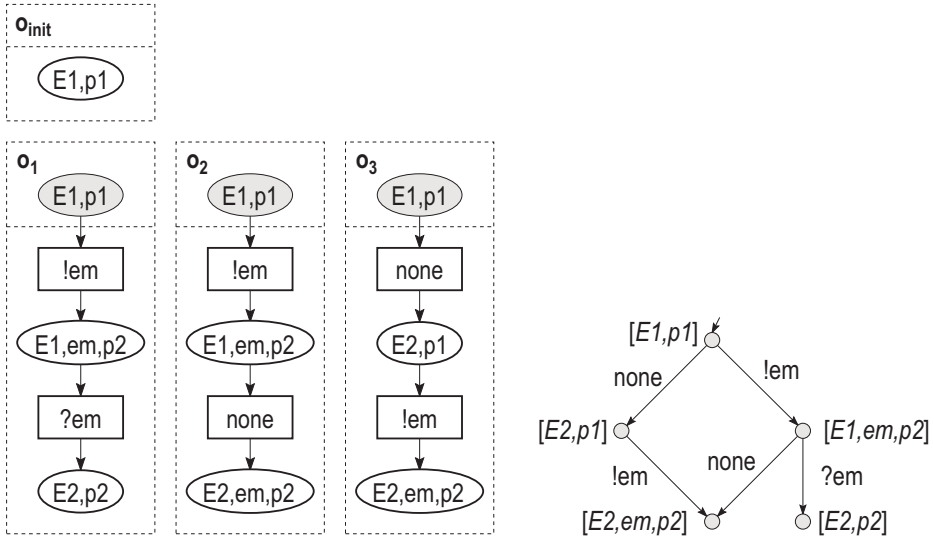
4.6.3. Causality and confusion

This section illustrates how the behavioral model of *distributed runs* of oclets allows to specify causality and specific effects in distributed systems.

We want to specify the following behavior. When the Emergency Management System (EMS) of the preceding examples is asked to return information about a pending emergencies, the following two scenarios may occur:

1. The EMS may get notified by a phone service about an emergency. In this case, this emergency will be handled.
2. If there is no notification about an emergency, then the result of the check is 'none'.

The oclet specification $\mathsf{O}_{\mathsf{conf}}$ in Figure 4.29 abstractly captures this behavior, and the depicted Petri net system Σ_{conf} implements $\mathsf{O}_{\mathsf{conf}}$.



(a) specification $\mathsf{O}_{\mathsf{seq}}$ (b) state space of Σ_{conf}

Figure 4.30. Petri net system Σ_{conf} of Fig. 4.29 does not implement $\mathsf{O}_{\mathsf{seq}}$ although both describe the same sequential runs.

The point of interest in this example is the choice between actions none and ?em. Both actions are alternative. More specifically, the transitions none and ?em of Σ_{conf} are in conflict about the token on place E1. Though, this conflict is not "objective" because ?em is not enabled initially. The enabling of ?em depends on an occurrence of !em which is concurrent to none. This situation is called *confusion*: *whether two transitions actually compete for the same token or not depends on a third transition that is concurrent to one of them.*

Confusion expresses a situation where a component may have to make a choice "against time". In the example of Figure 4.29, the EMS may choose at any time to evaluate the check to none. Or it may wait a little longer until a notification about an emergency arrives. Though, the EMS cannot wait until a notification arrives because it does not know whether one will arrive at all. Confusion inevitably occurs in distributed system. To avoid erroneous behavior, the system needs to be improved with further actions. For instance, to handle the case that an emergency arrives right after the EMS evaluated the check to none.

We close this example with a discussion how confusion depends on the behavioral model of distributed runs. Figure 4.30 depicts an alternative specification $\mathsf{O}_{\mathsf{seq}}$. $\mathsf{O}_{\mathsf{seq}}$ and $\mathsf{O}_{\mathsf{conf}}$ specify *different* systems which make choices in a different way. $\mathsf{O}_{\mathsf{seq}}$ specifies a system where the EMS and the phone service share a global state, actions occur in a sequential order, and choices are made based on the global state. In contrast, $\mathsf{O}_{\mathsf{conf}}$ specifies a system having local states for the EMS and the phone service, where local actions and local states are ordered by their *cause-effect relations*, and where choices are made based on local knowledge.

This difference between $\mathsf{O}_{\mathsf{seq}}$ and $\mathsf{O}_{\mathsf{conf}}$ can only be seen on the behavioral model of distributed runs. The Petri net system Σ_{conf} does not implement $\mathsf{O}_{\mathsf{seq}}$. However, both specifications describe the same sequential behavior. Figure 4.30 depicts the state space of the Petri net system Σ_{conf} from Figure 4.29: the state space describes all behavior of Σ_{conf} *sequentially*. The information about confusion, which is present in $\mathsf{O}_{\mathsf{conf}}$ vanishes in the state space. The choice between none and ?em becomes objective in the state space because only state $[\mathsf{E1}, \mathsf{em}, \mathsf{p2}]$ enables both actions; in $[\mathsf{E1}, \mathsf{p1}]$ action none is "alternative" to action !em. The *state space* of Σ_{conf} "implements" $\mathsf{O}_{\mathsf{seq}}$.

To conclude, the formal model of oclets that is based on distributed runs allows to specify the behavior of distributed systems more precisely than techniques which are based on the behavioral model of sequential runs.

4.6.4. Tool support

The results of this thesis have been implemented in a software prototype called Greta that is available at `http://service-technology.org/greta`. We present specific features of Greta in the respective chapters. Greta has a graphical interface for specifying distributed systems with oclets as presented in this chapter. Figure 4.31 shows the main graphical editor window of Greta with the oclet specification of Section 4.6.

Greta is based on the *Eclipse* software platform and can be extended with *plug-ins* for further functionality. We present several plug-ins for Greta in the subsequent chapters. The graphical interface of Greta and its software technical foundation have been developed by Wolf using a *model-driven approach* [128].

4.7. Comparison to Other Models and Conclusion

In Chapter 3, we reviewed existing scenario-based techniques and identified a minimal set of notions for specifying system behavior with scenarios. The model of *oclets* formalizes this minimal set of notions in terms of *distributed runs*.

Section 4.7.1 briefly summarizes these results, Section 4.7.2 discusses the model of oclets wrt. other scenario-based techniques, and Section 4.5.2 concludes this chapter.

4.7.1. Summary

The model of oclets is based on three simple ideas: (1) An *oclet* formalizes a scenario as a *partial distributed run*; a *specification* is a set of oclets. (2) An oclet distinguishes a *history*. And (3) a set of runs *satisfies* a specification if every run that ends with an oclet's history has a continuation with the entire oclet.

A satisfying set of runs can contain more runs than specified. In this respect oclets specify a "lower bound" to system behavior. A system S that exhibits less behavior than specified does not implement the specification; if S exhibits more behavior, then S implements the specification. Thus, a system designer may use oclets to specify *expected* system behavior which is the purpose of scenarios

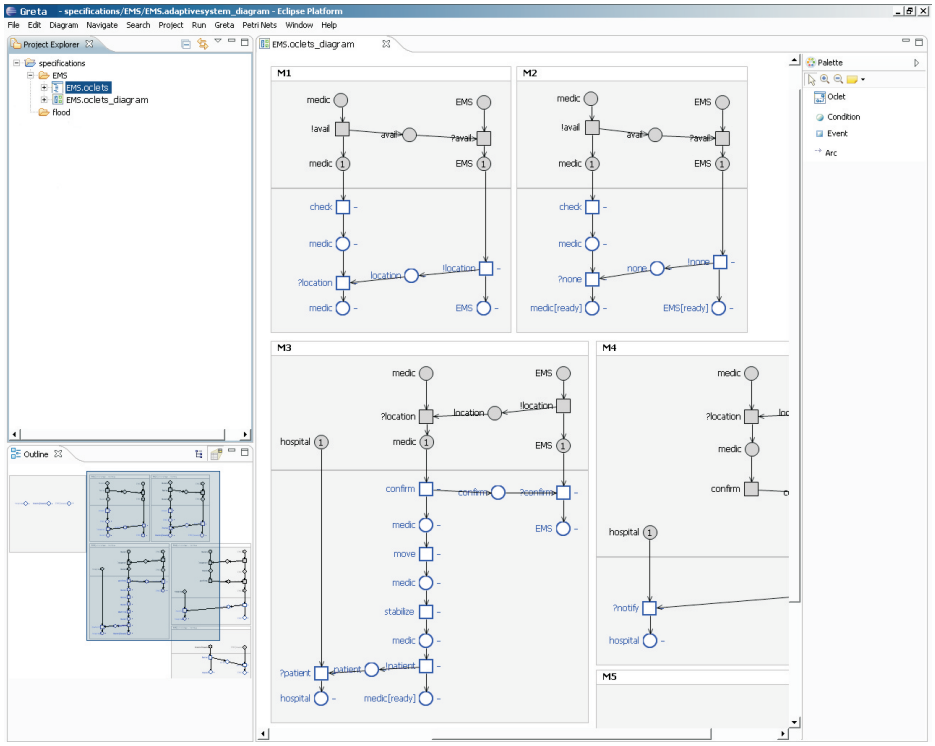


Figure 4.31. The graphical editor of GRETA for specifying distributed systems with oclets.

as discussed in Chapter 3. The idea to require continuations is novel in the scenario-based approach and necessary for specifying the *complete* behavior of a *distributed* system as discussed in Section 3.6.2.

Section 4.4 studied several basic properties of oclets and formally defined *existential* and *universal view* for oclets. Section 4.5 presented *progress* and *invisible actions* as extensions of oclets. We have shown in Section 4.6 how to specify system behavior with oclets and how to compose oclet specifications.

By using oclets in the manner presented, we obtain a flexible scenario-based specification technique that uses the same technical notions to denote scenarios, system behavior, and implementations: Petri nets. This solves the first research goal of this thesis and paves the way for solving the synthesis problem. The definitions presented in this chapter form the foundation for the forthcoming chapters.

4.7.2. Discussion

This section compares the formal model of oclets to other formal techniques for scenarios wrt. syntax and semantics. We specifically highlight that the formal model of oclets employs a *minimal set* of notions to define syntax and semantics of scenarios.

Syntax

All scenario-based techniques propose notions for graphically specifying local actions, local states, and their dependencies. Most techniques adapt (and extend) the industry standard of *Message Sequence Charts* (MSCs) [65]. Oclets use the syntax of Petri nets to denote a basic scenario, i.e., a course of actions in the system.

In direct comparison, the syntax of oclets allow for specifying arbitrary courses of actions whereas the syntax of MSCs restricts scenarios by the notion of components and some syntactic constraints. Some constraints of MSCs may be lifted using additional notions like *co-regions* [65]. Other approaches omit the notion of a local state and denote a scenario just as partial order of actions [13]; this syntax is as liberal as oclets but less expressive as discussed in Section 2.3.

Explicit Composition. Most scenario-based techniques provide some structuring technique to express when a scenario may occur wrt. other scenarios. Two approaches exist. One approach is to *hierarchically compose* smaller scenarios to larger (sets of) scenarios using explicit *operators* like sequential, alternative, parallel composition and finite iteration. Among others, *Hierarchical MSCs* (HMSCs) [65] follow this approach; see Section 3.5. The second approach provides the notion of an *inline scenario*, i.e., an expression within a scenario that contains one or more scenarios together with an operator. UML sequence diagrams [95] and *Live Sequence Charts* (LSCs) [25] follow this approach; see Figure 3.13 on page 66 for an example.

History. Oclets use the notion of a history to express when a scenario may occur. Several techniques apply this notion as well [107, 45, 31, 32]; the most prominent representative with formal semantics are LSCs [25]. The interpretation of a history is the same in all techniques: if the specified behavior (a single message or a partial order of actions) occurs, then the scenario *may* or *must* occur. Existing techniques express *alternative* scenarios using an additional notion like hierarchical composition [107, 32] or inline scenarios [25, 45].

The declared aim of this part of this thesis was to identify a *minimal* set of notions for specifying behavior with scenarios. Oclets use only the notion of a history to express when a scenario may occur. The ordering of scenarios follows from their inner logic. An oclet's history expresses *sequential* ordering in a natural way. Two oclets with the same history that diverge at some point describe *alternative* scenarios. Otherwise, they can occur together and describe *concurrent* or *overlapping* scenarios. Chapter 5 shows that these notions suffice for specifying the *complete* behavior of a distributed system. Oclets succeeded with this minimal set of notions because of their semantics which we discuss next.

Semantics

Behavioral model: causality. Oclets are interpreted on the behavioral model of distributed runs. In a distributed run, an event describes an occurrence of a local

action, and a condition describe that a local state is visited; events and conditions are ordered by *causality*. Semantics that use this model of distributed runs are also called *true-concurrency semantics*; semantics based on labeled partial orders are usually referred to as *partial order semantics*; the most common behavioral model are sequential runs which leads to *interleaving semantics*; see [91, 114] and Chapter 2.

Most formal techniques define an *interleaving semantics of scenarios on the behavioral model of sequential runs*. Specifically expressivity and complexity of scenario-based techniques has been researched on sequential runs in terms of *formal languages* over an alphabet of actions [60, 16]. Several partial order semantics for (H)MSCs have been defined [6, 67, 61, 59]. True-concurrency semantics for (H)MSCs follow from a translation to Petri nets [62, 33]. LSCs and other techniques with history use an interleaving semantics [25, 45, 107].

To our best knowledge, *oclets are the first technique which define a true-concurrency semantics for scenarios with history*. Section 4.6.3 discussed that only a partial order semantics or a true-concurrency semantics allows to clearly distinguish alternative actions from (interleaved) concurrent actions. As a result, oclets allow to specify intricate phenomena of distributed systems like *confusion*. We shall exploit the more precise representation of behavior in terms of distributed runs when synthesizing components from a specification in Chapter 8.

Relation between syntax and semantics. Oclets use the same notions from Petri nets to formalize syntax and semantics. This paradigm has become standard for techniques using hierarchical composition [65, 58, 13] because it reduces the technical overhead for relating syntax to semantics. Oclets, are the first technique applying this paradigm to scenarios with history. The specific advantage of oclets is that their behavioral model is "technically compatible" with Petri nets which will simplify synthesis in Chapter 8.

Composition is intersection. The semantics of oclets is compositional: a set R of distributed runs satisfies an oclet specification O iff R satisfies each oclet of O. In other words, the semantics of O is the *intersection* of the semantics of its oclets.

All other formal scenario-based techniques have compositional semantics as well, though requiring more than just intersection. All techniques using hierarchical composition or inline scenarios achieve compositionality by sequential, alternative, parallel composition, and finite iteration;[3] some extensions provide more refined operators with compositional semantics [72]. In history-based techniques, the main composition operator is intersection as well. However, to express alternative scenarios LSCs [25] and Template MSCs use inline scenarios, and Triggered MSCs [107] use hierarchical alternative composition.

Formal proofs over the semantics of specifications require induction and/or case distinction wrt. the involved composition operator. Here, oclets benefit from having intersection as the only composition operator. The proofs in this chapter

[3] These techniques are MSCs [87], HMSCs [61, 59, 60], Causal MSCs [44], UML sequence diagrams [33], scenarios over LPOs [13], scenario trees [64], Glinz' statechart scenarios [46], Somé's scenarios [31].

already exploited this property. We have shown in Section 4.4.6 that composition in oclets is in line with composition in temporal logic specifications [1, 76]. Again, oclets succeed with a *minimal* set of notions compared to other scenario-based techniques. The reason is that oclets use a simple branching time semantics with an open world assumption.

Branching time behavior. The semantics of oclets define a very simple branching time property: whenever the history of an oclet occurs at the end of a run, then *there exists* a continuation with this oclet. The example in Section 3.6.2 demonstrated that expressing branching time behavior matters for specifying the complete behavior of distributed systems.

Hélouet et al. [59] provide branching time semantics for HMSCs in terms of event structures and graph grammars. Techniques using the notion of a history mostly define a linear time semantics; see Section 3.5.2. This requires an additional notion such as alternative composition to describe alternative scenarios which is not necessary in oclets. Only Triggered MSCs [107] use a branching time semantics in terms of an *acceptance tree*; though, branching in Triggered MSCs follows from an explicit composition operator as well.

Open world assumption. The semantics of oclets is based on an *open world assumption* which supposes that each scenario only describes a partial course of actions. A system may implement more behavior than specified as long as all requirements of all scenarios are satisfied; see Section 4.4.4.

An open world assumption is common to all techniques using the notion of a history [25, 107, 45]. Damm and Harel coined this open world assumption *existential view* in which a specification describes "sample behavior" of the system [25], or only the behavior of some part of a system or a certain interaction. In contrast, "MSC system specifications are viewed as complete elaborations of system behavior that allow no deviation" [107, p.588]. This *closed-world assumption* also applies to (H)MSCs and all other techniques which explicitly compose scenarios; Damm and Harel called this view the *universal view* [25].

We could show that the universal view of oclets follows from their existential view by intersection with a canonical safety property; see Section 4.4.5. Together with LSCs, oclets are the only technique providing universal and existential view in one model.

Progress. Section 4.5.1 extended oclets by the notion of *progress* which allows a system designer to distinguish runs where the system may terminate from runs where the system has to progress.

Most techniques generally assume progress. In MSCs, the system progresses until it reaches the end of the specified scenario; in HMSCs, the system progresses until it reaches the end of a composition of scenarios. Template MSCs and Triggered MSCs generally assume progress for the specified scenarios [107, 45]. LSCs introduce modalities to distinguish *provisional*, *mandatory*, and *forbidden* behavior. A mandatory scenario (action) requires progress for this scenario (action). Though, mandatory behavior also entails a notion of priority: a mandatory scenario (action)

must occur if its pre-conditions hold. Thus, modalities in LSCs specify more than progress.

In comparison to LSCs, oclets define progress separated from the semantics as an additional notion like in Petri nets. Again, the separation simplifies the basic model of oclets and demonstrates how oclets can be extended towards a more expressive specification language.

4.7.3. Conclusion

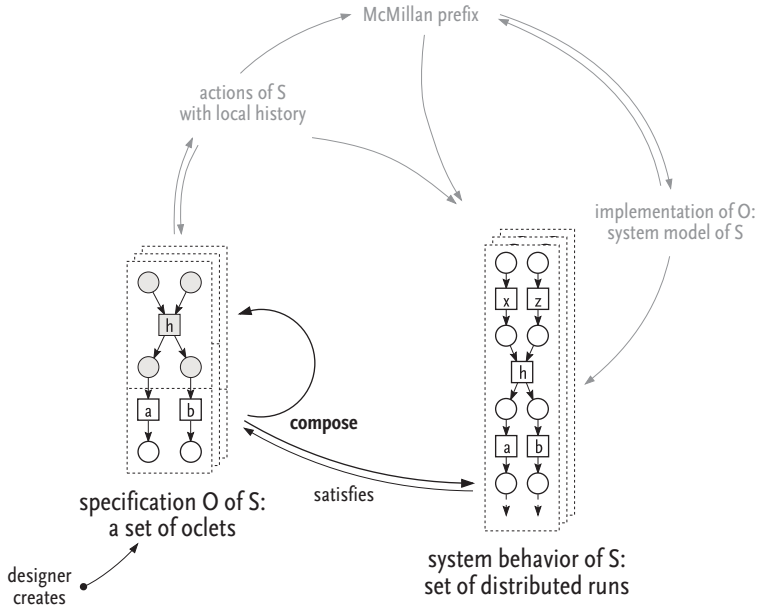
The formal model of oclets introduced in this chapter is not radically different from existing techniques. Thus, our approach can build on existing notions. Yet, in each aspect where oclets differ, oclets use either less notions than comparable techniques (e.g., composition is intersection) or "recycle" notions that are already present in the model (e.g., distributed runs). Altogether, oclets define a formal model of scenarios that uses less technical notions than other techniques, specifically compared to techniques using a history.

According to this conclusion, the formal model of oclets developed in this chapter solves our first goal of this thesis: we identified a kernel of scenarios that allows for specifying the complete behavior of a distributed system in a flexible way. Oclets cast this kernel into a formal model using a minimal set of technical notions. Most importantly for the upcoming results of this thesis, oclets use the *same* technical notions to describe scenarios, system behavior and implementation: Petri nets. These results shall help us to define an operational semantics for scenarios and to solve the synthesis problem.

Part III.

Modeling Distributed Systems with Scenarios

5. Constructing Behavior From Scenarios



The preceding Chapters 3 and 4 identified a kernel of scenario-based specifications and formalized it in the model of *oclets*. A system designer *specifies* with oclets how components of a distributed system interact with each other. This and the next chapter turn oclets into a flexible *modeling technique* for distributed systems. To this end, we show that oclets generalize the notion of a *distributed run* and a *transition* likewise. This chapter introduces a simple operator to *compose* two oclets to a larger oclet. We then show that oclet composition corresponds to oclet semantics: repeatedly composing all oclets of a specification O yields the least set of runs that satisfies O. These results are fundamental for solving the synthesis problem for oclets.

5.1. A Constructive Approach to System Behavior

The main challenge addressed in this thesis is to synthesize a state-based system model Σ from a scenario-based specification O. More precisely, we have to provide an algorithm that *constructs* a system model Σ s.t. the behavior of Σ satisfies O. Because O describes only behavioral properties, there are many different system models which satisfy O. The synthesized system model preferably exhibits the *minimal* behavior that satisfies O; any additional behavior would be redundant.

In this chapter, we take a first step towards synthesis and introduce a composition operator to construct from the specification O the minimal behavior that satisfies O.

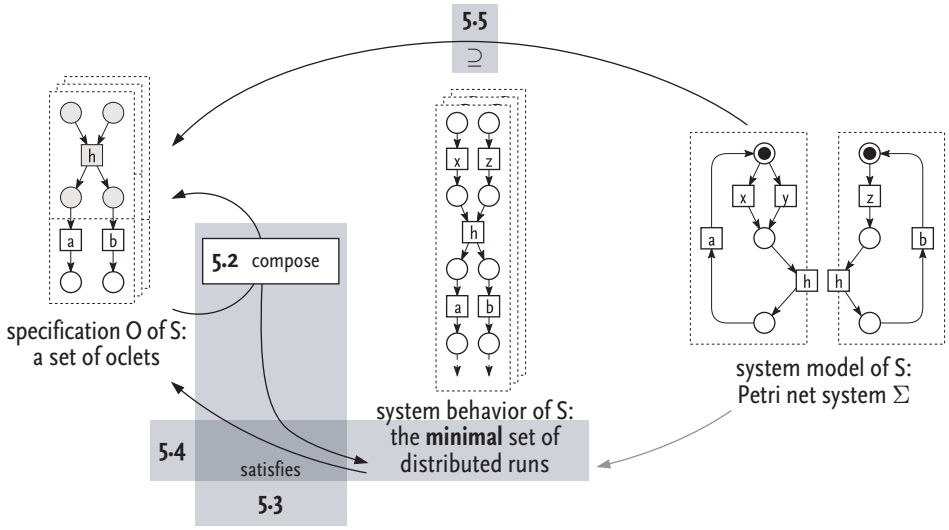


Figure 5.1. Oclet composition corresponds to oclet semantics.

What is given?

Chapter 4 introduced *oclets* as a formal model for scenario-based specifications. One oclet describes one scenario as a *distributed run* in the syntax of Petri nets. Each oclet distinguishes a *history* which describes when the scenario may *occur*; the history may be empty. A *specification* is a set O of oclets. The *semantics* of oclets defines when a system behavior (a set of distributed runs) *satisfies* an oclet specification O, as illustrated in Figure 5.1.

We have shown in Sections 4.4.3-4.4.5 that each oclet specification O has, among all its satisfying behaviors, a unique least set of runs $\min \mathcal{R}(O)$ which satisfies O. Moreover, O contains only runs that are described by O. Figure 5.3 illustrates the situation.

What do we want?

In the context of synthesizing an implementation Σ from a specification O, the set $\min \mathcal{R}(O)$ of runs is the system behavior that Σ preferably exhibits. Currently, we have only proven the *existence* of $\min \mathcal{R}(O)$ and not given a synthesis algorithm that *automatically constructs* Σ.

We approach a solution to the synthesis problem from the "behavioral side". In this and the following chapter, we

construct system behavior from oclets.

The techniques that we develop make explicit which behavioral information is contained in an oclet specification O. As a consequence, we obtain an *operational semantics* for oclets which turns a set O of oclets into a *system model*. By the help of this operational semantics, we will then be able to construct an equivalent Petri net system Σ that implements O from Chapter 7 onwards. We proceed as sketched in Figure 5.1.

Road map

The results of this chapter follow from a simple observation: oclets generalize distributed runs and transitions likewise. An oclet with an empty history describes a distributed run. An oclet o with a non-empty history describes a scenario. However, we may read o like a Petri net transition: o's history denotes its preconditions, and o's contribution denotes what happens when o occurs. This observation leads to a canonical *composition operator* on oclets by which we can "plug" oclets together as illustrated in Figure 5.2. The resulting oclet o_{14} denotes the distributed run that follows from consecutive occurrences of the oclets o_1-o_4.

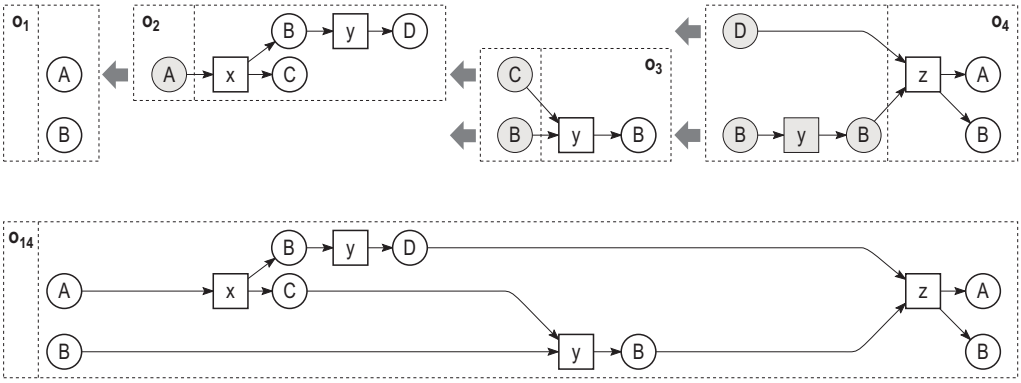


Figure 5.2. Composing oclets along their histories yields a distributed run.

Section 5.2 defines this composition of oclets and proves basic properties. In Section 5.3, we prove that composition of oclets is *equivalent* to the semantics of oclet specifications. Then we show how to

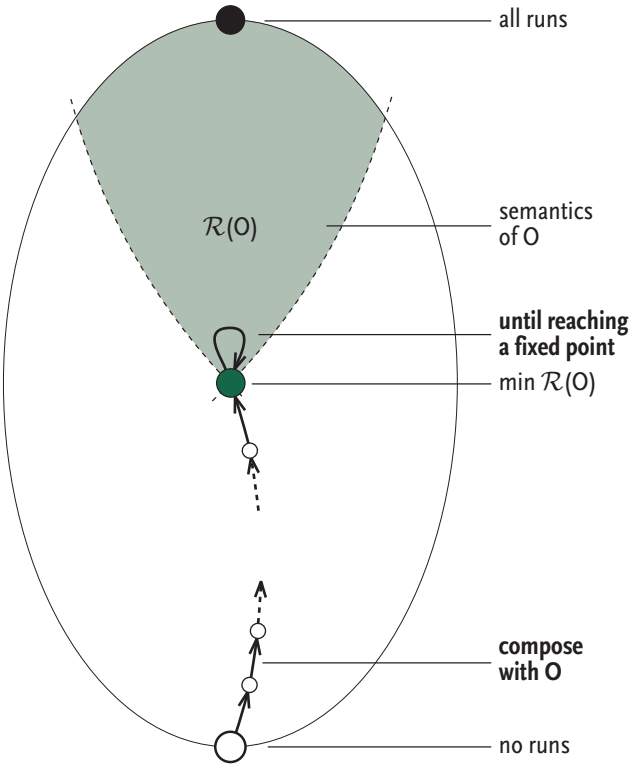


Figure 5.3. Main result of this chapter: a composition operator on oclets that constructs the unique minimal set R(O) that satisfies an oclet specification O.

> *construct the unique minimal set of runs* $\min \mathcal{R}(O)$ *that satisfies* O*.*

The constructions begins with the ε-oclets of O (describing a set of initial runs). Composing these with each oclet of O extends the given set of runs with more, longer runs. Iterating the composition step eventually reaches a fixed point $R(O)$ s.t. no more run can be added to $R(O)$ by composition as illustrated in Figure 5.3. We prove in Section 5.4 that $R(O) = \min \mathcal{R}(O)$. We conclude this chapter in Section 5.5 with an important result regarding the expressive power of oclets.

> *For every Petri net system* Σ*, there exists an equivalent oclet specification* O *that describes the behavior of* Σ*.*

In other words, each distributed system that can be described by a Petri net, can also be specified with oclets. We then show that there are specifications O which cannot be implemented in a distributed manner without extending behavior: any implementation of O inevitably exhibits more behaviors than $\min \mathcal{R}(O)$.

In Chapter 6, we extend composition of oclets to an *operational semantics*. This operational semantics constructs the least behavior that satisfies O *and*

can be implemented by a distributed system. Altogether, these results lay the foundation for an algorithmic synthesis of a distributed system from a scenario-based specification from Chapter 7 onwards.

5.2. Composing Oclets

5.2.1. The idea

This section introduces a composition operator $\triangleright$ on oclets for constructing a distributed run from an oclet specification. Technically, this operator appends an oclet o_2 to an oclet o_1 which results in a larger oclet $o_1 \triangleright o_2$.

The central idea for the composition of oclets is to generalize the constructive semantics of Petri nets from Section 2.4.1 to oclets. Depending on its history, an oclet takes a different role.

- A ε-oclet which has an empty history generalizes a *distributed run* because both notions describe the same behavioral information.
- An oclet with a non-empty history generalizes a Petri net transition as follows. We say that an oclet o_2 is *enabled* at an oclet o_1 if the entire history of o_2 occurs at the end of o_1. We *compose* o_1 with an enabled oclet o_2 to $o_1 \triangleright o_2$ by appending o_2's entire contribution to o_1.

Figure 5.2 illustrates these notions by an example. Composing $\mathsf{o}_1 \triangleright \mathsf{o}_2 \triangleright \mathsf{o}_3 \triangleright \mathsf{o}_4$ results in oclet o_{14}.

5.2.2. The definitions

This section provides the formal definitions for composing oclets. Oclet composition follows from a binary *partial* operator $\triangleright : \mathit{Oclets} \times \mathit{Oclets} \nrightarrow \mathit{Oclets}$. We first define a valid composition $o_1 \triangleright o_2$ of two given oclets o_1 and o_2. It is basically the union of o_1 and o_2 under the assumption that both oclets overlap "in the right way" (hence, the partiality of $\triangleright$). Afterwards, we generalize this definition to *sets of oclets* and *oclet classes*.

Definition 5.1 (Enabled oclet). Let $o_1 = (\pi_1, \mathit{hist}_1)$ and $o_2 = (\pi_2, \mathit{hist}_2)$ be two oclets. o_2 is *enabled* at o_1, written $o_2 \in \mathit{enabled}(o_1)$, iff

1. $\pi_1 \xrightarrow{\mathit{hist}_2} \rceil$ (i.e., π_1 ends with hist_2, so $\mathit{hist}_2 \subseteq \pi_1$ and $\max \mathit{hist}_2 \subseteq \max \pi_1$, Def. 4.4), and
2. $\pi_1 \cap \mathit{con}_2 = \emptyset$ (i.e., o_2's contribution is disjoint from o_1) ⌟

The second condition in this definition ensures that o_1 and o_2 together do not form a cycle. Enabled oclets define the *domain* of oclet composition.

Definition 5.2 (Oclet composition). Let *Oclets* denote the universe of all oclets. We define the partial operator $\triangleright : \mathit{Oclets} \times \mathit{Oclets} \nrightarrow \mathit{Oclets}$ with the *domain* $\mathit{dom}(\triangleright)$ s.t. $(o_1, o_2) \in \mathit{dom}(\triangleright)$ iff o_2 is enabled at o_1. For all $o_1 = (\pi_1, \mathit{hist}_1), o_2 = (\pi_2, \mathit{hist}_2)$ with $(o_1, o_2) \in \mathit{dom}(\triangleright)$, we define the *composition of* o_1 *with* o_2 as $o_1 \triangleright o_2 := (\pi_1 \cup \pi_2, \mathit{hist}_1)$. ⌟

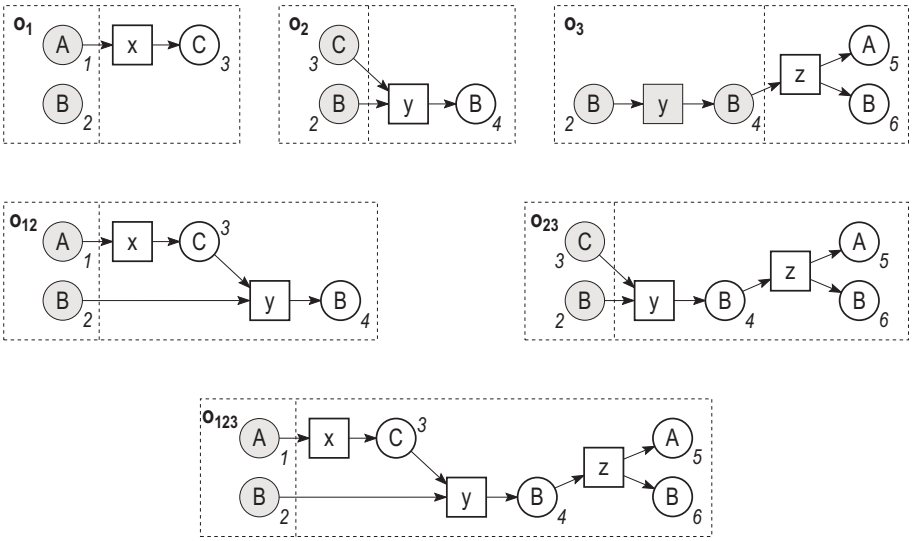


Figure 5.4. Composition of oclets.

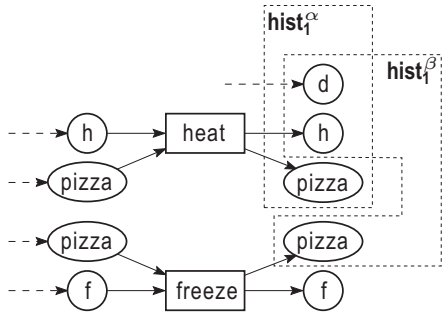
Composing two oclets $o_1 \rhd o_2$ appends the contribution of o_2 to o_1 while the history of o_2 is already part of o_1.

Figure 5.4 illustrates oclet composition by an example. Oclet o_2 is enabled at oclet o_1 and o_3 is enabled at o_2. Oclet o_3 is *not* enabled at o_1 because not all of o_3's history occurs at the end of o_1. The composition $\mathsf{o}_1 \rhd \mathsf{o}_2$ is the oclet o_{12} and the composition $\mathsf{o}_2 \rhd \mathsf{o}_3$ is the oclet o_{23}. Oclet o_3 is enabled at o_{12} and oclet o_{23} is enabled at o_1. We get $\mathsf{o}_{12} \rhd \mathsf{o}_3 = \mathsf{o}_1 \rhd \mathsf{o}_{23} = \mathsf{o}_{123}$.

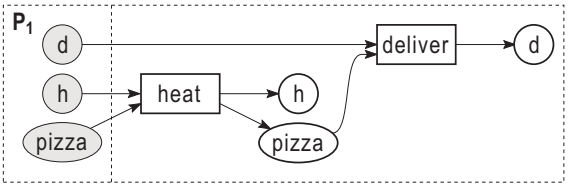
A special case: composing a distributed run with an oclet. In the following, we will frequently compose an ε-oclet, which represents a distributed run, with an oclet.

To simplify notations, we fix the following notational convention. For an arbitrary distributed run π, we also write $\pi := (\pi, \varepsilon)$ for the ε-oclet that represents π. If o is an oclet that is enabled at the ε-oclet π, then $\rho = \pi \rhd o$ denotes the composition of π with o. ρ is an ε-oclet because the first operand π determines the history of ρ (see Definition 5.2). Mildly abusing notation, we read ρ as a distributed run (instead of an ε-oclet) whenever confusion between both notions is safely avoided.

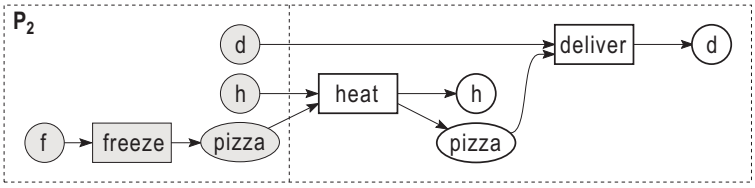
Composition and oclet classes. Chapter 4 introduced an oclet o as a *semantic notion* consisting of concrete events and conditions in a partial order. The corresponding *syntactic notion* is its *oclet class* $[o]$ which denotes all oclets that are isomorphic to o. We strictly distinguish syntax and semantics: two *concrete oclets compose strictly by the identity of their events and conditions*. For instance in Figure 5.4, oclet o_1 is *not* enabled at o_3; the small numbers indicate that hist_1 does *not* occur at then end of o_3. To compose o_1 and o_1 as suggested by their syntax, we generalize oclet composition to oclet classes (which are sets of oclets) in the canonical way.



(a) distributed run π_1



(b) oclet P_1



(c) oclet P_2

Figure 5.5. Oclet P_1 is enabled at the run π_1 at two locations α and β; oclet P_2 is enabled at π_1 only at the lower location β.

Definition 5.3 (Composition of sets of oclets). Let O_1 and O_2 be two sets of oclets. The *composition of* O_1 *with* O_2 is the set $O_1 \rhd O_2 := \{o_1 \rhd o_2 \mid o_1 \in O_1, o_2 \in O_2, (o_1, o_2) \in dom(\rhd)\}$. As additional notation, we say that O_2 *is enabled at* O_1 iff there exist $o_1 \in O_1, o_2 \in O_2$ s.t. o_2 is enabled at o_1. ⌟

Composition of sets of oclets is a *total operation* $\rhd : 2^{Oclets} \times 2^{Oclets} \to 2^{Oclets}$. Though, $O_1 \rhd O_2$ is empty if O_2 is not enabled at O_1. This notion allows to *compose two oclet classes* $[o_1]$ and $[o_2]$. The composition $[o_1] \rhd [o_2]$ denotes the set of all valid compositions $o_1^\alpha \rhd o_2^\beta$ of all oclets $o_1^\alpha \in [o_1], o_2^\beta \in [o_2]$. Correspondingly, $[o_2]$ *is enabled at* $[o_1]$ iff there exists an oclet $o_2^\beta \in [o_2]$ s.t. o_2^β is enabled at o_1.

For example, the history of oclet P_1 of Figure 5.5 occurs in π_1 at two different locations α and β (see Def. 4.2) — both locations are highlighted. Thus, the composition $[\pi_1] \rhd [\mathsf{P}_1]$ is the *union* of the two oclet classes $[\pi_2] \cup [\pi_3]$ shown in Figure 5.6. In other words, composition of oclet classes is *nondeterministic* producing *all possible compositions*.

In contrast, *composition of concrete oclets allows to determine the result of the composition*. For this reason, we refine our notions of oclet composition to consider a specific occurrence $o_2^\alpha \in [o_2]$ as introduced in Section 4.3.1. We say for two given oclets o_1 and o_2 that o_2 *is enabled at* o_1 *at location* α iff oclet $o_2^\alpha \in [o_2]$ is enabled at o_1. Then, the *composition of* o_1 *with* o_2 *at location* α is $o_1 \rhd o_2^\alpha$. This notational convention completes our definitions for composing oclets.

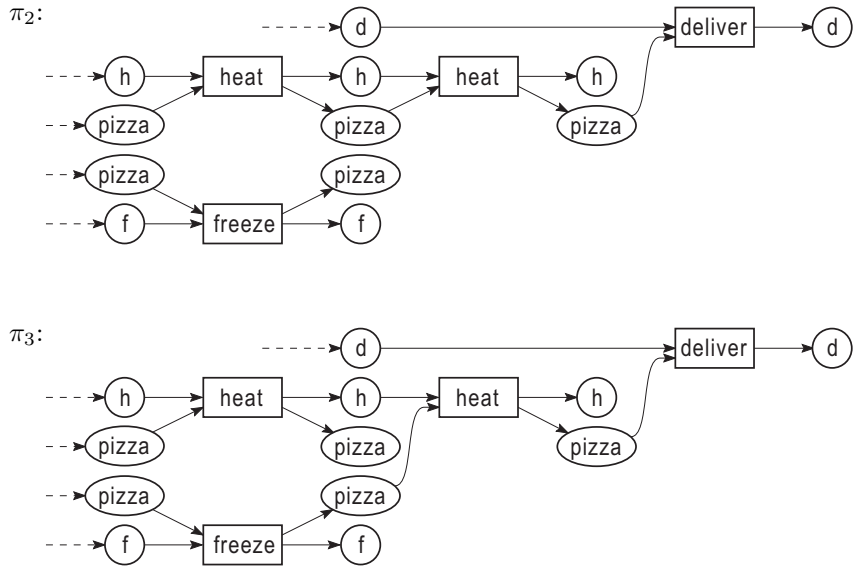


Figure 5.6. Two distributed runs obtained by composing π_1 of Fig. 5.5 with oclets P_1 and P_2: $\pi_2 = \pi_1 \rhd \mathsf{P}_1^{\alpha}$, $\pi_3 = \pi_1 \rhd \mathsf{P}_1^{\beta}$, $\pi_3 = \pi_1 \rhd \mathsf{P}_2^{\beta}$.

For example, Oclet P_1 shown in Figure 5.5 is enabled at the run π_1 at two locations α and β. Oclet P_2 is enabled at π_1 only at β. Composing π_1 with P_1 at location α yields the distributed run $\pi_2 = \pi_1 \rhd \mathsf{P}_1^{\alpha}$ of Figure 5.6; composing π_1 with P_1 at location β yields π_3. We can construct π_3 also by composing π_1 with P_2 at location β, i.e., $\pi_3 = \pi_1 \rhd \mathsf{P}_2{}^{\beta}$.

5.2.3. Properties of the composition

We close this section by stating and proving some basic properties of oclet composition.

Oclets are closed under composition

We introduced $\rhd : \mathit{Oclets} \times \mathit{Oclets} \nrightarrow \mathit{Oclets}$ as a partial composition operator on oclets which lifts to a total composition operator $\rhd : 2^{\mathit{Oclets}} \times 2^{\mathit{Oclets}} \to 2^{\mathit{Oclets}}$ on sets of oclets. The following lemma claims that the composition of two oclets is indeed an oclet again.

Lemma 5.4: *Let o_1 and o_2 be two oclets s.t. o_2 is enabled at o_1. Then $o_1 \rhd o_2$ is an oclet.* ⋆

The proof of this lemma relies on the following two technical lemmas on distributed runs. Figure 5.7 illustrates their propositions abstractly.

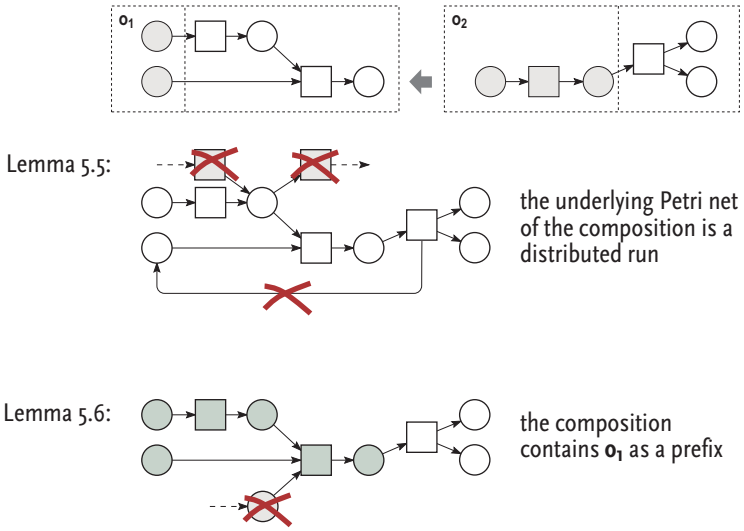


Figure 5.7. Illustration of Lemmas 5.5 and 5.6.

Lemma 5.5: *Let $o_1 = (\pi_1, hist_1)$ and $o_2 = (\pi_2, hist_2)$ be two oclets s.t. o_2 is enabled at o_1. Then the Petri net $\pi_1 \cup \pi_2$ that underlies the composition $o_1 \rhd o_2$ is a distributed run.* ⋆

To prove Lemma 5.5, we have to show that $\pi_1 \cup \pi_2$ is a causal net along the lines of Definition 2.6. The corresponding formal proof is rather technical and is given in Appendix A.7.

Lemma 5.6: *Let $o_1 = (\pi_1, hist_1)$ and $o_2 = (\pi_2, hist_2)$ be two oclets s.t. o_2 is enabled at o_1. Then the distributed run that underlies o_1 is a prefix of the composition $o_1 \rhd o_2$, i.e., $\pi_1 \sqsubseteq (\pi_1 \cup \pi_2)$.* ⋆

This lemma's proposition is shown along the lines of Definition 2.10. The proof is straight forward and given in Appendix A.7.

Proof (of Lemma 5.4). Let $o_1 = (\pi_1, hist_1)$ and $o_2 = (\pi_2, hist_2)$ be two oclets s.t. o_2 is enabled at o_1. Let $o_{12} := o_1 \rhd o_2 = (\pi_1 \cup \pi_2, hist_1)$ be their composition. We have to show that o_{12} is an oclet: (1) $\pi_1 \cup \pi_2$ is a distributed run, (2) $hist_1$ is a prefix of $\pi_1 \cup \pi_2$, and (3) either $hist_1 = \varepsilon$ or $\min(\pi_1 \cup \pi_2) = \min hist_1$ (by Definition 4.1).

(1) From Lemma 5.5 follows that $\pi_1 \cup \pi_2$ is a distributed run.

(2) From Lemma 5.6 follows that π_1 is a prefix of $\pi_1 \cup \pi_2$. Because o_1 is an oclet, $hist_1$ is a prefix of π_1: $hist_1 \sqsubseteq \pi_1 \sqsubseteq (\pi_1 \cup \pi_2)$. It follows straight from Lemma 2.11 that $hist_1 \sqsubseteq (\pi_1 \cup \pi_2)$.

(3) If $hist_1 = \varepsilon$, then the proposition holds trivially. If $hist_1 \neq \varepsilon$, then $\min \pi_1 = \min hist_1$ because o_1 is an oclet. From o_2 being enabled at o_1 follows: $\varepsilon \neq hist_2 \subseteq \pi_1$

and $\min \pi_2 = \min hist_2 \subseteq X_1$. Thus, π_1 and $\pi_1 \cup \pi_2$ have the same minimal nodes. We conclude $\min(\pi_1 \cup \pi_2) = \min \pi_1 = \min hist_1$, and the proposition holds. □

Oclet composition is associative

The following lemmas are central to calculating with oclets; they tell that composing oclets is associative in both the partial composition operator $\rhd : Oclets \times Oclets \nrightarrow Oclets$ and the total composition operator $\rhd : 2^{Oclets} \times 2^{Oclets} \to 2^{Oclets}$

Instead of first composing an oclet o_1 with an oclet o_2, and then composing o_3 with the result, we may also first compose o_2 and o_3 to a larger oclet o_{23}, and compose o_{23} with o_1. The resulting oclet will be the same. Figure 5.4 already illustrated an example.

Lemma 5.7 (Oclet class composition is associative): *Let $[o_1]$, $[o_2]$, $[o_3]$ be oclet classes with $[o_2]$ being enabled at $[o_1]$, and $[o_3]$ being enabled at $[o_2]$. Then*

1. *$[o_3]$ is enabled at $([o_1] \rhd [o_2])$ and $([o_2] \rhd [o_3])$ is enabled at $[o_1]$, and*
2. *$([o_1] \rhd [o_2]) \rhd [o_3] = [o_1] \rhd ([o_2] \rhd [o_3])$* ⋆

Basically the same propositions hold for the composition of concrete oclets — up to isomorphism.

Lemma 5.8 (Oclet composition is associative): *Let o_1, o_2, o_3 be oclets with o_2 being enabled at o_1, and o_3 being enabled at o_2. Then there exist oclets $o_i' \in [o_i], i = 1, 2, 3$ s.t.*

1. *o_3' is enabled at $(o_1' \rhd o_2')$ and $(o_2' \rhd o_3')$ is enabled at o_1',*
2. *$(o_1' \rhd o_2') \rhd o_3' = o_1' \rhd (o_2' \rhd o_3')$, and*
3. *$(o_1' \rhd o_2') \rhd o_3' \in ([o_1] \rhd [o_2]) \rhd [o_3]$.* ⋆

In case the concrete oclets are all chosen right, we obtain the following corollary that helps us in forthcoming proofs.

Corollary 5.9: *Let o, o_1, o_2, o_3 be oclets s.t. $(o_1 \rhd o_2) \rhd o_3 \in [o]$. Then $(o_1 \rhd o_2) \rhd o_3 = o_1 \rhd (o_2 \rhd o_3)$.* ⋆

The proofs. In the remainder of this section, we prove the associativity of oclet composition. The convinced reader may continue at Section 5.3.

We first prove the propositions of Lemma 5.8; Lemma 5.7 then follows by generalization to oclet classes. Lemma 5.8 essentially holds by the following technical lemma. It states that only the third operand o_3 has to be chosen right to ensure a valid composition $o_1 \rhd o_2 \rhd o_3$.

Lemma 5.10: *Let o_1, o_2, o_3 be oclets with o_2 enabled at o_1 and o_3 enabled at o_2 s.t. additionally $con_3 \cap \pi_1 = \emptyset$. Then*

1. *o_3 is enabled at $(o_1 \rhd o_2)$ and $(o_2 \rhd o_3)$ is enabled at o_1, and*
2. *$(o_1 \rhd o_2) \rhd o_3 = o_1 \rhd (o_2 \rhd o_3)$.* ⋆

Proof. (*1.*) First, we show that o_3 is enabled at $(o_1 \rhd o_2)$. Let $o^* = (\pi^*, hist^*) := o_1 \rhd o_2$. From o_3 being enabled at o_2 follows:

- $con_3 \cap \pi_2 = \emptyset$ (by Def. 5.1) and $con_3 \cap \pi_1 = \emptyset$ (by assumption). Thus, $con_3 \cap \pi^* = con_3 \cap (\pi_1 \cup \pi_2) = \emptyset$.
- $hist_3 \subseteq \pi_2 \subseteq \pi_1 \cup \pi_2 = \pi^*$
- $\max hist_3 \subseteq \max \pi_2$ by o_3 being enabled at o_2. Further, $\max \pi_2 \subseteq \max \pi^*$ by the following two cases. Let $b \in \max \pi_2$.
  (i) $b \in con_2$. Then $b \notin \pi_1$. Thus, $\nexists (b, e) \in F_1$ (by $b \notin \pi_1$), $\nexists (b, e) \in F_2$ (by $b \in \max \pi_2$), hence $\nexists (b, e) \in F_1 \cup F_2 = F^*$.
  (ii) $b \notin con_2$. Then $b \in \max hist_2 \subseteq \max \pi_1$ because o_2 is enabled at o_1. Thus, $\nexists (b, e) \in F_1$ (by $b \in \max \pi_1$), $\nexists (b, e) \in F_2$ (by $b \in \max \pi_2$), hence $\nexists (b, e) \in F_1 \cup F_2 = F^*$.

  In either case there exists no arc $(b, e) \in F^*$, thus $b \in \max \pi^*$. Thus, $\max hist_3 \subseteq \max \pi_2 \subseteq \max \pi^*$.

Altogether, $hist_3 \subseteq \pi_1$ and $\max hist_3 \subseteq \max \pi^*$ are equivalent to o_3 being enabled at $o^* := (\pi_1 \cup \pi_2, hist_1)$.

The second proposition, that $(o_2 \rhd o_3)$ is enabled at o_1, holds as follows: $(o_2 \rhd o_3) = (\pi_2 \cup \pi_3, hist_2)$. From o_2 being enabled at o_1 follows $hist_2 \subseteq \pi_1$ and $\max hist_2 \subseteq \max \pi_1$. Thus, o_1 ends with the history of $(o_2 \rhd o_3)$ which implies the proposition.

(*2.*) Firstly, $(o_1 \rhd o_2) \rhd o_3 = ((\pi_1 \cup \pi_2), hist_1) \rhd o_3 = ((\pi_1 \cup \pi_2 \cup \pi_3), hist_1)$. Likewise, $o_1 \rhd (o_2 \rhd o_3) = (\pi_1, hist_1) \rhd ((\pi_2 \cup \pi_3), hist_2) = ((\pi_1 \cup \pi_2 \cup \pi_3), hist_1)$. □

Proving Lemma 5.8 is now straight forward.

Proof (Proof of Lemma 5.8). The first two propositions follow from Lemma 5.8 by choosing $o_1' = o_1$, $o_2' = o_2$, and $o_3' \in [o_3]$ s.t. $con_3' \cap \pi_1' = \emptyset$. By Def. 5.3 holds $(o_1' \rhd o_2') \in ([o_1] \rhd [o_2])$. By the same argument follows $(o_1' \rhd o_2') \rhd o_3' \in ([o_1] \rhd [o_2]) \rhd [o_3]$. □

Proof (Proof of Lemma 5.7). Let o_1, o_2, o_3 be oclets. Let $[o_2]$ be enabled at $[o_1]$ and $[o_3]$ be enabled at $[o_2]$.

(*1.*) There exist oclets $o_i' \in [o_i], i = 1, 2$ s.t. o_2' is enabled at o_1' and $o_i'' \in [o_i], i = 2, 3$ s.t. o_3'' is enabled at o_2'' by Def. 5.3. Because $[o_3]$ contains all oclets that are isomorphic to o_3, we may choose $o_3' \in [o_3]$ s.t. o_3' is enabled at o_2'. From Lemma 5.8 follows $[o_3']$ is enabled at $([o_1'] \rhd [o_2'])$ and $([o_2'] \rhd [o_3'])$ is enabled at $[o_1']$. This implies the first proposition by $[o_i'] = [o_i]$, for $i = 1, 2, 3$.

(*2.*) Associativity of oclet class composition follows from Definition 5.3 and Lemma 5.8. $o \in (([o_1] \rhd [o_2]) \rhd [o_3])$
iff $o = o_{12}' \rhd o_3'$ and $o_{12}' \in ([o_1] \rhd [o_2])$ and $o_3' \in [o_3]$ and $(o_{12}', o_3') \in dom(\rhd)$
iff $o = ((o_1' \rhd o_2') \rhd o_3')$ and $o_i' \in [o_i], i = 1, 2, 3$ and $(o_1', o_2'), (o_2', o_3') \in dom(\rhd)$
[by the lemma's assumption and the $[o_i]$ being oclet classes]
iff $o = (o_1' \rhd (o_2' \rhd o_3'))$ and $o_i' \in [o_i], i = 1, 2, 3$ and $(o_1', o_2'), (o_2', o_3') \in dom(\rhd)$

[by Lemma 5.8]
iff $o = (o'_1 \rhd o'_{23})$ and $o'_1 \in [o_1]$ and $o'_{23} \in ([o_2] \rhd [o_3])$ and $(o'_1, o'_{23}) \in dom(\rhd)$
iff $o \in [o_1] \rhd ([o_2] \rhd [o_3])$. □

Deterministic composition of oclets

The example of Figures 5.5 and 5.6 has shown that composition of two oclet classes $[o_1]$ and $[o_2]$ is non-deterministic and may yield two different results $o_1 \rhd o_2^\alpha, o_1 \rhd o_2^\beta \in [o_1] \rhd [o_2]$. This happens whenever o_1 and o_2 can be composed at different locations α and β. The following lemma formally proves that if o_1 and o_2 can be composed in only one way, then composition of oclet classes is deterministic. In this case, $[o_1] \rhd [o_2]$ describes exactly what we would intuitively write as $o_1 \rhd o_2$ (without referring to any location α).

Lemma 5.11:

1. *Let O_1 and O_2 be sets of oclets. Then $O_1 \rhd O_2 \neq \emptyset$ iff O_2 is enabled at O_1.*
2. *Let o_1 and o_2 be oclets. If $[o_2]$ is enabled at $[o_1]$ then there exists an oclet $o_2^\alpha \in [o_2]$ s.t. o_2^α is enabled at o_1 and $o_1 \rhd o_2^\alpha \in [o_1] \rhd [o_2]$.*
3. *Let o_1 and o_2 be oclets s.t. o_2's history occurs exactly once at the end of o_1, i.e., for all $hist_2^\alpha, hist_2^\beta \in [hist_2]$ with $\pi_1 \xrightarrow{hist_2{}^\alpha}$ and $\pi_1 \xrightarrow{hist_2{}^\beta}$ holds $hist_2^\alpha = hist_2^\beta$. Then $[o_1] \rhd [o_2]$ is an oclet class and there exists $o_2^\alpha \in [o_2]$ s.t. $[o_1 \rhd o_2^\alpha] = [o_1] \rhd [o_2]$.* ⋆

Proof. (*1.*) Holds by Definition 5.3 as follows. (⇐) Let O_1 and O_2 be sets of oclets. O_2 is enabled at O_1 iff there exist $o_1 \in O_2, o_2 \in O_2$ s.t. o_2 is enabled at o_1, by Def. 5.3. Thus, $(o_1 \rhd o_2) \in O_1 \rhd O_2$. (⇒) If $o \in O_1 \rhd O_2$, then there exist $o_1 \in O_1, o_2 \in O_2$ with $o = o_1 \rhd o_2$ and o_2 enabled at o_1. Thus, O_2 is enabled at O_1.

(*2.*) Let $[o_2]$ be enabled at $[o_1]$. From (1) follows that there exist $o_1^\alpha \in [o_1], o_2^\beta \in [o_2]$ s.t. $o_2^\beta \in [o_2]$ is enabled at o_1^α. Because $[o_1]$ and $[o_2]$ contain all oclets that are isomorphic to o_1 and o_2 respectively, there also exist $o_1 \in [o_1], o_2^\beta \in [o_2]$ s.t. o_2^β is enabled at o_1. From Def. 5.3 follows $o_1 \rhd o_2^\beta \in [o_1] \rhd [o_2]$.

(*3.*) Let o_1 and o_2 be as assumed. Thus, for all oclets $o_2^\gamma, o_2^\delta \in [o_2]$ that are enabled at o_1 holds: $hist_2^\gamma = hist_2^\delta$ and $con_2^\gamma \cap \pi_1 = \emptyset$ and $con_2^\delta \cap \pi_1 = \emptyset$. Thus, $o_1 \rhd o_2^\gamma$ and $o_1 \rhd o_2^\delta$ are isomorphic by Def. 5.2. This argument applies wrt. all oclets $o_1^\alpha \in [o_1]$. Thus, for all oclets $o_1^\alpha, o_1^\beta \in [o_1], o_2^\gamma, o_2^\delta \in [o_2]$ s.t. o_2^γ enabled at o_1^α and o_2^δ enabled at o_1^β holds: $o_1^\alpha \rhd o_2^\gamma$ and $o_1^\beta \rhd o_2^\delta$ are isomorphic. Thus, all compositions in $[o_1] \rhd [o_2]$ are isomorphic. Further, $[o_1] \rhd [o_2]$ is closed under isomorphism: let $o = o_1^\alpha \rhd o_2^\beta \in [o_1] \rhd [o_2]$. Let $o^\gamma \in [o]$. Then $o^\gamma = o_1^{(\gamma \circ \alpha)} \rhd o_2^{(\gamma \circ \beta)}$ and $o_1^{(\gamma \circ \alpha)} \in [o_1]$ and $o_2^{(\gamma \circ \beta)} \in [o_2]$. Thus, $o^\gamma \in [o_1] \rhd [o_2]$ and we conclude that $[o_1] \rhd [o_2]$ is an oclet class. From (2) follows that there exists $o_2^\alpha \in [o_2]$ s.t. $o_1 \rhd o_2^\alpha \in ([o_1] \rhd [o_2])$. From $[o_1] \rhd [o_2]$ being an oclet class follows $[o_1 \rhd o_2^\alpha] = [o_1] \rhd [o_2]$. □

5.3. Oclet Composition Implements Oclet Semantics

The preceding section introduced oclet composition. Composing two oclets o_1 and o_2 appends o_2 to o_1 which results in a larger oclet $o_1 \rhd o_2$. In this section, we prove that oclet composition is *semantically relevant*. More precisely, we show that the semantics of oclets from Chapter 4 can equivalently be expressed by oclet composition.

5.3.1. The idea

The composition of a distributed run π with an oclet o *constructs* the run $\rho = \pi \rhd o$; we shall prove that ρ continues π with o as defined in Sect. 4.3, i.e., $\pi \xrightarrow{o} \rho$. This allows us to replace the semantic *characterization* when some run ρ *continues* π *with* o by the *constructive* notion $\pi \rhd o$.

As an example, the run π_3 of Figure 5.6 continues the run π_1 with oclet P_2 of Figure 5.5 on page 131. We also obtain π_3 by composition, i.e., $\pi_3 = \pi_1 \rhd \mathsf{P}_2^{\beta}$.

Next, we lift oclet composition to construct a *set* of distributed runs that satisfies each oclet o of a specification (Def. 4.6), as follows. Definition 5.3 lifted oclet composition to sets of oclets. For a given set R of runs and an oclet class $[o]$, the set $\mathit{Prefix}(R) \rhd [o]$ denotes the continuations $\pi \rhd o^{\alpha}$ for all prefixes π in R which end with o's history. Then, R is closed under composition with o if $\mathit{Prefix}(R) \rhd [o]$ is already contained in $\mathit{Prefix}(R)$.

Definition 5.12 (Oclet-closed). Let R be a set of distributed runs. Let $[o]$ be an oclet class with a nonempty history. The set R is *o-closed* iff $(\mathit{Prefix}(R) \rhd [o]) \subseteq \mathit{Prefix}(R)$.

Let $O^{\rhd}$ be an oclet specification of oclet classes with nonempty history. The set R of runs is *$O^{\rhd}$-closed*, denoted $(\mathit{Prefix}(R) \rhd O^{\rhd}) \subseteq \mathit{Prefix}(R)$, iff $(\mathit{Prefix}(R) \rhd [o]) \subseteq \mathit{Prefix}(R)$, for each $[o] \in O^{\rhd}$. ⌟

In the following, we will always write $O^{\rhd}$ for an oclet specification consisting only of oclets with a non-empty history. An $O^{\rhd}$-closed set R of distributed runs denotes a *fixed point* of the oclet composition $\rhd$. For each run $\pi \in R$ and each oclet class $[o] \in O^{\rhd}$ s.t. an oclet $o' \in [o]$ is enabled at π, the composition $\pi \rhd o'$ is also a run in R. Because $\pi \rhd o'$ denotes a run that continues π with o', we conclude that an $O^{\rhd}$-closed R is *closed under continuations* with each oclet class in $O^{\rhd}$, that is R satisfies each $[o] \in O^{\rhd}$. Altogether, we obtain the following theorem about the relation between oclet composition and oclet semantics.

Theorem 5.13 (Constructive characterization of oclet semantics). *Let $O^{\rhd}$ be a set of oclets with non-empty history. A set R of distributed runs satisfies $O^{\rhd}$ iff R is $O^{\rhd}$-closed.* ⋆

This theorem allows to rephrase the semantics of oclets in terms of their composition. We specifically use this theorem in Section 5.4 to *construct* the unique minimal set $\min \mathcal{R}(O)$ of runs that satisfies an oclet specification O. Constructing this behavior is the basis for algorithmically synthesizing an implementation from a scenario-based specification in Chapter 8. The remainder of this section is devoted to proving Theorem 5.13. The convinced reader may continue at Section 5.4.

5.3.2. Continuing a run by oclet composition

We prove Theorem 5.13 by first proving that the notions $\pi \xrightarrow{o} \rho$ (run ρ continues run π with oclet o) and $\pi \rhd o$ (composition of π with o) are equivalent. Section 5.3.3 lifts this equivalence to sets of runs.

Composition is correct

We begin by proving that the composition $\pi \rhd o$ continues π with o. In other words, the composition of oclets is *correct* wrt. the semantics of oclets; see Section 4.3

Lemma 5.14 (Oclet composition is correct): *Let π be a distributed run and let o be an oclet that is enabled at π. Then $\pi \xrightarrow{o} (\pi \rhd o)$.* ⋆

Proof. Let $(\rho, \varepsilon) := \pi \rhd o$, thus $\rho = (\pi \cup \pi_o)$. We prove $\pi \xrightarrow{o} \rho$ by proving (1) $\pi \xrightarrow{hist_o}]$, (2) $o \subseteq_{\mathbf{c}} \rho$, and (3) $\pi \sqsubseteq \rho$ according to Def. 4.5.

(1) $\pi \xrightarrow{hist_o}]$ holds by the assumption that o is enabled at π, Def. 5.1.

(2) $\rho = (P_\pi \cup P_o, T_\pi \cup T_o, F_\pi \cup F_o)$ by Def. 5.2. Thus, $P_o \subseteq P_\rho \wedge T_o \subseteq T_\rho \wedge F_o \subseteq F_\rho$ which is equivalent to $\pi_o \subseteq \rho$, by Def. 2.4. Moreover, $con_o \cap \pi = \emptyset$ because o enabled at π, Def 5.1. Thus, for all $e \in E_{con_o}$ holds $pre_\rho(e) = pre_\pi(e) \cup pre_o(e) = pre_o(e)$. Correspondingly, $post_\rho(e) = post_o(e)$. Thus $o \subseteq_{\mathbf{c}} \rho$ by Def. 4.3.

(3) $\pi \sqsubseteq \rho = \pi \rhd o$ holds by Lem. 5.6. □

Composition is complete

The converse of Lemma 5.14 is more involved. We have to show that whenever a run ρ continues a run π with an oclet o, i.e., $\pi \xrightarrow{o} \rho$, the longer run ρ begins with the composition $\pi \rhd o$. In other words, oclet composition is *complete* wrt. the semantics of oclets.

Lemma 5.15 (Oclet composition is complete): *Let o be an oclet with non-empty history. Let π and ρ be distributed runs s.t. $\pi \xrightarrow{o} \rho$. Then $\pi \rhd o$ is a prefix of ρ.* ⋆

The proof of this lemma is technically involved because we have to show that $\pi \rhd o$ is a prefix of *every possible* continuation ρ. To this end, we apply Definition 2.10 (Prefix).

Proof. We first have to show that we are allowed compose π with o, i.e., that o is enabled at π. Let $o = (\pi_o, hist_o)$. Definition 4.5, $\pi \xrightarrow{o} \rho$ implies that π ends with $hist_o$, i.e., $hist_o \subseteq \pi$ and $\max hist_o \subseteq \max \pi$. Because $hist_o$ is a prefix of π_o, we conclude that con_o and π are disjoint, i.e., $con_o \cap \pi = \emptyset$. Thus, o is enabled at π by Definition 5.1.

We prove $(\pi \rhd o) \sqsubseteq \rho$ by applying Lemma 2.11. Let $(\pi_2, \varepsilon) := \pi \rhd o$, thus, $\pi_2 = \pi \cup \pi_o = (B_2, E_2, F_2)$. To show that $\pi \rhd o$ is a prefix of ρ, we have to show (1) $\pi_2 \subseteq \rho$, (2) $F_2 = F_\rho|_{X_\rho \times X_2}$, and (3) $\forall e \in E_2 : post_2(e) = post_\rho(e)$.

(1) The following holds: $\pi \subseteq \rho$ and $\pi_o \subseteq \rho$ by $\pi \xrightarrow{o} \rho$ (Def. 4.5). Thus, $\pi \cup \pi_o \subseteq \rho$ by Def. 2.4.

(2) We prove $F_2 = F_\rho|_{X_\rho \times X_2}$ in two steps. We first show $F_2 = F_\rho|_{X_2 \times X_2}$ by contradiction; then we prove that for each $(x, y) \in F_\rho$ with $y \in X_2$ holds $x \in X_2$.

Assume $(x, y) \in F_\rho \setminus F_2$ with $x, y \in X_2$. There are three cases:

1. $x, y \in X_\pi$. By $F_2 = F_\pi \cup F_o$ holds $(x, y) \notin F_\pi$. This directly contradicts $F_\pi = F_\rho|_{(X_\pi \times X_\pi)}$ of Def. 2.10 because $\pi \sqsubseteq \rho$ by $\pi \xrightarrow{o} \rho$ (Def. 4.5).
2. $y \in X_2 \setminus X_\pi$. Thus $y \in X_{con_o}$ by (2). From $\pi \xrightarrow{o} \rho$ follows $(x, y) \in F_\rho \Leftrightarrow (x, y) \in F_o$ (by Def. 4.5). This contradicts $(x, y) \in F_\rho \setminus F_2 = F_\rho \setminus (F_\pi \cup F_o)$.
3. $y \in X_\pi$ and $x \in X_2 \setminus X_\pi$. Thus $x \in X_{con_o}$ by (2). But from $\pi \xrightarrow{o} \rho$ follows $\pi \sqsubseteq \rho$ (Def. 4.5). Thus by Def. 2.10, for each $x \in y{\downarrow_\rho}$ holds $x \in X_\pi$ which contradicts the assumption $x \in X_2 \setminus X_\pi$ of this case.

Thus, $F_2 = F_\rho|_{X_2 \times X_2}$ holds. Now, let $(x, y) \in F_\rho$ and $y \in X_2 = X_\pi \cup X_{con_o}$. Thus, there are two cases:

1. $y \in X_\pi$. From $\pi \sqsubseteq \rho$ follows: $(x, y) \in F_\rho$ and $y \in X_\pi$ imply $x \in X_\pi$ (Def. 2.10). Thus, $x \in X_2$.
2. $y \in X_{con_o}$. From $\pi \xrightarrow{o} \rho$ follows $o \subseteq_{\mathbf{c}} \rho$ (Def. 4.5) which implies $(x, y) \in F_\rho \Leftrightarrow (x, y) \in F_o$ by Def. 4.3; thus, $x \in X_o \subseteq X_2$.

Thus, altogether $F_2 = F_\rho|_{X_2 \times X_2} = F_\rho|_{X_\rho \times X_2}$.

(3) We have to show: $\forall e \in E_2 : post_2(e) = post_\rho(e)$.

From $\pi \sqsubseteq \rho$ (by $\pi \xrightarrow{o} \rho$, Def. 4.5) and $\pi \sqsubseteq \pi_2$ (by Lem. 5.6) follows that for all $e \in E_\pi$ holds: $post_2(e) = post_\pi(e) = post_\rho(e)$. From $\pi \xrightarrow{o} \rho$ follows $o \subseteq_{\mathbf{c}} \rho$ (by Def. 4.5). Thus, for all $e \in E_2 \setminus E_\pi = E_{con_o}$ holds: $post_o(e) = post_\pi(e)$ by Def. 4.3. Hence the proposition holds. This concludes the proof that $\pi_2 = \pi \rhd o$ is a prefix of ρ. □

With Lemmas 5.14 and 5.15 we have proven that the composition of oclets is *correct* and *complete* wrt. the semantics of oclets defined in Section 4.3.

5.3.3. Closed under continuations by oclet composition

In the preceding Section 5.3.2, we proved that the composition $\rho = \pi \rhd o$ of a distributed run π with an oclet o equivalently expresses that ρ continues π with o as defined in the semantics of oclets. In the following, we lift this result to *sets of runs* and show that an o-closed set satisfies o.

By Definition 4.6, a set R of runs satisfies an class $[o]$ of oclets, written $R \models [o]$, if for each run $\pi \in R$ that ends with o's history exists a run $\rho \in R$ that continues π with o. The corresponding notion using oclet composition is an o-closed set of runs (Definition 5.12). R is o-closed, written $(\mathit{Prefix}(R) \rhd [o]) \subseteq \mathit{Prefix}(R)$, if for each prefix $\pi \in R$ where an oclet $o' \in [o]$ is enabled, also the composition $\pi \rhd o$ is a prefix in R. Both notions are equivalent.

Lemma 5.16: *Let $[o]$ be an oclet class with nonempty history and let R be a set of distributed runs. Then $R \models [o]$ iff $(Prefix(R) \rhd [o]) \subseteq Prefix(R)$.* ⋆

Proof. Let R be a set of distributed runs with $R \models [o]$. By Def. 4.6, this is equivalent to

$$\begin{aligned}
& \forall \pi \in Prefix(R)\ \forall hist_o^\beta: \\
& \qquad (\pi \xrightarrow{hist_o{}^\beta}] \Rightarrow \exists \rho \in R\ \exists o^\alpha : (hist_o^\alpha = hist_o^\beta \wedge \pi \xrightarrow{o^\alpha} \rho)) \\
\Leftrightarrow\ & \forall \pi \in Prefix(R)\ \forall hist_o^\beta: \\
& \qquad (\pi \xrightarrow{hist_o{}^\beta}] \Rightarrow \exists \rho \in R\ \exists o^\alpha : (hist_o^\alpha = hist_o^\beta \wedge (\pi \rhd o^\alpha) \sqsubseteq \rho)) \\
& \quad \textit{[}\Rightarrow \textit{ by Lem. 5.15, } \Leftarrow \textit{ by Lem. 5.14]} \\
\Leftrightarrow\ & \forall \pi \in Prefix(R)\ \forall o^\alpha : (o^\alpha \in enabled(\pi) \Rightarrow \exists \rho \in R : (\pi \rhd o^\alpha) \sqsubseteq \rho) \\
& \quad \textit{[by definition of enabled oclet and oclet composition; Defs. 5.1, 5.2]} \\
\Leftrightarrow\ & \forall \pi \in Prefix(R)\ \forall o^\alpha : (o^\alpha \in enabled(\pi) \Rightarrow \pi \rhd o^\alpha \in Prefix(R)) \\
& \quad \textit{[by definition of Prefix(R); Def. 2.13.]}
\end{aligned}$$

The last line is equivalent to $(Prefix(R) \rhd o) \subseteq Prefix(R)$ by Definition 5.12. □

With the equivalence of o-closed sets to sets that satisfy o, we have all arguments for proving Theorem 5.13. The theorem made the following claim: Let $O^\rhd$ be an oclet specification of oclets with history. A set R of distributed runs satisfies $O^\rhd$ iff R is $O^\rhd$-closed.

Proof (of Theorem 5.13). Let $O^\rhd$ be as assumed. Let R be a set of distributed runs. The following statements are equivalent.

- $R \in \mathcal{R}(O)$.
- R satisfies each $[o] \in O^\rhd$, by Def. 4.9.
- R is o-closed, for each $[o] \in O^\rhd$, by Lemma 5.16.
- R is $O^\rhd$-closed by Def. 5.12. □

With Theorem 5.13 we have proven that oclet composition is *correct* and *complete* in the semantics of oclets.

5.4. Constructing the Least Behavior that Satisfies a Specification

The preceding sections extended the model of oclets by a composition operator $\rhd$. We have proven that oclet composition is equivalent to the semantics of oclets. In the following, we use oclet composition to *construct* the *unique minimal* set $\min \mathcal{R}(O)$ of distributed runs that satisfies a given oclet specification O — by composing the oclets in O. This result will be central for synthesizing a system model Σ that implements O and exhibits a *minimal* behavior.

To construct $\min \mathcal{R}(O)$, we split O into its ε-oclets O_ε and all other oclets $O^\rhd$. Then, we first set an initial set of runs by $R(O) := O_\varepsilon$ and repeatedly compose $R(O) := R(O) \rhd [o]$ with each $[o] \in O^\rhd$ until reaching a fixed point. The reached fixed point $R(O)$ is $\min \mathcal{R}(O)$. Figure 5.8 illustrates our approach.

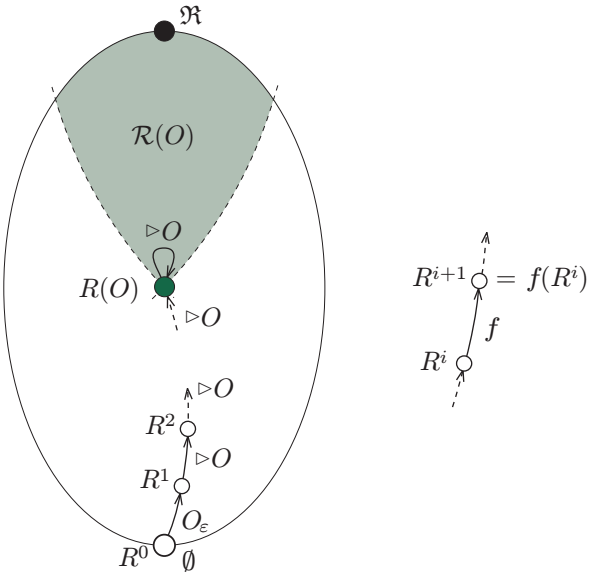


Figure 5.8. Composing oclets to the least behavior $R(O) = \min \mathcal{R}(O)$ that satisfies O.

Definition 5.17 (Construction of the least satisfying behavior). Let O be an oclet specification. Let $O_\varepsilon := \{[o] \in O \mid \mathit{hist}_o = \varepsilon\}$ and $O^\triangleright := \{[o] \in O \mid \mathit{hist}_o \neq \varepsilon\}$. We define a sequence $R^1, R^2, R^3, \ldots$ of sets of distributed runs as follows:

1. $R^1 := \mathit{Prefix}(O^*_\varepsilon)$ for the canonical representatives $O^*_\varepsilon = \{o \mid [o] \in O_\varepsilon\}$ of the oclet classes in O_ε.
2. $R^{i+1} := R^i \cup \mathit{Prefix}(\{R_i \triangleright O^\triangleright\})$, for all $i > 0$.

We define $R(O)$ as the least set of distributed runs containing each $R^i, i = 1, 2, \ldots$.⌟

Two remarks. There are two things notable about Definition 5.17. First, the construction of $R(O)$ is initialized by picking the canonical representative of each ε-oclet in O. For an explicit technical construction of canonical representatives see [38, 36]. Choosing canonical representatives is sound: oclet semantics (Def. 4.6 and 4.7) and minimal behavior (Def. 4.14) are defined only up to isomorphism. As a consequence, each run in $R(O)$ has a canonical beginning that is defined by a canonical representative $o \in O^*_\varepsilon$.

Second, R^1 assumes all ε-oclets of an oclet specification to be *alternative* initial runs. This is the weakest assumption that can be made about satisfying behavior and it yields minimality as the example in Figure 5.9 illustrates. The least set of runs that satisfies the specification $\{\mathsf{o}_1, \mathsf{o}_2, \mathsf{o}_3\}$ is the set $\{\pi_1, \pi_2, \pi_3\}$. Arguably, the set $\{\pi_4, \pi_3\}$ might be preferred over $\{\pi_1, \pi_2, \pi_3\}$. But π_1 and π_2 are prefixes of π_4, so $\{\pi_4, \pi_3\}$ is not minimal.

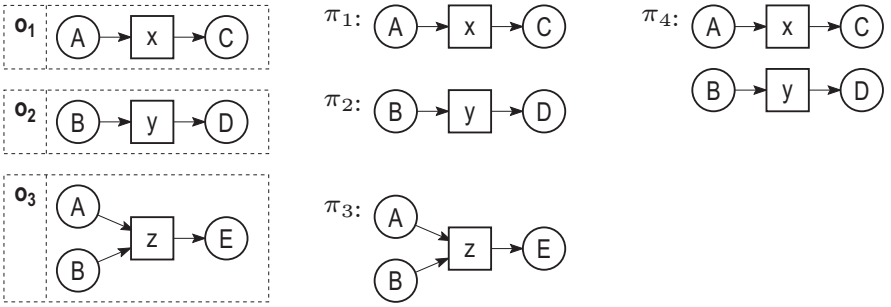


Figure 5.9. Oclets $\mathsf{o}_1, \mathsf{o}_2, \mathsf{o}_3$ occur as alternative runs in the least satisfying behavior $\{\pi_1, \pi_2, \pi_3\}$.

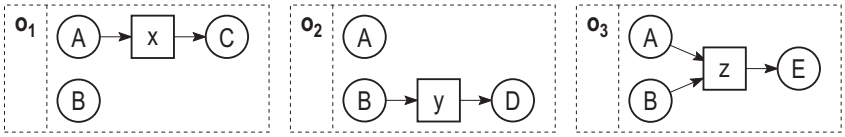


Figure 5.10. Oclets specifying a unique initial state.

This mismatch between intuition and formal definition may arise because the minimal set of runs $\{\pi_1, \pi_2, \pi_3\}$ has three *alternative initial states* [A], [B], and [A, B]. A system designer who wants to specify system behavior with a *unique initial state* can do so explicitly. For example, the oclets in Figure 5.10 specify a unique initial state. If we restricted Definition 5.17 to oclet specifications having a unique initial state, then our approach would be incomplete wrt. all possible specifications. Chapter 6 shows how derive the *least common initial state* of an oclet specification.

The proof. The following theorem states that Definition 5.17 yields the least set of runs that satisfies an oclet specification.

Theorem 5.18. *Let O be an oclet specification. The following properties hold:*

1. *$R(O)$ exists.*
2. *$R(O) \in \mathcal{R}(O)$ and $R(O) \in \mathcal{U}(O)$.*
3. *$R(O) = \min \mathcal{R}(O)$.* ⋆

Proof. Figure 5.8 sketches the strategy to prove that $R(O)$ is the least set of runs satisfying O. We prove the claims of this theorem by the help of the Knaster-Tarski Theorem (Thm. A.1), originally in [109]. We consider the power-set lattice $L = (2^{\mathfrak{R}}, \subseteq, \cup, \cap, \emptyset, \mathfrak{R})$ over all distributed runs $\mathfrak{R}$; see Sect. A.2. We show that the sequence $R^1, R^2, R^3, \ldots$ defined in Def. 5.17 follows from a monotone function f in L. According to Knaster-Tarski, this sequence has a least fixed point, which is $R(O)$.

(*1.*) We show that there exists a least fixed point R^* of the sets $R^1, R^2, R^3, \ldots$. First, define a function $g : 2^{\mathfrak{R}} \to 2^{\mathfrak{R}}$ with $g(R) := R \cup (R \rhd O^{\rhd})$.

Claim: g is monotone in L, i.e., $R \subseteq S \Rightarrow g(R) \subseteq g(S)$. Let $Y := S \setminus R$.

$$\begin{aligned} g(S) &= S \cup (S \rhd O^{\rhd}) \\ &= S \cup \{\pi \rhd o^{\alpha} \mid \pi \in S, [o] \in O^{\rhd}, o^{\alpha} \in [o], o^{\alpha} \in \mathit{enabled}(\pi)\} \\ &\quad \textit{[by Def. 5.12 and Def. 5.3]} \\ &= R \cup Y \cup \{\pi \rhd o^{\alpha} \mid \pi \in R, [o] \in O^{\rhd}, o^{\alpha} \in [o], o^{\alpha} \in \mathit{enabled}(\pi)\} \\ &\qquad \cup \{\pi \rhd o^{\alpha} \mid \pi \in Y, [o] \in O^{\rhd}, o^{\alpha} \in [o], o^{\alpha} \in \mathit{enabled}(\pi)\} \\ &= R \cup (R \rhd O^{\rhd}) \cup Y \cup (Y \rhd O^{\rhd}) \\ &\quad \textit{[by Def. 5.12 and Def. 5.3]} \\ &= g(R) \cup g(Y) \end{aligned}$$

Thus, $g(R) \subseteq g(R) \cup g(Y) = g(S)$ which implies that g is monotone. We lift g to the function $f : 2^{\mathfrak{R}} \to 2^{\mathfrak{R}}$ by $f(R) := \mathit{Prefix}(O^*_{\varepsilon} \cup g(R)) = \mathit{Prefix}(R \cup O^*_{\varepsilon} \cup (R \rhd O^{\rhd}))$. The function $f(.)$ is monotone because $g(.)$ and $\mathit{Prefix}(.)$ are. Set $R^0 := \emptyset$; so, for all sets $R^i, i > 0$ that are defined in Def. 5.17 holds: $R^i = f^i(\emptyset)$.

By Knaster-Tarski (Thm. A.1) holds: f has a least fixed point $R^* := \mathrm{LFP}(f)$ s.t. $f(R^*) \subseteq R^*$. Moreover, from Lem. A.2 follows that there exists a least ordinal λ s.t. $R^* = R^{\lambda} = R^{\lambda+1}$. Because f is monotone, $R^i \subseteq R^*$ for all i. Thus, R^* is the least fixed point as claimed in Def. 5.17, i.e., R^* exists.

(*2.*) Claim: $R^* \in \mathcal{R}(O)$ and $R^* \in \mathcal{U}(O)$. From $R^* := \mathrm{LFP}(f)$ and the definition of f follows:

$$\mathit{Prefix}(R^* \cup O^*_{\varepsilon} \cup (R^* \rhd O^{\rhd})) \subseteq R^* \tag{5.1}$$

By Def. 2.13 holds $(R^* \rhd O^{\rhd}) \subseteq \mathit{Prefix}(R^* \rhd O^{\rhd})$; furthermore $R^* = \mathit{Prefix}(R^*)$ by construction. Thus, (5.1) implies

$$(\mathit{Prefix}(R^*) \rhd O^{\rhd}) \subseteq \mathit{Prefix}(R^* \rhd O^{\rhd}) \quad \subseteq \quad R^* = \mathit{Prefix}(R^*), \text{ and} \tag{5.2}$$

$$O^*_{\varepsilon} \quad \subseteq \quad R^* = \mathit{Prefix}(R^*) \tag{5.3}$$

Equation (5.2) implies $(\mathit{Prefix}(R^*) \rhd O^{\rhd}) \subseteq \mathit{Prefix}(R^*)$. Thus, $R^* \in \mathcal{R}(O^{\rhd})$ by Thm. 5.13. Equation (5.3) implies that each ε-oclet of O occurs as a prefix in R^*, i.e., $\mathit{Prefix}(R^*) \models O_{\varepsilon}$. Thus, $R^* \in \mathcal{R}(O_{\varepsilon})$ by Def. 4.9. So, Cor. 4.11 implies $R^* \in \mathcal{R}(O^{\rhd} \cup O_{\varepsilon}) = \mathcal{R}(O)$. Moreover, R^* is O-covered (Def. 4.17) by construction. Thus, $R^* \in \mathcal{U}(O)$ by Def. 4.18.

(*3.*) It remains to show that R^* is minimal wrt. the semantics of oclets. To show $R^* = \min \mathcal{R}(O)$, we have to show that there exists no $R' \in \mathcal{R}(O)$ with $\mathit{Prefix}(R') \subset R^*$. According to Lem. 4.19, it is sufficient to consider prefix-closed sets R'.

- R^* is the least fixed point of f. Thus, there exists no $R' \subset R^*$ s.t. $O_{\varepsilon} \subseteq \mathit{Prefix}(R')$ and $(\mathit{Prefix}(R') \rhd O^{\rhd}) \subseteq \mathit{Prefix}(R')$.
- From Def. 4.7 and Thm. 5.13 follows: there exists no $R' \subset R^*$ s.t. $R' \in \mathcal{R}(O_{\varepsilon})$ and $R' \in \mathcal{R}(O^{\rhd})$.
- Cor. 4.11 implies: there exists no $R' \subset R^*$ s.t. $R' \in \mathcal{R}(O)$.

Altogether, the least fixed point $R(O) := R^*$ constructed according to Definition 5.17 is the least behavior $\min \mathcal{R}(O)$ that satisfies O, i.e., $R(O) = R^* = \min \mathcal{R}(O)$ up to isomorphism. □

5.5. Specifications vs. Distributed Systems

This section discusses the results of this chapter regarding the overall goal of synthesizing a distributed system Σ from a scenario-based specification O. We argued in the introduction of this chapter that the synthesized system Σ preferably exhibits the *least* behavior that satisfies O.

According to Theorem 5.18, Σ preferably exhibits exactly the runs $R(\Sigma) = R(O)$ because $R(O)$ is the least set of runs that satisfies O. Any smaller set $R' \subset R(O)$ does not satisfy O and any larger set $R' \supset R(O)$ contains more behaviors than necessary to satisfy O. In this section, we consider the relation between oclet specifications and distributed systems in more detail.

5.5.1. Petri nets are embedded in oclets

We first establish an important result regarding the expressive power of oclet specifications. Oclets are expressive enough to precisely specify the complete behavior of any Petri net system.

Lemma 5.19 (Petri nets are embedded in oclets): *For each Petri net system Σ exists an equivalent oclet specification O with $R(\Sigma) = R(O)$.* ⋆

Proof. We prove the proposition by constructing a specification O with the desired property. The oclets of O follow from the constructive semantics of Petri net systems presented in Section 2.4.1.

The constructive semantics of $\Sigma = (N, m_0)$ defines for each transition $t \in T_N$ a local t-step A_t. A_t is a distributed run consisting of one event e with its pre- and post-conditions (Def. 2.20). A_t induces the oclet $o_t := (A_t, hist_t)$ where $hist_t = ({}^\bullet e, \emptyset, \emptyset)$ is the prefix of A_t consisting of e's pre-conditions. Its corresponding oclet class $[o_t]$ specifies the behavior of transition t.

Further, the constructive semantics of Σ represents the initial marking m_0 as the distributed run π_0 that provides conditions $b_1^p, \ldots, b_k^p$ with label $\ell(b_i^p) = p, i = 1, \ldots, k$, for each place p of Σ with $m_\Sigma(p) = k$; see Def. 2.21. π_0 induces the ε-oclet $\pi_0 = (\pi_0, \varepsilon)$. Its oclet class $[\pi_0]$ specifies the initial marking m_0 of Σ.

We define the oclet specification $O_\Sigma := \{\, [\pi_0]\} \cup \{[o_t] \mid t \in T_N\}$. By construction holds that o_t is enabled at a distributed run π iff t is enabled π at some location α. The corresponding oclet $o_t^\alpha \in [o_t]$ expresses an occurrence of t by some local t-step A_t: $\pi \rhd A_t = \pi \rhd o_t^\alpha$. Thus, the minimal behavior $R(O_\Sigma)$ of O_Σ and the behavior of the Petri net system Σ are equal: $R(O_\Sigma) = R(\Sigma)$. This property follows by simple induction on the definitions of $R(\Sigma)$ (Def. 2.21) and $R(O_\Sigma)$ (Def. 5.17). □

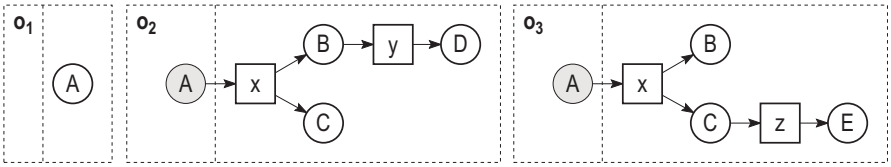
This result is remarkable wrt. the expressive power of oclets. Oclet formalize a kernel of scenarios that is based on a minimal set of notions. By Lemma 5.19, we

know that despite their limited expressive power, oclets are expressive enough to specify any Petri net. In other words, the kernel of scenarios identified in Chapter 3 is expressive enough to specify any distributed system (which can also be modeled as a Petri net).

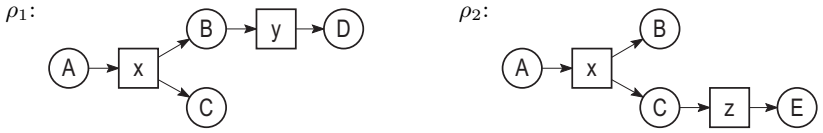
5.5.2. A non-implementable specification

We have just proved that every Petri net system has an equivalent oclet specification. The converse is not true. There are oclet specifications O s.t. no implementation Σ of O exhibits exactly $R(\Sigma) = R(O)$.

Figure 5.11 depicts an oclet specification O which has no equivalent implementation. The minimal behavior that satisfies O is the set $R(\mathsf{O}) = \mathit{Prefix}(\{\rho_1, \rho_2\})$. We obtain $R(\mathsf{O})$ by composing the oclets of O as defined in this chapter.



(a) oclet specification $\mathsf{O} = \{\mathsf{o}_1, \mathsf{o}_2, \mathsf{o}_3\}$



(b) behavior $\mathsf{R}_1 = \{\rho_1, \rho_2\}$

Figure 5.11. Oclet specification O has the least behavior $R(\mathsf{O}) = \mathit{Prefix}(\{\rho_1, \rho_2\})$.

There exists no Petri net system which exhibits exactly the distributed runs $R(\mathsf{O})$. The Petri net system Σ shown in Figure 5.12 exhibits the runs $R(\mathsf{O})$, and additionally the run ρ_3. However, Σ is *minimal* wrt. $R(\mathsf{O})$. Minimal means that there exists no Petri net system Σ' with $R(\mathsf{O}) \subseteq R(\Sigma')$ and $R(\Sigma') \subset R(\Sigma)$.

Every Petri net system that has ρ_1 and ρ_2 as runs also has ρ_3 as a run, for the following reason. According to the constructive semantics of Petri nets in Section 2.4.1, every Petri net system that exhibits ρ_1 as run also exhibits the local step A_{y} consisting of the event y with ${}^\bullet\mathsf{y} = \{\mathsf{B}\}$ and $\mathsf{y}^\bullet = \{\mathsf{D}\}$. The local step A_{y} is enabled at ρ_2. The continuation $\rho_2 \rhd A_{\mathsf{y}}$ inevitably yields to the additional run ρ_3 that is not specified in $R(\mathsf{O})$.

Where does this gap between $R(\mathsf{O})$ and $R(\Sigma)$ come from? According to the semantics of oclets, o_2 and o_3 are alternatives; hence, either y occurs or z occurs. In contrast, the *structure* of each oclet tells that both actions y and z are independent because their pre- and post-conditions are disjoint. Thus, an occurrence of y cannot influence an occurrence of z and vice versa; consequently both events eventually occur in the same run after x. In other words, in a distributed system

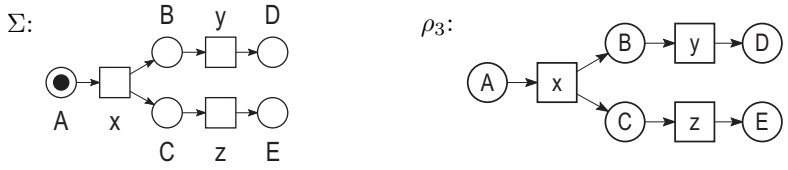


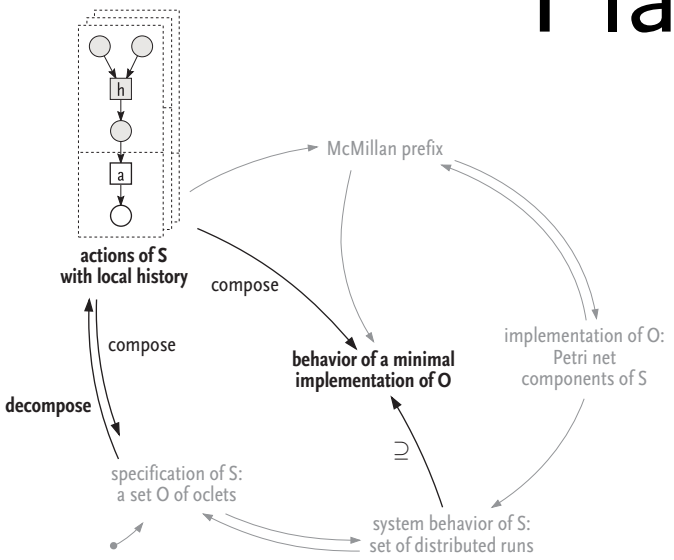
Figure 5.12. The Petri net system Σ has the behavior $R(\Sigma) = \mathit{Prefix}(\rho_3)$.

that implements a specification, the occurrence of a local action is not tied to the scenario in which this action is specified. Rather, each local action only depends on *its own* local pre-conditions.

Conclusions for the synthesis problem

The preceding example in Figure 5.12 presented an oclet specifications O s.t. no Petri net implementation Σ exhibits *exactly* the runs $R(\Sigma) = R(O)$. Any distributed system that implements O *inevitably exhibits* more behavior than O. Consequently, any solution to the synthesis problem can only construct a system that satisfies O and exhibits *as little additional behavior* as possible. The next chapter answers the question whether this additional behavior is *unique*.

6. Scenario Play-Out



The preceding Chapter 5 introduced oclet composition as a technique for constructing larger scenarios from small scenarios. We have shown that oclet composition allows to construct the unique minimal behavior that satisfies a specification. Though, there remains a gap between a specification and its implementations: for some specifications, any implementation exhibits more than the minimal behavior.

In this chapter, we show that this gap follows from different assumptions how system behavior is defined. The behavior of a specification follows from occurrences of *entire scenarios consisting of multiple actions*; the behavior of an implementation follows from occurrences of *single actions*. To bridge this gap, we introduce *oclet decomposition* to decompose a scenario into its single actions. In combination with oclet composition, we obtain *operational semantics* for oclets in the spirit of Petri nets. The operational semantics makes two contributions. (1) A set of scenarios becomes an operational system model. We present an *execution engine* for *scenario-based models*. (2) The operational semantics construct the behavior exhibited by a *minimal implementation* of a specification. In Chapter 7, we extend the operational semantics to a synthesis algorithm.

6.1. An Action-Centric View on System Behavior

The preceding chapter brought a significant progress towards a solution to the synthesis problem. We introduced a composition operator on oclets. With this operator, we can construct for any oclet specification O the unique minimal set $\min \mathcal{R}(O)$ of runs that satisfies O by composing the oclets of O. Thus, we could read O as a *system model* which describes a system S, and we could describe the behavior of S by composing O's oclets to the runs $\min \mathcal{R}(O)$.

In this chapter, we argue that $\min \mathcal{R}(O)$ does *not* describe the behavior of the actual system S. We then show that by canonically *decomposing* O into its single actions, we can bridge this gap. With this slight change, a set O of oclets becomes a system model as illustrated in Figure 6.1. By these results, we also precisely define which behavior a *minimal implementation* of O exhibits. This notion will be crucial to solve the synthesis problem in the next two chapters.

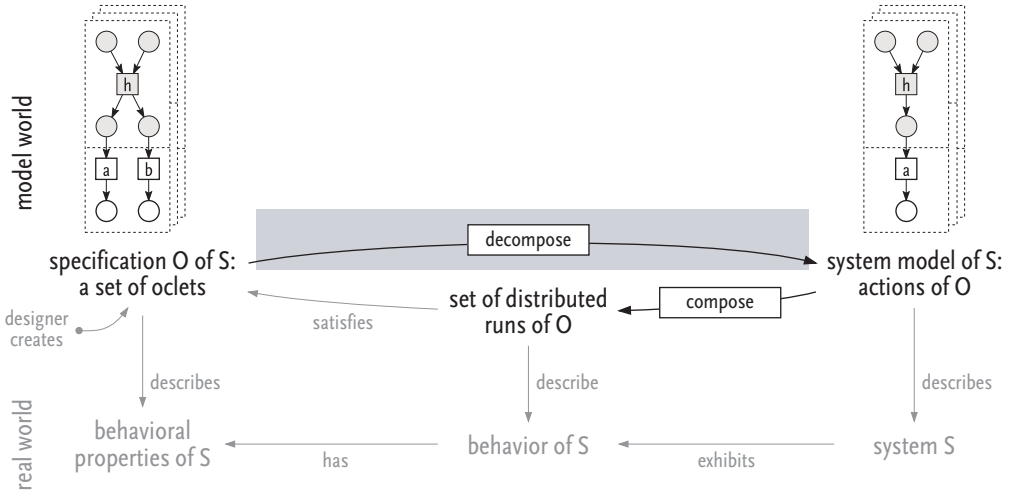


Figure 6.1. A specification becomes a system model by decomposition into its set of actions.

Behavior of a minimal implementation

Although we can construct the minimal behavior $\min \mathcal{R}(O)$ that satisfies O, we are still facing a *gap between a specification O and an implementation of O*, for instance, as a Petri net. Section 5.5 showed that there are specifications O for which no implementation Σ exhibits exactly the runs $R(\Sigma) = \min \mathcal{R}(O)$. More precisely, no *distributed system* exhibits $\min \mathcal{R}(O)$. Instead, any distributed system Σ that implements O inevitably exhibits *more* behavior than necessary to satisfy O.

So, we have to change our view on the synthesis problem. We call an implementation Σ of O *minimal* if no implementation Σ' of O exhibits less behavior than Σ. A minimal implementation Σ satisfies O and exhibits as few additional behavior

as possible. This additional behavior inevitably occurs in Σ because of being a distributed system. This observation triggers two questions. Is there a unique behavior exhibited by all minimal implementations? And if so, can we construct it?

An implementation executes single actions

In the preceding chapter, we used oclet composition to construct the least behavior $\min \mathcal{R}(O)$ that satisfies an oclet specification O. Each run in $\min \mathcal{R}(O)$ is constructed by composing *entire oclets* of O where each oclet consists of multiple actions. In other words, $\min \mathcal{R}(O)$ assumes that *all actions of an oclet occur together after the oclet's entire history occurred.* In contrast, a classical system model like a Petri net constructs a run based on a different assumption: *each single action occurs individually when its local precondition holds.*

The next steps

In this chapter, we show that the behavioral assumptions of a classical system model can be adapted to oclets: we decompose an oclet specification O into single actions, and define an operational semantics for oclets in the style of Petri nets. Figure 6.2 illustrates the idea: the oclet o can be decomposed into three *basic oclets.* Each basic oclet describes an action of o with a local history. The basic oclets recompose to the original oclet o.

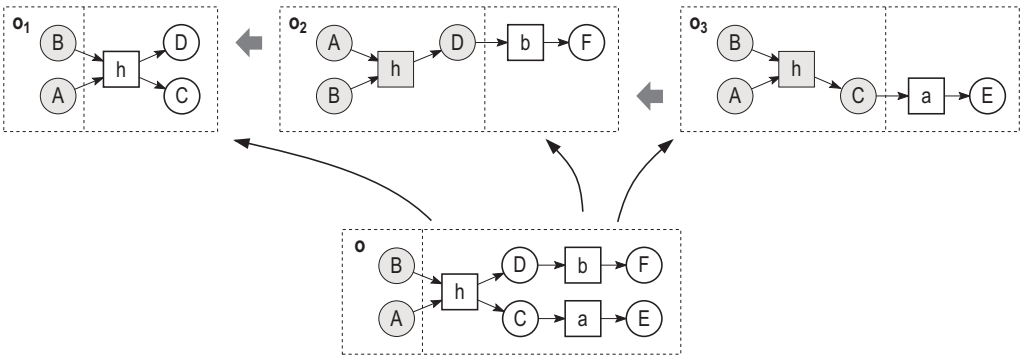


Figure 6.2. Decomposition of oclet o into its three basic oclets.

We proceed as sketched in Figure 6.3. In Section 6.2, we explain how the composition of basic oclets allows to *execute* an oclet specification O action by action. This way, system behavior is described in terms of actions instead of scenarios. Thus, we establish an *action-centric view* on system behavior.

Section 6.3 presents the formal definitions for decomposing an oclet specification O into its set of basic oclets. We then define *operational semantics* for O by composing O's basic oclets to distributed runs. The set of distributed runs defined by these operational semantics satisfies O as sketched in Figure 6.4. We prove this

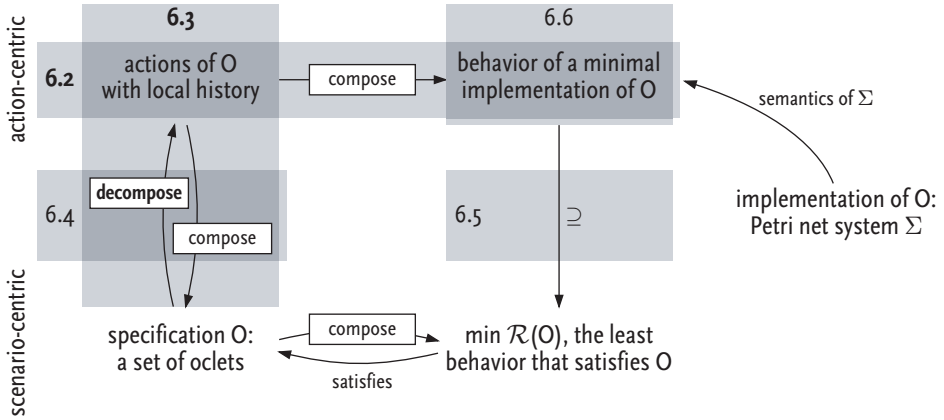


Figure 6.3. Chapter overview: by decomposing an oclet specification O into its single actions, we can construct the behavior of a minimal implementation of O by oclet composition.

result in Section 6.5 based on some properties of oclets, and their composition and decomposition shown in Section 6.4. We then argue in Section 6.6 that

> *every minimal implementation of O exhibits the behavior defined by the operational semantics of O.*

In other words, occurrences of single actions bridge the gap between the least behavior $\min \mathcal{R}(O)$ that satisfies O and the behavior of a minimal implementation of O. Figure 6.4 illustrates these results.

Altogether, this chapter makes the following two contributions. (1) By the operational semantics, oclets become a technique for *modeling* the behavior of distributed systems with scenarios. Thus, we solve the second research goal of this thesis: to define a technique for executing a scenario-based specification without synthesizing components. Section 6.7 presents a run-time engine for executing oclet specifications and demonstrates the flexibility of this approach. (2) The operational semantics of oclets determines and constructs the behavior of a minimal implementation.

Section 6.8 discusses the results of this chapter, and the role of declarative oclet semantics (of Chapter 4), oclet composition (of Chapter 5), and operational semantics of oclets (of this chapter) in system design. The next two Chapters 7 and 8 extend the operational semantics to a synthesis algorithm which constructs a minimal implementation from any given oclet specification.

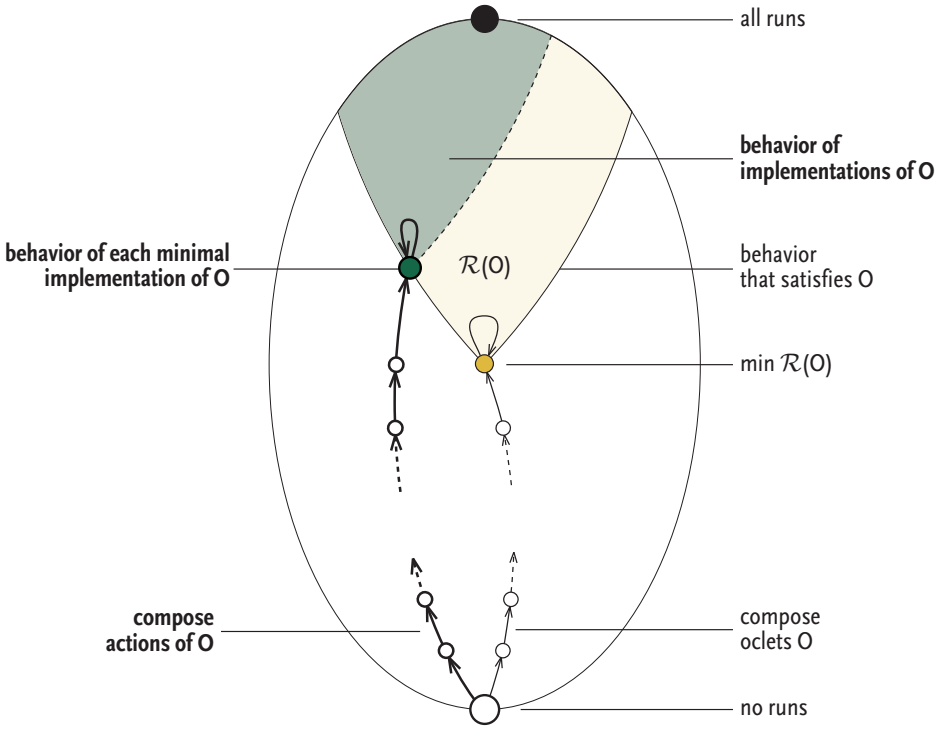


Figure 6.4. The operational semantics of oclets composes all individual actions of an oclet specification O to the behavior of a minimal implementation of O.

6.2. Play-out: Executing Scenarios

The idea of constructing behavior of scenarios from their individual actions is not new. It has been proposed first by Harel and Marelly who developed *scenario play-out* for LSCs [56].

The idea of play-out is the following. A scenario-based specification S is equipped with a *runtime state* r. The state r defines whether an action a of a scenario in S is *enabled* at r. If a is enabled at r, then a may occur. An occurrence of a in r changes the runtime state to a successor state r', i.e., defines a *step* $r \xrightarrow{a} r'$ of the specification S. A sequence of steps is an *execution* of S.

The specific trick of play-out for scenarios is that the runtime state r "remembers" how much behavior of each scenario in S has already occurred in an execution. This information determines which actions of the scenarios in S are enabled. In this way, the specification S becomes an *executable* system model like an automaton or a Petri net. The term "executable" here refers to the idea that if several actions are enabled at a runtime state, then a user may pick or trigger a specific action for execution.

In the following, we adapt play-out to oclets. In oclets, a "runtime state" is a finite distributed run which we interpret as the history of the execution that leads to the current state. Oclet play-out is based on the following two notions:

1. At any moment during an execution, i.e., at the end of every distributed run, the set of all *enabled* events of an oclet specification is well-defined.
2. If an event is enabled, then an *occurrence* of a *single* event of an oclet yields a well-defined step that continues the execution.

We will show that both notions canonically follow from distributed runs and oclets. Our approach finally amounts to an *operational semantics* for oclets that allows to directly execute an oclet specification. In the following, we present the idea of oclet play-out at an informal level by an example. The next Section 6.3 provides the corresponding formal definitions.

Playing oclet events

The key to oclet play-out is the *local history* of an event e in an oclet o. Intuitively, the local history of e are all conditions and events that necessarily have to occur before e can occur. In terms o: the local history of e is the set of all transitive predecessors of e in o.

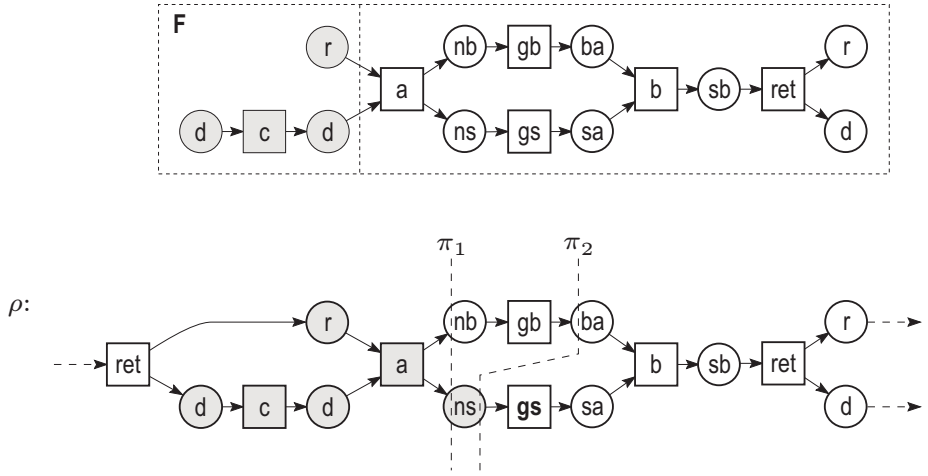


Figure 6.5. Oclet F occurs in the run ρ. Action gs is enabled at the prefixes π_1 and π_2, because both runs end with the local history of gs (highlighted grey).

We illustrate this notion by the example of oclet F of Figure 6.5. Oclet F will be the running example in this section, it describes one complete execution cycle of the flood alert example introduced in Section 2.1. Oclet F occurs in the distributed run ρ, so action gs occurs in ρ as well. We now consider *when* gs occurs in a system that executes ρ. The system first has to reach the prefix π_1 of ρ before gs can occur; the local pre-conditions of gs do not hold earlier. In π_1 all pre-conditions of gs do hold and gs can occur. Action gs can also occur after the system moved

to π_2 with action gb. Thus, action gs is *enabled* at both π_1 and π_2 because both runs end with the local history of gs that is highlighted grey in ρ.

That the enabledness of an action depends on its entire local history and not only on its immediate pre-conditions is illustrated by action a of F. In Figure 6.6, a is only enabled at π_4 but not at π_3 although both runs reach the same global state [r, d].

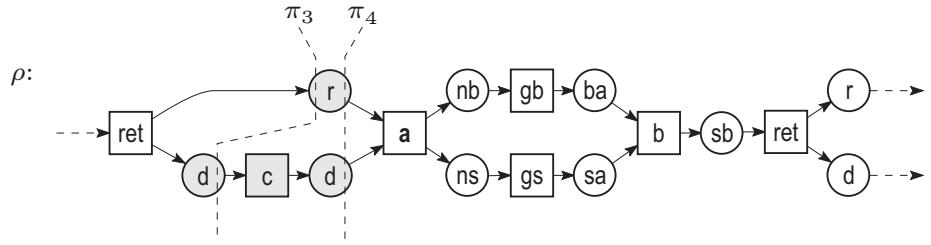


Figure 6.6. The local history of an action matters: a is enabled at the prefix π_4 but not in the prefix π_3 although both reach the same global state [r, d].

Basic oclets. In the following, we reduce the *occurrence of a single event of a scenario* to the model of oclets. We represent each event e of an oclet o together with its local history as an oclet $o[e]$ again. We call an oclet of this kind *basic*. The history of oclet $o[e]$ consists of the local history of e; the contribution of $o[e]$ consists of e and e's post-conditions. Figure 6.7 depicts all basic oclets of oclet F.

The semantics of the basic *oclet* $o[e]$ defines when event e is enabled and what happens when e occurs: event e is *enabled* at a distributed run π iff $o[e]$ is enabled at π. We express an *occurrence* of e at π by composing π with $o[e]$ to $\pi \rhd o[e]$ as defined in Chapter 5.

The following example illustrates these notions. The runs in Figure 6.8 are constructed by playing out oclet F. Event a is enabled at the run π_1 — because π_1 ends with the history of the basic oclet F[a]. An occurrence of event a at π_1 extends π_1 to π_2 — which is technically expressed by composing π_1 with the basic oclet F[a]: $\pi_2 = \pi_1 \rhd$ F[a]. By the same means, both events gb and gs are enabled at π_2, the local history of gs is highlighted grey. An occurrence of gs yields the run π_3 by composing π_2 with F[gs]. We obtain $\pi_4 = \pi_3 \rhd$ F[gb] by playing out gb. In this way, we could continue playing out all events until the entire oclet F occurs.

Alternative runs. Oclet play-out also constructs alternative runs. Event nob of oclet fail in Figure 6.10 is alternative to event gb of oclet F. Both gb and nob are enabled at π_3 of Figure 6.8. Playing out gb at π_3 yields the run π_4 shown in Figure 6.8 whereas playing out nob yields the run $\pi_5 = \pi_3 \rhd$ fail[nob] shown in Figure 6.10. The runs π_4 and π_5 are alternative to each other.

Play-out yields more behavior. The run π_5 is *not* part of the minimal behavior that satisfies a specification which contains oclets F and none. π_5 contains actions gs

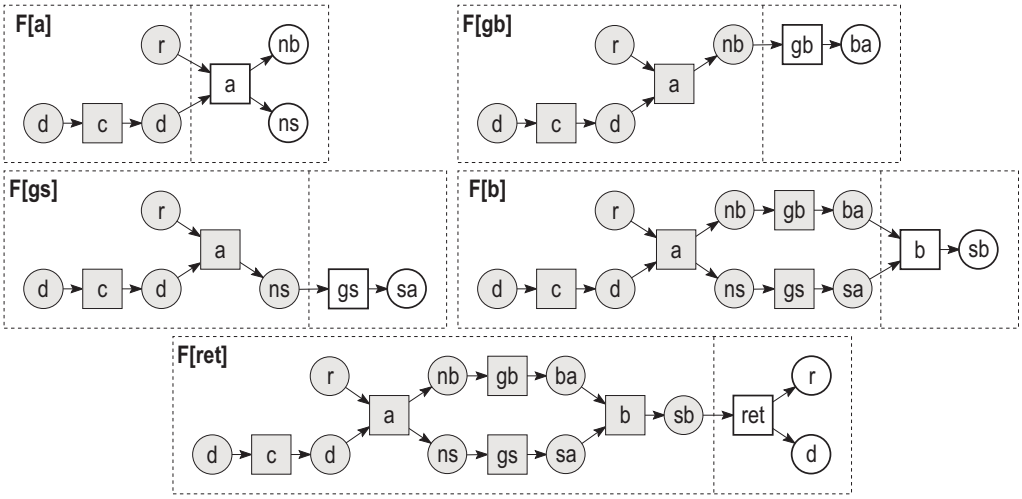


Figure 6.7. All basic oclets of oclet F from Fig. 6.5; each basic oclet represents on event of F with its local history.

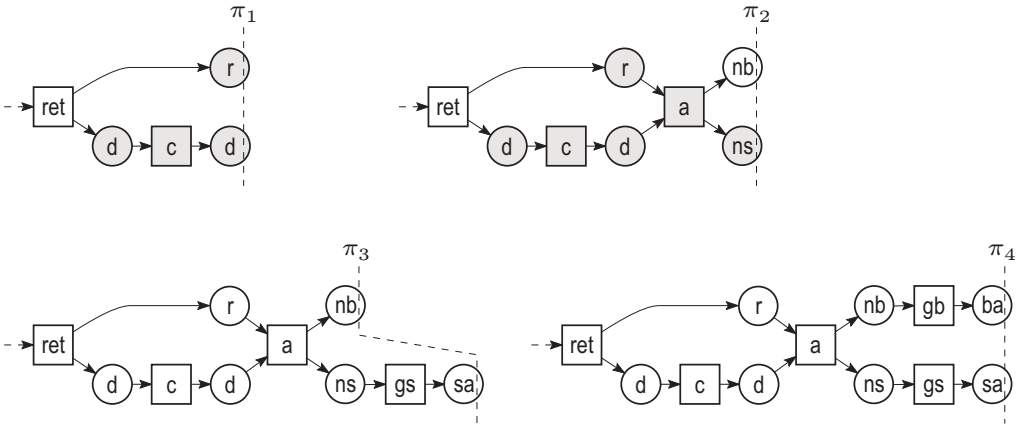


Figure 6.8. Distributed runs constructed by play-out of oclet F of Fig. 6.5.

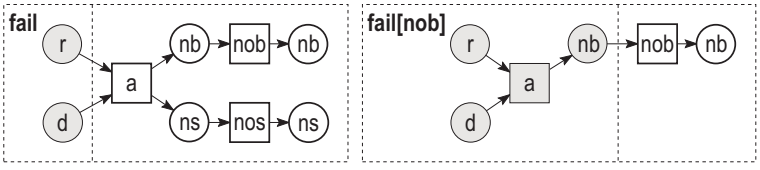


Figure 6.9. Oclet fail describes a failure of the flood alert process.

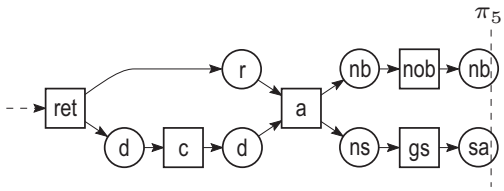


Figure 6.10. The run π_5 is alternative to the run π_4 of Fig. 6.8; π_5 is constructed by play-out of the alternative event nob.

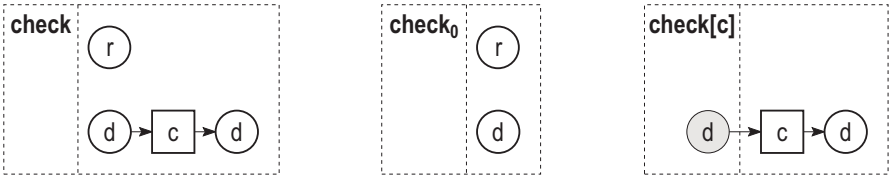


Figure 6.11. The ε-oclet check decomposes into its basic oclet check[c] and the initial state check$_0$.

and nob from F and none, respectively. Their minimal behavior only requires that either events gb and gs occur together (as in π_4) or that alternatively events nob and nos occur together (not depicted). Technically, this minimal behavior follows from composing the run π_1 either with F or with none. The "mixed" run π_5 cannot be constructed in this way. However, π_5 occurs in the system when assuming that behavior follows from occurrences of *single actions*. Then the choice between gb and nob is made in the local state nb, and action gs occurs *independently* of this choice. Hence, π_5 may occur in play-out but not by composition of scenarios.

Initial states

Up to now, we explained when an event of an oclet is enabled at a distributed run and how an occurrence of a single event continues a run. This approach needs an initial run at which the first event to be played is enabled.

The construction of the unique minimal behavior that satisfies an oclet specification O begins with the oclets of O that have an empty history (ε-oclet). In the same spirit, oclet play-out also begins with ε-oclets — but with a twist.

Figure 6.11 depicts an example. The ε-oclet check has one event c that can be played by the basic oclet check[c]. The part of check that is not covered by play-out yet are its minimal conditions; these constitute the special ε-oclet check$_0$, which describes an *initial state*. Oclet play-out of a specification which contains check begins in the initial state check$_0$. From there play-out continues with all enabled events of the specification. Playing event c yields the distributed run check$_0 \triangleright$ check[c] = check, so check becomes an *initial run*.

The least common initial state. There is one more important aspect to initial states in play-out. In general, an oclet specification has several ε-oclets describing

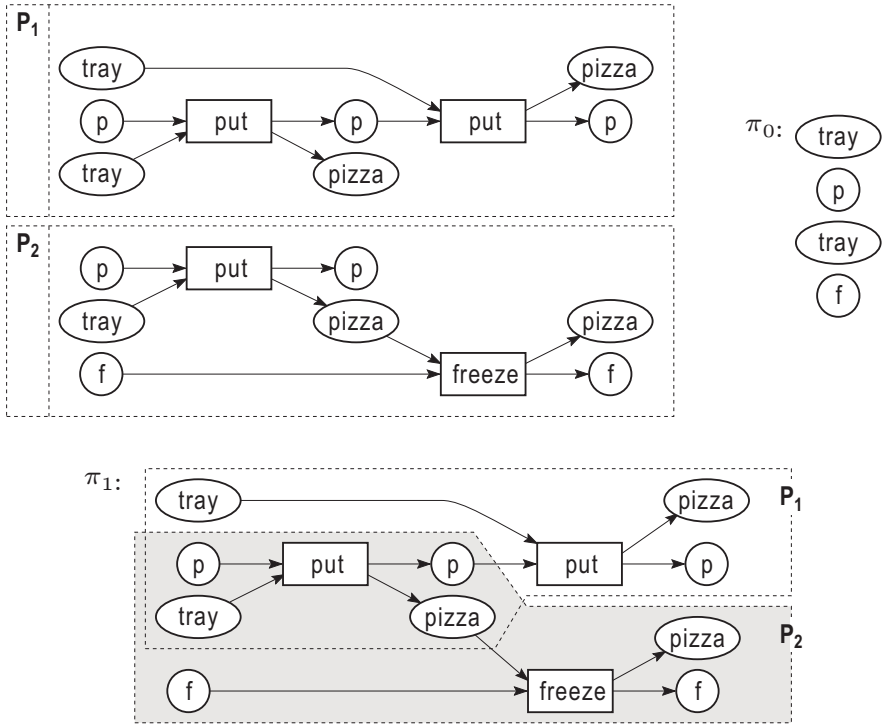


Figure 6.12. The alternative ε-oclets P_1 and P_2 yield different initial runs. Their least common initial state π_0 is unique. The run π_1 continues π_0 with P_1 and P_2.

different initial runs. In Chapter 3, we decided to interpret ε-oclets as *alternative initial runs*. Oclet composition of Chapter 5 puts all ε-oclets as true alternatives which yields the least behavior that satisfies an oclet specification. However, there are specifications where the ε-oclets together do not describe a *unique initial state*. Figure 6.12 depicts an example. Reducing oclets P_1 and P_2 to their respective minimal conditions would yield two alternative initial states $[2\mathsf{tray}, \mathsf{p}]$ and $[\mathsf{p}, \mathsf{tray}, \mathsf{f}]$ among which play-out had to choose prior to an event occurrence. This would be counter-intuitive to the idea of an implementation that initially resides in *one unique* state.

The aim of this chapter is to adapt the behavioral assumptions of an implementation to scenarios. If we do not want to exclude specification $\{\mathsf{P}_1, \mathsf{P}_2\}$ from our approach we have to *derive the least common initial state* of a specification O. A *common initial state* π_0 of O is a state which contains each initial state of each ε-oclet of O. The state π_0 is *minimal* wrt. O if there exists no smaller initial state of O. Obviously, each oclet specification has a unique least common initial state (up to isomorphism). Figure 6.12 shows the least common initial state π_0 of $\{\mathsf{P}_1, \mathsf{P}_2\}$.

A least common initial state is just large enough to be continued with every ε-oclet in O. O's ε-oclets then occur as alternative or overlapping continuations of

π_0 depending on their inner structure — just like oclets with history as discussed in Section 3.4. For example, by playing out out P_1 and P_2 in their unique initial state π_0 shown in Figure 6.12, we obtain the run π_1 in which P_1 and P_2 occur overlappingly. The occurrences of P_1 and P_2 are *prefixes* of π_1. Thus, π_1 also contains the initial runs of the specification $\{\mathsf{P}_1, \mathsf{P}_2\}$ that are constructed using the approach of Chapter 5.

Design decision: From this Chapter onwards, we assume that behavior of a scenario-based specification follows from occurrences of *single actions* and begins in a *unique initial state*.

This concludes our informal introduction to oclet play-out. The next sections present the corresponding formal definitions and prove the correctness of oclet play-out wrt. oclet semantics.

6.3. Formal Definitions for Oclet Play-Out

This section provides the formal definitions for oclet play-out. Oclet play-out describes when an event of an oclet is enabled and how to continue a run with *one* enabled event at a time.

6.3.1. Basic oclets represent events

Every oclet o contributes several events $e_1, \ldots, e_k$ in a specific partial order. For play-out of these events, we *decompose* o into smaller oclets $o[e_1], \ldots, o[e_k]$. The history of each oclet $o[e_i]$ consists of all predecessors of e_i in o; the contribution of $o[e_i]$ consists of e_i and e_i's post-conditions. An oclet of this kind is a *basic* oclet.

Definition 6.1 (Basic oclets). An oclet $o = (\pi_o, hist_o)$ with $\pi_o = (B_o, E_o, F_o)$ is *basic* iff its contribution contains exactly one event e s.t. the history of o consists of all predecessors of e, i.e., $X_{hist_o} = e{\downarrow_o} \setminus \{e\}$ (Def. 2.5).

Let o be an arbitrary oclet. Let $e \in E_{con_o}$ be an event of o's contribution. Event e *induces* the basic oclet $o[e] = (\pi_e, hist_e), \pi_e = (B_e, E_e, F_e)$ which has

1. the nodes $X_e = e{\downarrow_o} \cup \{e\} \cup post_o(e)$ s.t. $E_e = E_o \cap X_e$, $B_e = B_o \cap X_e$, $F_e = F_o|_{X_e \times X_e}$, and
2. the history $hist_e \sqsubseteq \pi_e$ with $X_{hist_e} = e{\downarrow_o} \setminus \{e\}$.

The basic oclets of o are the set $\hat{o} := \{o[e] \mid e \in E_{con_o}\}$. ⌟

Because a basic oclet $o[e]$ is an oclet, the notion of its *oclet class* is well-defined. We write $[\hat{o}] := \{[\, o[e] \,] \mid o[e] \in \hat{o}\}$ for the oclet classes of the basic oclets of o.

Figure 6.7 depicts the set of all basic oclets of our running example oclet F from Figure 6.5. A basic oclet represents a single event of a larger oclet. From the semantics of oclets follows (1) when an event (of an oclet) is enabled at a distributed run, and (2) how to continue a run with an enabled event.

Definition 6.2 (Enabled oclet event, occurrence of an oclet event). Let π be a distributed run. Let o be an oclet and let $e \in E_{con_o}$ be a contributed event of o.

1. The event e is *enabled* at π iff $o[e]$ is enabled at π (Def. 5.1).
2. If e is enabled at π, then e may *occur* at π which yields the distributed run $\pi \rhd o[e]$. ⌟

This definition canonically extends to all basic oclets $o[e]^\alpha$ in the oclet class of $o[e]$. Like for the composition of oclets in Section 5.2, we say that *event e is enabled at π at location α* iff $o[e]^\alpha$ is enabled at π. Then, *e may occurr in π at location α* by $\pi \rhd o[e]^\alpha$. We simplify notation and write $\pi \rhd e^\alpha$ instead of $\pi \rhd o[e]^\alpha$ if e is enabled at π at α.

Figure 6.8 illustrates the event-wise construction of a distributed run from the events of oclet F of Figure 6.5. At π_1, event a is enabled because π_1 ends with o[a]'s history. The basic oclet o[a] is depicted in Figure 6.7. a is the only enabled event at π_1. The occurrence of a yields the run $\pi_2 = \pi_1 \rhd$ a. At π_2, events gs and gb are enabled; the history of gs is highlighted in π_2. We obtain $\pi_3 = \pi_2 \rhd$ gs where gb is still enabled. In this way, we could continue π_3 to a complete occurrence of oclet F.

The preceding definitions canonically extend to an oclet specification O: decompose each oclet $o \in O$ into its basic oclets $\hat{o}$. The set of all basic oclet classes of O is $\hat{O} := \bigcup_{[o] \in O} [\hat{o}]$ of O. $\hat{O}$ is an oclet specification in which each oclet describes one event of O. In principle, the behavior of $\hat{O}$ defines the play-out behavior of O — except for the initial state of O which we consider next.

6.3.2. Minimal conditions denote the least common initial state

We decided to infer the least common initial state of an oclet specification O from its ε-oclets. This least common initial state is the smallest set B of conditions s.t. the minimal conditions of each ε-oclet in O occur in B.

As a technicality, Chapter 2 introduced the assumption that a distributed run always begins with a set of *conditions*. Correspondingly, we assume without loss of generality that each ε-oclet begins with conditions only.[1] More precisely,

$$\text{for each oclet } o = (\pi_o, hist_o) \text{ with } hist_o = \varepsilon \text{ holds } \min \pi_o \subseteq B_o. \tag{6.1}$$

In general, an oclet has an arbitrary number of equally labeled minimal conditions. Thus, the oclet with the maximal number of conditions with label a determines the number of a-labeled conditions in the least common initial state. Figure 6.12 depicts a general example.

Definition 6.3 (Initial state of an oclet specification). Let O be an oclet specification, let O_ε be the ε-oclets of O. The *initial state* of O is the distributed run $\pi(O) :=$

[1] If a specification O contains an ε-oclet $[o]$ with an event $e \in \min \pi_o$, then o can be extended to satisfy our assumption: add a new pre-condition b^* to e in o, and assign b^* a new label that is not used in O. Because b^* has a new label, only oclet $[o] \in O$ depends on an occurrence of b^*. The semantics of O remains unchanged up to occurrences of b^*, because the ε-oclet o occurs exactly once in the minimal behavior of O.

$(B, \emptyset, \emptyset)$ with $|\{b \in B \mid \ell(b) = a\}| = \max_{[o] \in O_\varepsilon} |\{b \in \min \pi_o \mid \ell(b) = a, o = (\pi_o, hist_o)\}|$, for all labels a. ⌟

Corollary 6.4: *Let O be an oclet specification. Let O_ε be the ε-oclets of O.*

1. *For each $[o] \in O_\varepsilon$ holds $\min \pi_o$ occurs in the initial state $\pi(O)$.*
2. *$\pi(O)$ is minimal wrt. O, i.e., there exists no strict prefix $\pi' \sqsubset \pi(O)$, $\pi' \neq \pi(O)$ s.t. for each each $[o] \in O_\varepsilon$ holds $\min \pi_o$ occurs in π'.* ⋆

According to our notational convention of Chapter 5, we also write $\pi(O)$ for the ε-oclet that represents $\pi(O)$.

6.3.3. Play-out behavior of oclets

Up to this point, all formal notions for oclet play-out of an oclet specification have been defined.

Each oclet specification O canonically decomposes into its set $\hat{O}$ of basic oclets; moreover, O defines a unique initial state $\pi(O)$. These notions together define a *normal form* of an oclet specification. The normal form defines the set of *all* distributed runs that can be constructed by play-out of O's events.

Definition 6.5 (Normal form, play-out behavior). Let O be an oclet specification. The *normal form* of O is the oclet specification $\mathrm{NF}(O) := (\hat{O} \cup \{[\pi(O)]\})$ consisting of the basic oclet classes $\hat{O}$ of O and the class of the initial state $[\pi(O)]$ of O. The *play-out* behavior of O is $\hat{R}(O) := R(\mathrm{NF}(O))$, i.e., the minimal behavior that satisfies $\mathrm{NF}(O)$ according to Def. 5.17. ⌟

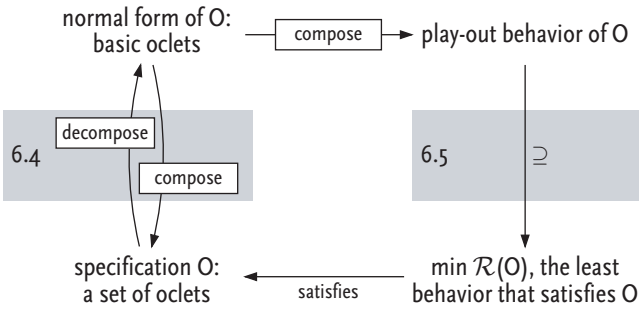
An oclet specification in normal form has exactly one ε-oclet consisting of a set of conditions; each other oclet $o \in \mathrm{NF}(O)$ is basic, i.e., o contributes one event e and o's history consists of all predecessors of e. By definition, any two oclet specifications O_1 and O_2 which have the same normal form $\mathrm{NF}(O_1) = \mathrm{NF}(O_2)$ have the same play-out behavior.

Theorem 6.6. *Let O be an oclet specification. The play-out behavior of O satisfies O, i.e., $\hat{R}(O) \in \mathcal{R}(O)$.* ⋆

We prove this theorem in the next two sections; the convinced reader may continue at Section 6.6.

6.4. Decomposing and Comparing Oclets

We just introduced oclet play-out to construct distributed runs from basic oclets of an oclet specification O. We prove that the play-out behavior of O satisfies O as follows.



In this section, we complement oclet composition from Chapter 5 with *oclet decomposition* and a relation to *compare oclets*. These operations and relation can be used to *calculate* with oclets, to develop oclet specifications systematically, and to reason about the behavior of oclet specifications on the syntactical level rather than the semantic level. We specifically use them in Section 6.5 where we show that the least behavior $\min \mathcal{R}(O)$ that satisfies O can be composed from O's basic oclets. This inclusion yields a proof of Theorem 6.6.

6.4.1. Decomposing an oclet along a cut

An oclet o can be split into two oclets along a cut B of o. The first half $o^-[B]$ of o ends at the cut B, the second half $o^+[B]$ of o continues from B. Figure 6.13 illustrates this operation.

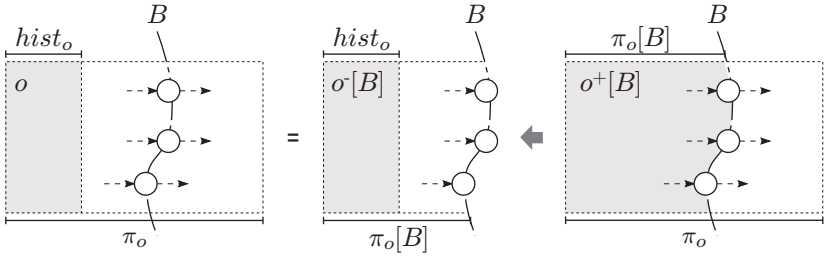


Figure 6.13. Decomposition of an oclet o along a cut B.

Technically, the decomposition of an oclet along a cut B is based on the notion of the prefix $\pi[B]$ of the run π that is *induced* by cut B, i.e., $\pi[B]$ is the prefix of π that consists of the nodes B and all nodes that causally precede B in π; see Definition 2.14. Formally, relation $x \leq_o y$ expresses when a node x causally precedes a node y in oclet o; see Definition 2.8.

Definition 6.7 (Cut decomposition). Let $o = (\pi_o, hist_o)$ be an oclet, let B be a cut of π_o s.t. $\max hist_o \leq_o B$, i.e., for each $x \in \max hist_o$ exists $y \in B$ with $x \leq_o y$. Then the oclets $o^+[B] := (\pi_o, \pi_o[B])$ and $o^-[B] := (\pi_o[B], hist_o)$ are the *cut decomposition* of o along B. ⌟

Obviously, the decomposed oclets compose again to the original oclet.

Lemma 6.8: *Let o be an oclet and let B be a cut of π_o s.t. $\max hist_o \leq_o B$. Then (1) $o^-[B]$ and $o^+[B]$ are oclets, (2) $o^+[B]$ is enabled at $o^-[B]$, and (3) $o = o^-[B] \rhd o^+[B]$.* ⋆

The proof of this lemma straight forward. Several technical details of the proof follow from a lemma on induced prefixes $\pi[B]$, see Lemma A.7 in Appendix A.5.

Proof. (1) From $hist_o \sqsubseteq \pi_o$ and $\max hist_o \leq_o B$ follows $hist_o \sqsubseteq \pi_o[B]$ by Lem. A.7-6. Thus, $o^-[B] = (\pi_o[B], hist_o)$ is an oclet. Further, $\pi_o[B] \sqsubseteq \pi_o$, by Lem. A.7-4, which implies that $o^+[B] := (\pi_o, \pi_o[B])$ is an oclet.

(2) $\pi_o[B]$ trivially ends with $\pi_o[B]$. Thus, $o^+[B] = (\pi_o, \pi_o[B])$ is enabled at $o^-[B] = (hist_o, \pi_o[B])$ by Def. 5.2.

(3) $o^-[B] \rhd o^+[B] = (\pi_o[B] \cup \pi_o, hist_o) = (\pi_o, hist_o)$ by Def. 5.2 and by $\pi_o[B] \subseteq \pi_o$ (Lem. A.7-4). □

The main difference of the decomposition of o along a cut B to the decomposition into basic oclets (Def. 6.1) is the following: the oclet $o^+[B]$ has *all* behavior that precedes the cut B as its history; the history of a basic oclet $o[e]$ consists only of all predecessors of e. Thus, its history is "smaller", or in other words, the prerequisites for an occurrence of $o[e]$ are *weaker*. This observation suggests to compare oclets wrt. their histories.

6.4.2. Comparing oclets

An oclet's history describes which behavior has to occur prior to an occurrence of the oclet. If two oclets differ only in their history, then the oclet with the smaller history requires less behavior prior to its occurrence (and hence may occur "more often"). This observation gives rise to formally compare two oclets wrt. their histories: an oclet o_1 is *weaker than* an oclet o_2, written $o_1 \preceq o_2$, if both oclets contribute the same behavior, but o_1's history is smaller than o_2's history. The idea of this relation is sketched in Figure 6.15 where o_1 is weaker than o_2.

Technically, the *weaker than* relation requires more than $con_1 = con_2$ to compare two oclets o_1 and o_2. Strictly speaking, also the arcs between history and contribution of o_1 and o_2 are contributed by o_1 and o_2, respectively. Thus, both oclets have to be identical on the *connection* between history and contribution as well. To this end, we introduce the technical notion of an extended contribution con_o^+ of an oclet o: con_o^+ contains con_o and the maximal conditions of $hist_o$ with an arc into con_o. Thus, $con_1^+ = con_2^+$, expresses that oclets o_1 and o_2 contribute the same course of actions depending on the same conditions of their histories.

Definition 6.9 (Weaker history).

1. Let o be an oclet. Let $\vec{F}_o$ denote the arcs of o that begin in o's history and end in o's contribution. The arcs $\vec{F}_o$ begin in the conditions $\vec{B}_o \subseteq \max hist_o$, i.e., $\vec{F}_o := \{(x, y) \in F_o \mid x \in X_{hist_o}, y \in X_{con_o}\}$ and $\vec{B}_o := \{x \mid (x, y) \in \vec{F}_o\}$. The *extended contribution* of o is $con^+ = (B_{con_o} \cup \vec{B}_o, E_{con_o}, F_{con_o} \cup \vec{F})$.
2. Oclet o_1 is *weaker than* oclet o_2, written $o_1 \preceq o_2$, iff $hist_2 \xrightarrow{hist_1}]$ and $con_1^+ = con_2^+$. ⌟

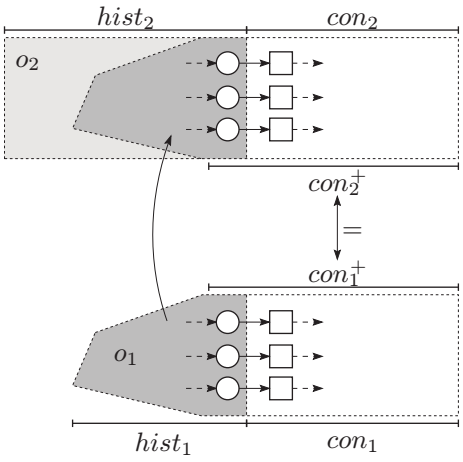


Figure 6.14. Oclet o_1 is weaker than oclet o_2.

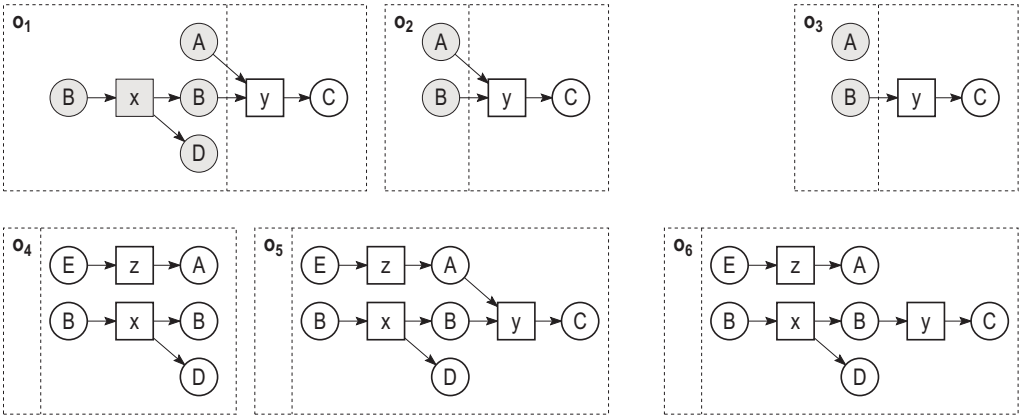


Figure 6.15. Oclet o_2 is weaker than oclet o_1.

The formal definition of con_o^+ is sound: the nodes $\vec{B}_o$ are conditions because $hist_o$ is a prefix of π_o. So each event of $hist_o$ also has *all* post-conditions in $hist_o$ (by Def. 2.10) and cannot be a maximal node of $hist_o$.

Figure 6.15 depicts three oclets o_1-o_3. Only o_2 is weaker than o_1, otherwise the oclets are incomparable, e.g., o_3 is not weaker than o_1. This example also illustrates the rationale of the extended contribution con_o^+. The oclets o_1-o_3 have the same contribution consisting of event y and condition C. Only o_1 and o_2 have the same extended contribution which also includes the pre-set of y. The effect is that the compositions $\mathsf{o}_4 \rhd \mathsf{o}_1 = \mathsf{o}_4 \rhd \mathsf{o}_2 = \mathsf{o}_5$ are equal whereas $\mathsf{o}_4 \rhd \mathsf{o}_3 = \mathsf{o}_6$ is different from o_5.

The following lemma shows that the intention of the weaker-than relation $\preceq$ also holds semantically: if $o_1 \preceq o_2$, then o_1 is enabled whenever o_2 is enabled and both oclets yield the same compositions.

Lemma 6.10: *Let o, o_1 and o_2 be oclets. If $o_1 \preceq o_2$, then the following holds.*

1. *If o_2 is enabled at o, then o_1 is enabled at o.*
2. *If o_2 is enabled at o, then $o \rhd o_2 = o \rhd o_1$.* ⋆

Proof. (*1.*) By Def. 6.9 (weaker than), Def. 4.4 (ends with) and Def. 5.1 (enabled oclet) holds $hist_1 \subseteq hist_2 \subseteq \pi_o$ and $\max hist_1 \subseteq \max hist_2 \subseteq \max \pi_o$. Thus o ends with $hist_1$ which is equivalent to o_1 being enabled at o.

(*2.*) Let $o^* := o \rhd o_1$. Claim: $\pi^* = \pi_o \cup \pi_1 = \pi_o \cup con_1^+$.

$$\begin{array}{lcll} X_o \cup X^+_{con_1} & = & X_o \cup \vec{B_1} \cup X_{con_1} & \textit{[by Def. 6.9-1]} \\ & = & X_o \cup X_{hist_1} \cup \vec{B_1} \cup X_{con_1} & \textit{[by } o_1 \textit{ enabled at } o\textit{: } X_{hist_1} \subseteq X_o\textit{]} \\ & = & X_o \cup X_{hist_1} \cup X_{con_1} & \textit{[by Def. 6.9-1: } \vec{B_1} \subseteq X_{hist_1}\textit{]} \\ & = & X_o \cup X_1 & \\ & = & X^* & \textit{[Def. 5.2].} \end{array}$$

By the same arguments holds $F^* = F_o \cup F_1 = F_\pi \cup F_{con^+}$. Thus, we conclude $\pi^* = \pi \cup con^+$. By the same arguments holds, $\pi^* = \pi \cup con_2^+$. Thus, $o \rhd o_1 = (\pi_o \cup con_1^+, hist_o) = (\pi^*, hist_o) = (\pi_o \cup con_2^+, hist_o) = o \rhd o_2$. □

The preceding Lemma 6.10 particularly allows to *reduce* an oclet *o that contributes a single event* e to its basic oclet $o[e]$. This case is interesting if the history of o is *not* the local history of e. Then $o[e] \preceq o$ and we can replace o by $o[e]$.

Lemma 6.11 (Local history is weaker): *Let $o = (\pi_o, hist_o)$ be an oclet with $E_{con_o} = \{e\}$ and $B_{con_o} = e^\bullet$. Then $o[e] \preceq o$.* ⋆

Proof. Set $o[e] = (\pi_e, hist_e)$ with $\pi_e = (B_e, E_e, F_e)$. To prove $o[e] \preceq o$, we have to show (1) $con_o^+ = con_e^+$, and (2) $hist_o$ ends with $hist_e$.

[$con_o^+ = con_e^+$] Obviously, $con_o = con_e$ by the definition of basic oclets. Moreover each arc $(x, y) \in \vec{F_o}$ of Definition 6.9 ends in $y = e$. Thus for each arc $(x, e) \in \vec{F_o}$ holds $x \in e{\downarrow_o}$ which implies $\vec{F_o} = \vec{F_e}$. Thus $con_o^+ = con_e^+$.

[$hist_o$ ends with $hist_e$] To show: $hist_e \subseteq hist_o$ and $\max hist_e \subseteq \max hist_o$. By definition of $o[e]$ (Def. 6.1) holds $hist_e \subseteq (\pi_o - con_e) = (\pi_o - con_o) = hist_o$. Now, let $b \in \max hist_e$. By the definition of basic oclets, event e is the post-event of condition b in the basic oclet $o[e]$. So, e is the post-event of b in o. Because e is the only contributed event, we get that $b \in \max hist_o$. Thus, $\max hist_e \subseteq \max hist_o$.□

This proof concludes the basic operations and relations on oclets. Chapter 5 defined an associative composition operator $\rhd$ on oclets. This section contributed a corresponding decomposition of oclets along cuts, and a relation to compare oclets wrt. weaker histories. Using these notions, we now prove that decomposing an oclet into its basic oclets, and re-composing the basic oclets again yields an occurrence of the original oclet.

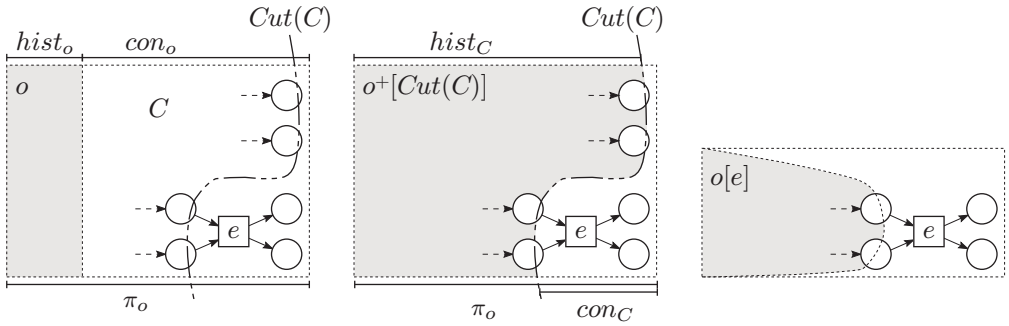


Figure 6.16. Illustration of the decomposition of oclet o in the proof of Lemma 6.12.

6.5. Oclet Play-Out Satisfies Oclet Specifications

This section proves that the play-out behavior of an oclet specification O satisfies O as defined in the semantics of oclets in Chapter 4. This general claim was already stated in Theorem 6.6 but not yet proved. The proof of this theorem follows directly from the following two lemmas. They essentially state that decomposing an oclet specification into its basic oclets always allows to reconstruct the original oclets.

For every oclet o, a set R of runs that is oclet-closed with the basic oclets $\hat{o}$ is also oclet-closed with the original oclet o (Def. 5.12).

Lemma 6.12 (Basic oclets compose to complete oclets): *Let o be an oclet with history and let R be a set of distributed runs. If $(Prefix(R) \rhd [\hat{o}]) \subseteq Prefix(R)$, then $(Prefix(R) \rhd [o]) \subseteq Prefix(R)$.* ⋆

Proof. Let o be an oclet with a history and let R be a set of distributed runs s.t. $(Prefix(R) \rhd [\hat{o}]) \subseteq Prefix(R)$. To show: $(Prefix(R) \rhd [o]) \subseteq Prefix(R)$.

Let $\pi \in Prefix(R)$. Without loss of generality, we pick that oclet o of its class $[o]$ s.t. o is enabled at π, i.e., π ends with o's history. To show the lemma's proposition, we have to show $(\pi \rhd o) \in Prefix(R)$. We do so by induction on the number $n = |E_{con_o}|$ of events in o's contribution.

$[n = 0]$. Then $\pi_o = hist_o$, and $con_o = \emptyset$. Thus, $\pi \rhd o = \pi \in Prefix(R)$.

$[n > 0]$. Let $e \in E_{con_o}$ be a maximal event, i.e., there is no $e' \in E_{con_o}$ with $e <_o e$. Then $C := E_o \setminus \{e\}$ is a configuration of π_o. Let $Cut(C) = (C^\bullet \cup \min \pi_o) \setminus {}^\bullet C$ be the cut of π reached by C. Figure 6.12 illustrates the situation.

1. Claim: $\max hist_o \leq_o Cut(C)$. To show: each $b \in Cut(C)$ has a predecessor in $\max hist_o$. Either b is a condition of the contribution, i.e., $b \notin X_{hist_o}$. Then from $\min \pi_o = \min hist_o$ of Def. 4.1 follows $\max hist_o \leq_o b$. In the other case: $b \in X_{hist_o}$. Because all events of $hist_o$ are in C, the $Cut(C)$ can only contain conditions of $\max hist_o$. Thus $b \in \max hist_o$.
2. We decompose o along $Cut(C)$ into $o^-[Cut(C)]$ and $o^+[Cut(C)]$ by Def. 6.7. The number of events in the contribution of $o^-[Cut(C)]$ is $n-1$. By inductive

assumption holds $\pi \rhd o^-[\mathit{Cut}(C)] \in \mathit{Prefix}(R)$. By the choice of e being a maximal condition, the contribution of $o^+[\mathit{Cut}(C)]$ consists of e and e's post-conditions. Thus, Lemma 6.11 implies $o[e] \preceq o^+[\mathit{Cut}(C)]$ and we can replace $o^+[\mathit{Cut}(C)]$ by $o[e]$.

3. Set $o^- := o^-[\mathit{Cut}(C)]$ and $o^+ := o^+[\mathit{Cut}(C)]$. By induction assumption, $\pi \rhd o^- \in \mathit{Prefix}(R)$.

$$\begin{array}{rrll} & o^+ & \text{enabled at} & o^- & \textit{[by cut decomposition, Lem. 6.8]} \\ \Rightarrow & o^+ & \text{enabled at} & \pi \rhd o^- & \textit{[by } \rhd \textit{ associative, Lem. 5.8]} \\ \Rightarrow & o[e] & \text{enabled at} & \pi \rhd o^- & \textit{[by } o[e] \preceq o^+\textit{, Lem. 6.10]} \end{array}$$

From the lemma's assumption, $(R \rhd [\hat{o}]) \subseteq \mathit{Prefix}(R)$, follows $(R \rhd [o[e]]) \subseteq \mathit{Prefix}(R)$ by Def. 5.12 because $o[e] \in \hat{o}$. Thus, $\rho := (\pi \rhd o^-) \rhd o[e] \in \mathit{Prefix}(R)$.

$$\begin{array}{rcll} \rho & = & (\pi \rhd o^-) \rhd o[e] & \\ & = & (\pi \rhd o^-) \rhd o^+ & \textit{[by } o[e] \preceq o^+\textit{, Lem. 6.10, and } o^+ \textit{ enabled at } o^-\textit{]} \\ & = & \pi \rhd (o^- \rhd o^+) & \textit{[by } \rhd \textit{ associative, Lem. 5.8]} \\ & = & \pi \rhd o & \textit{[by cut decomposition, Lem. 6.8]} \end{array}$$

Thus, $\pi \rhd o \in \mathit{Prefix}(R)$. □

We just have proved that for every oclet o with history, its basic oclets $\hat{o}$ can always be composed to a complete occurrence of o.

The corresponding proposition for an ε-oclet $[o] \in O$ reads as follows. If $\pi(O)$ is the least common initial state of R, and R is oclet-closed with the basic oclets $[\hat{o}]$, then o is an initial run of R as in Definition 4.7.

Lemma 6.13 (Basic oclets compose to initial run): *Let o be ε-oclet and let R be a set of distributed runs. If $\pi(O) \in \mathit{Prefix}(R)$ and $(\mathit{Prefix}(R) \rhd [\hat{o}]) \subseteq \mathit{Prefix}(R)$, then $o \in \mathit{Prefix}(R)$.* ⋆

Proof. Let O be an oclet specification, let $[o] \in O$ be the class of an ε-oclet $o = (\pi_o, \varepsilon)$. Further, let R be a set of distributed runs s.t. $\pi(O) \in \mathit{Prefix}(R)$ and $(\mathit{Prefix}(R) \rhd [\hat{o}]) \subseteq \mathit{Prefix}(R)$. To show: $o \in \mathit{Prefix}(R)$.

We first decompose o into its initial state o^- and the rest o^+, then show that o^- is a prefix in R, and finally conclude that $o^- \rhd o^+ = o$ is also a prefix in R.

Let $B^* := \min \pi_o$. By assumption (6.1) holds $B^* \subseteq B_o$; thus B_o is a cut of π_o. Let $o^-[B^*] = (\pi^*, \varepsilon)$ and $o^+[B^*] = (\pi_o, \pi^*)$ be the decomposition of o along B^* (by Def. 6.7).

1. By Def. 6.7, $o^-[B^*] = (\pi^*, \varepsilon)$ with $\pi^* = (B^*, \emptyset, \emptyset)$ and $o^+[B^*] = (\pi_o, \pi^*)$. Thus, $o^+[B^*]$ and o contribute the same events. Thus, the basic oclets of o and $o^+[B^*]$ are identical, and from Lem. 6.12 follows $(\mathit{Prefix}(R) \rhd [o^+[B^*]]) \subseteq \mathit{Prefix}(R)$.

2. By definition of the play-out semantics, $\mathit{Prefix}(R)$ contains the unique initial run $\pi(O)$ of O with $\pi(O) = (B_{\text{init}}, \emptyset, \emptyset)$ (up to isomorphism of B_{init}). By Def. 6.3, B_{init} contains B^*. Thus, $o^-[B^*]$ occurs as a prefix of $\pi(O) \in \mathit{Prefix}(R)$, hence $o^-[B^*] \in \mathit{Prefix}(R)$.

3. $o^+[B^*]$ is enabled at $o^-[B^*]$ by Lemma 6.8. Thus, $o^-[B^*] \rhd o^+[B^*] = o \in \mathit{Prefix}(R)$ by Lem. 6.12. □

Proof of Theorem 6.6: oclet play-out satisfies oclet specifications. By the help of these two lemmas, we can prove Theorem 6.6 which states that the play-out behavior $\hat{R}(O)$ satisfies O.

Proof (of Theorem 6.6). Set $R := \hat{R}(O)$. To show $R \in \mathcal{R}(O)$ it suffices to show (1) $(\mathit{Prefix}(R) \rhd [o]) \subseteq \mathit{Prefix}(R)$, for each $[o] \in O \setminus O_\varepsilon$ by Theorem 5.13, and (2) $o \in \mathit{Prefix}(R)$, for each ε-oclet $o \in O_\varepsilon$, by Def. 4.7.

(1) Let $[o] \in O \setminus O_\varepsilon$. By Def. 6.5, and Thm. 5.18 holds $(\mathit{Prefix}(R) \rhd [\, o[e]\,]) \subseteq \mathit{Prefix}(R)$ for each basic oclet $o[e] \in \hat{o}$. Thus, $(\mathit{Prefix}(R) \rhd [\hat{o}]) \subseteq \mathit{Prefix}(R)$ by Def. 5.12. Thus, Lemma 6.12 implies $(\mathit{Prefix}(R) \rhd [o]) \subseteq \mathit{Prefix}(R)$ and Thm. 5.13 implies $R \in \mathcal{R}([o])$.

(2) Let $[o] \in O_\varepsilon$. By the same arguments as above holds $(\mathit{Prefix}(R) \rhd [\hat{o}]) \subseteq \mathit{Prefix}(R)$. Further, $\pi(O) \in \mathit{Prefix}(R)$ by Def. 6.5. Thus, Lemma 6.13 implies $o \in \mathit{Prefix}(R)$. Thus, $R \in \mathcal{R}([o])$ by Def. 4.7 and Def. 4.9.

From Corollary 4.11 follows $R = \hat{R}(O) \in \mathcal{R}(O)$. □

By the proof of Theorem 6.6, oclet play-out is consistent with the semantics of oclets. By Theorem 5.18 and Corollary 6.4, the set $\hat{R}(O)$ is *the least set of distributed runs* that

1. has a unique initial state, and
2. is composed from the basic oclets of O.

6.6. Specifications vs. Distributed Systems Revisited

In Chapter 5 we introduced oclet composition and showed that repeatedly composing the oclets of a specification O yields the set $R(O)$ of runs that satisfies O, is minimal, and unique. Given the minimality of $R(O)$, we then approached the problem of a *minimal implementation*.

Definition 6.14 (Minimal implementations). Let O be an oclet specification and let Σ be an implementation of O, i.e., $R(\Sigma) \in \mathcal{R}(O)$. Σ is a *minimal implementation* of O iff there exists no implementation Σ' of O s.t. $R(\Sigma') \subset R(\Sigma)$. ⌟

We had seen in Section 5.5 that there may be no implementation Σ of O that exhibits exactly this minimal behavior $R(\Sigma) = R(O)$ but only a superset of $R(O)$. Put differently, the behavior of any minimal implementation of O may be strictly larger than $R(O) = \min \mathcal{R}(O)$. O may have multiple minimal implementations. However, in this section we show that all minimal implementations of O exhibit the same behavior as sketched in Figure 6.17.

The preceding sections bridged the gap between satisfying behavior of a specification O and behavior exhibited by its implementations. To this end, we adapted

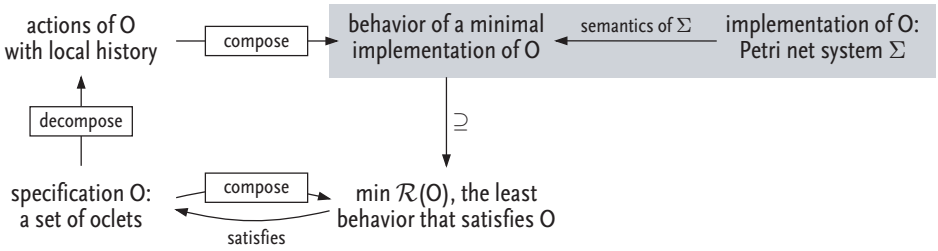


Figure 6.17. Every minimal implementation of an oclet specification O exhibits the play-out behavior of O.

behavioral assumptions of an implementation to scenarios: an implementation starts in a unique initial state and exhibits occurrences of single events instead of entire scenarios. To formalize these assumptions, we adapted the idea of scenario play-out to the formal model of oclets. This approach constructs the *play-out behavior* $\hat{R}(O)$ by composing the single events of O instead of entire oclets — starting in O's least common initial state. By construction, $\hat{R}(O)$ satisfies O, is minimal wrt. composition of single events, and unique. In this section, we discuss the relation between $R(O)$ and $\hat{R}(O)$ and finally conclude that

> *every distributed system with a unique initial state that implements O inevitably exhibits the play-out behavior of O.*

The play-out behavior of O contains more runs than the least behavior that satisfies O, i.e., $R(O) \subseteq \hat{R}(O)$, according to Lemmas 6.12 and 6.13. In general, the inclusion is *strict*, i.e., $R(O) \subset \hat{R}(O)$. In the following, we show that the additional behavior $\hat{R}(O) \setminus R(O)$ are those runs that inevitably occur in a minimal implementation Σ of O — if Σ is a *distributed system*.

The behavior of a distributed system is governed by the principle that it evolves by occurrences of *local actions*. Whether a specific local action occurs depends on the action's *local preconditions*. The play-out behavior of oclets exactly reflects this principle by decomposing a specification into its basic oclets. We highlight how oclet play-out reflects local actions and local preconditions by two examples.

A distributed system has local actions

Figure 6.18 repeats the oclet specification O from Figure 5.11. The least behavior that satisfies O has the maximal runs $\rho_1 := \mathsf{o_1} \rhd \mathsf{o_2}$ and $\rho_2 := \mathsf{o_1} \rhd \mathsf{o_3}$ (not depicted).

We already argued in Section 5.5 that the Petri net system Σ in Figure 6.18 is a minimal implementation of O: no implementation of O exhibits less behavior than Σ. But Σ inevitably exhibits the run ρ_3 which is not in $R(\mathsf{O})$. The reason is that Σ implements O only if it implements actions x, y, and z as single transitions with respective pre- and post-places. Whether a transition t of Σ occurs depends on t's pre-places only; an occurrence of t yields a *single* event. For this reason, transitions y and z occur independently after x occurred which yields ρ_3.

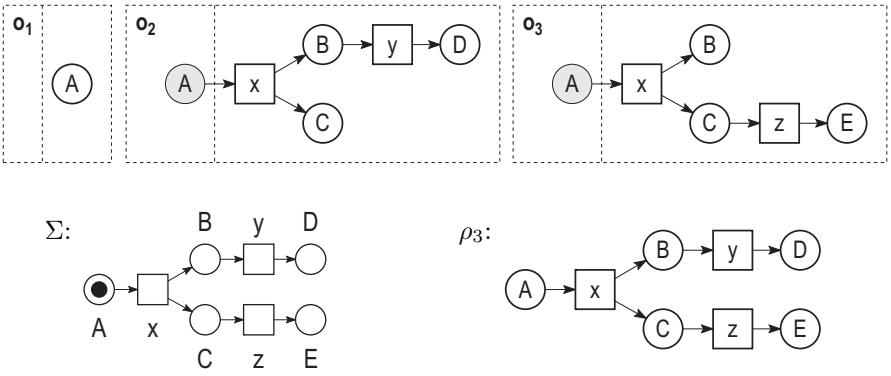


Figure 6.18. The Petri net system Σ implements the play-out behavior of specification specification $\mathsf{O} = \{\mathsf{o}_1, \mathsf{o}_2, \mathsf{o}_3\}$: $R(\Sigma) = \mathit{Prefix}(\rho_3) = \hat{R}(\mathsf{O})$.

The basic oclets of O exactly capture that each action of a distributed system occurs by a single event: each basic oclet contributes a single event that depends only on its local history. Playing out the single events x, y, and z of O yields exactly the inevitable run ρ_3.

An action of a distributed system has local preconditions

Each oclet distinguishes a history to describe when a specific scenario may occur. This history can describe behavior across several components. In contrast, in a distributed system, a single action may occur whenever its local preconditions hold, regardless of whether some history across several components has occurred or not.

We highlight the difference between the notion of a global history (of an oclet) and a local precondition (of an action) by comparing the oclet specification O_1 in Figure 6.19 to specification O_2 in Figure 6.20 wrt. their least satisfying behavior $R(.)$ and their play-out behavior $\hat{R}(.)$.

Both specifications describe a simple ping-pong protocol between two processes a and b. The special oclet drop allows b to terminate the protocol. We first show that O_1 and O_2 allow the same oclet compositions, i.e., both allow the same "composition terms". For instance, we can compose the infinite run $\rho_1 = \mathsf{init} \triangleright \mathsf{pong} \triangleright \mathsf{ping} \triangleright \mathsf{pong} \triangleright \ldots$ in O_1 and in O_2. Another run is $\rho_2 = \mathsf{init} \triangleright \mathsf{pong} \triangleright \mathsf{ping} \triangleright \mathsf{drop}$. Figures 6.19 and 6.20 depict the respective runs. In this respect, the minimal satisfying behavior $R(\mathsf{O}_1)$ is similar to the minimal satisfying behavior $R(\mathsf{O}_2)$, i.e., "$R(\mathsf{O}_1) \approx R(\mathsf{O}_2)$".

Thus, also a minimal implementation Σ_1 of O_1 is similar to a minimal implementation of O_2. But, Σ_1 inevitably exhibits more behavior than O_1 whereas a minimal implementation of O_2 is equivalent to O_2. This difference arises from the fact that both specifications differ wrt. how a and b exchange information. The play-out behavior of both specifications makes this difference between O_1 and O_2 on the one hand, and their minimal implementations on the other hand explicit.

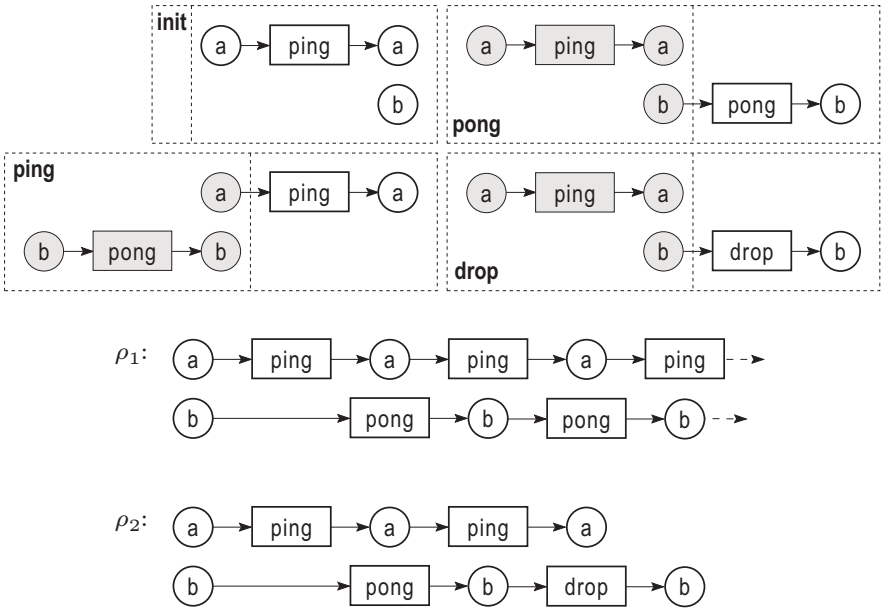


Figure 6.19. Composing entire oclets $\mathsf{O}_1 = \{\mathsf{init}, \mathsf{ping}, \mathsf{pong}, \mathsf{drop}\}$ yields runs such as $\rho_1 = \mathsf{init} \rhd \mathsf{pong} \rhd \mathsf{ping} \rhd \mathsf{pong} \rhd \ldots$, and $\rho_2 = \mathsf{init} \rhd \mathsf{pong} \rhd \mathsf{ping} \rhd \mathsf{drop}$.

Figure 6.21 depicts the basic oclets $\mathsf{ping}[\mathsf{ping}]$ of specification O_1 to the left and of O_2 to the right. Using $\mathsf{ping}[\mathsf{ping}]$ we can compose the distributed run $\rho_3 = \rho_2 \rhd \mathsf{ping}[\mathsf{ping}] \rhd \mathsf{ping}[\mathsf{ping}] \rhd \ldots$ in O_1 but not in O_2. In the play-out behavior, the occurrence of $\mathsf{ping}[\mathsf{ping}]$ of O_1 only depends on its local pre-condition a. In contrast, $\mathsf{ping}[\mathsf{ping}]$ of O_1 depends on a preceding occurrence of pong.

Considering the *causal dependencies* described in O_1 and O_2 respectively, we see that O_1 does not specify any dependencies between a and b. In other words, O_1 does not specify any flow of information between a and b. Thus, a minimal implementation of O_1 does not exchange information between a and b either. As a consequence, any implementation of O_1 inevitably exhibits the additional run ρ_3. In contrast, O_2 does specify that a and b exchange information.

In oclet play-out, the local history of a *basic oclet* $o[e]$ captures exactly all causal dependencies of event e when e is implemented in a distributed system. The local history of e contains all events and conditions on which it causally depends, i.e., it contains all flow of information implemented in a distributed system. Moreover, the local history of e contains *nothing else*, i.e., it contains *only*

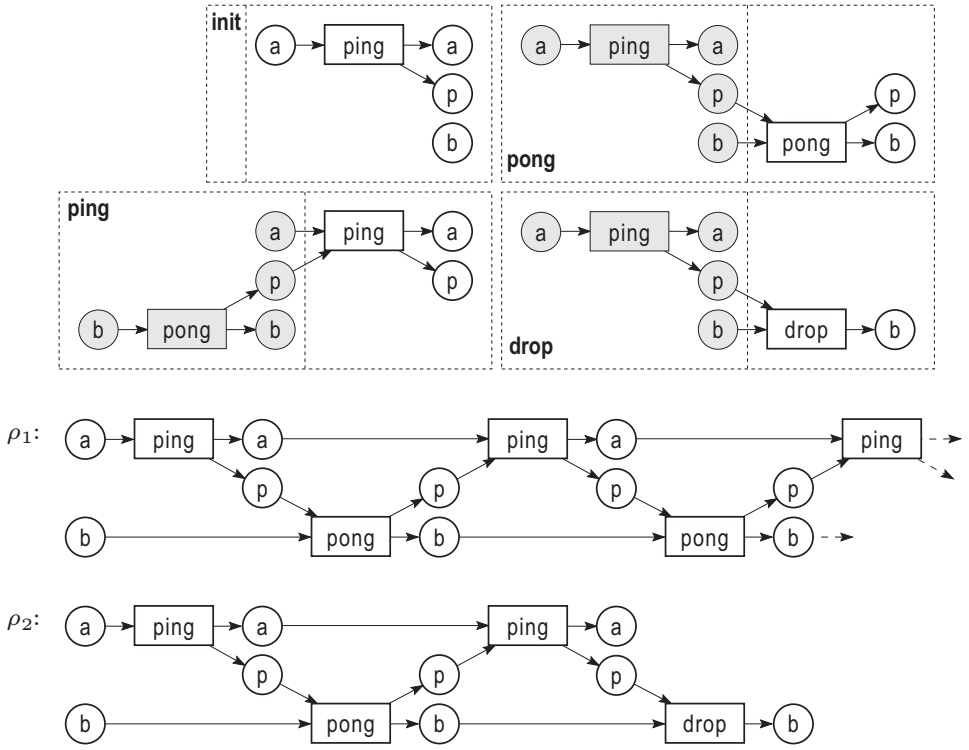


Figure 6.20. Composing entire oclets $\mathsf{O}_2 = \{\mathsf{init}, \mathsf{ping}, \mathsf{pong}, \mathsf{drop}\}$ yields runs such as $\rho_1 = \mathsf{init} \triangleright \mathsf{pong} \triangleright \mathsf{ping} \triangleright \mathsf{pong} \triangleright \ldots$, and $\rho_2 = \mathsf{init} \triangleright \mathsf{pong} \triangleright \mathsf{ping} \triangleright \mathsf{drop}$.

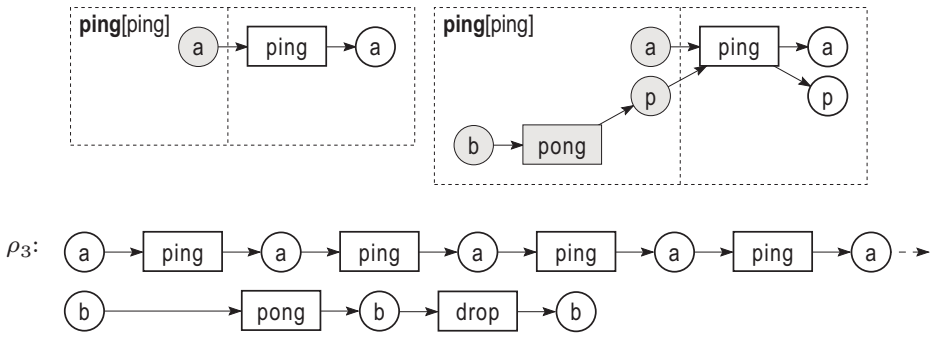


Figure 6.21. The difference between O_1 and O_2. The distributed run ρ_3 is a run in the play-out behavior of O_1 but not in the play-out behavior of O_2.

the flow of information implemented in a distributed system. For this reason we can conclude the following.

> The play-out behavior $\hat{R}(O)$ of an oclet specification O reflects exactly the behavior exhibited by *any minimal implementation* of O. (6.2)

This conclusion (6.2) is only a statement about the *behavior* of minimal implementations. An oclet specification O may have several minimal implementations Σ_1, Σ_2 exhibited the same behavior $R(\Sigma_1) = \hat{R}(O) = R(\Sigma_2)$.

Conclusion: what is a synthesis algorithm?

To conclude, the play-out behavior $\hat{R}(O)$ of an oclet specification O precisely describes the least set of runs that satisfies O and that can be implemented by a distributed system. In other words, $\hat{R}(O)$ describes the behavior of a *minimal implementation* of O. No implementation of O exhibits less behavior than $\hat{R}(O)$ — considering implementations with a *unique initial state* only. We know from Theorem 5.18 that $\hat{R}(O)$ is unique — based on the unique minimal initial state of an implementation of O by Corollary 6.4. Thus, we have found the *behavioral definition* of a *synthesis algorithm*.

> A *synthesis algorithm* is an algorithm Syn that returns for for every oclet specification O a system model Σ s.t. Σ exhibits exactly the play-out behavior of O, i.e., $R(\Sigma) = \hat{R}(O)$. (6.3)

6.7. Applying Play-Out

By introducing oclet play-out we have reached a point where an oclet specification no longer just describes desired behavior of some yet unknown system. By the two notions of (1) an initial state, and (2) individual events that may occur when enabled locally, an oclet specification turns into a system model with operational semantics. To emphasize this new quality, we introduce the notion of an *oclet system*.

Definition 6.15 (Oclet system). An *oclet system* $\Omega = (O, \pi_0)$ consists of a finite set O of classes of basic oclets, and a finite distributed run π_0. The run π_0 is called *initial* run. ⌟

In the most common case, π_0 contains no events, that is, π_0 describes an *initial state*. However, π_0 may contain events. This slight generalization of the normal form of an oclet specification (Def. 6.5) has technical reasons for the next chapters. Every oclet specification O induces its unique oclet system $\Omega(O) := (\hat{O}, \pi(O))$ consisting of the basic oclets $\hat{O}$ of O and the unique initial state $\pi(O)$ of O. Technically, the semantics of an oclet system is already defined by play-out behavior of oclets in

Definition 5.17. But again, to emphasize the operational nature of oclet systems, and for the results of the next chapters, we explicitly define the semantics of an oclet system.

Definition 6.16 (Distributed runs of an oclet system). Let $\Omega = (O, \pi_0)$ be an oclet system. The behavior of Ω is the set $R(\Omega)$ of distributed runs defined inductively as follows:

1. $\pi_0 \in R(\Omega)$
2. Let $\pi \in R(\Omega)$ and let $o \in O$ be a basic oclet. If o is enabled at π at location α, then $\pi \rhd o^\alpha \in R(\Omega)$. ⌟

Corollary 6.17: *Oclet systems and oclet specifications corresponds to each other as follows.*

1. *Let $\Omega = (O, \pi_0)$ be an oclet system. The behaviors of Ω are the play-out behaviors of the oclet specification $O_\Omega := (O \cup \{\ [\pi_0]\ \})$ that start with π_0, i.e., $R(\Omega) = \{\rho \in \hat{R}(O_\Omega) \mid \pi_0 \sqsubseteq \rho\}$. If the initial run π_0 of Ω contains no events, then $R(\Omega) = \hat{R}(O_\Omega)$.*
2. *Let O be an oclet specification. O induces the oclet system $\Omega(O) = (\hat{O}, \pi(O))$; see (Def. 6.5). The play-out behaviors of O and the behaviors of $\Omega(O)$ are equivalent, i.e., $R(\Omega(O)) = \hat{R}(O)$.* ⋆

In the following, we show how an oclet system Ω implement an oclet specification O by the operational semantics of Ω, i.e., we use oclets to model systems. Chapters 7 and 8 use oclet systems to *synthesize* a Petri net from O.

6.7.1. Oclet systems unify Petri net systems and oclet specifications

An oclet system not only relates a specification to its play-out behavior. Also every Petri net system Σ has an equivalent oclet system $\Omega(\Sigma)$. In Section 5.5 we showed that for every Petri net system Σ exists an equivalent oclet specification O_Σ so that Σ exhibits exactly the minimal behavior specified in O, i.e., $R(\Sigma) = R(O)$. This result canonically extends to oclet systems.

It is easy to see from the proof of the corresponding Lemma 5.19 that there exists an oclet specification O_Σ having one ε-oclet, and only basic oclets otherwise. Thus, the play-out behavior of O_Σ is the minimal satisfying behavior of O_Σ: $\hat{R}(O_\Sigma) = R(O_\Sigma)$. Consequently, the oclet system $\Omega(O_\Sigma)$ exhibits the same behavior as Σ. The converse does not hold in general. Intuitively, a large history of a (basic) oclet allows to express enabling conditions for an event that cannot be expressed by a marking of a Petri net. We prove that oclets and oclet systems are *strictly more expressive* than Petri nets in Chapter 8.

6.7.2. Implementing a specification by play-out

Corollary 6.17 from the beginning of this Section 6.7 establishes an unusual solution for the synthesis problem. The algorithm PlayOut with $\textsc{PlayOut}(O) := \Omega(O)$

constructs for every oclet specification O the oclet system $\Omega(O)$ as defined in Corollary 6.17. Algorithm PlayOut(O) is a synthesis algorithm that satisfies the requirements stated in (6.3) on page 171. There are several situations where directly executing a scenario-based specification by play-out suffices to implement the specification. For these situations, play-out is a *universal solution* to implement any scenario-based specification. We present two such situations in Section 6.7.3 in more detail. In this section, we present what is needed to implement an oclet specification by play-out.

The general idea of implementing a specification by play-out has been introduced first for LSCs in [56]; in that work, Harel and Marelly advocate play-out (in combination with the complimentary play-in) as a technique that applies to "many stages of system development, including requirements engineering, specification, testing, analysis and implementation." Play-out was implemented first for LSCs in the *play engine* [55].

A general execution engine for oclets

Like for LSCs, it needs an *execution engine* to implement an oclet specification O by play-out. This engine determines when an event of O is enabled an what happens when an enabled event occurs. Technically, such an engine has to implement Definition 6.16. Thus, O is first transformed into its oclet system $\Omega(O) = (\hat{O}, \pi(O))$ according to Corollary 6.17. Then O is executed as follows. A distributed run π denotes the *history of the current execution* of O.

1. The engine starts in the initial run $\pi := \pi(O)$ of O.
2. To extend the current execution, first determine the set of all events that are enabled at π: for each basic oclet class $[o] \in \hat{O}$ determine all $o^\alpha \in [o]$ s.t. π ends with $hist_o$.
3. Then choose one enabled event e (either non-deterministically or by user selection) and continue the current execution π by e, i.e., $\pi := \pi \triangleright o^\alpha$, for one enabled basic oclet o^α. Repeat steps (2) and (3) until no event is enabled.

The complexity of extending the current execution π by one event depends on the number of basic oclets in the oclet system and the complexity of checking whether a basic oclet $[o] \in \hat{O}$ is enabled. Whether an oclet $o = (\pi_o, hist_o)$ is enabled at π at some location α can be checked in non-deterministic polynomial time ($\mathcal{NP}$): guess a sub-net $N' \subseteq \pi$ at the end of π and check whether $hist_o$ and N' are isomorphic (by isomorphism α). If a basic oclet o^α is enabled at π, then extending π by o^α takes linear time in the size of the contribution of o^α.

An efficient execution engine for oclets

The preceding general execution engine can be improved for a special class of oclet specifications. Let O be an oclet specification s.t.

a) in each run $\pi \in \hat{R}(O)$ any two different conditions in $\max \pi$ have a different label, and

b) for each oclet $[o] \in O$ holds, no node x of o has two pre-nodes in ${}^\bullet x$ with the same label.

For oclet specifications of this kind, checking whether a basic oclet $o[e]$ is enabled at a distributed run π of $\Omega(O)$ requires only a simultaneous breadth-first search on $hist_o$ and π.

Algorithm 6.18.

input: The history $hist_o$ of an oclet o and a distributed run π.
output: The location α where $hist_o$ occurs in π or *false*.
begin
  $\alpha :=$ the empty mapping
  $Q :=$ the empty queue
  forall $y \in \max hist_o$ **do**
    if there exists $y' \in \max \pi$ with $\ell(y) = \ell(y')$ **then**
      $\alpha(y) := y'$
      append (y, y') to Q
    else return *false*
  endfor
  while Q not empty **do**
    $(x, x') := dequeue(Q)$
    forall $y \in {}^\bullet x$ **do**
      if there exists $y' \in {}^\bullet x'$ with $\ell(y) = \ell(y')$
          and for no y'' holds $\alpha(y'') = y'$ **then**
        $\alpha(y) := y'$
        append (y, y') to Q
      else return *false*
    endfor
  endwhile
return α
end

⌟

Lemma 6.19: *Let O be an oclet specification satisfying the above conditions a) and b). Let $\pi \in \hat{R}(O)$ be a distributed run.*

1. *A basic oclet $o[e] \in \hat{O}$ is enabled at π at α iff Algorithm 6.18 returns α for arguments $hist_{o[e]}$ and π.*
2. *Algorithm 6.18 takes time $\mathcal{O}(n+m)$, where $n = |X_{hist_{o[e]}}|$ and $m = |F_{hist_{o[e]}}|$ are the number of nodes and edges in the history of $o[e]$, respectively.* ⋆

Proof. (*1.*) We show that the result of Alg. 6.18 is correct and complete.

In the initial forall-loop holds: if there is a $y \in \max hist_o$ with no matching $y' \in \max \pi$, then π does not end with $hist_o$ which implies that o is not enabled at π (by Def. 5.1). Further there is at most one such y' by assumption a), i.e.,

the search is complete. Thus, at the end of the initial forall-loop, α denotes an injection of $\max hist_o$ into $\max \pi$.

In the breadth-first search holds: if there is a $y \in {}^\bullet x$ with no matching $y' \in {}^\bullet\alpha(x)$, then $hist_o$ does not occur in π which implies that o is not enabled at π. If there is a $y' \in {}^\bullet\alpha(x)$ which is matched by another $y'' \in {}^\bullet x'', y'' \neq y, x'' \neq x$, then α cannot be extended to an injection from $hist_o$ to π. Thus, $hist_o$ does not occur as a sub-net of π. Further there is at most one such y' by assumption b), i.e., the search is complete. Thus, after each iteration, α denotes an injection of $hist_o|_Y$ into $\max \pi$ for the current domain $Y = dom(\alpha)$ of α.

Because all nodes of $hist_o$ are transitive predecessors of $\max hist_o$ and because $hist_o$ is finite, Alg. 6.18 either terminates with an injection α from $hist_o$ to π s.t. $hist_o^\alpha \subseteq \pi$ and $\max hist_o^\alpha \subseteq \max \pi$, or Alg. 6.18 returns *false* if no such injection exists.

(*2.*) The breadth-first search completes after at most $|X_{hist_o}|$ iterations. In each iteration α is extended by one pair $\alpha(y) := y'$. To find y', the matching takes at most $k = |{}^\bullet x|$ steps. Further, each node of $hist_o$ is visited at most once. Altogether, the search takes $n + m$ steps with $n = |X_{hist_o}|$ nodes and $m = |F_{hist_o}|$ arcs. □

The conditions a) and b) on specification O given at the beginning of this section are not as restrictive as they may seem.

a) Having no two conditions with the same label s in $\max \pi$ intuitively demands that no component of the system is in the same state s twice at the same time. This restriction corresponds to the notion of a safe marking in Petri nets; Petri nets where only safe markings are reachable are considered as the practically most relevant class of nets [38].

b) The second restriction demands that no event has two pre-conditions with the same label. This restriction is violated, for instance, by an event that receives two indistinguishable messages at the same time. In Petri nets, it needs arc weights to let a transition consume two indistinguishable tokens from the same place. However, most systems in practice do not need this expressivity and satisfy b).

If requirements a) and b) hold, then executing one step in an oclet system $\Omega = (O, \pi_o)$ requires time $\mathcal{O}(|O| \cdot (n + m) + l)$, where n and m are the number of nodes and arcs, respectively, in the basic oclet $o \in O$ with the largest history, and l is the size of the largest contribution in O (needed for composing $\pi \rhd o$).

Despite the play-out solution, a system designer may still be interested in synthesizing a classical system model in a specific formalism, for instance, a Petri net. We present an algorithm for synthesizing a Petri net system from an oclet specification in Chapter 8. In the following, we present an implementation of an execution engine for oclets and how oclet systems and oclet play-out contribute to system design.

6.7.3. Tool support: an execution engine for oclets

We just described the principle idea of an execution engine for oclets. We implemented a prototype of an execution engine for oclets in our tool Greta. This section first presents some technical details of the implementation. Afterwards, we consider how an execution engine supports system modeling with scenarios, and how oclet play-out suits the domain of *process management*.

Combining scenario-based modeling and animated execution

Section 4.6.4 already mentioned our software prototype Greta which implements the results of this thesis. Greta provides a graphical editor for creating oclet specifications, and has a plug-in architecture which allows to extend its functionality. One of these plug-ins is an *execution engine* for oclets, which we explain in the following.

A system designer who uses Greta for modeling a distributed system with oclets first uses the graphical editor to model all system scenarios she has in mind as oclets. The graphical editor stores oclets and execution histories in separate data structures. These data structures and the user interface are exposed to plug-ins as illustrated in Figure 6.22.

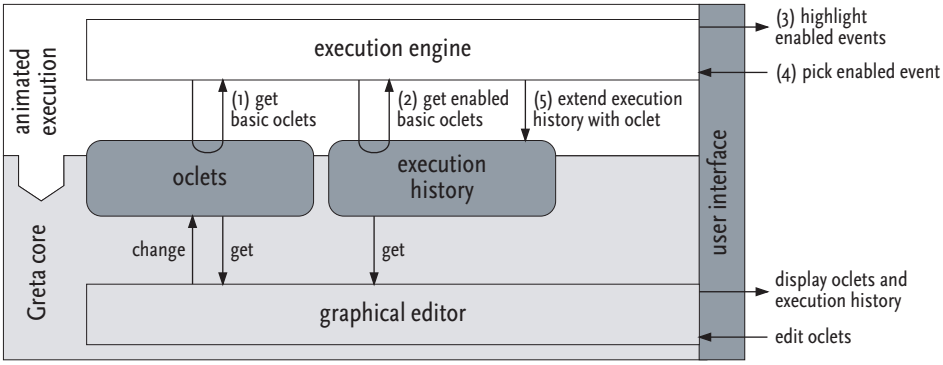


Figure 6.22. Greta implements animated execution as a plug-in to the graphical editor.

The execution engine allows to *validate* a model by *animated execution*. To execute one action of the modeled system, the engine performs the following steps. The execution engine (1) decomposes the current oclets in the model into basic oclets (Def. 6.1), (2) determines all enabled basic oclets (Alg. 6.18), and (3) highlights for each enabled basic oclet $o[e]$ the event e in the originating oclet o in the user interface. The basic oclets themselves are not shown to the system designer. The user (4) chooses an enabled event e, and the execution engine (5) extends the execution history by the corresponding basic oclet $o[e]$ (Def. 6.16). Greta's editor displays the extended execution history.

The system designer can start animated execution at any time during modeling. Figure 6.23 shows Greta's graphical interface during animated execution. Several controls allow to start, stop, pause and reset the execution; see Fig. 6.23(1). The

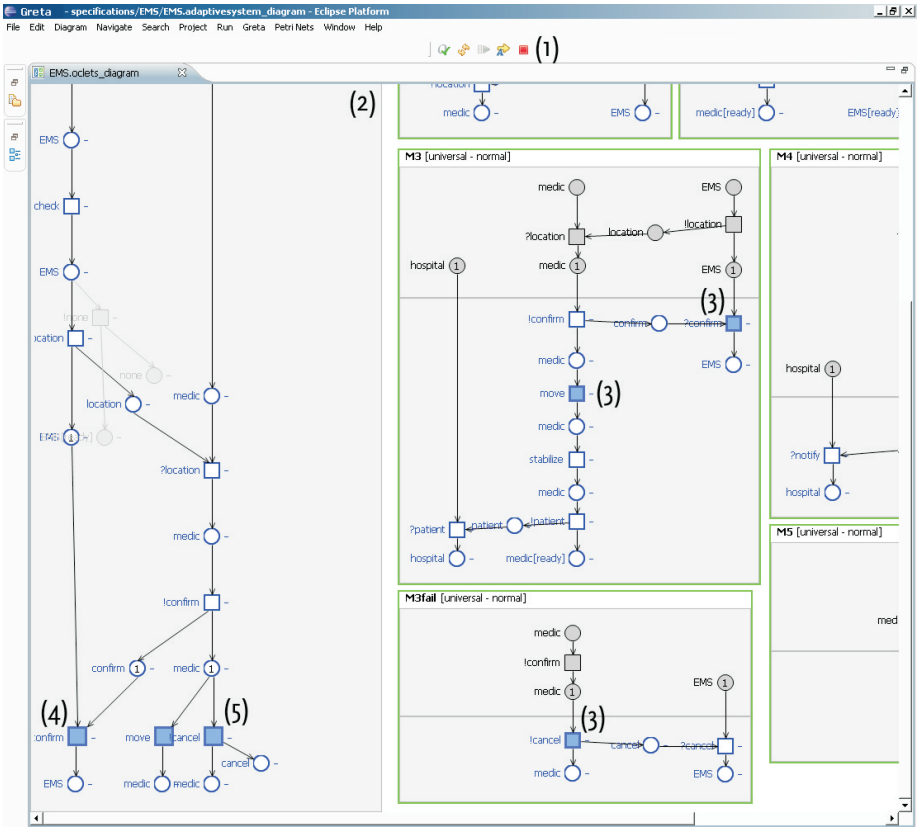


Figure 6.23. Animated execution of a scenario-based specification in GRETA. Simulation controls (1) start and stop simulation. The system designer can extend the current execution history (2) by clicking on enabled events (3). GRETA displays in each step the enabled events as tentative extensions (4,5) of the execution (2); alternative events (5) are shown directly in the execution (2).

current history of the execution is depicted on the left of the screen as a distributed run; see Fig. 6.23(2). GRETA displays the enabled events in two ways. First, by highlighting for each enabled basic oclet $o[e]$ the corresponding event e in its original oclet o; see Fig. 6.23(3). In addition, GRETA constructs a "lookahead" for the execution history: GRETA tentatively extends the execution history by all enabled basic events and highlights these; see Fig. 6.23(4) and (5). Thus, the system designer also sees which events are alternative in the current step; see Fig. 6.23(5). When the system designer clicks on a highlighted event, the clicked event occurs and all tentatively added events are removed. Animated execution continues until no event is enabled or the system designer stops the engine. Furthermore, execution may be paused at any time to edit the oclets.

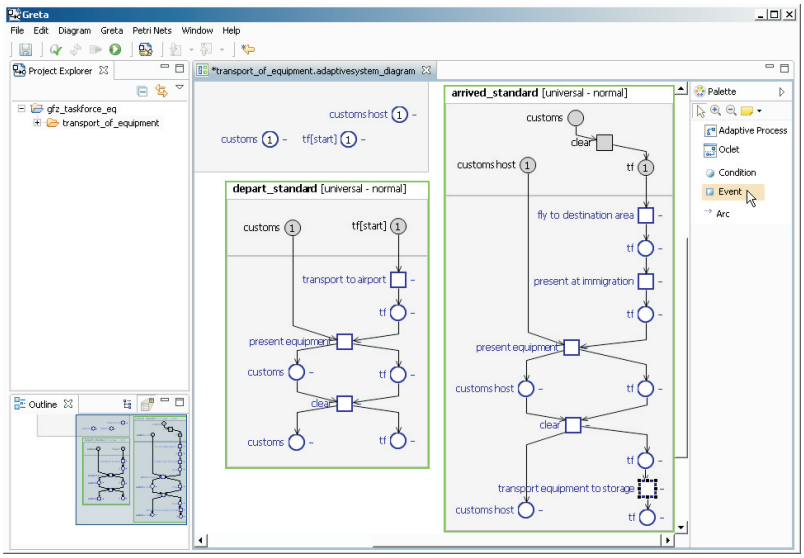


Figure 6.24. Modeling scenarios and variants with oclets. The "transport of equipment" process of the "Taskforce Earthquakes" describes how a scientific expert team handles the transport of scientific equipment from Germany to a disaster area. Oclet depart standard describes how the Taskforce clears its equipment at customs in Germany. Oclet arrived standard describes the corresponding handling in the host country. The upper left compartment describes the initial state of the process as an ε-oclet.

Flexible system modeling with oclets

This interplay of graphical editor and animated execution allows for a flexible approach to system modeling. The following example from the "Taskforce Earthquakes", which we already introduced in Chapter 1, illustrates the approach. A system designer first models standard scenarios of a system such as oclets depart standard and arrived standard of Figure 6.24. Once this is done, the system designer starts animated execution to validate the system model as shown in Figure 6.25; animated execution may be reset or stopped at any time.

When the system designer is satisfied with the current model, she may introduce variants and exceptions of system behavior one oclet at a time such as oclet depart devices shown in Figure 6.26. After new oclets have been introduced, animated execution helps to validate the changed model. If the system designer considers the extended model *invalid*, she may *pause* animated execution and *adapt* the process model as shown in Figure 6.27. Greta preserves the execution history of the animated execution in the meantime. When the system model has been adapted, the system designer may *resume* animated execution to continue validation as shown in Figure 6.28. The tight interplay of graphical editor and animated execution supports a system designer in systematically modeling a complex distributed system with scenarios.

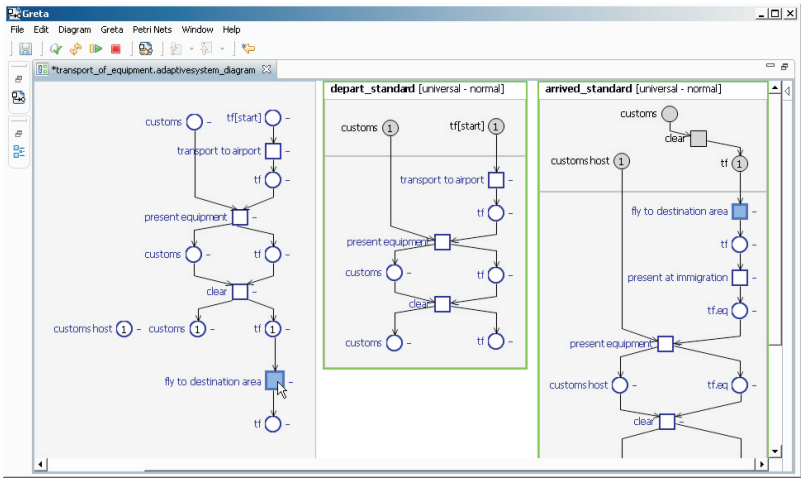


Figure 6.25. Executing a scenario-based model. The left compartment shows an execution history after transport to airport, present equipment, and clear have occurred. Now, fly to destination area is enabled. The user may execute fly to destination area by clicking.

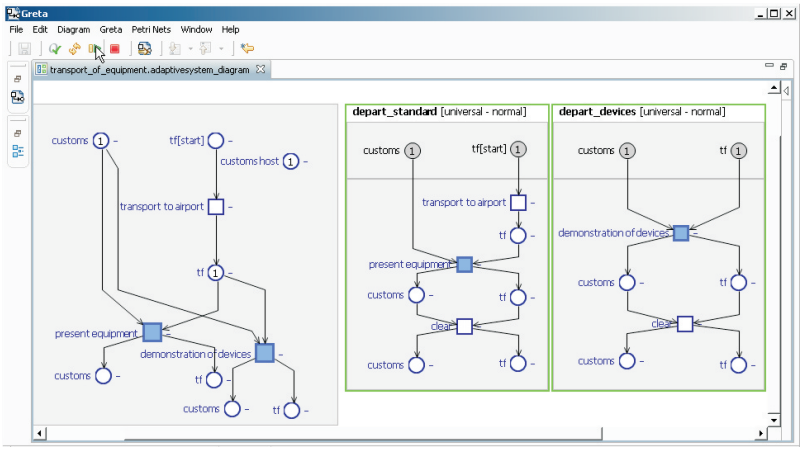


Figure 6.26. Modeling exceptions. Oclet depart devices describes a simple exception of depart standard where the Taskforce additionally has to demonstrate the devices before equipment is cleared. In the execution history on the left, present equipment and the exception demonstration of devices of Fig. 6.26 are enabled as alternative actions.

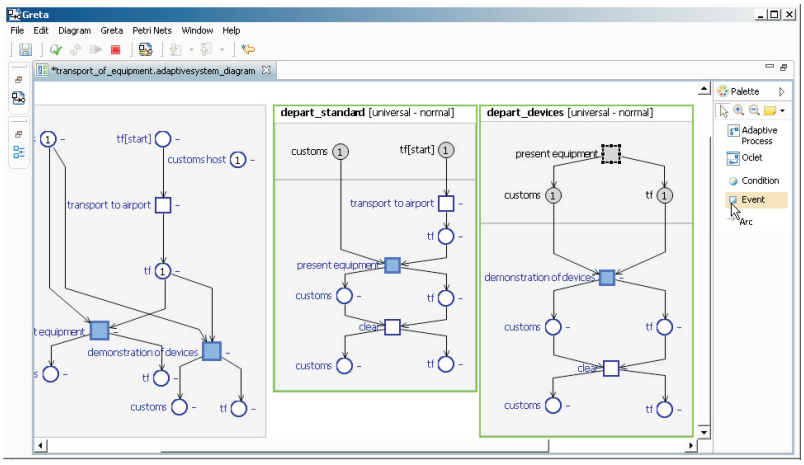


Figure 6.27. Adapting models during execution. In the actual process of the Taskforce, demonstration of devices should only occur *after* present equipment. The animated execution reveals that the model is invalid. The modeler may *pause* the execution of Fig. 6.26, and *change* oclet depart devices by extending its history. When *resuming* execution on the adapted model, action demonstration of devices is no longer enabled.

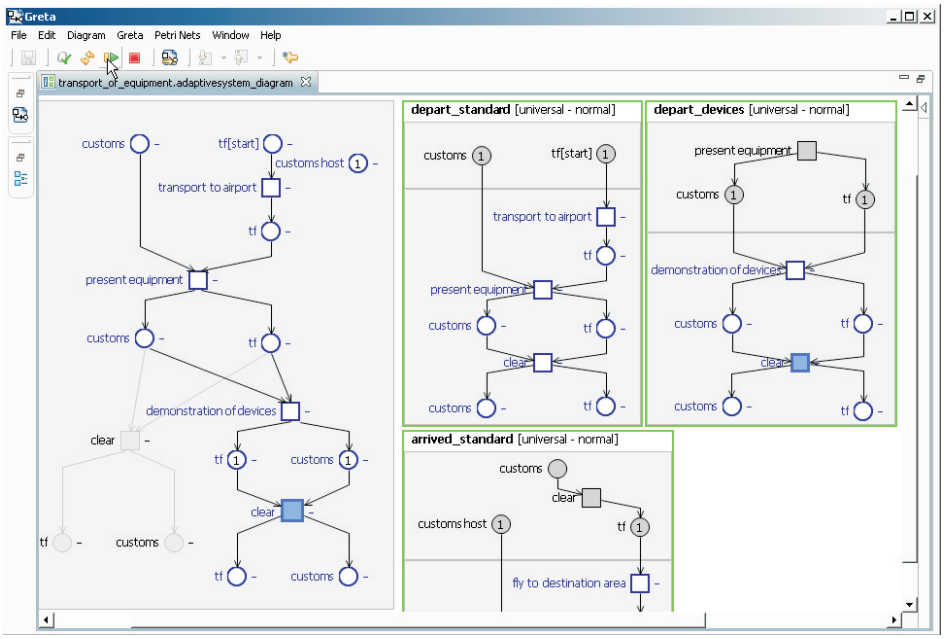


Figure 6.28. Continue execution with adapted model. After completing the adaptation of Fig. 6.27, the modeler may resume execution and continue in the execution history. In the instance on the left, the exception of oclet depart devices is finally handled by action clear as intended. This triggers oclet arrived standard of the standard behavior of Fig. 6.24.

Process management with oclets

The animated execution implemented in Greta can be extended to an actual *execution engine* for processes modeled with oclets. To this end, each event in the specification is associated to some system action such as calling a service, sending a message, waiting for a message, opening a dialogue box, or printing a document. Depending on the kind of action, the engine may automatically choose and execute an enabled event, or ask a user what to do next.

In this respect, the engine executes actions by *interpreting* the specification. This paradigm is used in the domain of process management where a (distributed) *process management system* coordinates the work in large organizations. The system interprets a *process model*, that describes which actions have to be executed in which order. In the majority of the cases, a process model is a (large) graph-based model, e.g., a Petri net [117, 116].

A future process management system could interpret an oclet specifications instead of a Petri net to coordinate work. The advantage of modeling with oclets becomes apparent in this setting. Oclets provide a structuring mechanism for complex system models. The system designer may group different oclets regarding their purpose. For instance, she may distinguish standard behavior from exceptional behavior as illustrated in the example of the process model of the "Taskforce Earthquakes" on page 6. If the actual process changes, also the process model has to be changed. To realize a specific change in an oclet specification, a system designer may follow a two step approach.

First, she focuses her attention on oclets from a specific group and ignores other oclets. Here, the specific advantage of oclets compared to Petri nets is that the system designer may structure her model in a non-hierarchical way. For instance, the Petri net process model of the "Taskforce Earthquakes" shown in Figure 1.4 on page 7 provides no hierarchical decomposition. The oclets of Figure 1.3 describe the same process in terms of behavioral fragments of reasonable size. So, the system designer may concentrate on those oclets directly related to the intended change, understand their behavior and modify the oclets to realize the intended change in the model.

The changed oclets may depend on other oclets in a model, and vice versa. So, in a second step, the system designer considers relations between oclets. Either by visually comparing two or more oclets with each other or by tool support such as animated execution presented previously. If she discovers an undesired interference of two oclets because of the change, she may decide which of the oclets has to be adapted, for instance, by changing one oclet's history to prevent the interference. The prospective advantage of this approach is that the modeling artifacts under consideration (i.e., the oclets) remain structurally simple. So they are easier to understand and to manipulate by the system designer than the entire model. More involved relations between oclets can be computed automatically; the next chapter presents an automated technique.

The oclets themselves may be shaped arbitrarily and may even *overlap*. This allows to follow good modeling practices such as having oclets of reasonable size (e.g., fitting on one screen) so that each oclet can be understood as a self-contained

story about the modeled system. We presented an example of such a specification in Section 4.6. The operational semantics of oclets (as implemented in Greta's execution engine) automatically integrates the behavior of all oclets.

6.8. Discussion

This and the preceding chapter extended the model of oclets by composition and decomposition operators and a relation to compare oclets. These notions allow to reason systematically about the behavior of an oclet specification. We applied oclet composition, decomposition, and comparison to establish several important results of oclets wrt. the synthesis problem. Section 6.8.1 reviews theses results and discusses their relation to other approaches. In Section 6.8.2, we discuss how oclet semantics of Chapter 4, oclet composition of Chapter 5, and oclet play-out of this chapter relate to each other.

6.8.1. Operations and relations on scenarios

Several scenario-based techniques construct larger scenarios from smaller scenarios by composition (usually requiring several composition operators), e.g., High-Level MSCs [67] and labeled partial orders [13]. Atir et al. [9] flatten a hierarchy of LSCs by LSC composition. We are not aware of any composition operator on history-based scenarios that resembles oclet composition. Further, scenario decomposition and comparison as well as their application to proofs is novel to the best of our knowledge. The following results have been established by the help of oclet composition, decomposition, and comparison.

Characterization of synthesis algorithms. We applied oclet composition and decomposition to characterize what a synthesis algorithm has to achieve in terms of the behavior of the implementation.

Oclet composition constructs from a given oclet specification O the unique minimal set $R(O)$ of runs that satisfies O. An implementation of O has to exhibit at least the behavior $R(O)$. There are oclet specifications O of which the least behavior $R(O)$ cannot be exactly implemented by a *distributed system*. In that case, an implementation of O inevitably exhibits more behavior. Oclet decomposition and composition *together* construct from O the *unique minimal* set $\hat{R}(O)$ of runs which is exhibited by a *minimal implementation* of O. In other words, we precisely characterized what a synthesis algorithm has to achieve: construct a system model Σ that exhibits the behavior $R(\Sigma) = \hat{R}(O)$. Furthermore, oclet decomposition and composition amounts to *operational semantics* of oclets which we extend in the next chapters to a synthesis algorithm for oclets.

Implied scenarios and distributed systems. Runs in $\hat{R}(O) \setminus R(O)$ are also known as *implied* scenarios in literature. Alur et al. [5] specifically point out that implied scenarios may contain undesired behavior and even lead to deadlocks; in that work a polynomial-time algorithm for detecting implied scenarios from finite MSCs

is defined. Uchitel et al. [111] detect implied scenarios of HMSC specifications by constructing and model-checking a labeled transition system; the result is extended in [112]. Other works follow a similar approach of first constructing the complete behavior and then checking for implied scenarios afterwards [108, 68]. The strong expressive power of LSCs requires a different notion of implied scenario detection: here, one scenario may locally allow to begin a run in a way that later on inevitably violates another scenario. The smart play-out technique in [51] applies model-checking to detect such kind of implied scenarios.

Implied scenarios wrt. minimal behavior and the difference between specification and implementation has not been considered yet systematically for scenarios with history to the best of our knowledge. We could show that, the distinction between specified behavior and implemented behavior follows directly from the decomposition into basic oclets. Thus, implied scenarios can be directly attributed to "syntactical reasons" of the specification.

Modeling systems with scenarios. The decomposition of oclet specifications into basic oclets automatically yields operational semantics for oclets. These semantics amounts to the notion of an *oclet system* which allows to directly execute the scenarios without synthesizing an implementation. This way, a system designer can *model* a distributed system with scenarios.

The general approach of directly executing scenarios has been first proposed for LSCs and is implemented in the play-engine [55]. Conceptually, LSC play-out is based on an interpretation algorithm of LSCs that instantiates LSCs and executes each LSC by letting its enabled events occur; an occurrence of an event in one LSC may trigger an instantiation of another LSC. Formally, this algorithm defines a transition system over sequential runs. LSC play-out has been improved in several works, specifically in terms of complexity [51, 52, 57, 50].

Other approaches obtain an operational semantics for scenarios by translating a specification into another formal model (like a Petri net [62, 33], a process algebraic expression [87], or a Statechart [46]); the operational semantics of the target model then defines the operational semantics of the scenarios (see the discussion in Sect. 4.7).

The specific contribution of oclet systems is that their operational semantics canonically follow from composition of basic oclets. As a result, the formal definitions for the operational semantics are conceptually lightweight and apply to every oclet specification. Secondly, the semantics is a *true concurrency* semantics that represents behavior by distributed runs instead of sequential runs. Section 6.7.3 presented an execution engine for oclets.

Expressive power of scenario-based specifications. We have proved that oclets embed Petri nets: for every Petri net exists an equivalent oclet specification that describes all behavior of the Petri net. This property is remarkable because we developed oclets as a kernel of scenario-based specifications using a minimal set of notions in Chapter 3. The kernel is expressive enough to specify any distributed system (that can also be modeled as a Petri net).

This result contributes to a long-lasting debate of how expressive a scenario-based technique has to be. While it is agreed that simple MSCs are too weak to specify meaningful system behavior [11, 25], its extensions HMSCs [65] and particularly LSCs [25] significantly extend expressive power. In fact, these extensions typically make many problems *undecidable*, e.g., the synthesis problem becomes undecidable [26, 18]. Following the extensive discussion in Chapter 3, we developed oclets as a rather weak model using a minimal set of notions. Yet, this result proves that oclets are already expressive enough to include Petri nets. We prove in the Chapter 8 that oclets are strictly more expressive than Petri nets.

Relation to formal languages. The specific structure of an oclet o consisting of a history and a contribution suggests to study expressive power of oclets also from the angle of formal languages.

The words of *context-sensitive* language over an alphabet Σ can be generated using a context-sensitive grammar, that has rules of the from $\alpha Z\gamma \to \alpha\beta\gamma$ where $Z \in \Gamma$ is a non-terminal symbol and $\alpha, \beta, \gamma \in (\Sigma \cup \Gamma)^*$ are finite words of terminals from Σ and non-terminals from Γ. The structure of an oclet resembles a *left context-sensitive rule*, which has the form $\alpha Z \to \alpha\beta$. Supposedly, the history $hist_o$ of an oclet o corresponds to the left context α, the maximal conditions $\max hist_o$ correspond to the non-terminal Z, and o's contribution corresponds to β. Every context sensitive language can be generated by a left-context sensitive grammar [20].

However, oclets are strictly more expressive than context-sensitive languages. In Appendix A.9, we show that every instance P of Post's Correspondence Problem (PCP) [100] can be translated to an oclet specification O s.t. P has a solution iff the oclets O can be composed to reach a specific state. Furthermore, every Turing machine T can be encoded as a PCP instance P. Thus, T can be encoded as an oclet specification O. Because every recursively enumerable language L can be accepted by some Turing machine T_L, there also exists an oclet specification O_L that accepts L by oclet composition. Because recursively enumerable languages strictly contain context-sensitive languages, we conclude that oclets are strictly more expressive than context-sensitive grammars.

The decisive difference between oclets and grammars is that a grammar rule $\alpha Z \to \alpha\beta$ only expresses a sequential context $\alpha \in (\Sigma \cup \Gamma)^*$ whereas an oclet $o = (\pi_o, hist_o)$ defines a partially ordered context $hist_o$ in which the presence or absence of causal dependencies matters.

Histories. Most results of this and the preceding chapter follow from composition and decomposition of oclets that respects the notion of a *history* of an event.

The notion of history-dependent transitions has been studied in Petri nets in several models. Hee et al. [120] define *history-dependent Petri nets* in which each token records its history. A transition t can have a history-dependent guard, expressed by an LTL formula, which restricts occurrences of t to those cases where the tokens in t's pre-places satisfy the guard. A basic oclet corresponds to a transition with a history-dependent guard. The history of a basic oclet is less expressive than LTL which allowed us to establish the results of this chapter. Yet,

it is worth researching how to increase the expressive power of oclets wrt. history while preserving the locality of events. In [122] it is shown how business process models benefit from history-dependent transitions.

The notion of a history of a state or of an event underlies many approaches that construct a system's *execution tree*, or a related structure like an event structure [127, 94, 10] or a branching process (implicit in [35] and explicit in [36]). *History-dependent automata* naturally express the semantics of process algebras like the π-calculus [92].

We continue using the notion of a history systematically for synthesizing an implementation from a specification in the next two chapters.

6.8.2. Semantics, composition, and play-out

With the oclet semantics of Chapter 4, oclet composition of Chapter 5, and oclet play-out of this chapter, we know three techniques to describe a system's behavior with scenarios. In the following, we briefly discuss which technique is appropriate for which purpose.

Describe all system behavior. The main purpose of oclets is to let a system designer describe the behavior of a distributed system S. For this purpose, oclet composition and play-out complement each other in system design. A system designer first describes each standard scenario of S as an oclet. Using oclet composition, the system designer derives the standard behavior of S by simply "plugging" these oclets together. So the standard behavior of S can be anticipated visually from the described scenarios in a "pen and paper" setting.

At a later stage, the designer describes exceptions and variants of the standard behavior of S. An exception usually occurs "in the middle" of a standard scenario, which cannot be expressed by composition of entire oclets. Moreover, oclet composition does not reflect that a distributed system S exhibits behavior action by action. Oclet play-out accurately describes the interplay of standard behavior and exceptions, and how S would implement the specified oclets. Altogether, the precise technique to describe all behavior of S is oclet play-out.

Describe partial system behavior. Oclet composition and oclet play-out yield reasonable results only if the specified oclets describe all behavior of S. If a system designer wants to describe only some aspects of the behavior, for instance, the behavior of a component of S or how S reacts in specific situations, then the declarative semantics of oclets are the appropriate technique. These are based on an open world assumption and allow S to exhibit more behavior than specified.

Develop scenario-based techniques. Oclet semantics also play an important role in the development of scenario-based techniques. We developed oclets with the aim of balancing the trade-off between expressive power and synthesis capabilities. Currently, oclets cannot describe global properties such as "action A will eventually occur" or "action B occurs at most three times", which can be specified in other techniques [55].

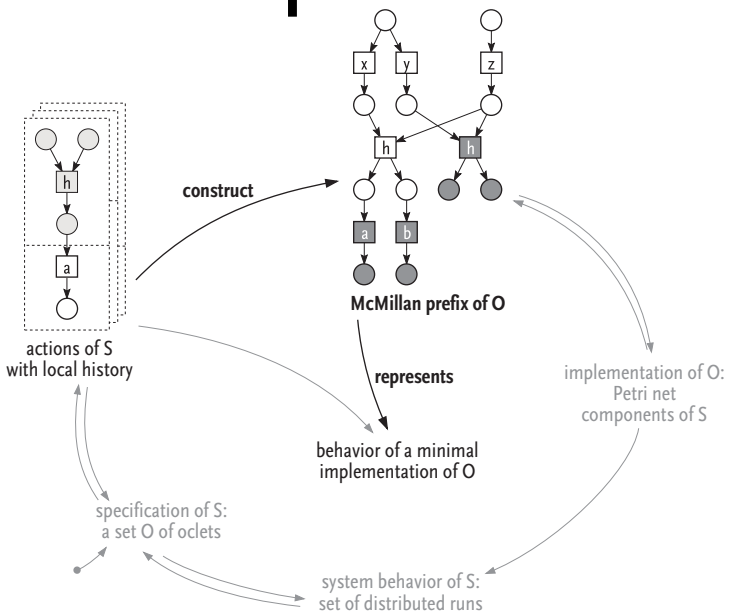
A systematic approach to extend oclets by new means of expression is the following. For example, we could introduce the notion of an *anti-oclet* which describes that a specific scenario must not occur. First, the semantics of oclets is extended wrt. the new notion: a set R of distributed runs *satisfies* an anti-oclet o iff there exists no run $\rho \in R$ s.t. o occurs in ρ. This definition directly characterizes the properties that R must have to satisfy o. Constructing a set R of runs that satisfies an oclet specification with an anti-oclet is more involved. We propose an operational definition of anti-oclets in [39].

Conclusion. Oclet play-out allows to describe all behavior of a distributed system S with oclets, which cannot be done with oclet composition. Moreover, oclet play-out constructs the behavior exhibited by all minimal implementations of an oclet specification, which cannot be done with oclet semantics. For this reason, we propose oclet play-out as the "decisive semantics" of oclets. In the next two chapters, we use oclet play-out to analyze an oclet specification O, and to synthesize a minimal implementation of O.

Part IV.

Analyzing and Synthesizing Distributed Systems

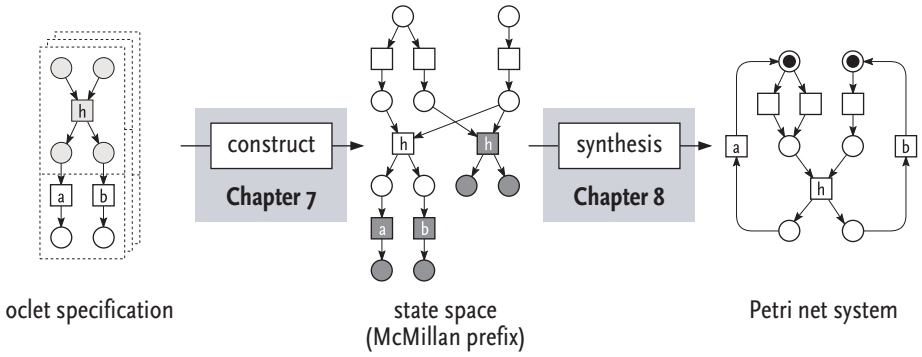
7. Analyzing Scenario-Based Specifications



The preceding chapters introduced oclets as a technique to describe the behavior of a distributed systems with scenarios. In the following, we want to analyze behavioral properties of an oclet specification O. A naïve analysis by *state space exploration* is infeasible because the state space of a distributed system grows exponentially with the size of the system. An approach by McMillan mitigates the problem for *Petri nets* by constructing a symbolic representation of the state space that is called *finite complete prefix*. In this chapter, we adapt McMillan's technique from Petri net theory to scenario-based specifications. We define an algorithm to construct a McMillan prefix of an oclet specification O, and show how to analyze behavioral properties of O on the prefix. The next chapter defines a synthesis algorithm that "folds" the McMillan prefix to a minimal implementation of O.

7.1. A Two-Step Approach

This and the next chapter address the main problem of this thesis: to synthesize a minimal implementation from a given scenario-based specification.



What is given?

The preceding chapters of this thesis paved the way for solving the synthesis problem. Chapter 4 introduced oclets as a formal model for scenarios. An oclet specification O is a set of oclets. We have shown in Chapter 5 that this model is expressive enough to specify every Petri net system. Chapter 6 characterized for each oclet specification O the unique minimal set $\hat{R}(O)$ of runs that is exhibited by a *minimal implementation* of O.

The formal vehicle to characterize $\hat{R}(O)$ were the notions of a *basic oclet* and an *oclet system*. Each oclet $o \in O$ decomposes into a set of basic oclets; each basic oclet represents one event of o with its *local history*. Each oclet specification O decomposes into an equivalent oclet system Ω_O consisting of all basic oclets of O. Ω_O has operational semantics which construct $\hat{R}(O)$ as a set of *distributed runs* by composing the basic oclets of O.

The next steps

This and the next chapter extend the operational semantics of an oclet system to an algorithm that synthesizes a Petri net from a given specification O. To this end, we adapt a technique from Petri net theory developed by McMillan. Figure 7.1 sketches our approach.

Section 7.2 of this chapter introduces the notion of a *branching process*. A branching process is basically the union of all distributed runs of $\hat{R}(O)$ to a canonical "execution tree" of distributed runs. A branching process can be infinite. Section 7.3 recalls the notion of a *complete prefix* of a branching process developed by McMillan [88]. A complete prefix is a finite representation of a branching process. Section 7.4 adapts McMillan's notion from Petri nets to oclets, and Section 7.5 presents an algorithm for computing a complete prefix of an oclet specification O. This prefix represents the minimal behavior $\hat{R}(O)$ of an implementation of O in a *finite* structure. Section 7.6 demonstrates how to analyze behavioral properties of

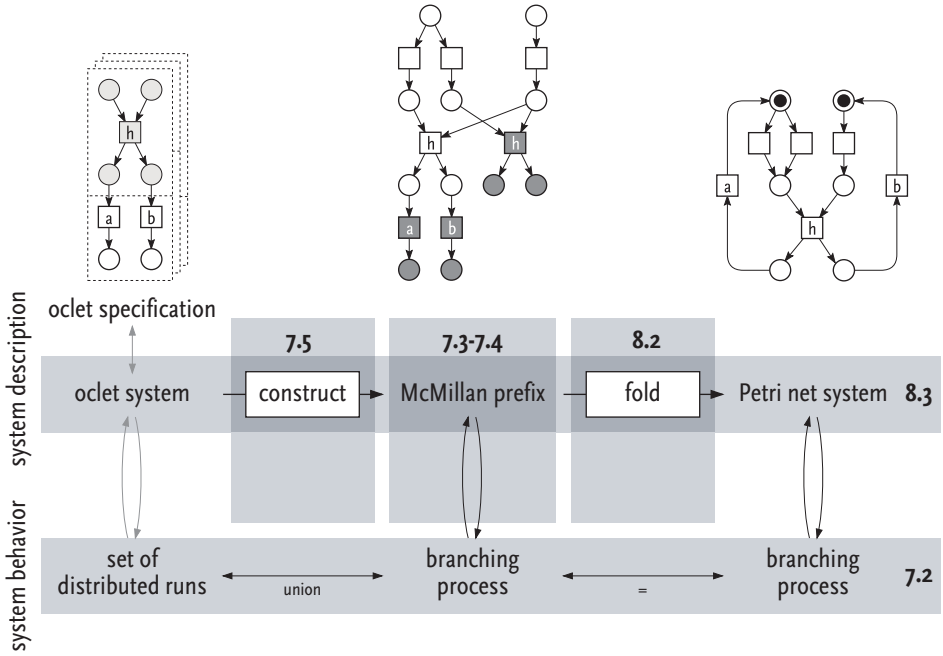


Figure 7.1. Chapter overview: synthesizing an implementation from a specification.

an oclet specification using McMillan's technique. In Section 7.7, we report on some experimental results showing the feasibility of this approach for analyzing the behavior of an oclet specification O.

Chapter 8 continues this approach and presents a synthesis algorithm. By folding the complete prefix of O (using a simple equivalence), we obtain a Petri net system Σ that exhibits the behavior $\hat{R}(O)$, i.e., Σ implements O. The folding operation is defined in Section 8.2. Section 8.3 combines all results of this chapter and defines the algorithm that solves the synthesis problem.

7.2. Branching Processes

Intuitively, a *branching process* represents a set of distributed runs to a tree-like structure. This notion is similar to representing a set $\{s_1, s_2, s_3, \ldots\}$ of *sequential runs* by their canonical *execution tree* T: any two sequential runs s_i and s_j are *merged* in T up to their maximal joint prefixes, then s_i and s_j diverge in T and never meet again. Correspondingly, a branching process β merges any two distributed runs ρ_1 and ρ_2 of a set R along their maximal joint prefixes. When ρ_1 and ρ_2 differ they diverge in β and never meet again. Figure 7.2 depicts a branching process β that merges the two distributed runs ρ_1 and ρ_2 in this respect.

From the technical perspective a branching process β is similar to a distributed run. The only difference is that a condition of β may have *two* post-events. This

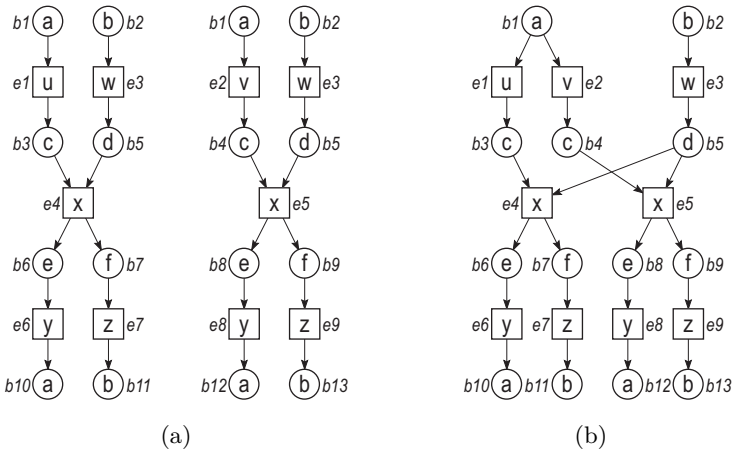


Figure 7.2. The distributed runs in (a) are merged to the branching process in (b).

happens whenever two merged runs diverge by different events. In the example of Figure 7.2, condition b_5 has two post-events e_4 and e_5. Section 7.2.1 formalizes the notion of a branching process, and Section 7.2.2 provides some canonical notions on branching processes. Section 7.2.3 recalls how the behavior of a Petri net system is represented in a branching process. We then generalize this result from Petri nets and define the branching processes of an oclet system in Section 7.2.4.

7.2.1. Basic definitions

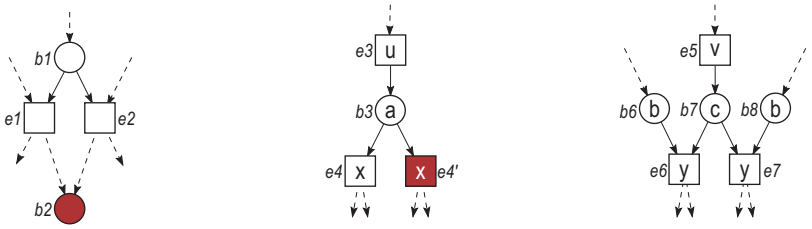
The notion a branching process is based on the *causal*, *conflict*, and *concurrency* relations between nodes of a Petri net. Causal dependency and concurrency have already been used in the preceding chapters; conflict is new.

Definition 7.1 (Causal relations). Let N be a Petri net.

1. Two nodes x_1 and x_2 of N are in *causal relation*, denoted $x_1 \leq x_2$, iff there exists a path from x_1 to x_2 along the arcs of N, i.e., $(x_1, x_2) \in F_N^*$; we write $x_1 < x_2$ if additionally $x_1 \neq x_2$. We also say x_2 *depends on* x_1.
2. x_1 and x_2 are *in conflict*, denoted $x_1 \# x_2$, iff N contains a place p with two post-transitions $t_1 \neq t_2$ s.t. $t_1 \leq x_1$ and $t_2 \leq x_2$.
3. x_1 and x_2 are *concurrent*, denoted $x_1 \,||\, x_2$ iff neither $x_1 \leq x_2$ nor $x_2 \leq x_1$ nor $x_1 \# x_2$. ⌟

We write $\leq_N$ etc. when we refer to a relation in a particular Petri net N. In Figure 7.2(b), transition e_6 depends on transition e_1, e_6 is in conflict with e_8, and e_6 is concurrent to e_7.

In a causal net (which underlies our notion of a distributed run), any two nodes are either in causal relation or concurrent. The new notion of conflict denotes that two events (conditions) cannot occur in the same run. The class of *occurrence*



(a) β_1, not an occurrence net

(b) β_2, not a branching process

(c) β_3, a branching process

Figure 7.3. Properties of an occurrence net (a) and a branching process (b) and (c).

nets generalizes the class of causal nets by allowing that two nodes may be in conflict — two conflicting nodes occur in alternative (or conflicting) runs.

Definition 7.2 (Occurrence net). A Petri net $\beta = (B, E, F)$ is an *occurrence net* iff

1. the causal relation $\leq = F^*$ is a partial order,
2. $\leq$ is finitely preceded, i.e., each $x \in X$ has only finitely many predecessors $y \leq x$ (see Def. 2.5),
3. $|{}^\bullet b| \leq 1$, for all $b \in B$, and
4. no node of β is in conflict with itself, i.e., $\forall x \in X : \neg\, x \# x$. ⌟

Like for causal nets, the elements of B are called *conditions* and the elements of E are called *events*. Figure 7.3(a) abstractly illustrates the structural properties of an occurrence net. The depicted net β_1 is not an occurrence net: condition b_2 is in self-conflict because the conflicting events e_1 and e_2 are both predecessors of b_2. All Petri nets in Figure 7.2 are occurrence nets (if we ignore the inscribed labels for a moment).

In Chapter 2, we labeled a causal net with names $\mathcal{L}_B$ of local states of a system and names $\mathcal{L}_E$ of actions of a system to describe a distributed run. Intuitively, an occurrence net represents a set of causal nets. To represent a *set of distribute runs*, we attach labels to the nodes of an occurrence net. The resulting structure is a *branching process*.

Definition 7.3 (Branching process). Let $\mathcal{L} = \mathcal{L}_B \cup \mathcal{L}_E$ be a set of names of local states $\mathcal{L}_B$ and actions $\mathcal{L}_E$, $\mathcal{L}_B \cap \mathcal{L}_E = \emptyset$. A *branching process* over $\mathcal{L}$ is a labeled occurrence net $\beta = (B, E, F, \ell)$ with labeling $\ell : B \cup E \to \mathcal{L}$ s.t.

1. $\ell(b) \in \mathcal{L}_B$, for all $b \in B$, and $\ell(e) \in \mathcal{L}_E$, for all $e \in E$, and
2. for any events e_1, e_2 of β, if ${}^\bullet e_1 = {}^\bullet e_2$ and $\ell(e_1) = \ell(e_2)$, then $e_1 = e_2$ (i.e., β contains no duplicate events). ⌟

Figures 7.3(b) and (c) abstractly illustrate the properties of the labeling of a branching process. The net β_2 is not a branching process because we find two events e_4 and e_4' with the same label x and the same pre-set ${}^\bullet e_4 = \{b_3\} = {}^\bullet e_4'$. In contrast, β_3 is a branching process: the pre-sets of the event e_6 and e_7 overlap on condition b_7 but they are not identical. All Petri nets in Figure 7.2 are branching

processes. It follows straight from the definition that every branching process without conflicting nodes is a distributed run. As for distributed runs, we do not re-label nodes of a branching process. Thus, we assume a fixed universal labeling ℓ that assigns each event or condition x of a distributed run "its" label $\ell(x) \in \mathcal{L}$. Accordingly, we simplify notation and write $\beta = (B, E, F)$ when denoting a branching process over $\mathcal{L}$.

7.2.2. Some canonical notions on branching processes

The possibility to have conflicts triggers the need to generalize some notions from distributed runs to branching processes. These generalizations are straight-forward; yet we give them here for the sake of completeness.

Definition 7.4 (Basic notions on branching processes). Let $\beta = (B, E, F)$ be a branching process.

1. A branching process $\beta' = (B', E', F')$ is a *prefix* of β, denoted $\beta' \sqsubseteq \beta$, iff
   a) $\beta' \subseteq \beta$ (β' is a sub-net of β) s.t.
   b) $\min \beta \subseteq X' = (B' \cup E')$ (the minimal nodes of β all occur in β'),
   c) $(y \in X' \wedge (x, y) \in F) \Rightarrow x \in X'$, for all $x, y \in X$ (β' is *causally closed* in β), and
   d) $\forall e \in E' : post_{\beta}(e) = post_{\beta'}(e)$ (β' is complete wrt. post-conditions of its events).

2. A *configuration* of β is a set C of events of β s.t.
   a) $\forall e \in C \forall e' \in E : e' \leq e \Rightarrow e' \in C$ (C is causally closed), and
   b) $\forall e, e' \in C : \neg\, e \# e'$ (C is *conflict-free*).

3. A set $B' \subseteq B$ of conditions is a *co-set* of β iff all conditions in B' are pair-wise concurrent. A maximal co-set of β is a *cut*.

4. Let C be a configuration of β. The *cut reached by* C is $Cut(C) := (\min \beta \cup C^{\bullet}) \setminus {}^{\bullet}C$. Conversely, let B be a cut of β. The *configuration that reaches* B is $C_B := \{e \in E_\beta \mid \exists b \in B : e < b\}$. ⌟

It is easy to see from the definitions that these notions on branching processes generalize the corresponding notions on distributed runs.

In Figure 7.2(b), $\{e_1, e_3, e_4, e_6, e_7\}$ is a configuration. The cut reached by this configuration consists of the conditions $\{b_{10}, b_{11}\}$. The branching process of Figure 7.2(b) is a prefix of the branching process of Figure 7.4(b).

A configuration is a set of non-conflicting events. Consequently, each configuration C of a branching process β induces the distributed run $\beta[C]$.

Definition 7.5 (Induced run). Let β be a branching process.

1. Let C be a configuration of β; the distributed run $\beta[C]$ *induced* by C consists of the minimal nodes of β followed by all events in C with their respective pre- and post-conditions, i.e., $\beta[C] := (B_\beta \cap Y, E_\beta \cap Y, F_\beta|_{Y \times Y})$ with $Y = \min \beta \cup {}^{\bullet}C \cup C \cup C^{\bullet}$.

2. Let B be a cut of β; the run *induced* by B is the run $\beta[B] := \beta[C_B]$ where C_B is the configuration that precedes the cut B.
3. The set of all runs of β is $R(\beta) := \{\beta[C] \mid C \text{ a configuration of } \beta\}$. ⌟

It is easy to see from the definitions that, for a given configuration C of β, $\beta[C]$ is a branching process and a prefix of β. Moreover, the events of $\beta[C]$ are exactly C. Because C is conflict-free, each condition of $\beta[C]$ has at most one post-event. Altogether $\beta[C]$ is a distributed run. The same reasoning applies to the distributed run $\beta[B]$ induced by a cut B of β. Moreover, all nodes of $\beta[B]$ precede B; thus, $\max \beta[B] = B$. These observations directly lead to the following corollary.

Corollary 7.6: *Let β be a branching process.*

1. *If C and C' are configurations of β with $C' \subset C$, then $\beta[C']$ is a prefix of $\beta[C]$.*
2. *Let $\beta' \neq \beta$ be a branching process. If β' is a prefix of β, then each configuration of β' is a configuration of β and $R(\beta') \subset R(\beta)$.* ⋆

In Figure 7.2(b), the configurations $\{e_1, e_3, e_4, e_6, e_7\}$ and $\{e_2, e_3, e_5, e_8, e_9\}$ induce the distributed runs of Figure 7.2(a), respectively.

7.2.3. Branching processes of a Petri net system

This section recalls the relation between a Petri net system and its branching processes. The distributed runs $R(\Sigma)$ of a Petri net system Σ are axiomatically characterized by a structural correspondence between each run $\rho \in R(\Sigma)$ and Σ; see Def. 2.16. The same structural correspondence allows to characterize the *branching processes of* Σ. A branching process β of a Petri net system Σ is labeled with the places and transitions of Σ. The minimal conditions of β represent the initial marking of Σ; each event e of β represents an occurrence of a transition t that consumes the tokens ${}^\bullet e$ from ${}^\bullet t$ and produces the tokens $e^\bullet$ on $t^\bullet$.

Definition 7.7 (Branching processes of a Petri net system). Let $\Sigma = (N, m_0)$ be a Petri net system. Let β be a branching process over the transitions and places of N. For a given cut B of β, denote by $\mathit{Mark}(B)$ the marking m that is represented by B, i.e., $m(p) = |\{b \in B \mid \ell(b) = p\}|$, for all $p \in P_N$. The branching process β is a branching process of Σ iff

1. $\mathit{Mark}(\min \beta) = m_0$,
2. $\ell(b) \in P_N$, for all $b \in B_\beta$, and $\ell(e) \in T_N$, for all $e \in E_\beta$, and
3. for each event $e \in E_\beta$ with $\ell(e) = t$, the labeling ℓ bijectively maps ${}^\bullet e$ to ${}^\bullet t$ and $e^\bullet$ to $t^\bullet$. ⌟

Figure 7.4 depicts an example. Also the branching process in Figure 7.2.(b) is a branching process of the Petri net system in Figure 7.4.

The definition of a branching process of Σ generalizes the characterization of a distributed run of Σ; see Definition 2.16. The actual difference between the branching processes of Σ and the distributed runs of Σ follows from the fact that an occurrence net allows conflicts.

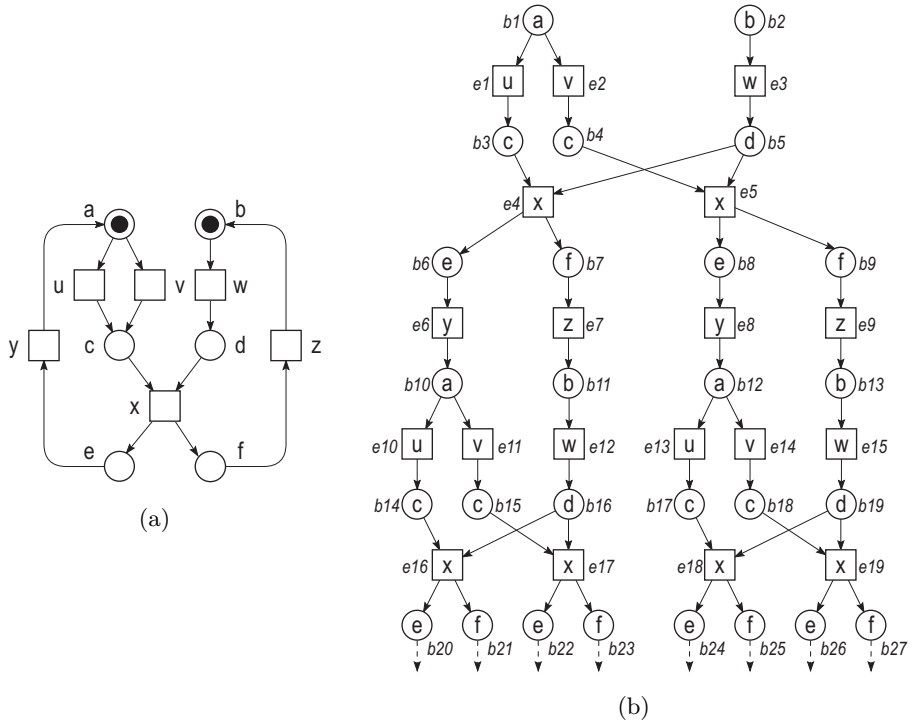


Figure 7.4. A Petri net system (a) and one of its branching processes (b).

Engelfriet has proven in [35] that the set of all branching processes of a Petri net system Σ forms a complete lattice wrt. the prefix-relation on branching processes. The largest branching process of Σ that contains all other branching processes of Σ is also called the *unfolding* of Σ; we denote it by $Unf(\Sigma)$. The unfolding of the Petri net system in Figure 7.4 is infinite.

The unfolding of a Petri net system Σ represents all behavior of Σ. Specifically, the configurations of $Unf(\Sigma)$ induce the set of distributed runs of Σ: $R(Unf(\Sigma)) = R(\Sigma)$.

7.2.4. Branching processes of an oclet system

This section generalizes the preceding definitions of a branching process from Petri nets to oclet systems. The aim is to define an unfolding $Unf(\Omega)$ of an oclet system $\Omega = (O, \pi_0)$. The unfolding $Unf(\Omega)$ should represent all behavior of Ω; specifically the configurations of $Unf(\Omega)$ shall induce exactly the distributed runs $R(\Omega)$ of Ω.

The idea

A branching process β of an oclet system $\Omega = (O, \pi_0)$ corresponds to the initial run π_0 and the basic oclets O as follows.

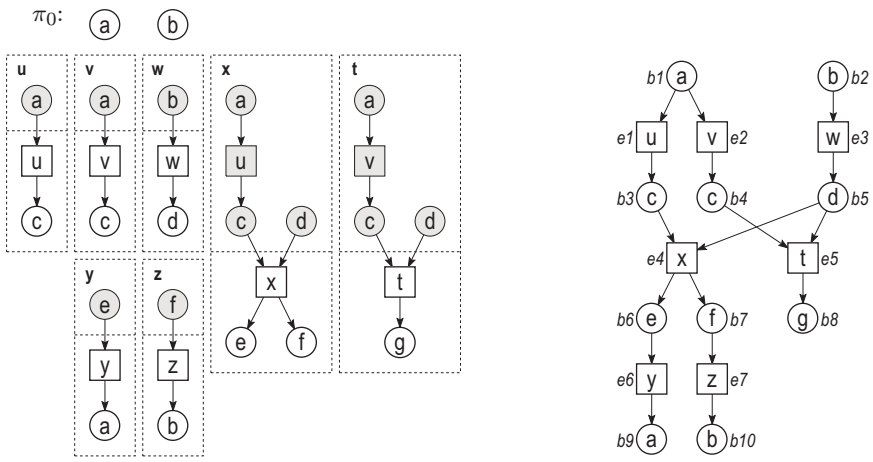


Figure 7.5. An oclet system in (a) and one of its branching processes in (b). Event e_4 of β is contributed by the basic oclet x.

- The beginning of β corresponds to the beginning of Ω, i.e., π_0 is a prefix of β; and
- each event e of β is contributed by an oclet o s.t. the local history of e ends with $hist_o$ and $e \cup e^\bullet$ represent the contribution of o.

Figure 7.5 depicts an example.

The largest branching process of an oclet system Ω is its unfolding $Unf(\Omega)$. The unfolding is intuitively the union of all distributed runs $R(\Omega)$ of Ω. Correspondingly, each configuration of $Unf(\Omega)$ induces a distributed run of Ω, i.e., $R(\Omega) = R(Unf(\Omega))$.

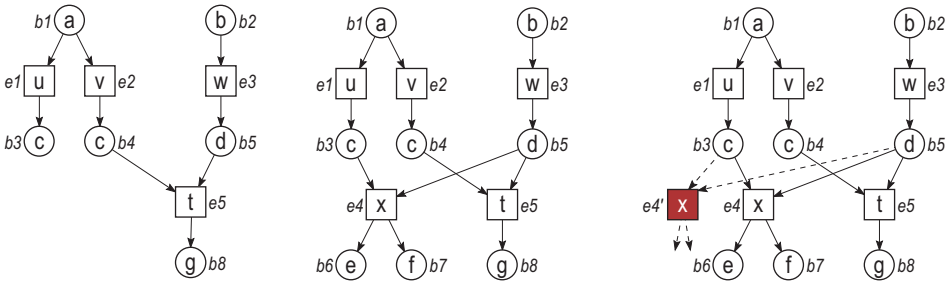
For instance, the configuration $\{e_1, e_3, e_4, e_6, e_7\}$ of Figure 7.5.(b) corresponds to the distributed run $\pi_0 \rhd \mathsf{u} \rhd \mathsf{w} \rhd \mathsf{x} \rhd \mathsf{y} \rhd \mathsf{z}$ of the oclet system in the same figure.

For the results of the next sections, specifically for the algorithmic synthesis of an implementation from an oclet specification, we need an inductive definition of $Unf(\Omega)$. The remainder of this section defines $Unf(\Omega)$ inductively and proves that $Unf(\Omega)$ is unique and contains any other branching process of Ω as a prefix.

The formal definitions

To define $Unf(\Omega)$, we adapt how the oclets of Ω are composed to a distributed run. As exercised in Chapters 4-6, we first define how a concrete oclet o of Ω relates to a branching process. Then we lift this notion to the oclet class $[o]$ of o and consider all isomorphic oclets $o^\alpha \in [o]$. On this level, we can relate the oclet system Ω which consists of classes of basic oclets to its unfolding $Unf(\Omega)$.

As shown in Chapter 6, the construction of a *distributed run of* Ω begins in the initial run π_0 of Ω and continues by repeatedly composing a run π with an oclet o



(a) β_1 (b) β_2 (c) β_3

Figure 7.6. Oclet x of Fig. 7.5 is a possible extension of β_1. Extending β_1 by x yields β_2. Oclet x is not a possible extension of β_2 (at conditions b_3, b_5) because of event e_4. If β_2 was extended by x, the resulting net β_3 would not be a branching process because of the duplicate events e_4 and e_4'.

of Ω to the run $\pi \rhd o$ if o is enabled "at the end" of π; see Definition 6.16. The composition $\pi \rhd o$ appends o's contribution to the *maximal* nodes of π.

The construction of a *branching process* β *of* Ω should also follow from repeatedly composing β with an oclet o of Ω to the branching process $\beta \rhd o$ if o is enabled. Though, we have refine the notion of when o is enabled. In contrast to a distributed run, β allows to express conflicts, i.e., a condition can have two post-events. So o may also be enabled "in the middle" of β — at any cut B of β which ends with o's history even if the conditions in B already have a post-event f. The composition $\beta \rhd o$ appends o's contribution to this cut B. To ensure that the result $\beta \rhd o$ is a branching process again, two conditions have to hold:

1. As in the case of distributed runs, o appends new nodes to β: $con_o \cap \beta = \emptyset$.
2. When composing $\beta \rhd o$, the oclet o must not introduce a duplicate event. In other words, the conditions B of β to which con_o is appended must not already have a post-event f, ${}^\bullet f = B$ with the same label as the contributed event of o.

If o satisfies these properties, then o is a *possible extension* of β. A possible extensions corresponds to the notion of an enabled oclet used in Chapter 5. The first condition ensures that $\beta \rhd o$ is an occurrence net (by not introducing cycles or self-conflicts). The second condition ensures that $\beta \rhd o$ is labeled correctly. Figure 7.6 illustrates the concept using an example.

Definition 7.8 (Possible extension). Let β be a branching process and let o be a basic oclet. Let $event(o) := e$ denote the unique event e of o's contribution, i.e., $E_{con_o} = \{e\}$.

Let B be a cut of β. Oclet $o = (\pi_o, hist_o)$ is *a possible extension* of β at B iff

1. the distributed run $\beta[B]$ ends with the history of o, i.e., $hist_o \subseteq \beta[B]$ s.t. $\max hist_o \subseteq B$,
2. $con_o \cap \beta = \emptyset$, and

3. there exists no event $f \in E_\beta$ with ${}^\bullet f = \max hist_o \subseteq B$ and $\ell(f) = \ell(e)$.

If o is a possible extension of β at B, then β can be *extended* with o by $\beta \rhd o := \beta \cup \pi_o$. ⌟

The composition operation $\rhd$ on branching processes is sound.

Lemma 7.9: *Let β be a branching process with a cut B and let o be an oclet that is a possible extension of β at B. Then $\beta \rhd o$ is a branching process and β is a prefix of $\beta \rhd o$.* ⋆

This lemma can be proven directly from the definitions. The main argument of the proof is that $\beta \rhd o$ appends o's contribution to β s.t. all new arcs in $\beta \rhd o$ only leave β but do not enter β. Thus, $\beta \rhd o$ is an occurrence net with the prefix β. That o is a possible extension of β (Def. 7.8) ensures that $\beta \rhd o$ contains no cut B having two post-events e and f with same labels and same pre-sets. Thus, $\beta \rhd o$ is a branching process.

A branching processes usually consists of several oclets of the same oclet class $[o]$. The notion of a possible extension o of β canonically lifts to all oclets o^α in the oclet class $[o]$. We say that $[o]$ *provides the possible extension* $o^\alpha \in [o]$ of β at cut B iff the oclet $o^\alpha \in [o]$ is a possible extension of β at B. There may be several possible extensions of β provided by $[o]$ at different cuts. Like for the composition of a distributed run with oclets in Section 5.2, we extend β by choosing a concrete possible extension $o^\alpha \in [o]$ of β and composing $\beta \rhd o^\alpha$.

A branching process of an oclet system Ω is constructed inductively by extending the system's initial run with the possible extensions provided by Ω's oclet classes.

Definition 7.10 (Branching processes of an oclet system). Let $\Omega = (O, \pi_0)$ be an oclet system. The branching processes of Ω are inductively defined.

1. The initial run π_0 is a branching process of Ω.
2. Let β be a branching process of Ω and let B be a cut of β s.t. $\pi_0 \sqsubseteq \beta[B]$. If an oclet class $[o] \in O$ provides a possible extension $o^\alpha \in [o]$ of β at B, then $(\beta \rhd o^\alpha)$ is a branching process of Ω. ⌟

If the oclet system Ω has infinitely many runs, then the construction of the branching processes of Ω does not terminate in finite time. But, like a Petri net system, an oclet system has a unique maximal branching process $Unf(\Omega)$ that contains all other branching processes of Ω. Though, the uniqueness of $Unf(\Omega)$ only holds if the oclets in Ω are labeled consistently; intuitively, two events with the same label should describe the same action that causes the same effects.

Figure 7.7 depicts an example where oclets are labeled *inconsistently*. The oclet system Ω has no unique unfolding.

Definition 7.11 (Label consistent). Let $N = (P, T, F, \ell)$ be a labeled Petri net. Two transitions t_1, t_2 of N are *label consistent* iff $\ell(t_1) = \ell(t_2)$ implies that the pre-sets (post-sets) of t_1 and t_2 are isomorphic wrt. the labeling ℓ, respectively. The entire Petri net N is *label consistent* iff all transitions of N are pair-wise label consistent.

Let O be a set of oclets; O is *label consistent* iff for any two oclets $o_1, o_2 \in O$ any two events e_1 of o_1's contribution and e_2 of o_2's contribution are label consistent. ⌟

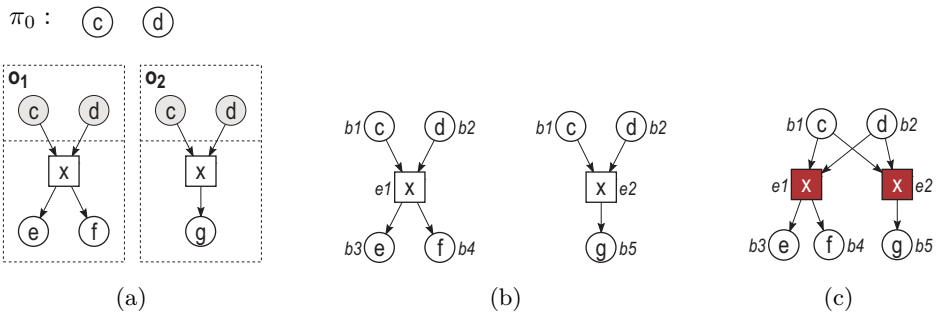


Figure 7.7. An oclet system $\Omega = (\{\mathsf{o}_1, \mathsf{o}_2\}, \pi_0)$ in (a) with an inconsistent labeling and its distributed runs (b). The union of the runs of Ω is not a branching process (c).

The oclets o_1 and o_2 in Figure 7.7 are *not* label consistent. A label consistent oclet system Ω has a unique unfolding $\beta(\Omega)$.

Theorem 7.12 (Unfolding of an oclet system). *Let $\Omega = (O, \pi_0)$ be an oclet system. If O is label consistent, then Ω has a unique largest branching process $Unf(\Omega)$ s.t. every other branching process β of Ω is a prefix of $Unf(\Omega)$. Moreover, $Unf(\Omega)$ represents all distributed runs of Ω, i.e., $R(Unf(\Omega)) = R(\Omega)$.* ⋆

The proof of this theorem follows directly from the propositions of the following lemma.

Lemma 7.13: *Let $\Omega = (O, \pi_0)$ be an oclet system with label consistent basic oclets O. The following properties hold.*

1. *Let β be a branching process of Ω and let C be a configuration of β. The induced distributed run $\beta[C]$ is a distributed run of Ω.*
2. *The runs represented by β are runs of Ω, i.e., $R(\beta) \subseteq R(\Omega)$.*
3. *Every monotone sequence $\beta^{(1)} \sqsubseteq \beta^{(2)} \sqsubseteq \ldots$ of branching process of Ω has a maximal element β^* s.t. $R(\beta^{(i)}) \subseteq R(\beta^*) \subseteq R(\Omega)$, for all $i > 0$.*
4. *Any two maximal, monotone sequences $\beta_1^{(1)} \sqsubseteq \beta_1^{(2)} \sqsubseteq \ldots \sqsubseteq \beta_1^*$ and $\beta_2^{(1)} \sqsubseteq \beta_2^{(2)} \sqsubseteq \ldots \sqsubseteq \beta_2^*$ eventually converge to $\beta_1^* = \beta_2^* = Unf(\Omega)$.*
5. *$R(Unf(\Omega)) = R(\Omega)$.* ⋆

Most propositions of Lemma 7.13 follow directly from the respective definitions. Only proposition 7.13-4 requires an argument that two possible extensions o_1 and o_2 of a branching process β can be added to β in any order. Hence, the following lemma.

Lemma 7.14 (Confluence of extension): *Let $\Omega = (O, \pi_0)$ be an oclet system with label consistent basic oclets O. Let $[o_1], [o_2] \in O$. Let β be a branching process of Ω having two possible extensions $o_1' \in [o_1]$ at a cut B_1 and $o_2' \in [o_2]$ at a cut B_2. Then either $\beta \rhd o_1' = \beta \rhd o_2'$, or $(\beta \rhd o_1') \rhd o_2' = (\beta \rhd o_2') \rhd o_1'$.* ⋆

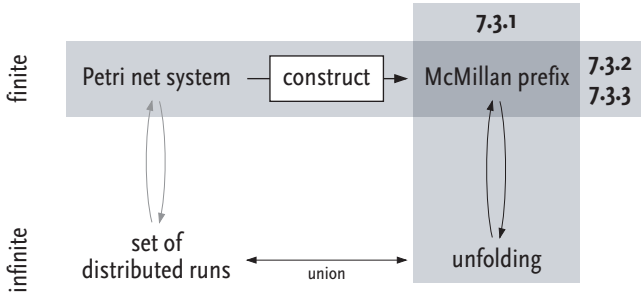
The proofs of both lemmas are given in Appendix A.8.

In the beginning of this section, we gave a characterization of the branching processes of an oclet system Ω. It is easy to see from the definitions that each branching process β that is constructed inductively satisfies this characterization. Conversely, every characterized branching process can be constructed.

Altogether, the branching processes of oclet systems generalize the branching processes of Petri net systems — because every Petri net system can be translated into an equivalent oclet system, see Section 6.7.1. The unique maximal branching process $Unf(\Omega)$ of an oclet system represents the set of all runs of Ω. Every condition in $Unf(\Omega)$ with two post-events e_1 and e_2 describes a point where two runs of Ω diverge because of the two alternative events e_1 and e_2. Like the set of all runs of Ω, $Unf(\Omega)$ may be infinite. The next section introduces a technique for computing an equivalent finite representation of $Unf(\Omega)$.

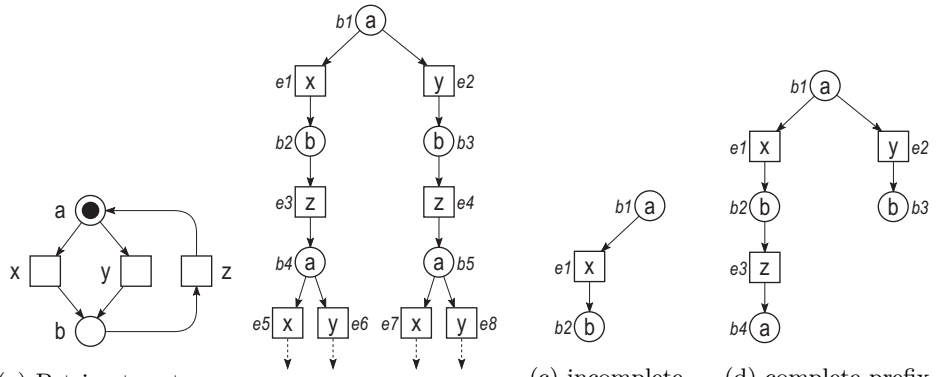
7.3. The McMillan Technique for Petri Nets

A Petri net system may have infinite runs, e.g., if the system has a cycle. Consequently, the system's unfolding may be infinite as well. By the same argument, an oclet system's unfolding may be infinite.



McMillan found in [88] that the unfolding of a Petri net system Σ is equivalently represented by a prefix β of the unfolding: β represents every *reachable marking* of Σ and every transition that occurs in Σ. This information is enough to reconstruct the unfolding from β. Such a prefix β is called *complete*. We present McMillan's approach in this section and generalize it to oclet systems from Section 7.3 onwards.

The specific contribution of McMillan's work is two-fold. Firstly, every Petri net system Σ which has a finite set of reachable markings also has a *finite* complete prefix of its unfolding. Section 7.3.1 introduces finite complete prefixes. Secondly, McMillan provided an algorithm to *construct* a finite complete prefix β. By construction, β represents all behavior of Σ. Algorithms on β may decide properties of the complete unfolding, i.e., behavioral properties of Σ. Section 7.3.2 presents McMillan's algorithmic idea, Section 7.3.3 provides the formal definitions. All formal notions in this section are based on [38].



(a) Petri net system Σ (b) its unfolding (c) incomplete prefix β_0 (d) complete prefix β_1

Figure 7.8. A Petri net system (a), its unfolding (b), a prefix (c), and a complete prefix (d).

7.3.1. Complete prefix of a Petri net system

This section briefly repeats the notion of a complete prefix of an unfolding of a Petri net system. We identify the central notions that are needed to generalize this technique to oclet systems.

Let Σ be a Petri net system. A prefix β of $Unf(\Sigma)$ is *complete* iff for every reachable marking m of Σ there exists a configuration C of β s.t.

1. $Mark(C) = m$, i.e., m is represented in β, and
2. for every transition t of Σ that is enabled at m there exists an event $e \notin C$ s.t. $\ell(e) = t$ and $C \cup \{e\}$ is a configuration of β [38].

The unfolding $\beta(\Sigma)$ is complete. The idea of a complete prefix β is that β contains just enough information to re-construct $\beta(\Sigma)$ from β. Intuitively, this is possible because every reachable marking m of Σ is represented in β and every occurrence of a transition t in marking m is represented in β.

The Petri net system Σ in Figure 7.8(a) has the infinite unfolding indicated in Figure 7.8(b). The prefix β_0 in Figure 7.8(c) is incomplete: β_0 represents every reachable marking of the system, but the occurrences of transitions y and z are not represented. Thus, the unfolding cannot be reconstructed from β_0. A complete prefix β_1 is depicted in Figure 7.8(d). This complete prefix β_1 is minimal but it is not unique, i.e., the system has other minimal complete prefixes.

Reconstructing the unfolding from a complete prefix. In the following, we present an intuitive argument why and how the unfolding Unf of Σ can be reconstructed from a complete prefix β of Unf. We generalize this argument in the next section to define a complete prefix of an oclet system.

The notion of a (finite) complete prefix of Unf relies on the following property of the unfolding: Let C_1 be a configuration of Unf. By $\Uparrow C_1$ we denote the *future*

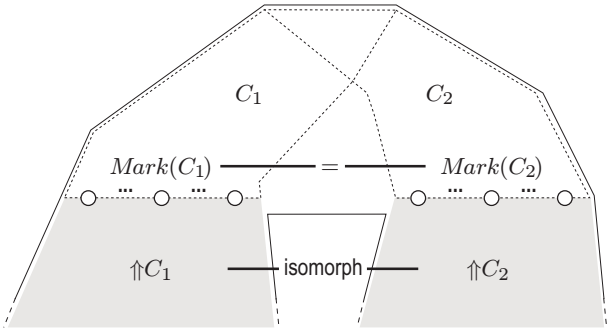


Figure 7.9. Configurations C_1 and C_2 reach the same marking $Mark(C_1) = Mark(C_2)$. Their futures $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic.

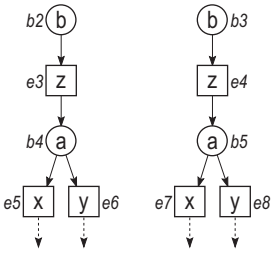


Figure 7.10. Futures $\Uparrow\{e_1\}$ (left) and $\Uparrow\{e_2\}$ (right) in the unfolding in Fig. 7.8(b).

of C_1 which is all behavior in *Unf* after $Cut(C_1)$. Technically, $\Uparrow C_1$ is the suffix of *Unf* that begins with $Cut(C_1)$ and contains all nodes (and arcs) of *Unf* that depend on $Cut(C_1)$. If a second configuration C_2 reaches the same marking as C_1, then their futures $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic [38]. Figure 7.9 illustrates this property. Figure 7.10 depicts the futures of the configurations $\{e_1\}$ and $\{e_2\}$ in the unfolding of Figure 7.8(b).

If $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic in the unfolding, then $\Uparrow C_2$ can be removed from *Unf* without losing any information about the behavior in *Unf*. Let $\beta := Unf - \Uparrow C_2$. Intuitively, the unfolding *Unf* can be reconstructed from the prefix β by appending (a copy of) the future $\Uparrow C_1$ to C_2. The result is *Unf* because $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic. This procedure of removing the future of a configuration from β can be iterated until every marking of Σ is represented only once. Each removal step can be undone by appending a copy of the future that is still represented in β.

This argument only provides an intuitive idea of how the unfolding can be reconstructed from a complete prefix. The actual technique for reconstructing the unfolding is more involved; details are given in [38].

7.3.2. Constructing a finite complete prefix: the algorithmic idea

We just recalled the notion of a complete prefix of a Petri net system Σ. A complete prefix β represents the entire behavior of Σ in the shape of a branching process. In the following, we present McMillan's algorithm to construct a finite complete prefix of Σ's unfolding. Figure 7.11 shows the running example of this section. For the Petri net in (a), McMillan's algorithm returns the finite complete prefix in (b). Figure 7.4 on page 196 shows the corresponding unfolding of the net.

McMillan's algorithm begins to construct the unfolding of Σ by extending a prefix β of the unfolding with one event at a time. It stops once β is complete. Each event in β represents an occurrence of a transition of Σ. Loosely speaking, after extending β with an event e, the algorithm checks whether "e reaches a marking"

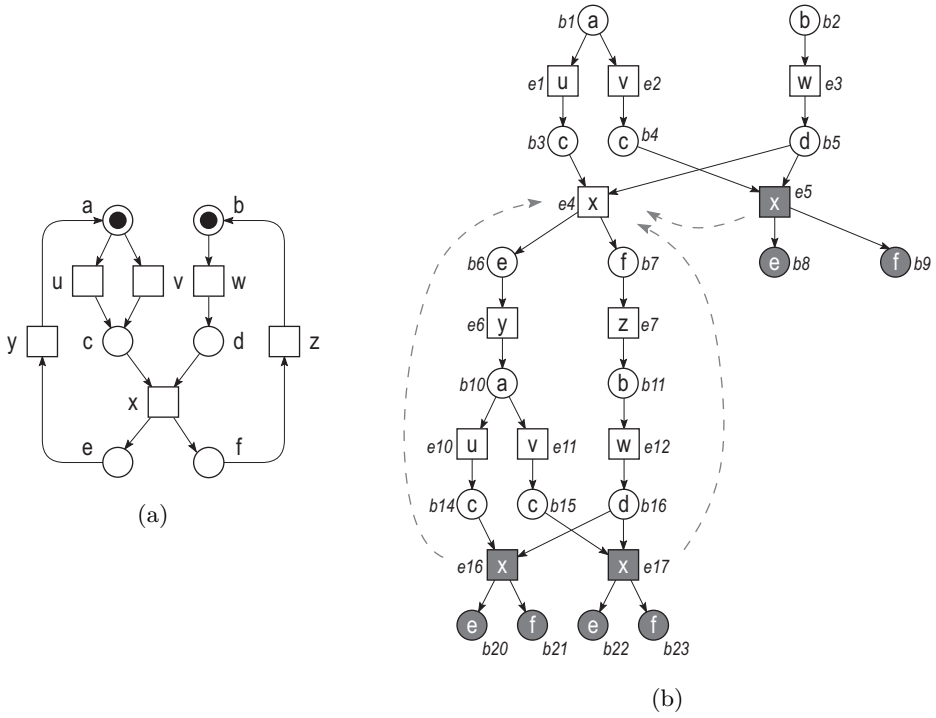


Figure 7.11. A Petri net system (a) and a finite complete prefix computed by McMillan's algorithm (b).

that has been "reached earlier" in β. If this is the case, then e is marked as a *cut-off event*. The algorithm does not extend β with any event that depends on a cut-off event. Eventually, if the net is bounded, every possible extension of β depends on a cut-off event and the algorithm terminates. The resulting prefix is complete. Cut-off events are highlighted grey in Figure 7.11.

McMillan provided two sophisticated notions for the two informal phrases "e reaches a marking" that has been "reached earlier". His choice yields an efficient technique as we explain next.

An event reaches a marking. Now, we consider one step in the construction algorithm. Let β be the current "intermediate" prefix and let e be the event that has just been added to β by the algorithm. Event e is part of some configuration C that reaches the marking $Mark(C)$. If the prefix β contains another configuration C' reaching the same marking $Mark(C') = Mark(C)$, then the future of event e, i.e., $\Uparrow e \subseteq \Uparrow C$, is already represented in β by $\Uparrow C'$ as explained in Section 7.3.1. So, the algorithm marks e as a *cut-off event* of β and no event depending on e is added to β.

Unfortunately, e may be part of many configurations C, and β contains exponentially many configurations C' to compare with C. A naïve algorithm for

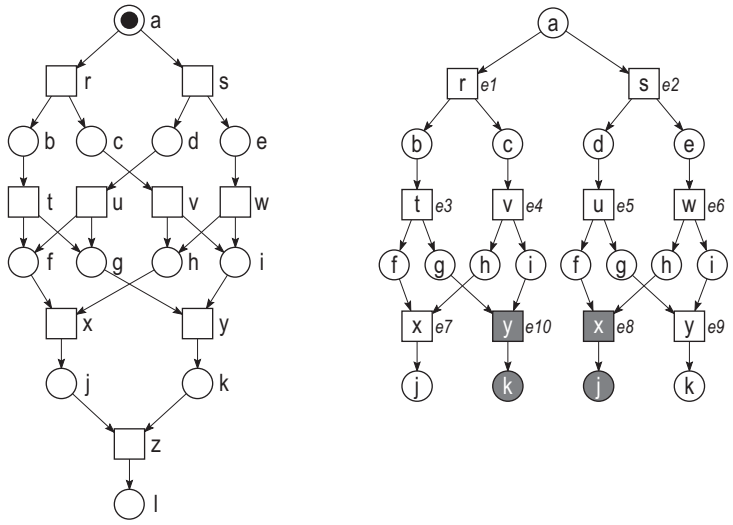


Figure 7.12. A Petri net system (a) and an incomplete prefix (b) of its unfolding.

detecting cut-off events would be inefficient. McMillan's solution is to consider only the *smallest* configuration that also contains e: this is the *local configuration* $[e]$ that consists of e and all events that precede e. Now, the marking "reached by e" is well-defined and unique by $Mark([e])$. In the example of Figure 7.11, the local configuration of event e_{10} is $[e_{10}] = \{e_{10}, e_6, e_4, e_3, e_1\}$; the reached marking is $Mark([e_{10}]) = [\mathsf{c}, \mathsf{f}]$ by the cut $\{b_{14}, b_7\}$ reached by $[e_{10}]$, see Def. 7.4-4.

The next step in the algorithm is to compare the local configuration $[e]$ of the added event e to other configurations C' in β. Although there are exponentially many configurations C', β contains only a linear number of local configurations $[e']$: one for each event e' of β. McMillan's first half of an efficient solution [88] is to compare the new local configuration $[e]$ only to existing local configurations $[e']$ of the prefix β.

A marking that has been reached earlier. However, it is insufficient to declare e as a cut-off event whenever β contains a local configuration $[e']$ with $Mark([e]) = Mark([e'])$. With this approach, the algorithm might declare too many cut-off events and terminate too early. Intuitively, the decision for a cut-off event is based on the behavior in a *local configuration* $[e]$ in β. However, some behavior in β only follows from a non-local configuration. Such behavior would not be represented in the prefix.

Figure 7.12 shows the standard counter-example taken from [38]. The marking [l] is reachable in the net. Depending on the order of adding events to the prefix, the algorithm could return the prefix of Figure 7.12(b) which does not represent [l]. The events are numbered in the order in which they were added to the prefix.

When adding e_8, the algorithm detects the local configuration $[e_7]$ that reaches the same marking $[\mathsf{j}, \mathsf{g}, \mathsf{h}]$ and marks e_8 as cut-off. Correspondingly, e_{10} is marked as cut-off because of $[e_9]$. An occurrence of transition z would depend either on the cut-off event e_8 or the cut-off event e_{10}; so the algorithm terminates and $[\mathsf{l}]$ is not represented. A correct algorithm ensures that either the global configuration $[e_7] \cup [e_{10}]$ or the global configuration $[e_8] \cup [e_9]$ does not contain a cut-off event.

McMillan's second half of the solution is to compare the local configuration $[e]$ to an existing local configuration $[e']$ only if $[e']$ contains less events than $[e]$. With this additional requirement, the algorithm terminates with a finite complete prefix. Esparza et al. generalize McMillan's idea in [38]. They define a partial order $\prec$ on the configurations, called *adequate* order. Event e is a *cut-off event* of β if β contains an event e' s.t. $[e'] \prec [e]$ and both local configurations reach the same marking $\mathit{Mark}([e]) = \mathit{Mark}([e'])$.

A sophisticated adequate order [38, Def. 5.1] yields the complete prefix in Figure 7.11. Events e_{16} and e_{17} are cut-off events because of the smaller local configuration $[e_4]$. Event e_5 is also a cut-off event because of $[e_4]$ although $[e_4]$ and $[e_5]$ have equal size. The trick is to compare the *words over transitions* in $[e_4]$ and $[e_5]$ "lexicographically": $\mathit{word}([e_4]) = \mathsf{uwx}$ and $\mathit{word}([e_5]) = \mathsf{vwx}$; the transitions in both words are ordered lexicographically; $\mathit{word}([e_4])$ is lexicographically smaller than $\mathit{word}([e_5])$, so e_4 has been reached "earlier" and e_5 is declared a cut-off event. Without lexicographical ordering, the prefix would be significantly larger.

7.3.3. The formal notions

The preceding section sketched McMillan's algorithm and introduced the notions of a *local configuration*, an *adequate order*, and a *cut-off event*. This section presents the corresponding formal definitions and the algorithm based on [38].

In the following let Σ be a Petri net system and let β be a branching process of Σ. Any configuration C of β induces the distributed run $\beta[C]$ that reaches the marking $\mathit{Mark}(C) := \mathit{Mark}(\beta[C])$. Two configurations that reach the same marking have isomorphic futures [38, Prop. 4.3]. Each event e induces its *local configuration* $[e]$ consisting of e and all events that precede e.

Definition 7.15 (Local configuration). The *local configuration* of an event e of a branching process is the configuration $[e] := \{e' \mid e' \leq e\}$. ⌟

The local configuration $[e]$ reaches the marking $\mathit{Mark}(e) := \mathit{Mark}([e])$. Thus, two events e_1 and e_2 that reach the same marking $\mathit{Mark}(e_1) = \mathit{Mark}(e_2)$ have isomorphic futures $\Uparrow[e_1]$ and $\Uparrow[e_2]$ in the unfolding of Σ.

A complete prefix β that contains two local configurations $[e_1]$ and $[e_2]$ that reach the same marking only has to contain one of the futures $\Uparrow[e_1]$ or $\Uparrow[e_2]$. The choice which future can be omitted is made on the basis of a partial order $\prec$. This partial order abstractly captures that an event e_2 "occurred earlier" than an event e_1 when $[e_2] \prec [e_1]$. Any partial order that satisfies the following three properties leads to a complete prefix.

Definition 7.16 (Adequate order, Definition 4.5 in [38]). A partial order $\prec$ on the finite configurations of the unfolding of an oclet system is an *adequate order* iff

1. $\prec$ is well-founded,
2. $C_1 \subset C_2$ implies $C_1 \prec C_2$, and
3. $\prec$ is preserved by finite extensions; if $C_1 \prec C_2$ and $\mathit{Mark}(C_1) = \mathit{Mark}(C_2)$, then the isomorphism α from $\Uparrow C_1$ to $\Uparrow C_2$ satisfies $(C_1 \cup E) \prec (C_2 \cup \alpha(E))$, for all finite sets E, $C_1 \cap E = \emptyset$, that extend C_1 to a configuration $C_1 \cup E$.⌟

The first condition (well-foundedness) ensures that any new configuration $[e]$ only has to be compared to finitely many smaller configurations $[e_n] \prec \ldots \prec [e_2] \prec [e_1] \prec [e]$ to check whether $\mathit{Mark}(e) = \mathit{Mark}(e_i)$, for some $i \in \{1, \ldots, n\}$. The second condition ensures that an adequate order captures "e_2 occurred earlier than e_1" in the objective case $[e_2] \subset [e_1]$. The third condition ensures that the order of "occurred earlier" is preserved by extensions. Two obvious candidates for an adequate order are the subset-relation, i.e., $C_1 \subset C_2$, and the size of the configurations, i.e., $|C_1| < |C_2|$. A *cut-off event* is defined wrt. the notion of an adequate order.

Let β be a prefix of the unfolding containing an event e. The event e is a *cut-off event* of β (wrt. $\prec$) iff β contains a local configuration $[e']$ s.t.

1. $\mathit{Mark}(e) = \mathit{Mark}(e')$ are equivalent, and
2. $[e'] \prec [e]$.

The algorithm that computes a finite complete prefix of the unfolding of $\Sigma = (N, m_0)$ begins the construction with the initial run $\beta := \pi_0$ that represents m_0 (see Def. 2.21). In the example of Figure 7.11, the initial run consists of the conditions $\{b_1, b_2\}$.

Each transition t of N defines a class of isomorphic t-steps. Each t-step is a small causal net A_t consisting of one event $e := \mathit{event}(A_t)$ and its pre- and post-conditions s.t. e represents t, ${}^\bullet e$ represents ${}^\bullet t$ and $e^\bullet$ represents $t^\bullet$; see Def. 2.20. Like for possible extensions of oclet systems (Def. 7.8), a t-step A_t is a *possible extension* of β if β contains a cut B that contains ${}^\bullet e$ s.t. there exists no event f of β with $\ell(t)$ and ${}^\bullet f = {}^\bullet e$. In the example of Figure 7.11, the u-step consisting of b_1, e_1, and b_3 is a possible extension of the initial run.

Let $\mathit{PE}(\beta)$ denote the set of all possible extensions of β. In each step, the algorithm first extends β by a possible extension A_t to $\beta \rhd A_t$ by appending $\mathit{event}(A_t)$ and its post-set to β. Then, the algorithm determines whether $\mathit{event}(A_t)$ is a cut-off event wrt. the chosen adequate order. A possible extension A_t is not added to β if the local configuration $[\mathit{event}(A_t)]$ in β contains a cut-off event. The algorithm terminates when no event can be added.

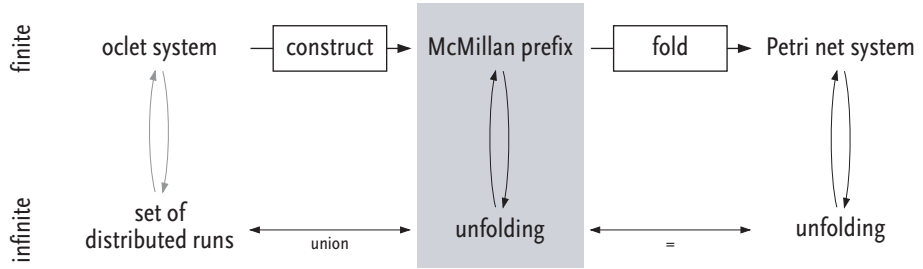
Algorithm 7.17 (Finite complete prefix of a Petri net, Algorithm 4.7 in [38]).

input: A bounded Petri net system $\Sigma = (N, m_0)$.
output: A complete finite prefix Fin of $Unf(\Sigma)$.
begin
  $Fin := \pi_0$;
  $pe := PE(Fin)$;
  $cut\text{-}off := \emptyset$;
  while $pe \neq \emptyset$ **do**
    choose a possible extension A_t in pe s.t. $[event(A_t)]$ is minimal wrt. $\prec$;
    if $[event(A_t)] \cap cut\text{-}off = \emptyset$ **then**
      $Fin := Fin \rhd A_t$;
      $pe := PE(Fin)$;
      if $event(A_t)$ is a cut-off event of Fin **then**
        $cut\text{-}off := cut\text{-}off \cup \{event(A_t)\}$
      endif
    else
      $pe := pe \setminus \{A_t\}$;
    endif
  endwhile
end ⌟

Algorithm 7.17 returns a complete finite prefix Fin of a bounded Petri net system. That Fin is indeed finite and complete is proven in [38] (Propositions 4.8 and 4.9).

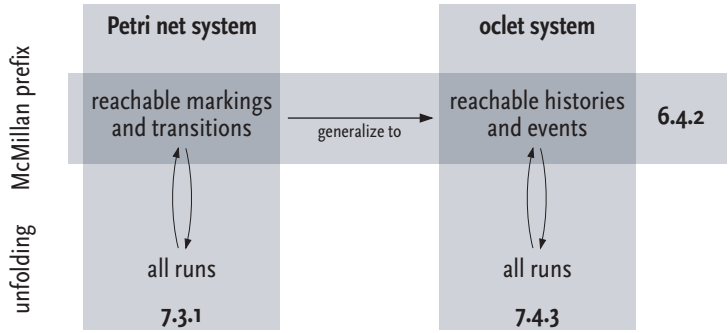
7.4. Complete Prefixes of an Oclet System

The preceding section recalled McMillan's technique for constructing a finite complete prefix of a Petri net system. In this section, we generalize the notion of finite complete prefix to oclet systems. Like for Petri nets, the aim is that a finite complete prefix β of an oclet system Ω represents all behavior of Ω. Section 7.5 presents an algorithm that *constructs* a finite complete prefix β of an oclet system Ω. Section 7.6 shows how to adapt this algorithm to decide behavioral properties of Ω. Chapter 8 uses β to *synthesize* a Petri net that exhibits the same behavior as Ω.



7.4.1. How to generalize complete prefixes?

This section discusses how to generalize the notion of a complete prefix of a Petri net system to oclet systems. A prefix β of the unfolding of a Petri net system Σ is only complete if β represents every reachable marking m of Σ. This characterization of a complete prefix does not apply to an oclet system Ω because the behavior of Ω is based on *histories* of events instead of markings. To generalize complete prefixes, we generalize the notion of a reachable marking of a Petri net system to a *reachable history* of an oclet system.



This section investigates the properties that a *reachable history* shall have in this context; Section 7.4.2 presents the formal notions. The complete prefixes of the unfolding of an oclet system Ω then follow canonically in Section 7.4.3. We show under which conditions Ω has a *finite* complete prefix.

Reachable history and future behavior. In a Petri net system, a marking determines the future behaviors of the system. Thus, any two configurations C_1 and C_2 that reach the same marking m have isomorphic futures $\Uparrow C_1$ and $\Uparrow C_2$. In this sense, the markings of Σ define an *equivalence relation* on the configurations of $Unf(\Sigma)$. And therefore the unfolding $Unf(\Sigma)$ can be reconstructed from β.

An oclet system Ω needs a corresponding notion that determines the future behavior of the system. This yet to be defined notion has to define an equivalence relation $\sim$ on the configurations of the unfolding of Ω s.t.

> any two equivalent configurations $C_1 \sim C_2$ of $Unf(\Omega)$ have isomorphic futures $\Uparrow C_1$ and $\Uparrow C_2$ in $Unf(\Omega)$. (7.1)

Figure 7.13 illustrates this property. The rationale of this equivalence $\sim$ is to distinguish two configurations of $Unf(\Omega)$ only if they have different futures. Thus, a prefix β of $Unf(\Omega)$ is *complete* if β contains for each equivalence class of $\sim$ a configuration of this equivalence class. If $\sim$ has only finitely many equivalence classes, then there exists a *finite* complete prefix β, because β contains only finitely many non-equivalent configurations.

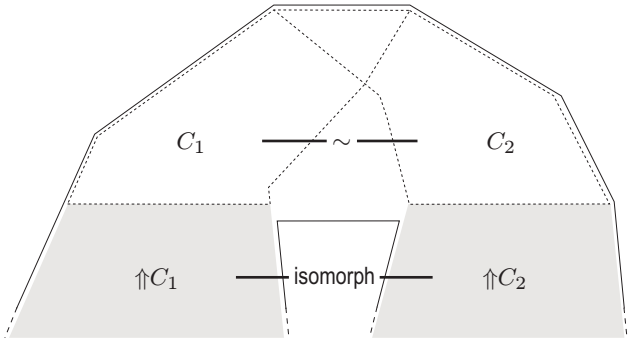


Figure 7.13. Generalization of the unfolding of Petri nets: configurations C_1 and C_2 are equivalent; their futures $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic.

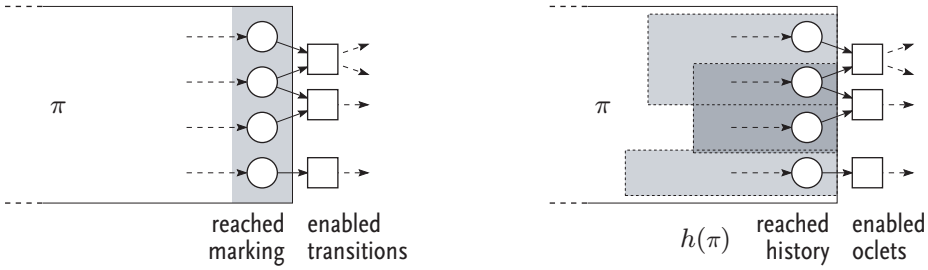


Figure 7.14. The history $h(\pi)$ is a suffix of π that contains the histories of all oclets that are enabled at π. The notion of a history generalizes the notion of a reached marking.

The next steps. In the next Section 7.4.2, we show that each oclet system Ω defines an equivalence relation $\sim$ that satisfies (7.1). Section 7.4.3 characterizes oclet systems in which $\sim$ has only finitely many equivalence classes. We then prove that if $\sim$ has only finitely many equivalence classes, then Ω has a finite complete prefix β of its unfolding $Unf(\Omega)$. This prefix β represents all behavior of Ω.

7.4.2. History equivalence

The behavior at the end of a run π of a *Petri net system* is determined by the marking that is reached by π. The behavior at the end of a run π of an *oclet system* Ω is determined by a *history* $h(\pi)$ of finite length — as we will show in this section. Two runs that have the same history allow for the same future behaviors (up to isomorphism). Thus, the history $h(.)$ defines the complete prefixes of Ω.

The idea

In an oclet system $\Omega = (O, \pi_0)$, each basic oclet o denotes one event $event(o)$ with a local history $hist_o$. An occurrence of o depends on the local history of o, i.e., o is *enabled* at the end of a distributed run π_1 iff π_1 ends with $hist_o$.

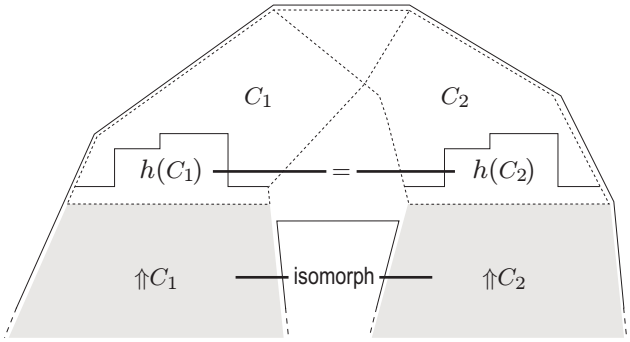


Figure 7.15. Equivalence for oclet systems: the runs induced by the configurations C_1 and C_2 have equal histories. The futures $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic.

Every oclet in O is finite. Consequently, whether o is enabled at the end of π_1 does not depend on the entire run π_1. Instead it is sufficient to consider a finite suffix $h(\pi_1)$ of π_1 that is large enough to contain $hist_o$ (and any other history of any other oclet in O). We call $h(\pi_1)$ the *characteristic history* of π_1 (wrt. O). The trivial suffix of π_1 is its set of maximal conditions which corresponds to the marking reached by π_1. The characteristic history of π_1 typically contains more nodes. Thus, the notion of a characteristic history generalizes the notion of a marking.

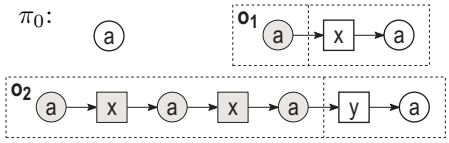
In the following, we define $h(.)$ s.t. each oclet o of Ω is enabled at π_1 iff o is enabled at $h(\pi_1)$. Thus, any two runs π_1 and π_2 of Ω that end with the same characteristic history $h(\pi_1) = h(\pi_2)$ continue in exactly the same way. If π_1 and π_2 end with different characteristic histories $h(\pi_1) \neq h(\pi_2)$, then also their continuations differ. Figure 7.14 depicts these concepts abstractly.

This characteristic history $h(.)$ of a run induces an equivalence relation to characterize a complete prefix of the unfolding Unf of Ω. Two configurations C_1 and C_2 are *h-equivalent* iff their induced runs $\pi_1 = Unf[C_1]$ and $\pi_2 = Unf[C_2]$ have equal characteristic histories $h(\pi_1) = h(\pi_2)$. From the definition of $h(.)$ follows that h-equivalent configurations C_1 and C_2 have isomorphic futures $\Uparrow C_1$ and $\Uparrow C_2$ as illustrated in Figure 7.15.

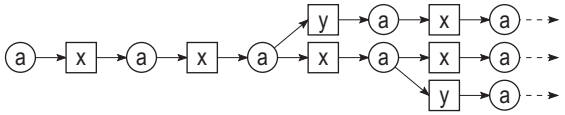
The remainder of this Section 7.4.2 formally defines $h(.)$ and proves the stated property.

The definitions

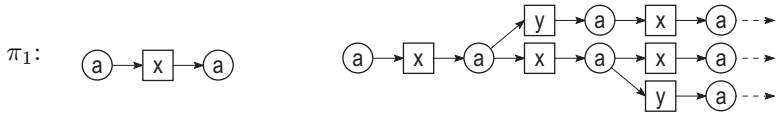
A naïve approach to the history of a run would be to define $h(\pi)$ as the union of all histories that occur at the end of π. This definition would imply: o is enabled at π iff o is enabled at $h(\pi)$. But this definition is too weak for (7.1) which requires that h-equivalent runs have isomorphic futures. Figure 7.16 depicts a counter-example. The oclet system Ω in (a) has the unfolding indicated in (b). Both runs π_0 and π_1 enable oclet o_1, i.e., π_0 and π_1 would be "equivalent". However, the future



(a) oclet system $\Omega = (\{\mathsf{o}_1, \mathsf{o}_2\}, \pi_0)$



(b) unfolding of Ω



(c) future of π_1

Figure 7.16. In π_0 and in π_1 oclet o_1 is enabled and o_2 is not enabled. The future of π_0 is the unfolding (b), the future of π_1 in (c) is not isomorphic to the unfolding.

of π_0 is the unfolding of Ω which is different from the future of π_1 depicted in Figure 7.16(c).

The counter-example of Figure 7.16 tells us that the future of a run π does not only depend on the complete histories of the enabled oclets. The future of π also depends on the *partial histories* that occur at the end of π. In Figure 7.16, the history of o_1 occurs completely at the end of π_1. Furthermore, we may "stick" the history of o_2 "partially into π_1" up to the local history of the second event x. Figure 7.17 depicts this partial history of o_2 as $hist_1$ explicitly. In other words, π_1 also ends with the partial history $hist_1$. We show in the following that partial histories suffice to determine whether two runs have isomorphic futures.

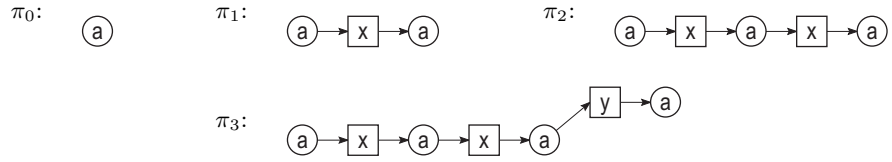
Definition 7.18 (Partial histories in a run). Let $o = (\pi_o, hist_o)$ be an oclet. Let e be an event of o (from o's contribution or o's history). The local history of e is a *partial history* of o.

Technically, each event $e \in E_o$ induces the *partial history* $hist_e = (B_e, E_e, F_e)$ consisting of all strict transitive predecessors $X_e = e{\downarrow_o} \setminus \{e\}$ of e, $B_e = X_e \cap B_o$, $E_e = X_e \cap E_o$, $F_e = F_o|_{X_e \times X_e}$. Each partial history $hist_e$ induces its isomorphism class $[hist_e]$.

Let $\Omega = (O, \pi_0)$ be an oclet system. The set of all partial histories of an oclet class $[o] \in O$ is $\mathcal{H}([o]) := \{[hist_e] \mid e \in E_o\}$. The set of all *partial histories of* O is the set $\mathcal{H}(O) := \bigcup_{[o] \in O} \mathcal{H}([o])$.

Let π be a distributed run. The set of *partial histories of* O *that occur at the end of* π is $\mathcal{H}(O, \pi) := \{hist^\alpha \mid [hist] \in \mathcal{H}(O), \exists hist^\alpha \in [hist] : \pi \xrightarrow{hist^\alpha}]\}$. ⌟

(a) the partial histories of o_1 and o_2 of Figure 7.16



(b) some distributed runs of Ω of Figure 7.16

Figure 7.17. The distributed runs π_0 and π_3 end with $hist_0$ only, π_1 ends with $hist_0$ and $hist_1$, π_2 ends with all partial histories.

The condition $\pi \xrightarrow{hist^\alpha}]$ in the definition of $\mathcal{H}(O,\pi)$ captures the enabling condition of oclets for partial histories: $hist_e^\alpha \subseteq \pi$ and $\max hist_e^\alpha \subseteq \max \pi$; see and *enabled oclet* (Def. 5.1).

The example in Figure 7.17 illustrates Definition 7.18. Figure 7.17(a) shows all partial histories of Ω in Figure 7.16: $\mathcal{H}(\mathsf{o}_1) = \{[hist_0]\}, \mathcal{H}(\mathsf{o}_2) = \{[hist_0], [hist_1], [hist_2]\}$, so $\mathcal{H}(\{\mathsf{o}_1, \mathsf{o}_2\}) = \mathcal{H}(\mathsf{o}_1) \cup \mathcal{H}(\mathsf{o}_2)$. Both runs π_0 and π_3 in Figure 7.17(b) end with $hist_0$ only, i.e., $\mathcal{H}(\{\mathsf{o}_1, \mathsf{o}_2\}, \pi_0) = \{hist_0'\}$ and $\mathcal{H}(\{\mathsf{o}_1, \mathsf{o}_2\}, \pi_3) = \{hist_0'\}$ for occurrences $hist_0', hist_0'' \in [hist_0]$. In contrast, the run π_1 ends with $hist_0$ and $hist_1$, and the run π_2 ends with all partial histories in $\mathcal{H}(\{\mathsf{o}_1, \mathsf{o}_2\})$. All runs π_0-π_3 are also runs of the oclet system Ω in Figure 7.16. On a more figurative level, the histories of the oclets o_1 and o_2 "stick" in these runs at different depths, formally expressed by the partial histories of o_1 and o_2. All partial histories of Ω "stick" in π_0 and π_3 *at the same depth*, i.e., they are *history equivalent* wrt. partial histories.

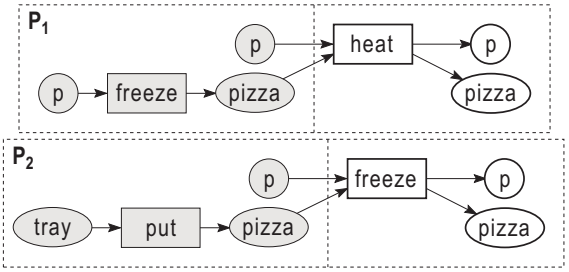
The preceding notions were chosen in a way s.t. each partial history $hist \in \mathcal{H}(O,\pi)$ that occurs at the end of a distributed run π is a sub-net of π. Let $\bigcup \mathcal{H}(O,\pi)$ denote the composition of all these sub-nets by union.

Definition 7.19 (Characteristic history). Let $\Omega = (O, \pi_0)$ be an oclet system and let π be a distributed run. The *characteristic history* of π is the suffix $h(\pi)$ of π that consists of $\max \pi$ and all partial histories of O that occur at the end of π, i.e., $h(\pi) := \max \pi \cup \bigcup \mathcal{H}(O,\pi)$.

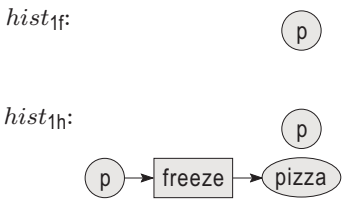
Two distributed runs π_1 and π_2 are *h-equivalent* iff $h(\pi_1)$ and $h(\pi_2)$ are equal (up to isomorphism), i.e., $[h(\pi_1)] = [h(\pi_2)]$. ⌟

In the example of Figure 7.17, the distributed runs π_0 and π_3 are h-equivalent. Their characteristic history is $hist_0$. Moreover, both runs have isomorphic futures in the unfolding of Figure 7.16.

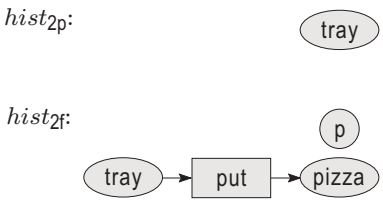
Figure 7.18 depicts a more involved example of a characteristic history. The partial histories of oclet P_1 of (a) are depicted in (b): $hist_{1\mathsf{f}}$ is the partial history of freeze, $hist_{1\mathsf{h}}$ is the partial history of heat. The partial histories of oclet P_2



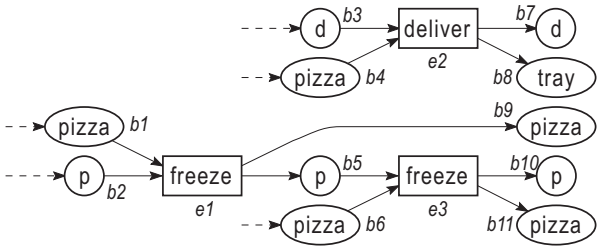
(a) two oclets P_1 and P_2



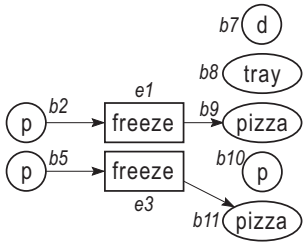
(b) partial histories $\mathcal{H}(\mathsf{P}_1)$



(c) partial histories $\mathcal{H}(\mathsf{P}_2)$



(d) a distributed run π



(e) characteristic history $h(\pi)$

Figure 7.18. The distributed run π in (d) has the characteristic history $h(\pi)$ in (e) wrt. the oclets P_1 and P_2 in (a).

are depicted in (c). The partial histories of $\{\mathsf{P}_1, \mathsf{P}_2\}$ that occur at the end of the run π of (d) are the following: $hist_{1\mathsf{f}}$ occurs once at condition $\{b_{10}\}$, $hist_{1\mathsf{h}}$ occurs twice at $\{b_9, b_{10}\}$ and at $\{b_{10}, b_{11}\}$, $hist_{2\mathsf{p}}$ occurs once at $\{b_8\}$, and $hist_{2\mathsf{f}}$ does not occur at the end of π. The characteristic history $h(\pi)$ of π consists of all these occurrences of the partial histories of $\{\mathsf{P}_1, \mathsf{P}_2\}$ *and* the condition b_7 because $h(\pi)$ contains all maximal nodes of π; Figure 7.18(e) shows $h(\pi)$.

In the remainder of this section, we prove that two runs which have the same characteristic history also have isomorphic futures. This property allows us to define a complete prefix of an oclet system using the notion of a characteristic history.

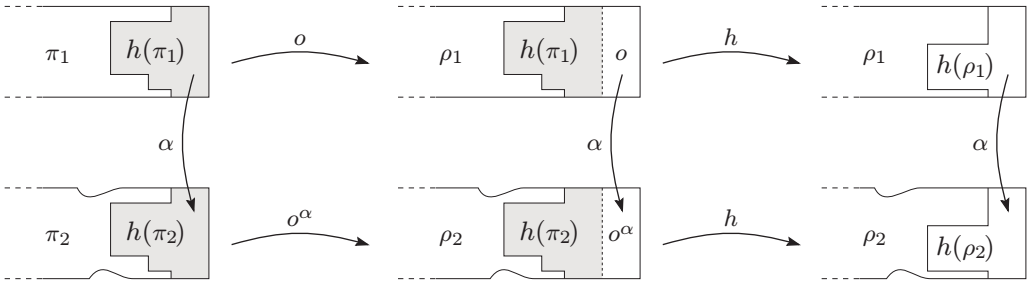


Figure 7.19. The propositions of Lemma 7.22.

Lemma 7.20: *Let $\Omega = (O, \pi_0)$ be an oclet system and let C_1 and C_2 be two configurations of the unfolding Unf of Ω s.t. $h(Unf[C_1])$ and $h(Unf[C_2])$ are isomorphic. Then $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic in Unf.* ⋆

The proofs

The following corollary states some basic properties of the characteristic history of a distributed run. It follows directly from Definitions 7.18 and 7.19.

Corollary 7.21 (Properties of the characteristic history): *Let $\Omega = (O, \pi_0)$ be an oclet system and let π be a distributed run. Then the following properties hold.*

1. *$h(\pi)$ is a distributed run with $h(\pi) \subseteq \pi$ and $\max h(\pi) = \max \pi$.*
2. *For each partial history $hist \in \mathcal{H}(O)$ holds: the run π ends with $[hist] \in \mathcal{H}(O)$ at location α iff $h(\pi)$ ends with $[hist] \in \mathcal{H}(O)$ at location α, i.e., $\pi \xrightarrow{hist^\alpha}]$ iff $h(\pi) \xrightarrow{hist^\alpha}]$ for $hist^\alpha \in [hist]$.*
3. *For each oclet $o \in O$ holds: o is enabled at π at location α iff o is enabled at $h(\pi)$ at location α.* ⋆

The proof of Lemma 7.20 (h-equivalent runs have isomorphic futures) will be done inductively. The subsequent lemma states all necessary properties for the induction step. Specifically, if two runs π_1 and π_2 are h-equivalent, then each continuation of π_1 by a single oclet defines an isomorphic continuation of π_2. Moreover, the resulting runs are h-equivalent again. Figure 7.19 illustrates these propositions.

Lemma 7.22: *Let $\Omega = (O, \pi_0)$ be an oclet system and let π_1 and π_2 be two distributed runs s.t. their characteristic histories $h(\pi_1)$ and $h(\pi_2)$ are isomorphic. Let α be the isomorphism from $h(\pi_1)$ to $h(\pi_2)$. Let $[o] \in O$ be an oclet class s.t. o is enabled at π_1. Then the following properties hold.*

1. *Oclet o is enabled at π_2 at location α, and $h(\pi_1) \rhd o$ and $h(\pi_2) \rhd o^\alpha$ are isomorphic.*
2. *The characteristic histories of $\pi_1 \rhd o$ and $\pi_2 \rhd o^\alpha$ are isomorphic, i.e., α extends to an isomorphism from $h(\pi_1 \rhd o)$ to $h(\pi_2 \rhd o^\alpha)$.* ⋆

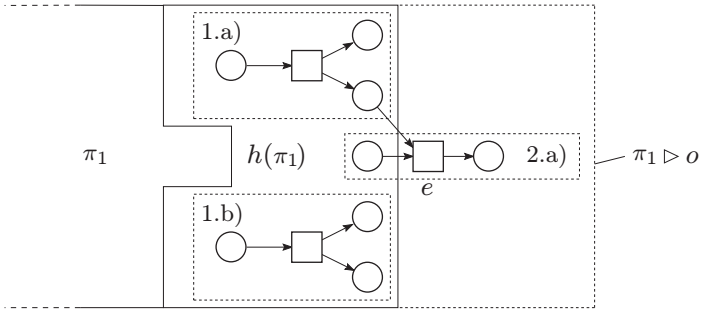


Figure 7.20. Three of the four cases in the proof of Lemma 7.22.

Proof. Let o be enabled at π_1 and let $\alpha : h(\pi_1) \to h(\pi_2)$ be an isomorphism.

(*1.*) From o enabled at π_1 follows that o is enabled at $h(\pi_1)$ by Corollary 7.21. By the isomorphism α holds that o^α is enabled at $\alpha(h(\pi_1)) = h(\pi_2)$. Thus o^α is enabled at π_2. The composition $h(\pi_1) \rhd o$ extends the isomorphism $\alpha : h(\pi_1) \to h(\pi_2)$ to an isomorphism from $(h(\pi_1) \rhd o)$ to $(h(\pi_2) \rhd o^\alpha)$.

(*2.*) We have to show that the characteristic histories $h(\pi_1 \rhd o)$ and $h(\pi_2 \rhd o^\alpha)$ are isomorphic by α. By the definition of the characteristic history $h(.)$ we have to show (*2.a*) $\max(\pi_1 \rhd o)$ and $\max(\pi_2 \rhd o^\alpha)$ isomorphic, and (*2.b*) $\mathcal{H}(O, \pi_1 \rhd o)$ and $\mathcal{H}(O, \pi_2 \rhd o^\alpha)$ are isomorphic. The isomorphism of $\max(\pi_1 \rhd o)$ and $\max(\pi_2 \rhd o^\alpha)$ already follows from (*1.*). Thus, we have to show that $hist \in \mathcal{H}(O, \pi_1 \rhd o)$ iff $hist^\alpha \in \mathcal{H}(O, \pi_2 \rhd o^\alpha)$. We distinguish four cases; Figure 7.20 illustrates three of them.

1. Let $hist \in \mathcal{H}(O, \pi_1)$. By the definition of $h(.)$ and the isomorphism α from $h(\pi_1)$ and $h(\pi_2)$ follows $hist^\alpha \in \mathcal{H}(O, \pi_2)$. There are two cases:
   a) $hist \notin \mathcal{H}(O, \pi_1 \rhd o)$. In other words, the run $\pi_1 \rhd o$ does not end with $hist$ although π_1 does. Thus, the event e that is appended to π_1 has a pre-condition b in $\max hist$; and e is a successor of b in $\pi_1 \rhd o$. But then is b a maximal condition of o's history. Thus, $hist \notin \mathcal{H}(O, \pi_1 \rhd o)$ iff $(\max hist \cap \max hist_o) \neq \emptyset$ iff $(\max hist^\alpha \cap \max hist^\alpha_o) \neq \emptyset$ iff $hist^\alpha \notin \mathcal{H}(O, \pi_2 \rhd o^\alpha)$.
   b) $hist \in \mathcal{H}(O, \pi_1 \rhd o)$. Thus, the event that is appended to π_1 has no pre-condition in $\max hist$. Hence, the equivalence of the first case holds: $hist^\alpha \in \mathcal{H}(O, \pi_2 \rhd o^\alpha)$.
2. Let $hist \notin \mathcal{H}(O, \pi_1)$. Like in the first case, $hist^\alpha \notin \mathcal{H}(O, \pi_2)$. There are two cases:
   a) $hist \in \mathcal{H}(O, \pi_1 \rhd o)$. This only happens if the oclet o contributes a node s.t. $hist$ occurs at the end of $\pi_1 \rhd o$, i.e., $hist \cap con_o \neq \emptyset$. Thus, $hist \in \mathcal{H}(O, \pi_1 \rhd o)$ iff $(hist \cap con_o) \neq \emptyset$ iff $(hist^\alpha \cap con^\alpha_o) \neq \emptyset$ iff $hist^\alpha \in \mathcal{H}(O, \pi_2 \rhd o^\alpha)$.

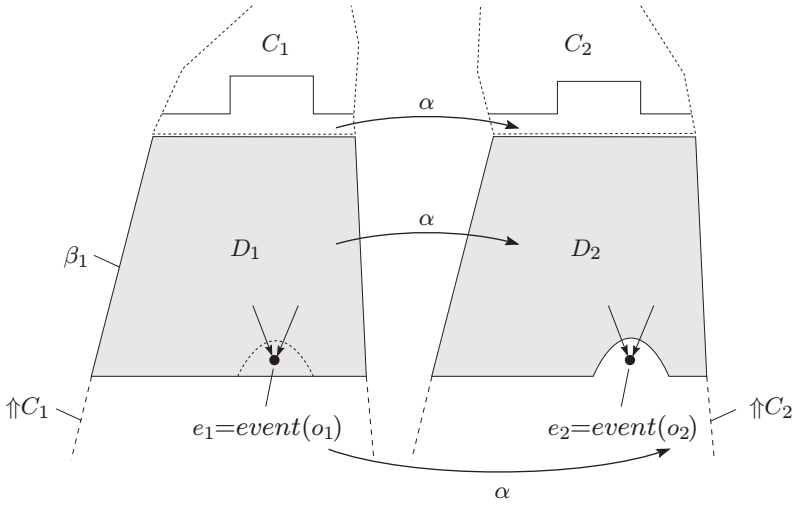


Figure 7.21. Proof idea for Lemma 7.20.

b) $hist \notin \mathcal{H}(O, \pi_1 \rhd o)$. By the inverse arguments of the first case, holds $hist^\alpha \notin \mathcal{H}(O, \pi_2 \rhd o^\alpha)$.

Altogether, α is an isomorphism from $\bigcup \mathcal{H}(O, \pi_1 \rhd o)$ to $\bigcup \mathcal{H}(O, \pi_2 \rhd o^\alpha)$ and from $\max(\pi_1 \rhd o)$ to $\max(\pi_2 \rhd o^\alpha)$. Thus, α is an isomorphism from $h(\pi_1 \rhd o)$ to $h(\pi_2 \rhd o^\alpha)$. □

By the preceding lemma, two runs π_1 and π_2 of an oclet system Ω that are h-equivalent are guaranteed to continue in an isomorphic manner. Moreover, h-equivalence is preserved under continuations. To prove Lemma 7.20, we lift this property to the complete future of π_1 and π_2 in the unfolding of Ω. Figure 7.21 sketches the proof idea.

Proof (of Lemma 7.20). Let $\Omega = (O, \pi_0)$ be an oclet system and let C_1 and C_2 be two configurations of the unfolding Unf of Ω s.t. $h(Unf[C_1])$ and $h(Unf[C_2])$ are isomorphic. We show that $\Uparrow C_1$ and $\Uparrow C_2$ are isomorphic by induction on the prefixes of $\Uparrow C_1$ and $\Uparrow C_2$.

[Base] The least prefix of $\Uparrow C_1$ is the $Cut(C_1) = \max Unf[C_1] = \max h(Unf[C_1])$. The least prefix of $\Uparrow C_2$ is $\max h(Unf[C_2])$. Both prefixes are isomorphic by assumption.

[Step] Let β_1 be a prefix of $\Uparrow C_1$ containing an event. We have to show that there exists an isomorphic prefix β_2 of $\Uparrow C_2$.

Let e_1 be a maximal event of β_1 s.t. for no other event e' of β holds $e_1 < e'$. Let β_1' be the prefix of β_1 where event e_1 and $e_1^\bullet$ were removed. By inductive assumption exist a prefix β_2' of $\Uparrow C_2$ s.t. β_1' and β_2' are isomorphic. We now show that there exist an event e_2 in $\Uparrow C_2$ s.t. extending β_2' with e_2 yields the prefix β_2 of Unf that is isomorphic to β_1.

Let $D_1 = \{e' \in E_1 \mid e' < e_1\}$ be the configuration of β_1 that precedes e_1. From $e_1 \notin D_1$ follows that D_1 is a configuration of β'_1. Thus, there exists an isomorphic configuration D_2 of β'_2.

By construction holds that $C_1 \cup D_1$ is a configuration of the unfolding Unf and $Unf[C_1]$ is a prefix of $Unf[C_1 \cup D_1]$. Correspondingly, $Unf[C_2]$ is a prefix of $Unf[C_2 \cup D_2]$. By assumption $h(Unf[C_1])$ and $h(Unf[C_2])$ isomorphic. By induction on the size of D_1 and Lemma 7.22 it follows that $h(Unf[C_1 \cup D_1])$ and $h(Unf[C_2 \cup D_2])$ are isomorphic.

The induced run $Unf[C_1 \cup D_1]$ is a branching process of Ω. All events that precede e_1 are in the configuration $C_1 \cup D_1$. Thus, there exists a possible extension o_1 of $Unf[C_1 \cup D_1]$ s.t. $event(o_1) = e_1$. Correspondingly, there exists a possible extension o_2 of $Unf[C_2 \cup D_2]$ that is isomorphic to o_1 —because $h(Unf[C_1 \cup D_1])$ and $h(Unf[C_2 \cup D_2])$ are isomorphic. The possible extension o_2 contributes the $event(o_2) = e_2$ of Unf in $\Uparrow C_2$. By construction, $D_2 \cup \{e_2\}$ is a configuration of $\Uparrow C_2$. Thus, adding e_2 and $e_2^\bullet$ to β'_2 yields the prefix β_2 of $\Uparrow C_2$ that is isomorphic to β_1. □

This proof concludes the definition of a characteristic history $h(\pi)$ of a distributed run π. Specifically, we have proven that two h-equivalent runs have isomorphic futures. Thus, the notion of h-equivalence allows us to define the complete prefixes of an oclet system.

7.4.3. Complete prefix of an oclet system

The preceding section introduced the notion of a *characteristic history* $h(\pi)$ of a distributed run π wrt. an oclet system Ω. It generalizes the notion of a marking (of a Petri net system). Two runs of Ω that reach the same characteristic history have isomorphic futures. Because of this property, we may now define the *complete prefixes* of an oclet system (wrt. characteristic histories).

The complete prefixes of an oclet system generalize complete prefixes of a Petri net system. We generalize the notion of a reachable marking to a *reachable characteristic history*. A complete prefix of a Petri net system represents every reachable marking. A complete prefix of an oclet system represents every reachable characteristic history.

Definition 7.23 (Reachable history). Let Ω be an oclet system. Let C be a finite configuration of the unfolding Unf of Ω. The characteristic history $hist = h(Unf[C])$ is *reachable* in Ω. Two finite configurations C_1 and C_2 reach the same history iff $h(Unf[C_1])$ and $h(Unf[C_2])$ are equal (up to isomorphism). Relaxing notation, we write $h(C_1) = hist = h(C_2)$. ⌟

Definition 7.24 (Complete prefix of an oclet system). Let $\Omega = (O, \pi_0)$ be an oclet system. A prefix β of the unfolding Unf of Ω is *complete* iff for every reachable characteristic history $hist = h(C)$ of some finite configuration C of Unf there exists a configuration C' of β s.t.

1. $h(C) = hist = h(C')$ (C and C' reach the same history, i.e., β represents the run $Unf[C]$ of Ω), and

2. for every oclet o of Ω that is enabled at $Unf[C]$ there exists an event $e' \notin C'$ with $\ell(e) = \ell(event(o))$ s.t. $C' \cup \{e'\}$ is a configuration of β. ⌟

Figure 7.5 on page 197 depicts an example. The depicted branching process in Figure 7.5(b) is a complete prefix of the unfolding of the oclet system in Figure 7.5(a).

Definition 7.24 corresponds to the notion of a complete prefix of a Petri net system given in Section 7.3.1. We only replaced "reachable marking" by "reachable characteristic history". Thus, Definition 7.24 generalizes complete prefixes of Petri net systems.

A complete prefix β of an oclet system Ω represents the same behavior as the unfolding Unf of Ω. The prefix β contains for every finite configuration C of Unf an h-equivalent configuration C'. By Lemma 7.20, the futures of C and C' are isomorphic. Moreover, β contains for each event e that extends C an equivalent event e' that extends C'. Thus, every extension of C with some events E to $C \cup E$ can be mimicked by equivalent events E' of β.

The unfolding of Ω is complete by definition. In the following, we are interested in *finite* complete prefixes of Ω. An algorithm can decide properties of Unf on a finite complete prefix β, i.e., decide behavioral properties of Ω on β.

Finite complete prefixes

Whether an oclet system Ω has a *finite* complete prefix only depends on how many different characteristic histories are reachable in Ω. If the set of reachable characteristic histories is finite, then Ω has a finite complete prefix. Otherwise, every complete prefix β represents infinitely many incomparable histories which means that β is infinite as well.

Intuitively, a characteristic history *hist* has only finite length (bounded by the longest history in Ω). But *hist* could have arbitrary width, i.e., the number of maximal conditions of *hist* may be unbounded. If there exists a bound k s.t. every cut of *hist* has at most k conditions, then every reachable history has width $\leq k$. In this case Ω reaches only finitely many incomparable characteristic histories and Ω has a finite complete prefix.

In the following, we formally characterize *k-bounded* oclet systems *where the size of every cut is bounded by a constant k*. Then, we prove that every k-bounded oclet system has a finite complete prefix.

Formal definitions and proofs

Petri nets know the notion of a k-bounded Petri net system Σ where every reachable marking of Σ puts at most k tokens on a place of Σ. We adapt this notion to oclet systems by bounding the conditions in a cut.

Definition 7.25 (k-bounded oclet system). Let Ω be an oclet system, let $k \in \mathbb{N}$. A cut B of $Unf(\Omega)$ is *k-bounded* iff each label occurs in B at most k times, i.e., $|\{b \in B \mid \ell(b) = a\}| \leq k$, for all labels a. The oclet system Ω is *k-bounded* iff every cut of $Unf(\Omega)$ is *k-bounded*. An oclet specification O is *k-bounded* iff its oclet system Ω_O is k-bounded; see Cor. 6.17. ⌟

In a k-bounded oclet system Ω, every cut has at most $k \cdot |\mathcal{L}_\Omega|$ conditions where $\mathcal{L}_\Omega$ is the set of labels occurring in Ω. Thus, width and length of every reachable characteristic history of Ω are bounded. This leads to the following two lemmas.

Lemma 7.26: *A k-bounded oclet system Ω has finitely many reachable histories.* ⋆

Proof. The characteristic history $h(.)$ induces the equivalence classes $[\pi]_h := \{\pi' \mid h(\pi) = h(\pi')\}$, for each $\pi = \mathit{Unf}[C]$ for some finite configuration C. All runs in one equivalence class of h reach the same history of Ω. Moreover, for each reachable history of Ω exists one equivalence class of $h(.)$. Thus, Ω has finitely many reachable histories iff $h(.)$ defines finitely many equivalence classes.

Claim: If Ω is k-bounded, then $h(.)$ defines finitely many equivalence classes.

Let *Unf* be k-bounded for some $k \in \mathbb{N}$. Let L be the number of different condition labels used in Ω. Every cut of *Unf* has at most $k \cdot L$ conditions. Let *hist* be the history of some oclet in Ω. A node x_0 of *hist* has *depth* d if there exists a longest sequence of nodes $x_0 < x_1 < x_2 < \ldots < x_d$ s.t. $x_d \in \max \mathit{hist}$. Let D denote the maximal depth of all nodes of all histories in Ω. Further, let P denote the size of the largest pre-set of all nodes in Ω. Every node of *Unf* has at most P pre-nodes.

Let π be some distributed run in *Unf*. The characteristic history $h(\pi)$ contains only nodes which have at most depth D. Thus, each node in $\max h(\pi)$ has at most P^D predecessors. Because $\max h(\pi)$ is a cut of *Unf*, $h(\pi)$ has at most $k \cdot L \cdot P^D$ nodes (usually much less). Because Ω has only finitely many oclets of finite size, there are only finitely many labels in $h(\pi)$. Thus, there are only finitely many non-isomorphic characteristic histories $h(\pi)$. We conclude that $h(.)$ has only finitely many equivalence classes. □

Lemma 7.27 (Finite complete prefix of an oclet system): *Let Ω be an oclet system. If Ω is k-bounded, then Ω has a finite complete prefix β of $\mathit{Unf}(\Omega)$.* ⋆

Proof. A complete prefix β of the unfolding *Unf* of Ω contains (1) one configuration C for each reachable characteristic history *hist* of Ω, and (2) one event e for each oclet o that is enabled at *hist*.

(1) If Ω is k-bounded, then there are only finitely many reachable histories $\mathit{hist}_1, \ldots, \mathit{hist}_n$. By definition, each characteristic history has a finite configuration C_i of *Unf* s.t. $\mathit{hist}_i = h(\mathit{Unf}[C_i])$, for all $i = 1, \ldots, n$. Let E be the union of all these finite configurations. The prefix β_E of *Unf* that consists of all events E is finite and represents each reachable history of *Unf*.

(2) For each configuration C of β_E, extend β_E by all possible extensions at $\mathit{Cut}(C)$. The resulting prefix β contains one occurrence of each oclet which satisfies the second requirement of a complete prefix (Definition 7.24). There are only finitely many possible extensions per configuration C. Thus β is a finite complete prefix of *Unf*. □

7.5. Construct a Finite Complete Prefix of an Oclet System

The preceding section introduced the notion of a complete prefix of an oclet system Ω. A complete prefix β represents the entire behavior of Ω in the shape of a branching process. In some respect, a complete prefix of Ω denotes the state space of Ω. Ω has a *finite* complete prefix β if the distributed runs of Ω are bounded in their width, i.e., every cut of Ω has at most k conditions with the same label.

In this section, we present an algorithm to *construct* a finite complete prefix of Ω if Ω has one. The constructed prefix is the basis for synthesizing a Petri net system that implements an oclet specification. The main result of this section is the following theorem.

Theorem 7.28. *For a k-bounded oclet system Ω, a finite complete prefix of Ω can be constructed algorithmically.* ⋆

To prove Theorem 7.28 we define an algorithm that constructs a finite complete prefix of a given k-bounded oclet system Ω. To this end, we adapt Algorith 7.17 for constructing a complete finite prefix of a Petri net system [88]. Figure 7.22 depicts an oclet system Ω and the complete finite prefix of Ω that is computed by the algorithm.

7.5.1. The idea

We generalize McMillan's algorithm for constructing a finite complete prefix of a Petri net system to oclet systems. In fact, the basic algorithmic idea which was presented in Section 7.3.1 remains the same. Beginning in the initial run π_0 of an oclet system $\Omega = (O, \pi_0)$, extend a prefix β of the unfolding of Ω with one event e at a time. If e reaches a situation that has been reached earlier, then mark e as a *cut-off event*. Do not extend β by any event that depends on a cut-off event. The prefix β is complete when β cannot be extended anymore.

All relevant notions for applying this algorithm to oclet systems have already been generalized in the preceding sections:

- branching processes of oclet systems generalize branching processes of Petri net systems,
- an occurrence of a basic oclet generalizes an occurrence of a transition — either occurrence extends a branching process by one event,
- the reachable characteristic histories of an oclet system generalizes the reachable markings of a Petri net system, and
- the correctness notion of a complete prefix of an oclet system generalizes the corresponding notion of a Petri net system.

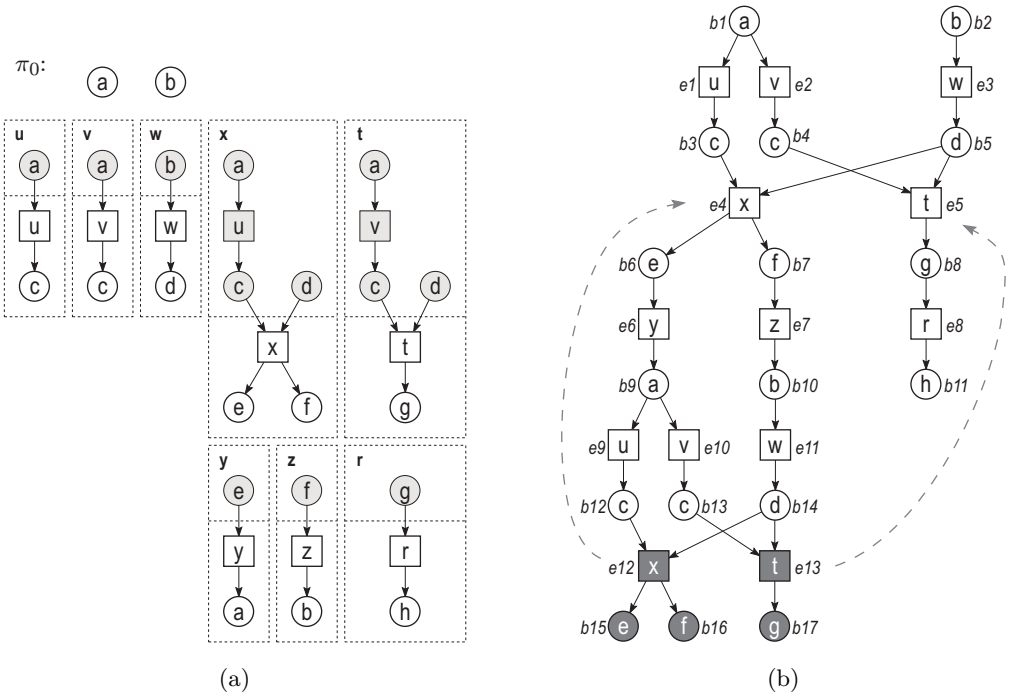


Figure 7.22. An oclet system (a) and its complete finite prefix (b) computed by McMillan's algorithm. The events are added to the prefix in the denoted order. Events e_{12} and e_{13} are cut-off events.

Thus, to generalize McMillan's algorithm to oclet systems we only have to adjust the notion of a *cut-off event*. Let β be an "intermediate" prefix of the algorithm and let e be the event that has just been added to β by the algorithm. The event e induces its *local configuration* $[e]$ which consists of e and all events that precede e. This configuration $[e]$ reaches the *history* $h([e])$ which determines how the system continues after e. If the prefix β contains another event e' that "occurred earlier" and reaches the same history $h([e']) = h([e])$, then the future of event e, i.e., $\Uparrow e \subseteq \Uparrow C$, is already represented in β by $\Uparrow e'$ (by Lemma 7.20). So, we mark e as a *cut-off event* of β and no event depending on e is added to β. Like in McMillan's algorithm, eventually every possible extension of β depends on a cut-off event and the algorithm terminates with a complete prefix of Ω.

Whether an existing event e' "occurred earlier" than the new event e follows from an *adequate order* — a special partial order over the configurations of β; see Definition 7.16. A simple adequate order defines that e' "occurred earlier" than e if $[e']$ contains less events than $[e]$. The example in Figure 7.22 was computed using this notion. Event e_{12} is a cut-off event because of event e_4 whereas event e_{13} is a cut-off event because of event e_5.

In the next section, we formally define a cut-off event for oclet systems and present the algorithm that computes a finite complete prefix of an oclet system. Afterwards, we prove the correctness of the algorithm and discuss its complexity.

7.5.2. The definitions and proofs

In the following let Ω be an oclet system and let β be a branching process of Ω. Any configuration C of β induces the distributed run $\beta[C]$ that has the characteristic history $h(C) := h(\beta[C])$. Two h-equivalent configurations have isomorphic futures in the unfolding of Ω (by Lemma 7.20).

Each event e of β induces the local configuration $[e]$ consisting of e and all events that precede e; see Def. 7.15. We say that *event e reaches the characteristic history* $h(e) := h([e])$. Thus, two events e_1 and e_2 reaching equivalent histories $h(e_1) = h(e_2)$ have isomorphic futures $\Uparrow[e_1]$ and $\Uparrow[e_2]$ in the unfolding of Ω.

The construction algorithm may stop extending a branching process β of Ω beyond an event e_1 if β already contains an equivalent event e_2 with $h(e_1) = h(e_2)$. In this case e_1 is a *cut-off event* of β — but only if e_2 "occurred earlier" than e_1. The second condition is formalized by an *adequate order*, i.e., a partial order over the configurations of β as defined in Definition 7.16. Two simple adequate orders are the subset-relation, i.e., $C_1 \subset C_2$, and the size of the configurations, i.e., $|C_1| < |C_2|$.

A *cut-off event* is defined wrt. the notion of an adequate order.

Definition 7.29 (Cut-off event). Let $\prec$ be an adequate order on the configurations of the unfolding of an oclet system. Let β be a prefix of the unfolding containing an event e. The event e is a *cut-off event* of β (wrt. $\prec$) iff β contains a local configuration $[e']$ s.t.

1. $h(e)$ and $h(e')$ are equivalent, and
2. $[e'] \prec [e]$. ⌟

The algorithm that computes a finite complete prefix of the oclet system $\Omega = (O, \pi_0)$ begins the construction with the initial run $\beta := \pi_0$. An oclet $o \in O$ is a *possible extension* of β if β contains a cut B that ends with the history of o and does not already have $event(o)$ as post-event; see Definition 7.8. Let $PE(\beta)$ denote the set of all possible extensions of β. Extending β by a possible extension o appends $event(o)$ and its post-set to β.

After each extension, the algorithm determines whether $event(o)$ is a cut-off event wrt. the chosen adequate order. A possible extension o is not added to β if the local configuration $[event(o)]$ in β contains a cut-off event. The algorithm terminates when no event can be added. The following algorithm directly generalizes Algorithm 7.17; the only differences are the notion of a possible extension (now basic oclets) and of a cut-off event (now considering characteristic histories).

Algorithm 7.30 (Finite complete prefix of an oclet system).

input: A k-bounded oclet system $\Omega = (O, \pi_0)$.

output: A complete finite prefix Fin of $Unf(\Omega)$.
begin
  $Fin := \pi_0$;
  $pe := PE(Fin)$;
  $cut\text{-}off := \emptyset$;
  while $pe \neq \emptyset$ **do**
    choose a possible extension o in pe s.t. $[event(o)]$ is minimal wrt. $\prec$;
    if $[event(o)] \cap cut\text{-}off = \emptyset$ **then**
      $Fin := Fin \rhd o$;
      $pe := PE(Fin)$;
      if $event(o)$ is a cut-off event of Fin **then**
        $cut\text{-}off := cut\text{-}off \cup \{event(o)\}$
      endif
    else
      $pe := pe \setminus \{o\}$;
    endif
  endwhile
  end ⌟

Algorithm 7.30 returns a complete finite prefix Fin of a k-bounded oclet system. Thus, Theorem 7.28 holds.

The proof that Fin is indeed finite and complete follows directly from the corresponding proof in [38] (Propositions 4.8 and 4.9). We only sketch the main arguments. The finiteness of Fin follows from Ω being k-bounded. Thus, the algorithm will see only finitely many local configurations $[event(o)]$ that have a characteristic history that is not in Fin yet. Thus, after finitely many extension steps, every other possible extension is declared as a cut-off. The completeness of Fin holds as follows:

1. Any two local configurations $[e_1]$ and $[e_2]$ with h-equivalent histories have isomorphic futures $\Uparrow[e_1]$ and $\Uparrow[e_2]$ with an isomorphism α from $\Uparrow[e_1]$ to $\Uparrow[e_2]$.
2. Then the arguments of [38] apply as follows. Every configuration C_1 of the unfolding can be split into a local configuration $[e_1]$ of a cut-off event e_1 and an extension E, $C_1 = [e_1] \cup E$. Then Fin contains a $\prec$-smaller local configuration $[e_2]$ that is h-equivalent to $[e_1]$. Thus, $C_2 := [e_2] \cup \alpha(E)$ is a configuration that is equivalent to C_1 and smaller. By iteration, there exists a $\prec$-least configuration C_k that is h-equivalent to C_1 and completely contained in Fin.
3. Every oclet that is enabled at $Cut(C_1)$ leads to a configuration $C_1 \cup \{e\}$. By the same argument as above, $C_1 \cup \{e\}$ is represented in Fin.

Minimal complete prefixes. Altogether, Algorithm 7.30 returns a finite complete prefix Fin of a k-bounded oclet system Ω. But Fin is not necessarily minimal, i.e., Fin may have a prefix that is complete. The algorithm returns a non-minimal prefix whenever a minimal prefix follows only from a "cut-off configuration" instead

of a "cut-off event." In the example of Figure 7.22 on page 222, a minimal prefix consists of the events $\{e_1, \ldots, e_8\}$. This minimal prefix cuts off all events after the configuration $[e_7] \cup [e_8]$. But the algorithm only considers $[e_7]$ and $[e_8]$ individually and continues until the cut-off events e_{12} and e_{13} are identified.

Algorithm 7.30 can be improved. The algorithm actually defines a family of procedures for constructing a complete finite prefix — depending on the chosen adequate order $\prec$. Esparza et al. discuss in [38] how the result of Algorithm 7.30 can be improved by choosing a "better" adequate order $\prec$.

7.5.3. Complexity of constructing a complete finite prefix

This section discusses the complexity of constructing a finite complete prefix of an oclet system. We adapt most complexity arguments from [38] and provide additional arguments that are specific to the generalization to oclet systems.

Let $\Omega = (O, \pi_0)$ be a 1-bounded oclet system, i.e., every reachable cut of Ω contains no two conditions with the same label. Let

- R be the number of reachable histories of Ω,
- E_Ω be the set of all events in all oclets in O,
- ξ be the size of the largest pre-set or post-set of the contributed events in O,
- $o_{\max}$ be the largest oclet in O

A complete finite prefix β of Ω can be constructed in time $\mathcal{O}(|O| \cdot R^{\xi+1} \cdot |E_\Omega| \cdot |o_{\max}|^2)$. The prefix β has a total of $\mathcal{O}(\xi \cdot R)$ conditions and non-cut-off events, and at most $\mathcal{O}(|O| \cdot R^\xi)$ cut-off events.

But in experiments, typically $\mathcal{O}(R)$ cut-off events are observed only. Moreover, in many practical systems, ξ is known to be small, usually having values 2 or 3 [36].

Most arguments for the running time and space required by Algorithm 7.30 have been provided by Esparza et al. for the original Algorithm 7.17 on Petri nets in [38]. We repeat their arguments wrt. the new algorithm for the sake of completeness. The dominating factor of the algorithm is the computation of the possible extensions. Deciding whether a branching process of an oclet system has a possible extension is $\mathcal{NP}$-complete [36]. In the following, the individual steps of the algorithm are considered in more detail.

We consider the prefix *Fin* returned by Algorithm 7.30. Let B be the set of conditions of *Fin* that do not depend on a cut-off event. In each iteration, the algorithm computes the possible extensions of *Fin*. To check whether an oclet o is an extension, the algorithm has to examine in the worst case all subsets B' of B s.t. B' equals the end of o's history, i.e., the pre-set of the contributed event e of o. Each B' could end with o's history, which means that o was a possible extension. If k is the size of e's pre-set, the algorithm may have to examine $\left(\frac{|B|}{k}\right)^k$ subsets B' of B to find all possible extensions due to oclet o [38].

Let ξ denote the size of the largest pre-set or post-set of the contributed events in O. If $B \gg \xi$, then $\left(\frac{|B|}{k}\right)^k \leq \left(\frac{|B|}{\xi}\right)^\xi$. So, the algorithm examines a total of

$\mathcal{O}(|O| \cdot \left(\frac{|B|}{\xi}\right)^{\xi})$ subsets of B to find all possible extensions due to all oclets O. For each subset B', the algorithm has to check whether B' ends with the oclet's history.

Assume that *Fin* was constructed using a total adequate order on the configurations of *Fin*. Then in each iteration, each new event e is compared to all other events *Fin* to determine whether e is a cut-off event. Thus, in the case of a 1-bounded oclet system and a total adequate order, *Fin* has at most R non-cut-off events, where R is the number of reachable histories. Each event contributes at most ξ post-conditions. So, $|B| \leq R \cdot \xi + |\pi_0|$ and under the natural assumption $|B| \geq |\pi_0|$ the algorithm checks at most $\mathcal{O}(|O| \cdot R^{\xi})$ subsets to construct *Fin* [38].

The notion of a history in oclets requires the following additional steps during the construction of *Fin*:

1. check whether a subset B' of the conditions B ends with the history of an oclet o (then o is a possible extension), and
2. check whether an added event is a cut-off event by comparing it to the characteristic histories of all other $\prec$-smaller events in *Fin*.

According to Lemma 6.19, checking 1. takes a breadth-first search in the size of the oclet's history for 1-bounded oclet systems. Thus, the check is bounded by the size of the largest oclet $o_{\max}$. So, the algorithm takes at most $\mathcal{O}(|O| \cdot R^{\xi} \cdot |o_{\max}|)$ steps to find and add all possible extensions.

To identify whether an added event e is a cut-off event, the algorithm compares in the worst case for every other event e' of *Fin* whether $h(e) = h(e')$. The check whether $h(e) = h(e')$ is equivalent to: the partial history *hist* occurs at the end of the local configuration $[e]$ iff *hist* occurs at the end of $[e']$, for the local history *hist* of every event f in O. This requires two breadth-first searches in the size of f's local history which is bounded by $|o_{\max}|$. So, the entire check whether $h(e) = h(e')$ takes $\mathcal{O}(|E_\Omega| \cdot |o_{\max}|)$ steps, where E_Ω is the set of all events in the oclets O. As the prefix contains at most R non-cut-off events e', checking whether e is a cut-off event requires $\mathcal{O}(R \cdot |E_\Omega| \cdot |o_{\max}|)$ steps.

Altogether, the algorithm terminates in $\mathcal{O}(|O| \cdot R^{\xi+1} \cdot |E_\Omega| \cdot |o_{\max}|^2)$ steps. The constructed prefix *Fin* without cut-off events has at most $\mathcal{O}(R \cdot \xi)$ events and conditions (assuming a total adequate order and a 1-bounded oclet system). The trivial upper bound for cut-off events of *Fin* is $\mathcal{O}(|O| \cdot R^{\xi})$.

7.6. Analyzing Oclet Specifications

The preceding sections introduced a technique to construct a *finite representation* of the play-out behavior of an oclet specification O. First translate O into its canonical oclet system Ω. The finite complete prefix *Fin* returned by Algorithm 7.30 *Fin* can be seen as a symbolic representation of the *state-space* of O. It contains all behavioral information to reconstruct every distributed run of O. So, algorithms on *Fin* may *decide behavioral properties* of O, if O is k-bounded for some $k \in \mathbb{N}$.

Providing full-fledged verification techniques for oclet specifications like model checking is a research topic on its and not considered in this thesis. In this section,

we show by three examples how slight modifications of Algorithm 7.30 allow to analyze non-trivial behavioral properties of an *oclet specification* O.

7.6.1. Deciding k-boundedness

Section 7.4.3 introduced the notion of a k-bounded oclet specification O where k is a natural number. The play-out behavior of a k-bounded oclet specification O contains only runs π in which each cut contains at most k conditions with the same label. For example, any message buffer specified by O contains at most k messages.

Whether O is k-bounded or not depends on *all runs* in the play-out behavior $\hat{R}(O)$ of O, that is, k-boundedness is a behavioral property. Whether O is k-bounded for a given natural number k can be decided by the following modification of Algorithm 7.30.

> After extending *Fin* with a new event e, compute for each condition b its co-set of conditions $co(b) := \{b' \mid b \mid\mid b', \ell(b) = \ell(b')\}$ with the same label. If $|co(b)| \geq k$, then O is not k-bounded; in this case abort the construction of *Fin*. (7.2)

If the modified algorithm terminates, then O is k-bounded. Deciding whether there exists some number k s.t. O is k-bounded is a harder problem. An unfolding-based solution for Petri nets is described by Desel et al. [29].

7.6.2. Detecting local deadlocks

The arguably most analyzed behavioral property of distributed systems is whether a system is *deadlock free*. A *global deadlock* is a reachable state of the system at which no action is enabled at all. In the following we want to detect when a component can deadlock *locally*.

Definition 7.31 (Local deadlock). Let O be an oclet specification and $\pi \in \hat{R}(O)$ be a distributed run of O. A condition $b \in \max \pi$ is a *local deadlock* iff $b \in \max \rho$, for every continuation $\rho \in \hat{R}(O)$ of π. The specification O is *deadlock free* iff no run of O contains a local deadlock. ⌟

In other words, a local deadlock is a local state b that is never left. Detecting deadlocks in a finite complete prefix *Fin* is $\mathcal{NP}$-complete by a reduction to 3-SAT [88].

For a practically relevant class of systems, local deadlocks can be detected in polynomial time on *Fin*. In this class of systems a choice between two alternative actions depends only on these two actions and no other side condition; these systems are called *free choice* [28]. For any basic oclet o, let $event(o)$ denote its contributed event and let $\ell({}^\bullet event(o))$ denote the labels of its pre-conditions.

Definition 7.32 (Free choice specification). An oclet specification O is *free choice* iff for any two basic oclets $[o_1], [o_2] \in \hat{O}$: if $\ell({}^\bullet event(o_1)) \cap \ell({}^\bullet event(o_2)) \neq \emptyset$, then ${}^\bullet event(o_1)$ and ${}^\bullet event(o_2)$ are isomorphic, and $hist_1$ ends with $hist_2$. ⌟

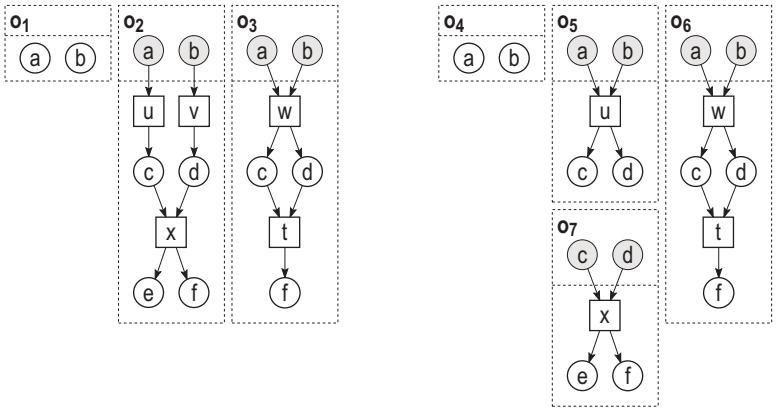


Figure 7.23. The specification $\mathsf{O_n} = \{\mathsf{o_1}, \mathsf{o_2}, \mathsf{o_3}\}$ is not free choice, the specification $\mathsf{O_f} = \{\mathsf{o_4}, \mathsf{o_5}, \mathsf{o_6}, \mathsf{o_7}\}$ is free choice.

This definition generalizes free choice Petri nets [28]. For example, the specification $\mathsf{O_n}$ in Figure 7.23 is *not* free choice. Events u and w share pre-condition a, but an occurrence of w also depends on b. Further, events x and t have the same pre-set, but their enabling depends on different histories. The specification $\mathsf{O_f}$ is free choice. Events u and w have the same pre-set. Further, the history of t ends with the history of x, so, x is enabled whenever t is.

The local deadlocks of a free choice oclet specification O can be detected efficiently on a finite complete prefix *Fin* of O.

Lemma 7.33: *Let O be a free choice oclet specification and let Fin be a complete prefix of O. A condition b of Fin is a local deadlock iff one of the following properties holds*

1. *$b^\bullet = \emptyset$ and b does not depend on a cut-off event,*
2. *there exists a post-event $e \in b^\bullet$, a pre-condition $b' \in {}^\bullet e$, and a condition c of Fin s.t. c is concurrent to b and in conflict with b'.* ⋆

Proof. The first property of Lemma 7.33 is obvious: b never gets a post-event under any circumstances, so the local state b is never left. The second property considers the case where b does have a post-event e, but e may be prevented from occurring. This is the case when a run π reaches b together with a concurrent condition c where c prohibits to reach all pre-conditions of e, i.e., some $b' \in {}^\bullet e$ is in conflict with c. This check is complete: let there be another event $f \in b^\bullet$ which could allow to leave b in π. But f and e share the pre-condition b. From O being free-choice follows that ${}^\bullet e = {}^\bullet f$. So, also f is prevent from occurring because of $b' \in {}^\bullet f$. □

An algorithm that checks whether O has a local deadlock first computes a finite complete prefix *Fin* of O using Algorithm 7.30, and then searches for a condition b of *Fin* satisfying one of the properties of Lemma 7.33.

7.6.3. Deciding soundness of process models

The third behavioral property which we consider in this section is *soundness*. The notion of soundness particularly applies to the domain of *process modeling*. A process describes *terminating behaviors*, i.e., behaviors that start in a well-defined *initial situation* and reach a well-defined *final situation* like a goal. Consequently, a *process model* is a system model that additionally describes a set of *final states*.

To model a process with scenarios, we equip an oclet specification O with a specification of final states. To this end, we distinguish a set of names $\mathcal{L}_{final}$ of final states. A distributed run $\pi \in \hat{R}(O)$ in the play-out behavior of O *reaches a final state* iff π is finite and the maximal conditions $\max \pi$ are labeled with names in $\mathcal{L}_{final}$, i.e., $\forall b \in \max \pi : \ell(b) \in \mathcal{L}_{final}$.

Soundness is a minimal behavioral requirement to any process model. Informally, a process is *sound* if (1) each run of the process can be continued to reach a final state, (2) when reaching a final state, the process exhibits no further behavior, and (3) for every action of the process exist a run in which this action occurs [124]. For free choice processes, soundness simplifies to the following conditions.

Definition 7.34 (Soundness of free choice specifications). A free choice oclet specification O with names $\mathcal{L}_{final}$ of final states is *sound* iff

1. O has a local deadlock,
2. each local deadlock b of O is part of a final state, i.e., $\ell(b) \in \mathcal{L}_{final}$, and
3. O is 1-bounded. ⌟

This notion of soundness for free choice specifications generalizes the corresponding notion of soundness free choice Petri nets [123]. Soundness of a free choice oclet specification O can be decided on its finite complete prefix using Lemma 7.33 and procedure (7.2). The next section reports on a case study in which we checked soundness of industrial business process models.

Boundedness, deadlock freedom, and soundness are only basic behavioral properties. Verifying more advanced behavioral properties, for instance expressed in temporal logics, demands further adaptations of Algorithm 7.30. Several algorithms to verify behavioral properties of Petri nets using finite complete prefixes are presented in [36].

7.7. Practical Results

This section reports on experimental results with our algorithm for constructing a complete finite prefix of an oclet system that was presented in Section 7.5.

7.7.1. Tool support

We implemented Algorithm 7.30 as a plug-in of our prototype tool Greta. A system designer who uses Greta may call the algorithm at any stage during modeling. The computed prefix is displayed graphically and allows the system designer to detect flaws in the current model.

Some implementation details. The implementation uses specific data-structures to efficiently represent an oclet specification and the finite complete prefix during computation. The principle data structure is described in [36]; we adapted it to the setting of oclets with their histories.

The computationally most expensive step in Algorithm 7.30 is to determine whether a set of conditions is a co-set. Our implementation explicitly stores the concurrency relation of the prefix as a hash table. With this design choice, we spend more memory for storing the prefix but gain computation speed and have the concurrency information readily available. Moreover, we implemented several heuristics that reduce the number of sets of conditions that need to be checked for being pair-wise concurrent when computing the possible extensions in Algorithm 7.30. Our implementation determines cut-off events using the lexicographic adequate order of [38] that was presented in Section 7.3.2.

The explicit concurrency relation helps in analyzing the behavior of an oclet specification. For instance, we can easily determine whether an oclet specification O is k-bounded. On the finite prefix Fin of O, we simply apply procedure (7.2) on each condition that has been added in a step. By the hash table, (7.2) takes constant time. So, k-boundedness is checked on the fly during the construction of Fin.

Some numbers. In the following we present some academic and industrial benchmark results regarding the performance of our implementation of Algorithm 7.30. In all practical examples, the algorithm returns a finite complete prefix in linear or polynomial time. The size of the computed prefix is also linear or polynomial in the size of the oclet specification.

Our example are the Dining Philosophers in two variants and the process model of the "Taskforce Earthquakes" from page 6. Oclet phil_2 of Figure 7.24 specifies one normal cycle of the second dining philosopher; for all other philosophers exists a corresponding oclet. In the second variant the philosophers behave *decently*: after eating, each philosopher waits until both his neighbors have eaten [104]. Oclet $\mathsf{phil}_{2\mathsf{g}}$ denotes that the second philosopher initially may take his forks greedily (he has has not eaten yet). But once he has eaten, he picks up the forks only decently after his left and right neighbor returned their forks as specified by $\mathsf{phil}_{2\mathsf{d}}$.

Table 7.1 lists the results. The "model" columns describe the *size of the contributions* of all oclets: $|T|$ denotes the number of contributed events, $|P|$ the number of contributed conditions, and $|F|$ the number of arcs occurring in a contribution. The number of oclets is irrelevant as these are decomposed into their basic oclets which follows from all contributed events. The "complete prefix" columns describe the size of the computed prefix ($|E|$, $|B|$, and $|F|$ for events, conditions, and arcs, respectively), the number $|E_{\text{cutoff}}|$ of cut-off events, and finally the running time of the algorithm in seconds.

7.7.2. Verifying soundness of industrial business process models

We applied our implementation in an industrial case study for verifying soundness of business process models using the algorithm described in Section 7.6.3. In total,

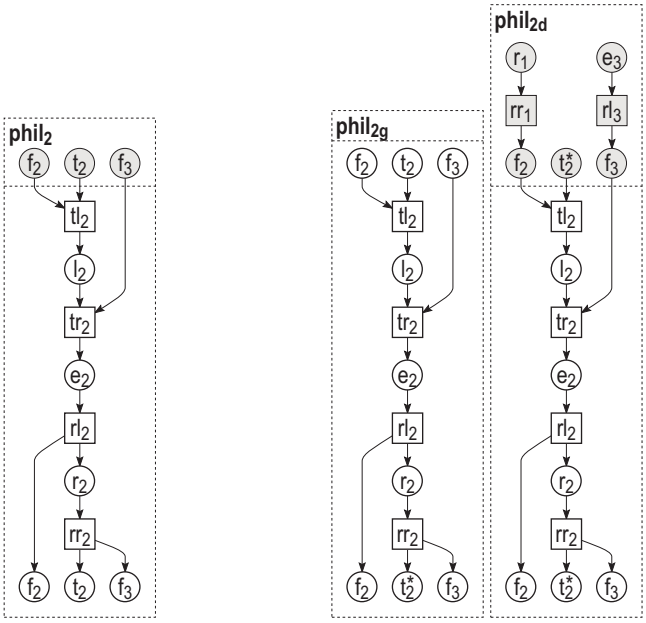


Figure 7.24. Oclet phil_2 describes one cycle of the second dining philosopher. Oclets phil_{2g} and phil_{2d} describe decent behavior of the second philosopher.

	model			complete prefix				
model	$\|P\|$	$\|T\|$	$\|F\|$	$\|E\|$	$\|B\|$	$\|F\|$	$\|E_{\text{cutoff}}\|$	time [secs]
Dining Philosophers (5)	30	20	60	65	115	195	20	0.062
Dining Philosophers (10)	60	40	120	280	480	840	90	0.219
Dining Philosophers (20)	120	80	240	1160	1960	3480	380	6.109
Dining Philosophers (30)	180	120	360	2640	4440	7920	870	54.000
Dining Philosophers (40)	240	160	480	4720	7920	14160	1560	261.703
Decent Philosophers (5)	60	40	120	510	800	1530	35	0.297
Decent Philosophers (7)	84	56	168	1722	2688	5166	112	1.992
Decent Philosophers (10)	120	80	240	6670	10350	20010	370	23.672
Decent Philosophers (14)	168	112	336	24906	38402	74718	1106	335.594
TaskForce Earthquakes	80	35	111	93	152	320	33	0.141

Table 7.1. Running time and size of complete prefixes for several benchmark processes.

we analyzed models from 5 process libraries of in total 735 processes. The models covered various industry domains such as financial services, automotive, telecommunications, construction, supply chain, health care, and customer relationship management. The same case study has been conducted using the explicit Petri net model checker LoLA, the soundness analysis tool Woflan, and a domain-specific structural analysis technique [40].

The process data. All process models in all five libraries were modeled in the IBM WebSphere Business Process Modeler using basic control-flow constructs. Each

	A	B1	B2	B3	C
Avg. / max. number of places	47 / 171	44 / 213	47 / 263	51 / 273	53 / 262
Avg. / max. number of trans.	30 / 104	39 / 145	30 / 144	34 / 179	55 / 284

Table 7.2. Structural size of the process libraries.

	A	B1	B2	B3	C
Processes in library	282	288	363	421	32
sound	152	107	161	207	15
unsound	130	181	202	214	17
Avg. / max. concurrency	2 / 13	8 / 14	16 / 66	14 / 33	2 / 4
Processes with >1000000 states	26	19	29	38	7
Processes with >1000000 states (only sound)	0	1	4	4	0
Avg. number of states (only sound, <1000000 states)	26	71	322	4911	680
Max. number of states (only sound, <1000000 states)	213	2363	28641	588507	8370

Table 7.3. Behavioral information about the process libraries.

process model was translated into a *free choice* Petri net system Σ; only places of Σ that have no post-transition may be marked in the final state of the process [40]. Figure 7.25 (left) shows a typical Petri net model of a process from the libraries.

For this experiment, we decomposed each Petri net system $\Sigma = (N, m_0)$ into a free choice oclet specification O using a simple procedure. Beginning in some transition t of N, repeatedly mark adjacent transitions and places as long as the marked sub-net π of N is a causal net (Def. 2.6). If π cannot be extended anymore, remove π from N — except for those places of π that still have a pre- or post-transition that is not in π. π is causal net which induces the oclet $o = (\pi, \min \pi)$ that is added to O. The events and conditions of π are labeled with their originating transitions and places.

The initial marking m_0 translates into the unique ε-oclet of O; see Section 5.5. The places of N that have no post-transition constitute the names $\mathcal{L}_{final}$ of the final state of O as introduced in Section 7.6.3. By construction, the oclet specification O exhibits the same behavior as Σ. The oclet specification on the right of Figure 7.25 corresponds to the Petri net on the left.

Table 7.2 displays some structural information about the process libraries. Table 7.3 summarizes some results about their behavior, specifically regarding the size of the state spaces of the models. The fourth line in Table 7.3 describes the degree of concurrency in the processes, i.e., how many transitions are enabled concurrently.

Verifying soundness with finite complete prefixes. According to Definition 7.34, a free choice oclet specification O is sound iff each local deadlock of O is a final state, O does reach a local deadlock, and O is 1-bounded.

For the experiment, we extended the implementation of Algorithm 7.30 in Greta. During the construction of *Fin*, the algorithm checks for 1-boundedness of O on the fly using procedure (7.2). If 1-boundedness is violated, then Greta terminates with a counter-example trace to the violating state. If the construction

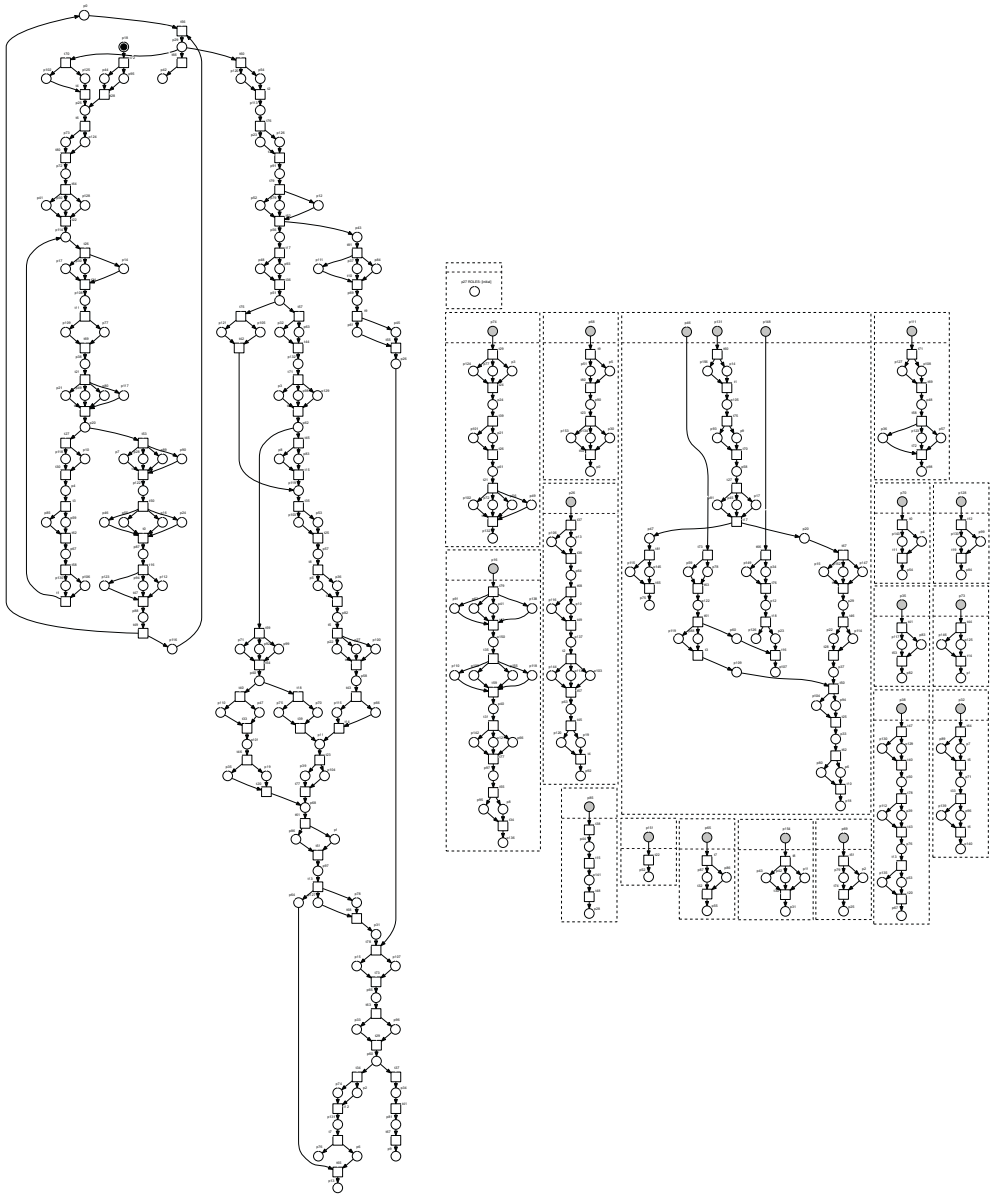


Figure 7.25. A process of average size as a Petri net (left) and its equivalent oclet specification (right).

	A	B1	B2	B3	C
size of prefix (avg./max.)					
events	34 / 189	20 / 125	19 / 144	24 / 325	115 / 1227
cut-off ev.	1 / 10	0 / 7	0 / 6	0 / 18	7 / 93
conditions	57 / 334	31 / 186	31 / 233	38 / 483	120 / 1238
arcs	108 / 644	59 / 357	59 / 357	74 / 960	235 / 2464
time [ms]					
avg.	7	6	6	10	102
max.	94	78	219	422	1703

Table 7.4. Experimental results.

of *Fin* succeeds without error, GRETA checks for local deadlocks in *Fin* according to Lemma 7.33. Again, if *Fin* contains a local deadlock that is not a final state, then GRETA returns a counter-example trace to deadlock.

Experimental setup. To run the experiment, we invoked GRETA's soundness verification algorithm for each process in each library. Besides the analysis result, we measured the time that was needed to decide soundness and the size of the constructed prefix, for each process. The experiment was conducted on a standard notebook with a 1.66 GHz processor and 2 GB RAM.

Experimental results. GRETA could verify soundness of all processes in the data set. Figure 7.26 and Table 7.4 summarize our experimental results which we discuss in the following.

Figure 7.26 relates for all processes in the data set, the size of the computed finite prefix (here, the number of events) to the analysis time required to construct the prefix. For a large number of processes, soundness could be verified instantaneously or within a few milliseconds. Only a one process from library C required analysis times of about 1.7 seconds. Table 7.4 presents some details for each library separately.

Our analysis succeeded: we could verify all process models with GRETA. The same experiment had been conducted earlier using the explicit model checker LoLA, the soundness checker Woflan, and a domain specific analysis technique [40]. That experiment aimed at instantaneous verification of business processes: analysis times of more than 5 seconds were considered *intractable*. GRETA provides tractable analysis for all processes (see Fig. 7.26).

Similar results had been obtained in the original experiment. GRETA could verify soundness of the processes within 6 to 10 milliseconds in average for libraries A and B, and within 102 milliseconds in average for library C which corresponds to the analysis times reported in [40]. The original experiment reported a number of intractable processes for each analysis tool in first place. There, analysis only became tractable by the help of additional techniques. Specifically, the model checker LoLA gained performance by the use of *partial order reduction* which has basically the same effect as finite complete prefixes: it avoids to explore all linearizations of a partially ordered run when this is not necessary for the property to be verified.

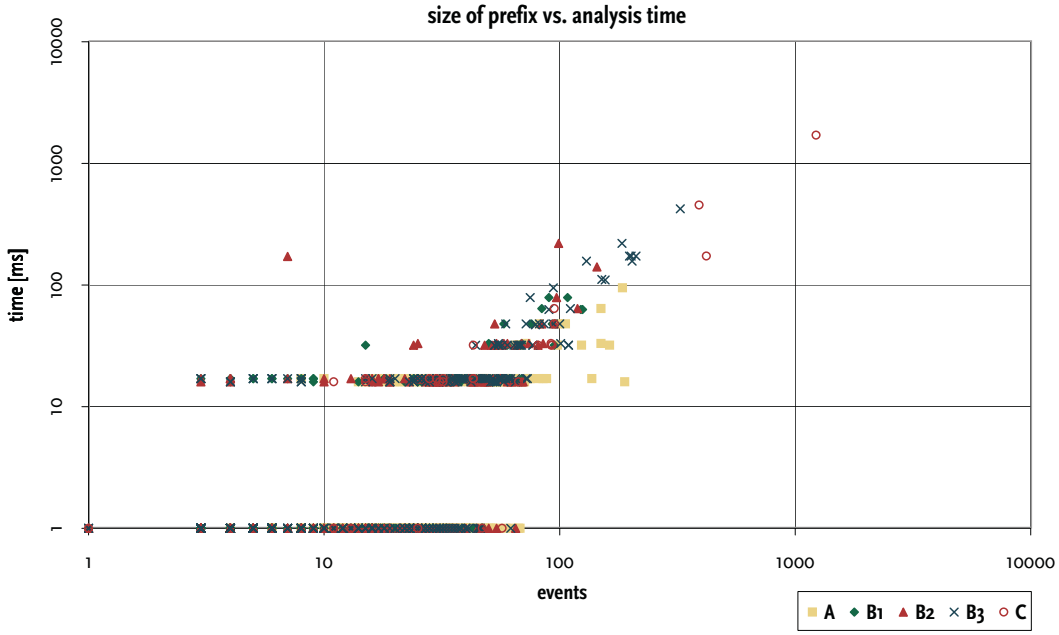
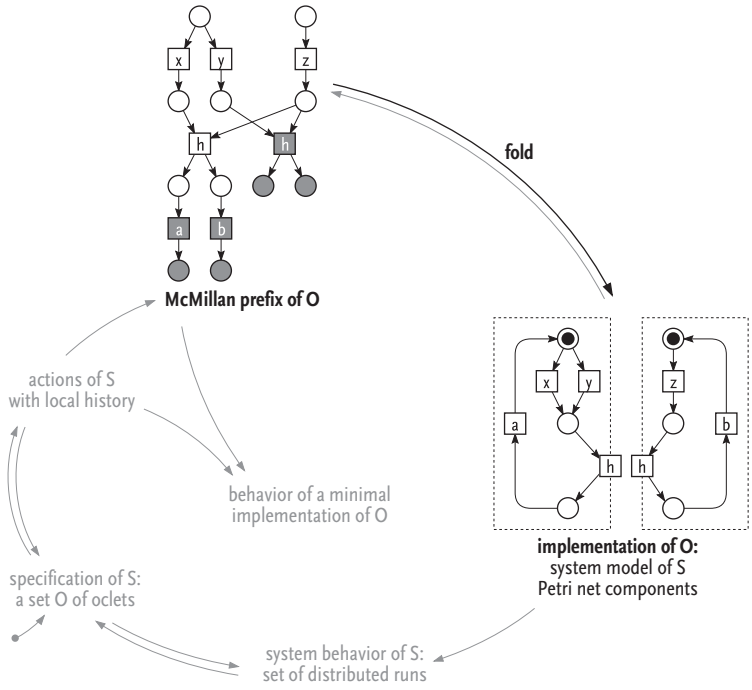


Figure 7.26. Size of computed prefixes vs. analysis times.

Greta exhibits slightly lower performance in library C compared to the analysis times reported in [40] on one unsound processes. The process has a rather large state-space and a local deadlock. Greta can detect this deadlock only after the finite complete prefix has been constructed whereas the techniques in [40] may detect a local deadlock during construction and terminate analysis as soon as it is detected.

Altogether, this experiment demonstrated how to apply finite complete prefixes for analyzing behavioral properties of oclet specifications. Our results showed that this analysis technique is feasible for analyzing process models in an industrial context.

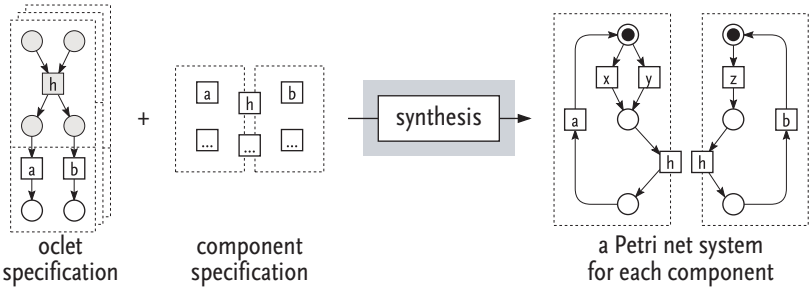
8. Synthesizing an Implementation from a Specification



This chapter defines an algorithm for automatically synthesizing an implementation from a scenario-based specification. The synthesized implementation is a model of a distributed system consisting of several components that interact as denoted in the scenarios.

8.1. The Synthesis Problem

Chapter 4 introduced *oclets* as a formal model for scenario-based specifications. Each oclet describes a scenario as a course of actions which depends on a *history*. An *oclet specification* O is a set of oclets. A system designer can use oclets to specify desired interactions among several components of a distributed system: by *additionally* associating each action in O to one or more components, each oclet describes "one story involving many components." An *implementation* of O consists of the described components so that each component implements "all stories of the same component." The *synthesis problem* is to automatically construct a distributed system that consists of the specified components, and exhibits the behavior described in O and as few additional behavior as possible.



Chapter 6 defined for a given oclet specification O the least behavior $\hat{R}(O)$ that is exhibited by any *minimal implementation* of O, see Definition 6.14. With this notion, the *general synthesis problem* reads as follows:

> Define an algorithm Syn that returns for every oclet specification O a finite Petri net system Σ s.t. $R(\Sigma) = \hat{R}(O)$. (8.1)

Hardness of the synthesis problem

The general synthesis problem (8.1) has no solution. This negative result follows from the stronger expressive power of oclets compared to Petri nets. For a (labeled) Petri net system Σ, the problem whether a specific marking m is reachable in Σ is decidable [37]. The corresponding problem for oclet specifications is undecidable.

Theorem 8.1 (Reachability is undecidable). *Let O be an oclet specification. Let M be a set of conditions labeled with names occurring in O (corresponding to a marking of a Petri net system). The problem "Is there a run $\pi \in \hat{R}(O)$ containing a cut that is isomorphic to M?" is undecidable.* ⋆

Proof. The undecidability of oclets follows from a reduction to *Post's Correspondence Problem* (PCP) [100]. It can be shown that every PCP instance P can be translated to an oclet specification O s.t. P has a solution iff O has a run that contains a specific cut M. Because PCP is undecidable, also the reachability of M is undecidable. Some more details on the proof are given in Appendix A.9. □

If there was a synthesis algorithm Syn that returned for every oclet specification O an equivalent Petri net system Σ, then we could decide reachability of M using Σ. This contradicts the preceding theorem.

Oclets are in good company with this negative result. The corresponding problem of deciding whether a given LSC specification can be implemented as a distributed system with pre-specified components is undecidable [18] — also by a reduction to PCP. Deciding whether an HMSC specification can be synthesized to a Petri net is likewise undecidable [26]. The difficulty of the synthesis problem seems to arise from the relation between specification and satisfying behavior. Alur and Yannakakis [7] have proved undecidability of a simple model-checking problem over MSC graphs with asynchronous composition. A corresponding problem for Template MSCs (MSCs with history) is undecidable as well [45].

The bounded synthesis problem

In the light of these results, only a restricted synthesis problem can be solved. An oclet specification O is *k-bounded* for a natural number k if every reachable cut (i.e., a cut of a run in $\hat{R}(O)$) has at most k conditions with the same label. O is *bounded* if O is k-bounded for some natural number k. The *bounded synthesis problem* reads as follows:

> Define an algorithm Syn that returns for every bounded oclet specification O a finite Petri net system Σ s.t. $R(\Sigma) = \hat{R}(O)$. (8.2)

A subsequent problem is to synthesize not a single system, but a distributed system consisting of components $\Sigma_1, \ldots, \Sigma_r$ that together yield the system $\Sigma_1 \oplus \ldots \oplus \Sigma_r$. Existing scenario-based techniques such as MSCs assign each action of a scenario to a specific component [65]. Let $\mathcal{L}$ be the names of actions and states occurring in an oclet specification O. A *component specification* of O is a family $(K_i)_{i=1}^r$ of names s.t. $\bigcup_{i=1}^r K_i = \mathcal{L}$. The *component synthesis problem* reads as follows:

> Define an algorithm Syn that returns for every bounded oclet specification O and every component specification $(K_i)_{i=1}^r$ of O a family of Petri net systems $(\Sigma_i)_{i=1}^r$ s.t. $R(\Sigma_1 \oplus \ldots \oplus \Sigma_r) = \hat{R}(O)$ and Σ_i contains only labels from K_i, for all $i = 1, \ldots, r$. (8.3)

The bounded synthesis problem is a special case of the component synthesis problem with a single component $K_1 = \mathcal{L}$.

The next steps

The main result of this chapter is an algorithm that solves the component synthesis problem. Figure 8.1 sketches our approach.

The preceding Chapter 7 adapted McMillan's technique of complete finite prefixes to oclets. A complete finite prefix *Fin* of an oclet specification O represents the behavior $\hat{R}(O)$ of a minimal implementation of O in a finite structure. We defined an algorithm to construct *Fin* for any bounded oclet specification O.

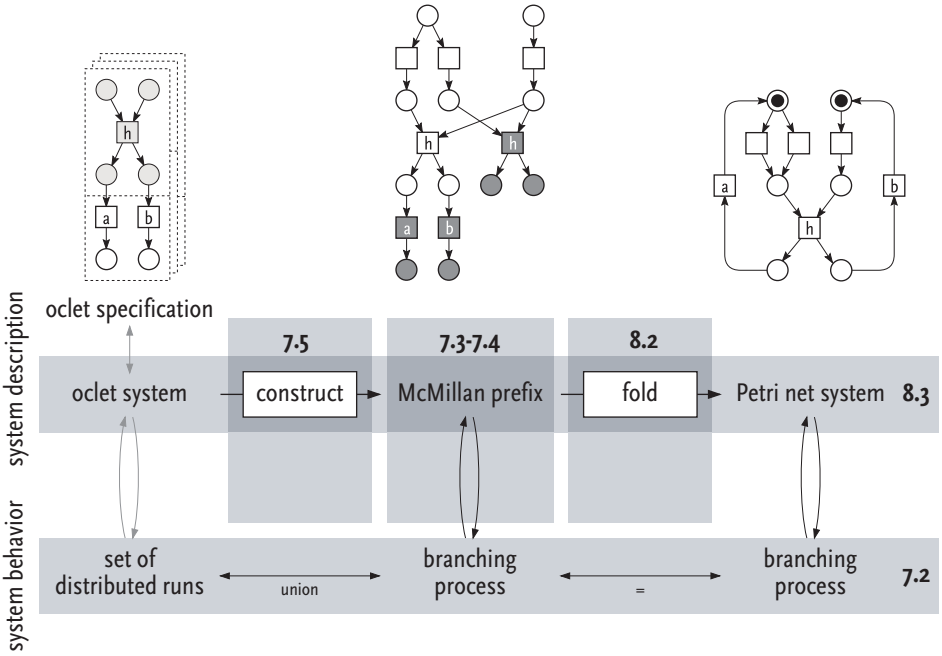


Figure 8.1. Chapter overview: synthesizing an implementation from a specification.

In the following, we obtain a minimal implementation Σ of O by folding *Fin* along a simple equivalence relation. Section 8.2 defines this equivalence and the corresponding folding algorithm. This algorithm is the last step to solve the bounded synthesis problem into a single component. Section 8.3 combines all results of this and the preceding chapter and defines the algorithm that solves the component synthesis problem, i.e., providing an implementation distributed over a set of components. The chapter closes with a discussion in Section 8.4.

8.2. Folding a Prefix to an Implementation

This section presents the final step of the synthesis algorithm in case of a *single component*. Section 7.5 defined an algorithm for computing a finite complete prefix *Fin* of an oclet specification O. Technically, the algorithm uses the oclet system Ω of O to construct *Fin* because Ω has an operational semantics. O and Ω are synonymous in this respect and we use Ω for our arguments of the behavior of O as shown in Section 6.7. *Fin* represents all reachability information of Ω as an acyclic Petri net. In this section, we define an algorithm that *folds Fin* into a Petri net system that exhibits the same behavior as Ω.

8.2.1. The idea

Let *Fin* be the finite complete prefix of Ω as constructed by Algorithm 7.30 of the preceding section. *Fin* is a Petri net. By putting a token on each minimal event of *Fin*, *Fin* exhibits an initial part of the behaviors of Ω. By adding a cycle to *Fin*, we let *Fin* exhibit more behaviors of Ω; adding all necessary cycles lets *Fin* exhibit all behaviors of Ω.

Each cycle follows from a *cut-off event* of *Fin* which is identified by Algorithm 7.30. The algorithm identifies for each *cut-off event* e_1 of *Fin* an equivalent event e_2 "earlier" in *Fin* s.t. the local configurations $[e_1]$ and $[e_2]$ have isomorphic futures in Ω; we write $e_1 \sim e_2$. Especially, each post-condition $b_1 \in Cut([e_1])$ of the local configuration $[e_1]$ has an equivalent post-condition $b_2 \in Cut([e_2])$ s.t. b_1 and b_2 have isomorphic futures; we write $b_1 \sim b_2$. By merging equivalent conditions $b_1 \sim b_2$, we add a cycle to *Fin* which lets *Fin* exhibit more behaviors of Ω. By merging the post-conditions of all equivalent local configurations we add all cycles s.t. *Fin* exhibits exactly the behavior of Ω. Thus, we fold *Fin* to an equivalent Petri net system $\Sigma = (N, m_0)$ as follows:

> for any two equivalent events $e_1 \sim e_2$ of *Fin*, pair-wise merge equivalent post-conditions $b_1 \sim b_2$ of the local configurations $[e_1]$ and $[e_2]$. The resulting Petri net is N. The initial marking m_0 puts on every minimal condition of *Fin* one token. (8.4)

Figure 8.2 illustrates this folding on the finite complete prefix *Fin* of the oclet system Ω of Figure 7.22 constructed in the preceding chapter.

In the example, the folded Petri net system Σ exhibits the same Ω by the following intuitive argument. Every run π that is represented in the unfolding of Ω starts in the prefix *Fin* of Figure 8.2. If π exceeds *Fin*, then π contains a cut-off event of *Fin*, for instance, event e_{12}. By Algorithm 7.30, *Fin* contains the equivalent event e_4 that has the same future as e_{12} in the unfolding of Ω. By letting e_{12} produce its tokens on the post-conditions of e_4, the successor events of e_4 mimic the successor events of e_{12} in π. In this way, the folded Petri net system Σ "automatically" mimics the future of e_{12} by the future of e_4. The same reasoning applies to all other cut-off events that occur in π. Thus, Σ exhibits exactly the behavior of $\Omega = \hat{R}(O)$ which is the behavior of a *minimal implementation* of O as shown in Section 6.6. We will show in the following that

> *for every bounded oclet specification O, the folded Petri net system Σ is a minimal implementation of O.*

The structural size of the synthesized Petri net system Σ is determined by the size of the finite complete prefix *Fin*. As discussed in Section 7.30, the size of *Fin*

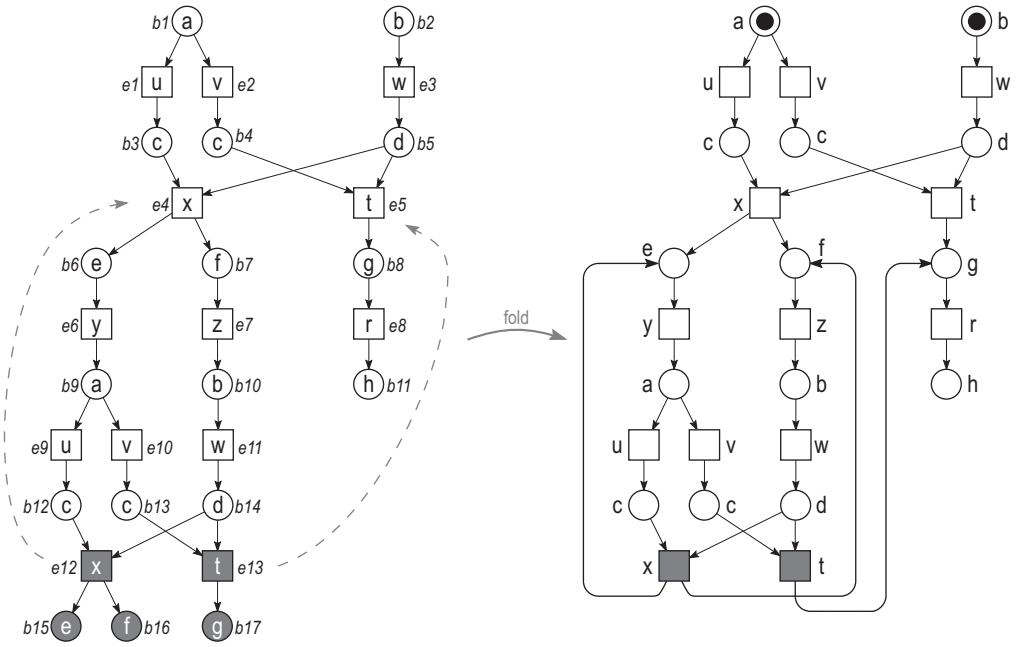


Figure 8.2. In the finite complete prefix (left), cut-off event e_{12} is equivalent to event e_4, and cut-off event e_{13} is equivalent to event e_5. Merging the post-conditions of the local configurations of equivalent events yields the Petri net system on the right. The Petri net system exhibits the same behavior as the oclet system in Figure 7.22.

can be influenced by the ordering relation $\prec$ used in Algorithm 7.30. Regardless of the structural size of Σ, its behavior is minimal wrt. the oclet specification O.

Optimizing the synthesis results wrt. additional criteria like the structural size of the implementation Σ is beyond the scope of this thesis. Choosing a different equivalence relation $\sim$ on the complete prefix of Figure 8.2 would yield the Petri net system in Figure 8.3. Although the Petri net in Figure 8.3 is smaller than the Petri net in Figure 8.2, both nets exhibit the same behaviors. The folding algorithm defined in the following states sufficient properties of $\sim$. Thus, synthesis can be adjusted according to the synthesis goal.

8.2.2. The folding algorithm

The algorithm to fold a prefix β of the unfolding of an oclet system Ω to a Petri net system is fairly simple: it merges post-conditions of equivalent local configurations. The decisive information for folding comes from an equivalence relation $\sim$ on the events of β. We present two abstraction requirements on $\sim$ and β which hold whenever β is a complete prefix of Ω. These two requirements capture the notion of a *complete prefix* of the unfolding of Ω as presented in Definition 7.24 in a form that suits the folding operation. If both requirements are satisfied, then *Fin* can

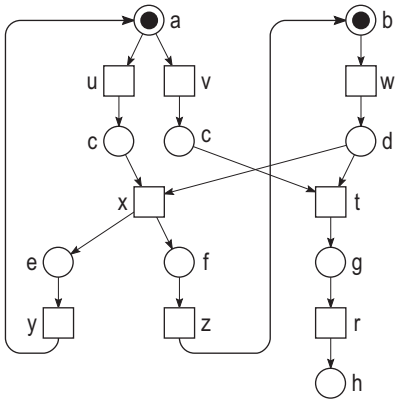


Figure 8.3. A structurally minimal Petri net system that exhibits the same behavior as the oclet system in Figure 7.22. This system is obtained by folding in the prefix b_9 to b_1 and b_{10} to b_2. That these conditions have equivalent futures cannot be detected from events of the prefix because the pre-set of b_1 and b_2 is empty.

be folded to an equivalent Petri net system. The output of Algorithm 7.30 *does* satisfy these requirements.

An alternative characterization of complete prefixes

Intuitively, two events e_1 and e_2 of a prefix β of the unfolding of Ω are equivalent if their futures are isomorphic in the unfolding. Thus, an occurrence of e_1 can be replaced by an occurrence of e_2, and vice versa. However, to reconstruct all behaviors of Ω from β, and to fold β to an implementation of Ω, the equivalence relation $\sim$ and the prefix β have to satisfy all of the following properties.

Definition 8.2 (Future equivalence). Let Ω be an oclet system with its unfolding *Unf*. Let β be a prefix of *Unf*. An equivalence relation $\sim$ on the nodes of β is a *future equivalence* (wrt. *Unf*) iff

1. if two nodes x_1, x_2 of β are equivalent, then they have isomorphic futures in *Unf*, i.e., $x_1 \sim x_2$ implies that $\Uparrow x_1$ and $\Uparrow x_2$ are isomorphic in *Unf*; and
2. for each maximal node $x_1 \in \max \beta$ either (a) $x_1 \in \max \mathit{Unf}$ or (b) there exists an equivalent node x_2 of β, $x_2 \notin \max \beta$, $x_1 \sim x_2$. ⌟

The notion of a future equivalence applies to every prefixes of an unfolding, i.e., β does not have to be finite or complete. Figure 8.4 illustrates this definition in a technical example.

Algorithm 7.30 identifies a future equivalence $\sim$ when constructing a finite complete prefix *Fin* of an oclet system. For each cut-off event e_1 the algorithm identifies an equivalent non-cut-off event e_2 s.t. their local configurations $[e_1]$ and $[e_2]$ have isomorphic futures. Set $e_1 \sim e_2$ and extend $\sim$ to the post-conditions of $\mathit{Cut}([e_1])$ and $\mathit{Cut}([e_2])$, respectively. By construction of *Fin*, every condition b_1 that is reached by the local configuration $[e_1]$ of a cut-off events e_1 has a

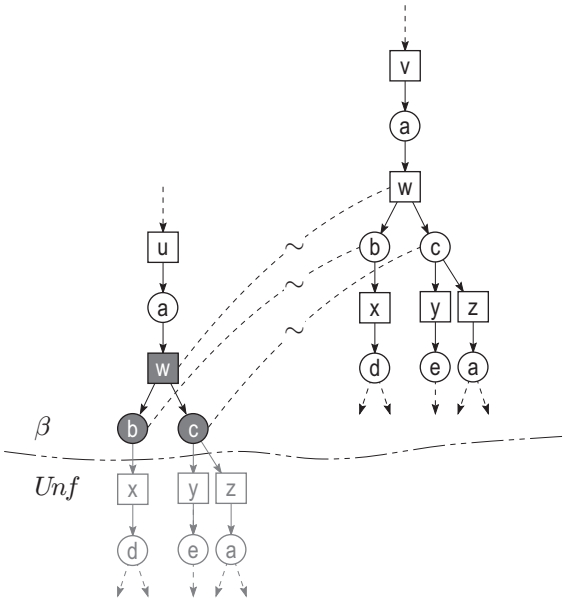


Figure 8.4. Illustration of the future equivalence (Definition 8.2).

$\sim$-related post-condition b_2 that is reached by the local configuration $[e_2]$ of a non-cut-off event e_2. Especially, each post-condition of a cut-off event is related to a post-condition of a non-cut-off event. This satisfies the second requirement of Definition 8.2. The reflexive, transitive, and symmetric closure of $\sim$ is a future equivalence. Figure 8.5 illustrates a future equivalence coming from Algorithm 7.30. It defines $e_{12} \sim e_4$ and $e_{13} \sim e_5$ (extended to their respective post-conditions).

A future equivalence $\sim$ ensures that only nodes of β with equivalent futures belong to the same equivalence class, and that nodes which have no successors are also represented as such in β. To ensure that β is a *complete* prefix (Def. 7.24), also all successors of a node x have to be represented in β. More precisely, if β does not represent all successors of x in Ω, then there exists an equivalent node x' of β that does.

Definition 8.3 (Successor complete prefix). Let Ω be an oclet system with its unfolding Unf. Let β be a prefix of Unf with a future equivalence $\sim$. β is *successor complete* iff for each node x of β exists an equivalent node $x' \sim x$ of β s.t. for each $y \in post_{Unf}(x)$ exists $y' \in post_{\beta}(x')$ and $\Uparrow y$ and $\Uparrow y'$ are isomorphic. ⌟

For instance, the prefix in Figure 8.5 is successor complete. The successors of b_{15}, b_{16}, and b_{17} are represented by their respective equivalent conditions b_6, b_7, and b_8.

A future equivalence of a successor complete prefix β of an oclet system Ω suffices to reconstruct all behaviors of Ω from β. Conversely, the finite complete prefixes constructed by Algorithm 7.30 are successor complete. We prove the following two lemmas at the end of Section 8.2.3.

Lemma 8.4: *Let Unf be an unfolding of an oclet system. Every future equivalence $\sim$ on a successor complete prefix β of Unf extends to a future equivalence $\simeq$ on Unf* ⋆

Lemma 8.5: *Every finite complete prefix Fin computed by Algorithm 7.30 is successor complete.* ⋆

Folding

A successor complete prefix β can be folded to a Petri net system Σ by the help of a future equivalence $\sim$. Every equivalence class of $\sim$ defines a node of Σ: equivalent conditions define a place, equivalent events define a transition. The future equivalence guarantees that the arcs between places and transitions respect the behavior of β.

Though, we have to be careful when determining the equivalence class of an event. Each event of β represents an occurrence of a transition. So, we may only fold those events of β to the same transition of Σ that are *labeled consistently* wrt. their pre- *and* post-sets; see Def. 7.11. A future equivalence $\sim$ only ensures that the post-conditions of two equivalent events $e_1 \sim e_2$ are pairwise equivalent as well, i.e., e_1 and e_2 are label consistent wrt. their post-conditions. This does not necessarily hold for the pre-conditions of e_1 and e_2. If ${}^\bullet e_1$ and ${}^\bullet e_2$ are not pairwise equivalent, then e_1 and e_2 represent occurrences of different transitions. So, to fold e_1 and e_2 to the same transition, we demand that $e_1 \sim e_2$, *and* that their pre-conditions ${}^\bullet e_1$ and ${}^\bullet e_2$ are pairwise equivalent. Under this requirement, the folded Petri net system Σ exhibits exactly the behavior represented by β.

Definition 8.6 (Folded Petri net system). Let Ω be an oclet system. Let β be a successor complete prefix of the unfolding of Ω with a future equivalence $\sim$. We define the following equivalence classes of $\sim$:

- $\langle b \rangle_\sim := \{b_2 \mid b \sim b_2\}$, for all conditions $b \in B_\beta$,
- denote $\langle B \rangle_\sim = \{\langle b \rangle_\sim \mid b \in B\}$ for sets B of conditions, and
- $\langle e \rangle_\sim := \{e_2 \mid e \sim e_2, \langle {}^\bullet e \rangle_\sim = \langle {}^\bullet e_2 \rangle_\sim\}$, for all events $e \in E_\beta$.

The *folded Petri net system* (wrt. $\sim$) is $\beta_\sim := (N_\sim, m_\sim)$ with the Petri net $N_\sim = (P_\sim, T_\sim, F_\sim, \ell_\sim)$ s.t.

- $P_\sim := \{\langle b \rangle_\sim \mid b \in B_\beta\}$,
- $T_\sim := \{\langle e \rangle_\sim \mid e \in E_\beta\}$,
- $F_\sim := \{(\langle x \rangle_\sim, \langle y \rangle_\sim) \mid (x, y) \in F_\beta\}$,
- $\ell_\sim(\langle x \rangle_\sim) := \ell(x)$, for all $x \in X_\beta$,

and the initial marking $m_\sim(\langle b \rangle_\sim) := |\{b' \in \langle b \rangle_\sim \mid b' \in \min \beta\}|$, for all $b \in B_\beta$. ⌟

Figure 8.5 depicts an example. The dashed lines indicate the future equivalence $e_{12} \sim e_4$ and $e_{13} \sim e_5$ (extended to their respective post-conditions) on the depicted prefix *Fin*. Folding this prefix along $\sim$ yields the Petri net system to the right of Figure 8.5. Although, e_{12} and e_4 are equivalent, they are not folded to same transition. Here, the requirement $\langle {}^\bullet e_{12} \rangle_\sim = \langle {}^\bullet e_4 \rangle_\sim$ of Definition 8.6 is

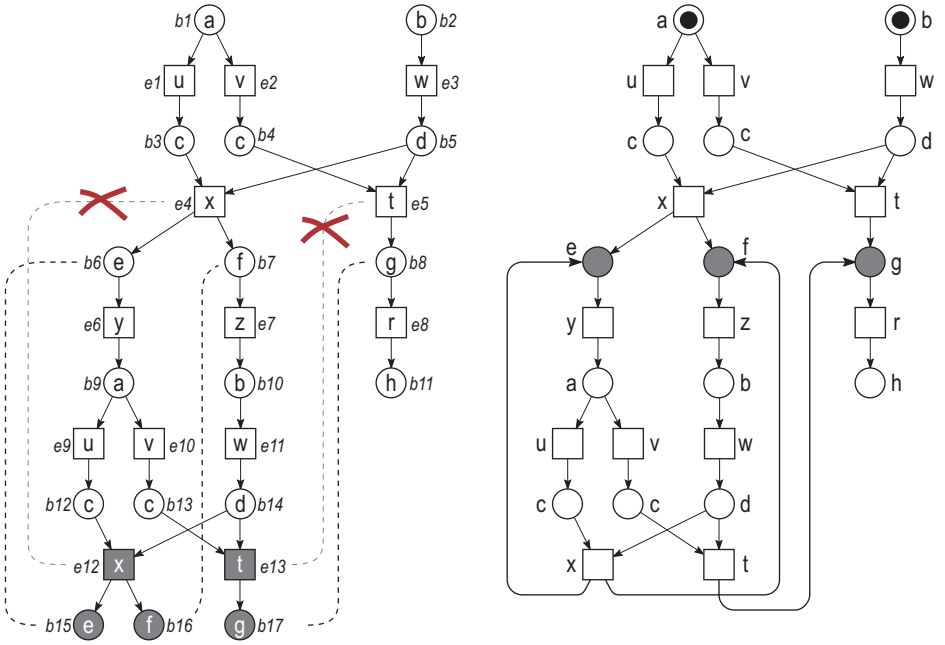


Figure 8.5. The finite complete prefix with the future equivalence on the left folds to the Petri net system on the right.

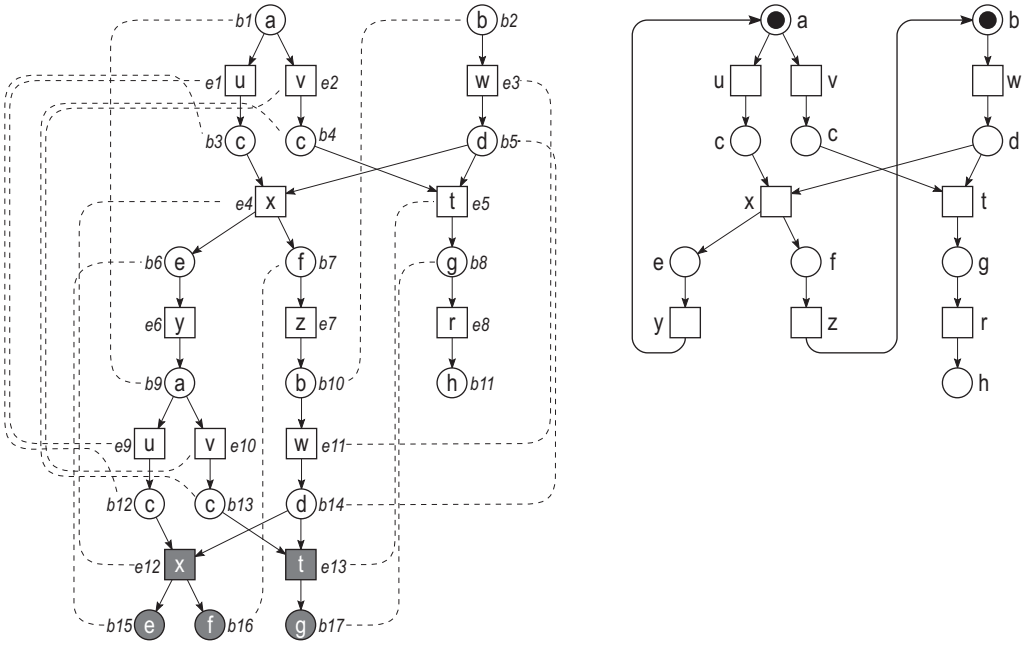


Figure 8.6. The finite complete prefix with the future equivalence on the left folds to the Petri net system on the right.

violated. If we folded e_{12} and e_4 to the same transition t, then t would have the places $\langle b_{12}\rangle_\sim$ *and* $\langle b_3\rangle_\sim$ with $\langle b_{12}\rangle_\sim \neq \langle b_3\rangle_\sim$ in its pre-set which is not equivalent to the behavior represented by *Fin*.

There is an even coarser future equivalence of this prefix *Fin*. Its equivalence classes are $[e_1, e_9], [e_2, e_{10}], [e_3, e_{11}], [e_4, e_{12}], [e_5, e_{13}], [e_6], [e_7], [e_8]$ (and the equivalence classes of the respective post-conditions) as depicted in Figure 8.6 to the left. The corresponding folded Petri net system is depicted to the right.

Folding a successor complete prefix β of an unfolding *Unf* along a future equivalence preserves the behavior of *Unf*.

Theorem 8.7. *Let Ω be an oclet system. Let β be a successor complete prefix of $Unf(\Omega)$ with a future equivalence $\sim$ on β.*

1. *The folded Petri net system $\beta_\sim$ exhibits the same behavior as Ω, i.e., $Unf(\beta_\sim) = Unf(\Omega)$.*
2. *If β is finite, then $\beta_\sim$ is finite.*
3. *If Ω is k-bounded, then $\beta_\sim$ is k-bounded.* ⋆

We prove this theorem in Section 8.2.3.

Labeled vs. unlabeled Petri nets

The folded Petri net system $\beta_\sim$ is labeled. Specifically several transitions or places of $\beta_\sim$ carry the same label.

A system designer might prefer that each action A which occurs in an oclet specification should be implemented as a unique action of the system, i.e., one transition A. In some specifications, this wish contradicts the general goal of the synthesis: to construct a minimal implementation.

Figure 8.7 depicts an example adapted from [96]. For the given oclet system Ω, there exists no *unlabeled* Petri net system that exhibits the behavior of Ω. The depicted Petri net system Σ is the least (labeled) Petri net system that implements Ω. This example also shows that oclet systems are strictly more expressive than unlabeled Petri net systems.

A system designer may turn a labeled Petri net into an unlabeled Petri net by introducing additional labels. A common technique is to "refine" a label A to a set of labels $A_1, A_2, \ldots$ by introducing one new label A_i for each transition that is labeled with A. This way, action A is implemented as a set of similar, but distinct actions. Places can be relabeled in the same way.

8.2.3. The folded system implements the specification

This section proves that folding a finite complete prefix of an oclet system Ω yields a finite Petri net that implements Ω.

We prove this proposition with the following strategy. We first show that every successor complete prefix β of the unfolding of Ω with a corresponding future equivalence allows to reconstruct all behaviors of Ω. Technically, we show that $\sim$ canonically extends to a future equivalence $\simeq$ on the entire unfolding *Unf*. Conversely, we show that every prefix *Fin* constructed by Algorithm 7.30

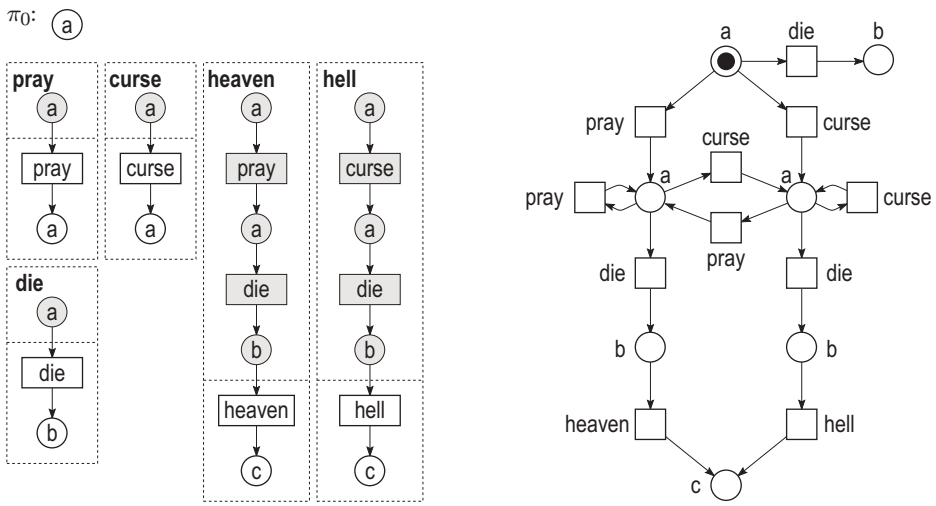


Figure 8.7. For the oclet system Ω (left) exists no equivalent unlabeled Petri net with the same behavior. The Petri net system (right) is the least labeled Petri net that implements Ω.

is successor complete with a corresponding future equivalence. Thus, future equivalence and successor completeness are an alternative characterization of finite complete prefixes of oclet systems.

We then show that the extension of $\sim$ on β to $\simeq$ on *Unf* introduces no new equivalence classes, i.e., the nodes of *Unf* are put into existing equivalence classes only. Thus, folding β and folding *Unf* yields the same system $\beta_\sim = Unf_\sim$. Then the unfolding of $Unf_\sim$ is *Unf* again by a simple inductive argument.

Successor complete prefixes are complete prefixes

Proof of Lemma 8.4. Let *Unf* be an unfolding of an oclet system. Every future equivalence $\sim$ on a successor complete prefix β of *Unf* extends to a future equivalence $\simeq$ on *Unf*.

Proof. The extended future equivalence $\simeq$ on *Unf* is defined co-inductively.

[Base.] If $x \sim y$, then $x \simeq y$, for any two nodes x, y of β. Because $\sim$ is a future equivalence, also $\simeq$ is a future equivalence.

[Extend by conditions.] Let $e \simeq f$ be two events of *Unf*. $\Uparrow e$ and $\Uparrow f$ are isomorphic in *Unf* by inductive assumption. Thus, for each post-condition $b \in e^\bullet$ exists a corresponding post-condition $c \in f^\bullet$. We extend $\simeq$ and set $b \simeq c$.

[Extend by events.] Let $\{b_1, \ldots, b_k\}$ and $\{c_1, \ldots, c_k\}$ be sets of pair-wise equivalent conditions $b_i \simeq c_i$, for $i = 1 \ldots, k$, s.t. there exist events e and f with $\{b_1, \ldots, b_k\} = {}^\bullet e$ and $\{c_1, \ldots, c_k\} = {}^\bullet f$ and the same label $\ell(e) = \ell(f)$. Let $\alpha_i : (\Uparrow b_i) \to (\Uparrow c_i), i = 1, \ldots, k$ be the corresponding isomorphisms. Then $\alpha = \bigcap_{i=1}^{k} \alpha_i$ is an

isomorphism from $\Uparrow e$ to $\Uparrow f$ because $e \in \bigcap_{i=1}^{k} \Uparrow b_i$ and $f \in \bigcap_{i=1}^{k} \Uparrow c_i$ and *Unf* is label consistent (Def. 7.11). We extend $\simeq$ and set $e \simeq f$. □

Proof of Lemma 8.5. For the folding operation (Def. 8.6) to be applicable for a synthesis algorithm, we have to show that the prefix *Fin* of an oclet system Ω that is constructed by Algorithm 7.30 is successor complete.

Proof. Let $\sim$ be the future equivalence identified by Alg. 7.30. An event of *Fin* has all its post-conditions by construction. If a condition b has a post-event $e_1 \in post_{Unf}(b_1)$ that is not *Fin*, then e_1 depends on a cut-off event f_1 of *Fin* and $b_1 \in \Uparrow[f_1]$. By Alg. 7.30, there exists a smaller event f_2 of *Fin* with $[f_2] \prec [f_1]$ that represents f_1: $\Uparrow[f_1]$ and $\Uparrow[f_2]$ are isomorphic by an isomorphism α. Let $b_2 \in \Uparrow[f_2]$ with $b_2 = \alpha(b_1)$. Thus, $\Uparrow b_1$ and $\Uparrow b_2$ are isomorphic, and we can extend $\sim$ by $b_1 \sim b_2$. Thus, there exists a corresponding event $e_2 \in post_{Unf}(b_2)$ with $e_2 = \alpha(e_1)$, and $\Uparrow e_1$ and $\Uparrow e_2$ are isomorphic. The local configuration $[e_2]$ is smaller than the local configuration $[e_1]$ wrt. the adequate order $\prec$ in Alg. 7.30, because $[f_2] \prec [f_1]$ and $\prec$ is preserved under finite extensions $[e_1] = [f_1] \cup E_1 \prec [f_2] \cup \alpha(E_1) = [e_2]$.

If e_2 is not in *Fin*, then there exists another cut-off event f_3 and we can iterate the procedure and reach another event e_3. By Def. 7.16, the step from e_i to e_{i+1} can be iterated only finitely many times, because there are only finitely many smaller configurations $[e_k] \prec \ldots [e_2] \prec [e_1]$. Thus, we eventually reach $b_k \sim b_1$ and $e_k \in post_{Fin}(b_k)$ with $\Uparrow e_1$ and $\Uparrow e_k$ being isomorphic. □

A finite complete prefix folds to an equivalent finite Petri net system

Lemma 8.8: *Let Unf be an unfolding of an oclet system. If $\sim$ is a future equivalence on a successor complete prefix β of Unf, and $\simeq$ the extension of $\sim$ to Unf, then $\beta_\sim$ and $Unf_\simeq$ are isomorphic.* ⋆

The proof of this lemma follows by co-induction on *Unf*, beginning in β. Proving that $\beta_\sim$ and $Unf_\simeq$ are isomorphic requires that β is successor complete. Otherwise $\simeq$ would introduce new equivalence classes in *Unf* that have no match in $\sim$ on β.

Proof. We have to show that $\sim$ and $\simeq$ have the same number of equivalence classes.

By Lem. 8.4, $\simeq$ has all equivalence classes of $\sim$. In the induction base, for each pair $x \simeq y$, there trivially exists a node z of β s.t. $z \simeq x$.

For the induction step, we have to show that whenever $\simeq$ is extended by a new pair (x, y), then there already exists a pair $(x, z) \in \simeq$ with z being a node of the prefix β. In this case, y only extends the equivalence class $[z]$ of β and introduces no new equivalence class. Figure 8.8 illustrates the subsequent arguments for the extensions by equivalent conditions $b \simeq c$ and equivalent events $e \simeq f$.

The induction step that extends $\simeq$ to conditions $b \simeq c$ requires two pre-events $e \simeq f$, $e \in {}^\bullet b, f \in {}^\bullet c$. By inductive assumption, there exists an event g of β s.t. $g \simeq e$. Event g has a post-condition $d \in g^\bullet$ s.t. $d \simeq b$ by construction of $\simeq$. Because β is a prefix of *Unf*, the post-condition d is a node of β as well. Consequently, d occurs in an equivalence class of $\sim$ and extending $\simeq$ to conditions $b \simeq c$ adds

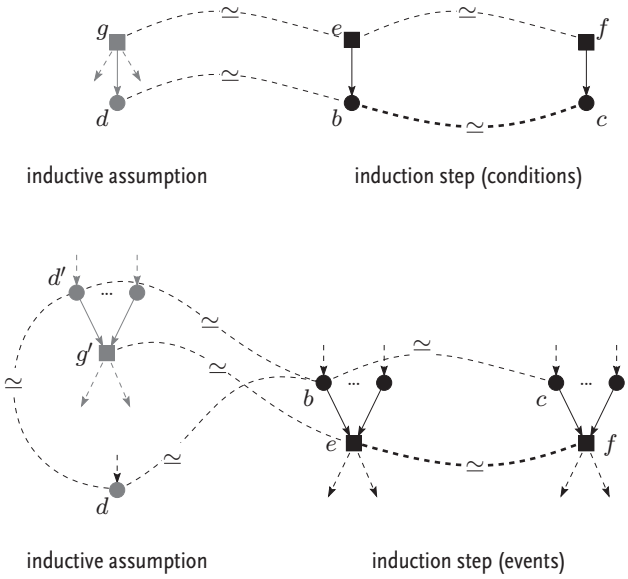


Figure 8.8. Illustration of the induction step: extending $\simeq$ by equivalent conditions $b \sim c$ or equivalent events $e \sim f$ preserves the equivalence classes of $\simeq$.

b and c to this class, but does not add any new equivalence class. Also the arcs (e, b) and (f, c) in Unf have an equivalent counter-part (g, d) in β.

Extending $\simeq$ to events $e \simeq f$ requires pre-conditions $b \in {}^\bullet e$ and $c \in {}^\bullet f$ with $\simeq$. By the same argument, there exists a condition d of β with a post-event $g \in d^\bullet$ s.t. $d \simeq b$ and $g \simeq e$. There are two cases:

1. $g \in post_\beta(d)$. Then g is trivially a node of β.
2. $g \notin post_\beta(d)$. Because β is successor complete (Def. 8.3), there exists a condition $d' \simeq d$ of β with $g' \in post_\beta(d')$ s.t. $\Uparrow g$ and $\Uparrow g'$ are isomorphic. Thus, $g \simeq g'$ by Def. 8.2 and Lem. 8.4. The condition d' is equivalent to b and its post-event g' is equivalent to e by inductive assumption. So, $g' \simeq e$ and g' is an event of β.

Like for conditions, events e and f are added to the equivalence class of g which is an equivalence class of $\sim$. Likewise, the arcs (b, e) and (c, f) in Unf have an equivalent counter-part (d, g) in β. Altogether, $\sim$ has the same equivalence classes as $\simeq$.

Proposition: $\beta_\sim$ and $Unf_\simeq$ are isomorphic.

- Places. Because $\sim$ and $\simeq$ have the same equivalence classes, $\beta_\sim$ and $Unf_\simeq$ define the same sets of places.
- Transitions. Every new pair of equivalent events $e \simeq f$ of Unf requires that their pre-conditions are pair-wise equivalent. Thus, $\beta_\sim$ and $Unf_\simeq$ define the same sets of transitions. See Def. 8.6.

- Arcs. By the co-inductive extension of $\sim$ to $\simeq$ on Unf, every arc (x, y) in Unf has an equivalent counter-part (x', y') in β with $x' \in \langle x \rangle_{\simeq}$ and $y' \in \langle y \rangle_{\simeq}$. Thus, $\beta_{\sim}$ and $Unf_{\simeq}$ define the same sets of arcs.
- Labeling. By definition of the equivalence classes.
- Initial marking. By $\min \beta = \min Unf$ because β is a complete prefix of Unf. □

Proof of Theorem 8.7. By the preceding Lemma 8.8, the folded unfolding $Unf_{\simeq}$ and the folded prefix $\beta_{\sim}$ have the same behaviors because $Unf_{\simeq}$ and $\beta_{\sim}$ are isomorphic, i.e., $Unf(Unf_{\simeq}) = Unf(\beta_{\simeq})$. Thus, we prove Theorem 8.7 using the folded unfolding $Unf_{\simeq}$: Ω and $\beta_{\simeq}$ have the same behavior because $Unf(\Omega) = Unf = Unf(Unf_{\simeq}) = Unf(\beta_{\simeq})$.

Proof (of Theorem 8.7). By induction. Let $Unf_{\simeq} = (N_{\simeq}, m_{\simeq})$.

The future equivalence $\simeq$ defines a labeling $\lambda : Unf \to N_{\simeq}$ s.t. each run $\pi = (B, E, F) \in R(Unf)$ of Ω is a run $(\pi, \lambda) = (B, E, F, \lambda)$ of $Unf_{\simeq}$. The labeling is canonical: each event e of Unf is labeled with its transition $\lambda(e) = \langle e \rangle_{\simeq}$ of $Unf_{\simeq}$, and each condition b of Unf is labeled with its place $\lambda(b) = \langle b \rangle_{\simeq}$ of $Unf_{\simeq}$.

Let $\pi \in R(Unf)$ be a distributed run of Ω. We show that (π, λ) is a distributed run of the Petri net system $Unf_{\simeq}$ by the axiomatic characterization of the distributed runs of Petri nets (Definition 2.16).

1. Each condition b is labeled with a place $\langle b \rangle_{\simeq}$ and each event e is labeled with a transition $\langle e \rangle_{\simeq}$ by definition of λ.
2. $\min \pi$ represents the initial marking $m_{\simeq}$ via the labeling λ by definition of the folded system $Unf_{\simeq}$ (Def. 8.6).
3. Let e be an event of π. By construction of transition $\langle e \rangle_{\simeq}$, the labeling λ bijectively maps ${}^\bullet e$ to ${}^\bullet \langle e \rangle_{\simeq}$ (Def. 8.6). Further, all equivalent events $f \in \langle e \rangle_{\simeq}$ have isomorphic futures. So, $(e, c) \in F_{Unf}$ iff $(\langle e \rangle_{\simeq}, \langle c \rangle_{\simeq}) \in F_{\simeq}$, for all places $\langle c \rangle_{\simeq}$ of $N_{\simeq}$. Thus, λ bijectively maps $e^\bullet$ to $\langle e \rangle_{\simeq}^\bullet$ and event e represents an occurrence of transition $\lambda(e) = \langle e \rangle_{\simeq}$.

Altogether, π is a distributed run of $Unf_{\simeq}$ by Definition 2.16. Further by the properties of the future equivalence $\simeq$, $Unf_{\simeq}$ only exhibits runs of Unf.

It follows directly from Definition 8.6, that the folded Petri net system $Unf_{\simeq} = \beta_{\sim}$ has at most as many nodes as β. Thus, if β is finite, then also $\beta_{\sim}$ is finite. Further, Ω is k-bounded iff $\beta_{\sim}$ is k-bounded for all natural numbers k, because $R(Unf)$ and $R(\beta_{\sim})$ have the same runs. □

8.3. Synthesizing Components from a Scenario-based Specification

This section summarizes the results of the preceding sections and defines an algorithm for synthesizing a distributed system consisting of *several components* from a scenario-based specification. Figure 8.9 illustrates the synthesis algorithm and the outline of this section.

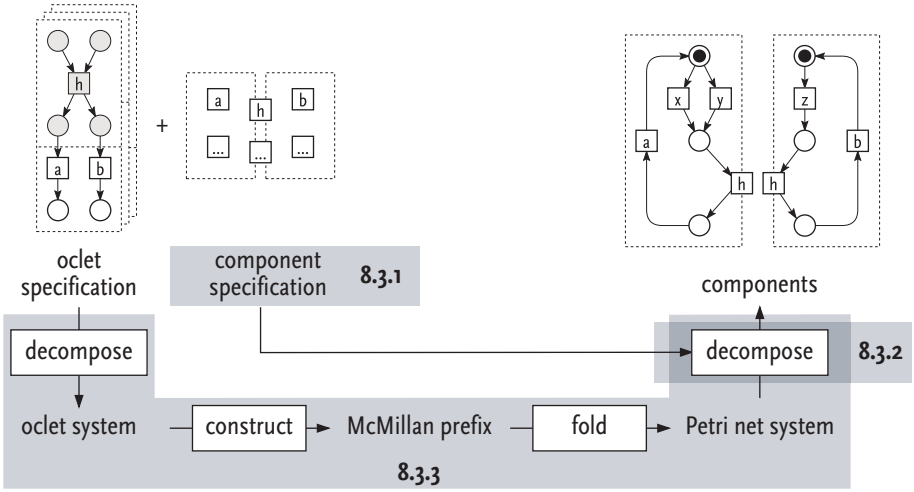


Figure 8.9. From scenarios to components: overview of the synthesis algorithm.

Recall: the single component case. The preceding section just closed the circle in our approach for synthesizing a minimal implementation Σ of an oclet specification O. Our solution of the bounded synthesis problem (8.2) is the following algorithm. Let O be a k-bounded oclet specification for some $k \in \mathbb{N}$. The synthesis algorithm returns $\Sigma = \textsc{Syn}(O)$ as follows:

1. The first step is to decompose O into its canonical oclet system Ω according to Corollary 6.17.
2. Then construct the finite complete prefix *Fin* of Ω using Algorithm 7.30.
3. The cut-off events of *Fin* define a future equivalence $\sim$ according to Section 8.2.2.
4. Fold *Fin* along $\sim$ to the Petri net system $\Sigma := \mathit{Fin}_{\sim}$ according to Definition 8.6.

We know from Theorem 8.7 that the Petri net system Σ exhibits the behavior of Ω, i.e., Σ is a minimal implementation O. Currently, Σ does not distinguish different components. Yet, a solution to the *component synthesis problem* (8.3) is just a small step away.

The component synthesis problem was to synthesize for *every scenario-based specification* O the system components $\Sigma_1, \ldots, \Sigma_r$ that are specified in O so that the composed system $\Sigma_1 \oplus \ldots \oplus \Sigma_r$ is a *minimal* implementation of O. We first formalize in Section 8.3.1 how to specify and how to model components. Section 8.3.2 shows how to decompose a Petri net system into components, and Section 8.3.3 presents the algorithm that solves the component synthesis problem (8.3).

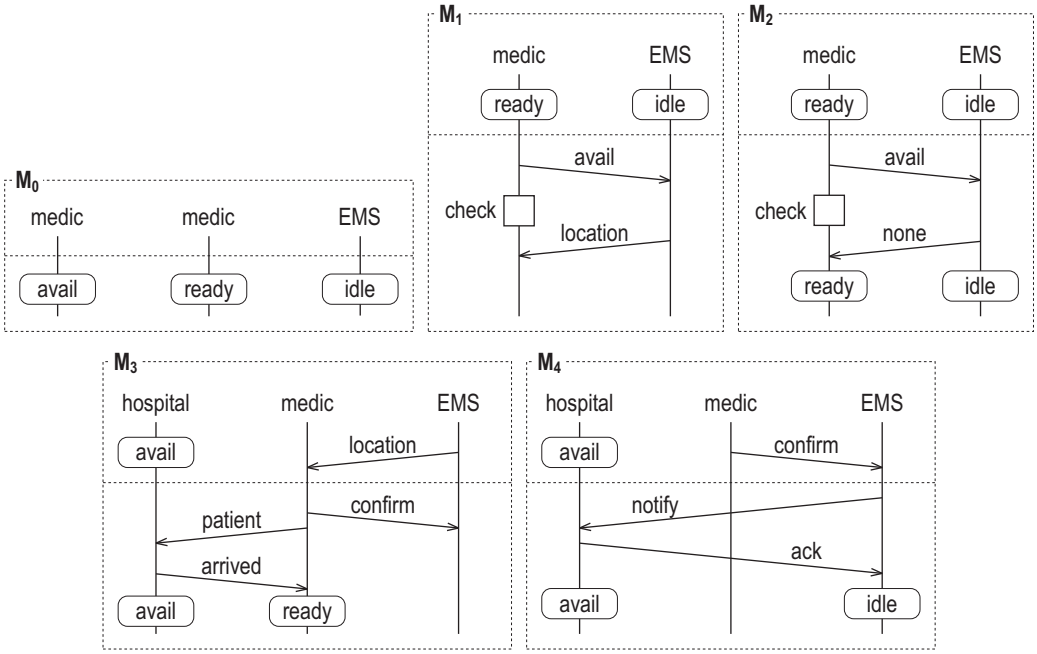


Figure 8.10. Specification of an emergency management system, adapted from Section 4.6.

8.3.1. Describing components

The standard approach to specify components with scenarios is to distinguish which actions of the specification shall be implemented in which component. Further, a scenario-based specification, at least implicitly, describes which component implements which state. Figure 8.10 shows an example in the syntax of MSCs. The following definition makes the assignment of actions and states to components explicit.

Definition 8.9 (Component specification). Let $\mathcal{L} = \mathcal{L}_E \cup \mathcal{L}_B$ be names of actions and states, respectively. A *component specification* over $\mathcal{L}$ is a family $(K_i)_{i=1}^r$ of names s.t. $\bigcup_{i=1}^r K_i = \mathcal{L}$. $(K_i)_{i=1}^r$ is a *component specification of an oclet specification* O iff

1. $\mathcal{L}$ are exactly the names occurring in O, and
2. for each arc (x, y) that occurs in an oclet in O holds: $\ell(x) \in K_i$ iff $\ell(y) \in K_i$, for all $i = 1, \ldots, r$ ⌟

Figure 8.11 shows the oclet specification that corresponds to Figure 8.10 together with its component specification.

Intuitively, the actions and states in K_i should be implemented by component i. However, the specification must not violate causality: an arc is always implemented by *one* component. Two components i and j *synchronize* on each action and each state $a \in K_i \cap K_j$. So, an event e may have a pre-condition in one component i

and a post-condition in another component j — as long as i and j synchronize on e. Correspondingly for a condition having pre- and post-condition in different components. An *asynchronous* specification satisfies $K_i \cap K_j \subseteq \mathcal{L}_B$, for all $1 \leq i < j \leq r$. A *monolithic* specification defines only one component (K_1) which contains all names.

There is a similarly canonic approach to *model components* of a distributed system. Basically, a component is a system model that distinguishes an *interface*. Several components with matching interfaces are a *component model*. All components in a component model *compose* to a larger distributed system by *union*. Here is the formalization in terms of Petri nets.

Definition 8.10 (Component model). A *component* is a Petri net system $\Sigma = (N, m_0)$ together with an *interface* $I \subseteq X_N$; we write (N, m_0, I) for the component. A *component model* is a family $(N_i, m_i, I_i)_{i=1}^r$ of components s.t. for any two components i and j holds:

1. each shared node $x \in X_i \cap X_j$ is in the shared interface $x \in I_i \cap I_j$,
2. the initial markings agree on shared places: $m_i(p) = m_j(p)$, for all places in $I_i \cap I_j$.

The *composition* of a component model $(N_i, m_i, I_i)_{i=1}^r$ is the Petri net system $(N, m) = (N_1, m_1) \oplus \ldots \oplus (N_r, m_r)$ where $P_N = \bigcup_{i=1}^r P_{N_i}$, $T_N = \bigcup_{i=1}^r T_{N_i}$, $F_N = \bigcup_{i=1}^r F_{N_i}$, $m(p) = m_i(p)$ for all $p \in P_N$ and each component i with $p \in P_{N_i}$. ⌟

The preceding definition is technically sound also for labeled Petri net systems: since Chapter 2 we assume strict typing and a universal labeling of all Petri net nodes.

A component model *implements* a component specification if no component implements an action that was specified for another component.

Definition 8.11 (Implementation of a component specification). Let $(K_i)_{i=1}^r$ be a component specification. Let $K_i^- := K_i \setminus \bigcup_{i \neq j} K_j$ denote the *internal names* of component $i = 1, \ldots, r$. Let $K = \bigcup_{i=1}^r K_i$ denote all names of the specification.

A component model $(N_i, m_i, I_i)_{i=1}^r$ *implements* $(K_i)_{i=1}^r$ iff for each component $i = 1, \ldots, r$ holds:

1. Only component N_i implements actions and states of K_i, i.e., for all $x \in X_i$ holds: if $\ell(x) \in K$, then $\ell(x) \in K_i$.
2. An internal name is not implemented as an interface node, and vice versa, i.e., for all $x \in X_i$ holds: if $\ell(x) \in (K_i \setminus K_i^-)$, then $x \in I_i$. ⌟

Definitions 8.9 and 8.11 are rather permissive. A component specification is unconstrained up to the requirement to assign each action and each state to at least one component. An implementation must not put an action or states into the wrong component. A component may implement additional actions that are not specified. These notions may allow for unusual or practically infeasible specifications.

However, our solution to the synthesis problem succeeds on these general notions. Any more constrained notion of specification and implementation is subsumed by our solution.

8.3.2. Synthesizing components

With the given formal notions, the component synthesis problem (8.3) reads as follows: given an oclet specification O and a component specification $(K_i)_{i=1}^r$ of O, synthesize components $(N_i, m_i, I_i)_{i=1}^r$ s.t. the composed system $\Sigma := (N_1, m_1) \oplus \ldots \oplus (N_r, m_r)$ is a minimal implementation of O and $(K_i)_{i=1}^r$.

The solution for the component synthesis problem builds on the single component case. The synthesis *preserves* all names occurring in O and the causal relations between events and conditions. So, the synthesized single Petri net system $\Sigma = \textsc{Syn}(O)$ can be decomposed into exactly those components that are specified in $(K_i)_{i=1}^r$.

The transitions and places of the synthesized system $\Sigma = \textsc{Syn}(O)$ are labeled with the action names and state names $\mathcal{L}$ that occur in O; we know $\mathcal{L} = K_1 \cup \ldots \cup K_r$ by Definition 8.9. The decomposition of $\Sigma = (N, m)$ into components $(N_i, m_i, I_i)_{i=1}^r$ follows from the sets $K_1, \ldots, K_r$. Component (N_i, m_i, I_i) contains all transitions and places that are labeled with names in K_i. Its interface I_i consists of those places and transitions which are shared with another component (N_j, m_j, I_j). Because each name that occurs in O is also assigned to at least one component i, also each node of Σ ends up in at least one component. The resulting components re-compose to Σ as stated in Definition 8.10. Figure 8.12 depicts an example.

Definition 8.12 (Decomposition into components). Let $\Sigma = (N, m)$ be a Petri net system and let $(K_i)_{i=1}^r$ be a component specification over the transition labels and place labels of Σ. The *decomposition of* Σ *by* $(K_i)_{i=1}^r$ is the component model $(N_i, m_i, I_i)_{i=1}^r$ where each component $i = 1, \ldots, r$ has

- transitions $T_i = \{t \in T_N \mid \ell(t) \in K_i\}$,
- places $P_i = \{p \in P_N \mid \ell(t) \in K_i\}$,
- arcs $F_i = F_N|_{X_i \times X_i}$,
- initial marking $m_i = m|_{P_i}$, and
- interface $I_i = \{x \in X_i \mid \exists j \in \{1, \ldots, r\}, i \neq j, \ell(x) \in K_j\}$. ⌟

Corollary 8.13 (Component synthesis solution): *Let O be an oclet specification, let $(K_i)_{i=1}^r$ be a component specification of O. Let $\Sigma = \textsc{Syn}(O)$ be a synthesized implementation of O and let $(N_i, m_i, I_i)_{i=1}^r$ be the decomposition of Σ by $(K_i)_{i=1}^r$. Then*

1. *the component model $(N_i, m_i, I_i)_{i=1}^r$ implements the component specification $(K_i)_{i=1}^r$,*
2. *$(N_1, m_1) \oplus \ldots \oplus (N_r, m_r) = \Sigma$, and*
3. *$(N_1, m_1) \oplus \ldots \oplus (N_r, m_r)$ implements O.* ⋆

The first two propositions follow from the respective definitions. The third proposition holds because of Theorem 8.7 for the single component case.

8.3.3. The complete synthesis algorithm

This section summarizes all preceding results and presents the algorithm that solves the component synthesis problem. We also highlight two adjustable parameters of the synthesis algorithm and discuss its complexity.

Formally, a specification is an *oclet specification* O as defined in Chapter 4 together with a *component specification* $(K_i)_{i=1}^r$ (Def. 8.9). We assume the oclets in O to be *label consistent* (Def. 7.11). The algorithm Syn that solves the component synthesis problem (8.3) takes O and $(K_i)_{i=1}^r$ as input and returns a component model $(N_i, m_i, I_i)_{i=1}^r = \text{Syn}(O, (K_i)_{i=1}^r)$ as follows:

1. decompose O into its oclet system Ω according to Section 6.7;
2. compute the finite complete prefix *Fin* of Ω by Algorithm 7.30;
3. the cut-off events of *Fin* define a future equivalence $\sim$ on *Fin*;
4. fold *Fin* to $\Sigma := Fin_\sim$ according to Definition 8.6;
5. decompose Σ by $(K_i)_{i=1}^r$ into $(N_i, m_i, I_i)_{i=1}^r$ according to Definition 8.12.

The synthesis algorithm is only guaranteed to terminate if the oclet system Ω is k-bounded for some natural number k. To guarantee termination in any case, Syn can be extended to take an additional parameter k and to check during the construction of *Fin* whether *Fin* reaches a cut that is not k-bounded. If this is the case, then Syn terminates without result.

By Theorem 8.7 and Corollaries 6.17 and 8.13 our synthesis algorithm Syn satisfies the following property.

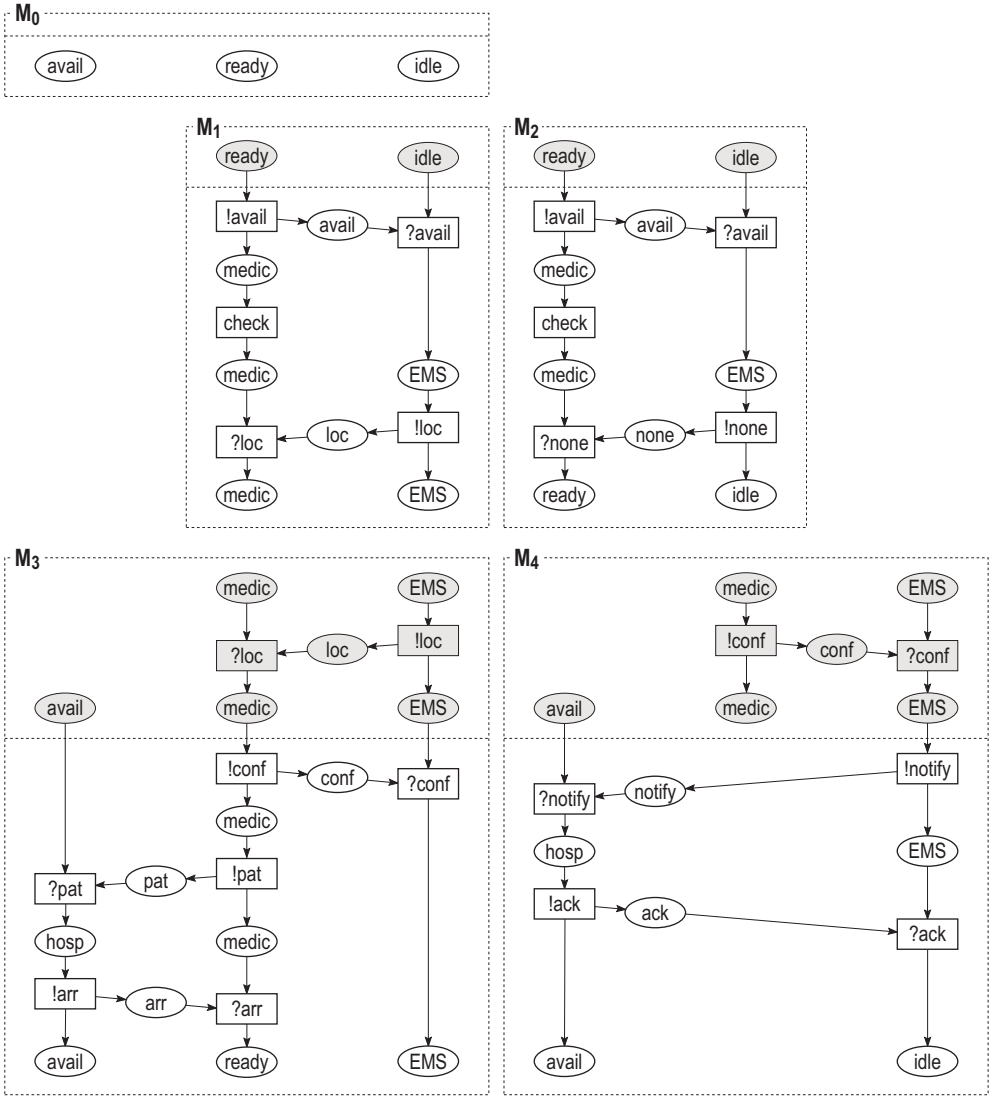
> For every k-bounded oclet specification O and every component specification $(K_i)_{i=1}^r$ of O holds: the synthesized components $(N_i, m_i, I_i)_{i=1}^r = \text{Syn}(O, (K_i)_{i=1}^r)$ implement $(K_i)_{i=1}^r$ and the composed system $\Sigma = (N_1, m_1) \oplus \ldots \oplus (N_r, m_r)$ exhibits the behavior $R(\Sigma) = \hat{R}(O)$. (8.5)

The behavior $\hat{R}(O)$ is the behavior of a minimal implementation of O, as discussed in Section 6.6. Thus, Syn solves the component synthesis problem.

Synthesis example

The following example illustrates our solution. Figure 8.10 on page 253 shows a specification in the syntax of MSCs. This specification is a variant of the emergency management example used throughout Chapters 3 and 4. The original specification presented there is *unbounded*; an implementation cannot be synthesized automatically. In contrast, the specification depicted in Figure 8.10 is 1-bounded. Figure 8.11 shows the corresponding oclet specification and a component specification. We abbreviated some of the names for conciseness.

Applying the synthesis algorithm Syn returns the components depicted in Figure 8.12. Components medic and EMS are fairly obvious whereas component hospital is more interesting. Component hospital implements the interaction with



component specification:

$$
\begin{aligned}
K_{\mathsf{medic}} &= \{\mathsf{medic}, \mathsf{ready}\} \cup \{\mathsf{!avail}, \mathsf{?loc}, \mathsf{?none}, \mathsf{!conf}, \mathsf{!pat}, \mathsf{?arr}\} \\
&\quad \cup \{\mathsf{avail}, \mathsf{loc}, \mathsf{none}, \mathsf{conf}, \mathsf{pat}, \mathsf{arr}\} \\
K_{\mathsf{EMS}} &= \{\mathsf{EMS}, \mathsf{idle}\} \cup \{\mathsf{?avail}, \mathsf{check}, \mathsf{!none}, \mathsf{?confirm}, \mathsf{!notify}, \mathsf{?ack}\} \\
&\quad \cup \{\mathsf{avail}, \mathsf{none}, \mathsf{conf}, \mathsf{notify}, \mathsf{ack}\} \\
K_{\mathsf{hospital}} &= \{\mathsf{hosp}, \mathsf{avail}\} \cup \{\mathsf{?pat}, \mathsf{!arr}, \mathsf{?notify}, \mathsf{!ack}\} \\
&\quad \cup \{\mathsf{pat}, \mathsf{arr}, \mathsf{notify}, \mathsf{ack}\}
\end{aligned}
$$

Figure 8.11. Specification of an emergency management system, adapted from Section 4.6, and a corresponding component specification.

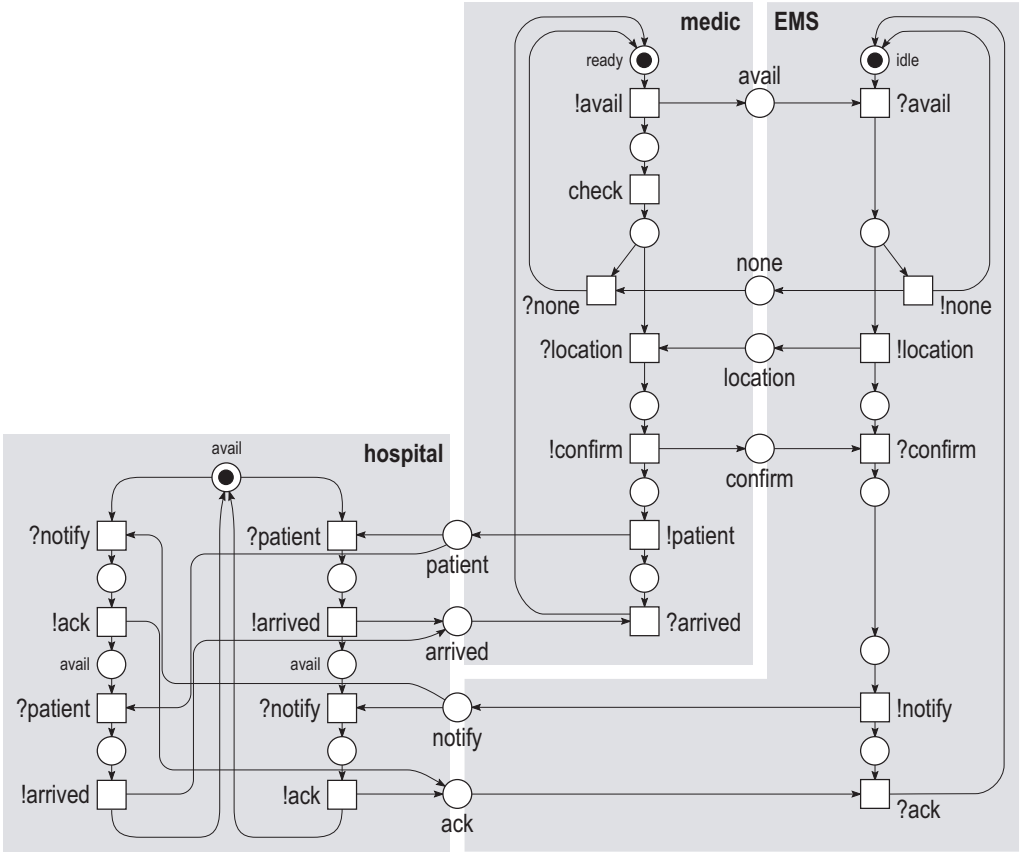


Figure 8.12. Synthesized Petri net Σ from the specification in Figure 8.11. Σ consists of three components hospital, medic, and EMS.

the medic and the EMS as two separate sequences. Both sequences differ in the order in which the hospital interacts with both components. The two explicit sequences follow from the specification. Only two explicit sequences implement that the hospital receives and acknowledges exactly one patient and one notification per location in arbitrary order. In other words, the hospital remembers that it already received a notification, but not yet a patient. This way, correctness and minimality of the component hospital does not depend implicitly on the other components medic and EMS. Instead, the hospital explicitly guarantees correctness and minimality for "its part" of the specification by design.

Parameters to the synthesis

Our solution Syn defines a family of synthesis algorithms because it has two parameters that can be chosen by a system designer regardless of the specific input O.

1. Algorithm 7.30 uses an adequate order $\prec$ on the configurations of *Fin* to determine the cut-off events. There are several adequate orders available as discussed in Section 7.5.2. Two simple adequate orders are $C_1 \prec_a C_2$ iff $C_1 \subset C_2$, and $C_1 \prec_b C_2$ iff $|C_1| < |C_2|$.
2. Algorithm 7.30 also defines a standard future equivalence $\sim$: $e_1 \sim e_2$ whenever e_1 is a cut-off event detected by the algorithm and e_2 is an event of *Fin* s.t. $[e_1]$ and $[e_2]$ have the same characteristic history. The future equivalence can be made coarser, i.e., making more events equivalent which results in smaller synthesized nets as shown in Section 8.2.

Each pair of an adequate order and a future equivalence defines a different algorithm. The synthesized Petri nets only differ syntactically.

Complexity

The core of the synthesis algorithm is the computation of the finite complete prefix. This step is also the dominating factor for the space and running time complexity of the synthesis. Despite its exponential worst-case complexity, the algorithm is known to have polynomial space and time complexity in most cases.

Decomposing oclet specification O into its oclet system Ω_O does not have to be done structurally: the structural information of a basic oclet $o[e]$ of O, where $[o] \in O$ and e a contributed event of o, is equivalently represented in o. Thus, the prefix of Ω_O can be computed directly from O.

According to Section 7.5.3, computing the finite complete prefix *Fin* of a 1-bounded oclet specification O takes time $\mathcal{O}(|E_{con}| \cdot R^{\xi+1} \cdot |E_O| \cdot |hist_{\max}|^2)$, where

- E_O are all events in the oclets O,
- E_{con} are the contributed events in the oclets O (corresponding to the number of basic oclets in Ω_O),
- R is the number of reachable histories in the oclet system Ω_O of O, i.e., the size of the state-space of O,
- ξ is the size of the largest pre-set or post-set of the contributed events in O, and
- $hist_{\max}$ is the largest local history of an event in O (corresponding to the size of the largest basic oclet in Ω_O).

The prefix has at most $\mathcal{O}(\xi \cdot R)$ conditions and non-cut-off events, and at most $\mathcal{O}(|E_{con}| \cdot R^{\xi})$ cut-off events. In experiments, usually $\mathcal{O}(R)$ cut-off events are observed, see [38] and Section 7.7.

The future equivalence $\sim$ can be computed on the fly together with the finite prefix *Fin*. Folding *Fin* to Σ requires merging the post-conditions of equivalent events. This requires at most one search over the entire prefix *Fin*. Thus, for 1-bounded oclet specifications folding has worst-case complexity $\mathcal{O}(\xi \cdot R) + \mathcal{O}(|E_{con}| \cdot R^{\xi}) = \mathcal{O}(|E_{con}| \cdot R^{\xi})$ and complexity $\mathcal{O}(\xi \cdot R) + \mathcal{O}(R) = \mathcal{O}(\xi \cdot R)$ is typically observed in experiments.

Parameter ξ indicates the degree of synchronization observed in the system, i.e., the largest number of tokens consumed or produced by a transition. ξ is usually small in practice (having values 2 or 3) [36]. If we consider ξ as a constant, then

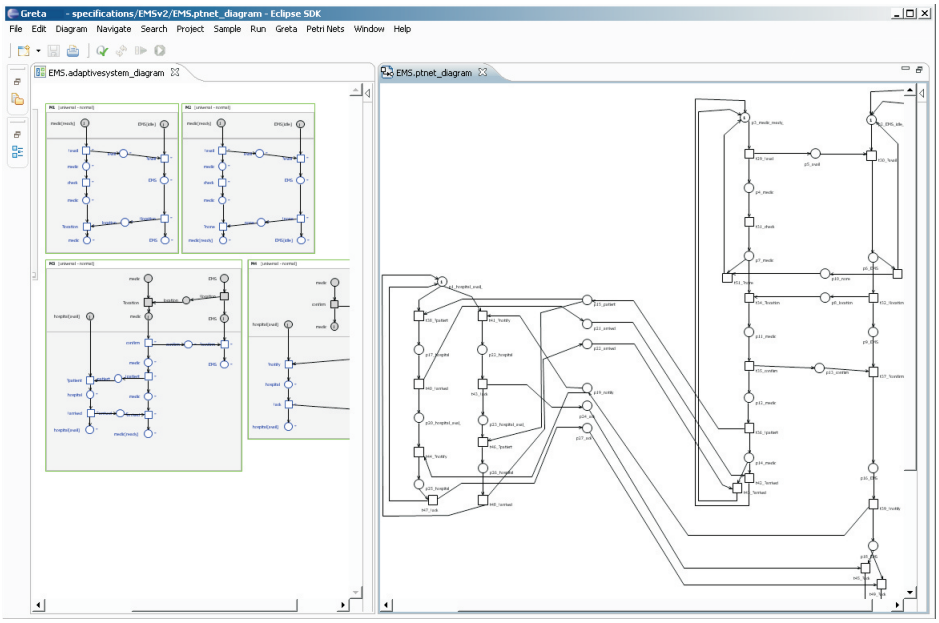


Figure 8.13. GRETA allows to automatically synthesize Petri nets from oclet specifications. The synthesized Petri net can be edited and refined.

the synthesis of O has *polynomial* complexity in the size of O's state-space (the number of reachable histories R).

8.3.4. Tool support

We implemented the synthesis algorithm from oclet specifications to Petri nets in our prototype tool GRETA, which we introduced in Section 4.6.4.

The implementation of the synthesis algorithm extends implementation of unfolding algorithm described in Section 7.7 as follows. We identify the future equivalence $\sim$ according to (8.4) during the construction of the finite complete prefix. Then, we fold the constructed prefix by $\sim$ according to Definition 8.6. The decomposition into components is not implemented yet, though the extension would be straight forward. The graphical user interface of GRETA allows to edit the synthesized Petri net as desired. Figure 8.13 shows GRETA's user interface with the oclet specification of Figure 8.11 and the synthesized Petri net side by side.

The performance results of Section 7.7 also hold for the synthesis algorithm because the additional folding step has only polynomial complexity in the size of the constructed prefix.

8.4. Discussion

The preceding chapter defined an algorithm for synthesizing an implementation from a scenario-based specification. The implementation is a Petri net that exhibits the least behavior that satisfies the specification and can be implemented by a distributed system. The synthesis algorithm preserves the locality of actions and their causal dependencies, so the implementation can be canonically distributed into components as specified. The entire approach is based on Petri nets and generalizes existing techniques from Petri net theory. In the following, we discuss our approach wrt. other synthesis algorithms, its limitations and possible extensions.

8.4.1. Synthesis

The synthesis problem for scenarios has been studied extensively in various works. Several synthesis algorithms exist for various kinds of input (the scenario notation) and output (the kind of system model to synthesize). Amyot and Eberlein [8] as well as Liang et al. [80] compare existing synthesis solutions in surveys. Our discussion concentrates on similarities with and differences to oclets.

Existing approaches

Four kinds of approaches to synthesis have been developed: integration of scenarios by common conditions, by common events, by composition of scenarios, and by behavioral semantics. These approaches are sketched in the following, Table 8.1 summarizes our discussion wrt. synthesis input (defined inter-scenario relations), output (inferred inter-scenario relations), and applied inference and synthesis techniques. Table 8.2 compares the techniques wrt. the quality of the synthesis result and the integration of the synthesis with verification techniques.

A *condition-based synthesis* assumes that a system designer explicitly defines conditions in the scenarios. All scenarios are merged pairwise along conditions with the same label. The merged scenarios describe the system model in which a condition describes a local state and an event describes a local transition of a component, see [126, 68, 73].

An *event-based technique* merges all scenarios of a specification along joint events. Technically, the techniques canonically infer pre- and post-conditions of the scenario's events. Then events with matching pre- and/or post-condition are merged as in the condition-based approach, see [70, 83, 84]. This approach originates in program synthesis from example computations and requires backtracking [15]. The technique of Mitchel et al. [90] also works on incomplete specifications.

Structured specifications like HMSCs require a *composition-based technique*. Each scenario is decomposed into *basic component models* by projection. States are synthesized as in the previous approaches. Then, all basic models of each component are composed along the structure of the specification, see [79, 21, 85, 129, 108, 86]. Composition- and condition-based techniques can be combined [113, 34] and extended [110].

In contrast to the preceding three techniques, a *semantics-based technique* infers an appropriate system model by the scenarios' behavioral semantics. Alur et al.

	inter-scenario relations		
	defined	inferred	inference- // synthesis technique
condition-based, MSCs	conditions		merge
event-based, MSCs	events	states	exploration, state-splitting, back tracking // merge
composition-based, HMSCs	high-level structure, composition operators		compose
semantics-based			
MSCs, [5]		states	MSC traces (finite) // automata construction
LPOs, [13]	LPOs of events (reg. expressions)	Petri net places	region theory // ILP problem
LSCs, [53]	history	states, transitions, coordinating messages	exploration by play-out, pruning // automata construction, state charts
LSCs, [19]	history	strategy per component	decomposition to events, two-player games // strategy synthesis
LSCs, [17]	history	states, transitions	decomposition, exploration by play-out, pruning // I/O automata construction
oclets	history	Petri net places, transitions	history-preserving decomposition, exploration by play-out, local equivalence // folding to Petri net

Table 8.1. Comparison of synthesis techniques from scenarios.

		synthesis result		
	overlapping scenarios	local/global	proved minimal/complete	verification during synthesis
condition-based, MSCs		local if conditions local		
event-based, MSCs		local if inferred conditions are local	both, by [15]	
composition-based, HMSCs	[90]	depends on composition operators		yes, technique unknown [113]
semantics-based				
MSCs, [5]	yes (on prefixes)	local	both	polynomial algorithm
LPOs, [13]	only finite scenarios	local, may implement additional dependencies	both, by region theory	
LSCs, [53]	yes	local, adds coordinating messages	both	symbolic model checking
LSCs, [19]	yes	global strategy for all components	both	yes, technique unknown
LSCs, [17]	yes	local	minimal, incomplete	symbolic model checking
oclets	yes	local	both	yes, Chapter 7

Table 8.2. Quality of synthesis results and system verification during synthesis.

synthesize concurrent state machines from finite MSCs along the traces accepted by the MSCs [5]. Desel et al. [13] synthesize Petri nets from scenarios using Petri net region theory: a specification S is a regular expression over labeled partial orders. The synthesis maps each action in S to a transition and synthesizes places and arcs that restrict the transitions according to S. The behavior of the synthesized system is minimal.

For the synthesis from scenarios with histories, the key challenge is to coordinate the start of a scenario across different components. There are techniques to synthesize automata from LSCs [49] and "High-Level LSCs" (corresponding to HMSCs) [16]. The most recent techniques synthesize components along the operational semantics of LSCs defined in play-out: scenarios are coordinated across components either by auxiliary activation messages [53], or by global strategies for all system components [19]. Bontemps and Heymans [17] decompose LSCs prior to synthesizing an implementation for each component by play-out; the decomposition trades efficiency for completeness.

Relation to oclets and conclusion

The approach by Bontemps and Heymans [17] is the closest to oclet synthesis. In direct comparison, oclet synthesis preserves an event's *history* throughout the synthesis and obtains an exact result: each oclet decomposes equivalently into its events (instead of its components), the event's local history preserves when the event may occur (where Bontemps and Heymans discard the history outside of the component). Consequently, events with local history compose equivalently to satisfying behavior. Oclet synthesis implements the events' histories structurally, i.e., by splitting places and transitions. Splitting of places and transitions follows monotonically from the notion of a history and requires no backtracking like in event-based synthesis.

The oclet-based approach may also apply beyond scenario-based design: Hee et al. [121] translate history-dependent Petri nets to equivalent history-free Petri nets by adding a global coordinating Petri net place that records the history of the system. The oclet-based approach with local histories may help preserving the concurrency in the system.

In terms of methodology, oclets are the first approach for *composing* an implementation from scenarios with history. Moreover, oclets consequently apply the notion of a history from the scenario syntax up to the synthesis algorithm. In other words, oclet synthesis preserves *behavioral* and *structural information* of the specification by implementing a condition by a Petri net place, and an event by a Petri net transition. This principle yields the following benefits:

- The synthesis preserves the locality of events given in the scenarios.
- The synthesis result corresponds structurally to the specification, i.e., the scenarios occur as substructures of the system model. Thus, the synthesized implementation remains readable for a system designer.
- The synthesized system is correct by construction: it exhibits the least behavior that satisfies the specification and can be implemented as a distributed system.

Oclets largely contribute to the *component synthesis problem* (8.3). Bontemps and Schobbens [18] prove that the problem is undecidable for LSC specifications and pre-defined component specifications. In contrast, oclet synthesis decides the problem constructively for all bounded oclet specifications and all component specifications.

The bounded synthesis problem (8.2) involving only one component is decidable for LSCs, but its solution is ExpTime-complete [54]. This complexity arises from the need to check consistency of the given LSC specification. Constructing a single run of an LSC specification via LSC play-out is Pspace-complete. Synthesis from an oclet specification O is polynomial in the size of the state-space of O if O is 1-bounded, cf. Section 8.3. However, the technique is known to have good average case complexity in practice as our results in Section 7.7 confirm.

Decidability and better complexity of oclet synthesis holds mainly due to the restricted expressive power of oclets compared to LSCs. Though the question remains whether this additional expressive power of LSCs is necessary. The synthesis algorithm of Section 8.3 is *complete* wrt. the class of systems we want to synthesize: every (bounded) Petri net system is the synthesis result of some oclet specification because every Petri net system has an equivalent oclet specification, see Section 5.5. Thus, the kernel of scenarios formalized in oclets is expressive enough to specify any distributed system that can be modeled as a bounded Petri net. Moreover, we can *synthesize* any bounded Petri net system. This completeness result has not been established for existing synthesis techniques. Thus, we succeeded in the main goal of this thesis: to find a general solution to the synthesis problem for a flexible and sufficiently expressive class of scenario-based specifications.

A final note on conditions: most of the discussed synthesis approaches introduce conditions into scenarios for the purpose of synthesis. Oclets include conditions from the very beginning explicating the need for this notion for synthesis. Various examples in this thesis show that conditions in oclets can be hidden by "syntactic sugar" like MSC syntax: most conditions follow canonically from events and the notion of a component.

8.4.2. Improving the synthesis

Input. Because of the undecidability result in Section 8.1, we restricted the synthesis problem to bounded specifications which yields bounded nets. There is room for improvement: Desel et al. [29] generalize McMillan's technique to an algorithm for computing a finite complete prefix of an unbounded Petri net system. This technique could help in generalizing oclet synthesis to some classes of unbounded oclet specifications. Petri net region theory might help characterizing oclet specifications that can be implemented as an unlabeled Petri net system. [26]

Output. Whether the synthesis algorithm of Section 8.3 returns a structurally minimal implementation Σ depends on a parameter of the synthesis algorithm: the *adequate order*. Several adequate orders are available [38, 24]. The theory of regions may help improving the synthesis results as well. The approach of Desel et

al. [13] could be applied on the finite complete prefix of an oclet specification to synthesize a Petri net system Σ' with a minimal number of transitions. Though, Σ' may not exhibit the same distributed runs as O, but only the same partial orders of events (see Section 2.3 for a discussion).

8.4.3. Relation to synthesis of open systems

Our solution for synthesis from scenarios only applies to *closed* systems. A closed distributed system Σ consists of components $\Sigma_1 \oplus \ldots \oplus \Sigma_r$ which interact with each other — and with nothing else.

In contrast, an *open* system Γ has an *interface* I via which it may communicate with its *environment* E by receiving messages from E, sending messages to E, or waiting for a synchronous step together with E. Put differently, an open system is a *component* according to Definition 8.10. Γ may itself consist of components, and is typically viewed as waiting for inputs from its environment and reacting to it by outputs. For this reason, Γ is also called a *reactive* system.

The composition of Γ with its environment would yield a closed system $\Gamma \oplus E$. The assumption, however, is that the behavior of E is not known. So, the behavior of a closed system Σ is completely defined by the states and steps of Σ whereas the behavior of an open system Γ is only partially determined by Γ and depends on the behavior of the unknown environment E.

A different synthesis problem

The *synthesis problem for open systems* resembles the component synthesis problem (8.3) with a notable addition:

> Given a component specification $(K_i)_{i=1}^r$, and a behavioral specification *Spec* over messages channels and actions in the interfaces of the K_i, construct components $\Gamma_1, \ldots, \Gamma_r$ s.t. under *arbitrary behavior of the environment* E, the composed systems $\Gamma_1 \oplus \ldots \oplus \Gamma_r \oplus E$ satisfies *Spec*. (8.6)

The problem of synthesizing open systems is due to Pnueli and Rosner [99] who were the first to consider "arbitrary behavior of the environment." Usually, the specification *Spec* is a temporal logic formula in which atomic propositions describe messages on interface channels or occurrences of actions in the interfaces. A variant of the problem is to construct a *controller* that restricts the behavior a given open system s.t. the specification is satisfied [81, 82].

The synthesis problem for open systems (8.6) is surprisingly hard. The problem is undecidable for (propositional) LTL specifications, even for two components that only communicate with the environment and not with each other. The problem is decidable for tree-like architectures, and a solution can be constructed in double exponential time in the size of the specification [98]. Several decidable sub-classes of the problem have been identified, for instance, by restricting specifications to subclasses [97] or other architectures [75]. Other solutions consider variants, where each component only knows some of the inputs provided by the environment [74],

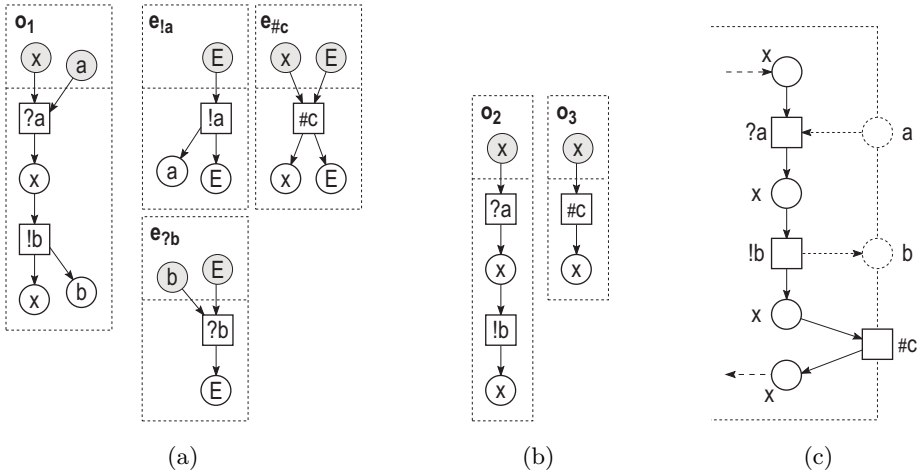


Figure 8.14. Oclets to describe interaction behavior (a) explicitly or (b) implicitly. In a synthesized Petri net, implicit interaction can be made explicit (c).

or enforce a component's reaction to follow only from the component's inputs [82]. LSCs also allow to specify open systems interacting with an environment. Synthesis of open systems from LSCs has been studied in this respect as well [49, 19], and is undecidable [18].

Oclets and synthesis of open systems

Describing an unconstrained environment with oclets. In the following, we discuss how oclets relate to open systems and the corresponding synthesis problem (8.6). We can use oclets such as $\mathsf{o_1}$ of Figure 8.14(a) to describe how a component x interacts with its environment. Oclets $\mathsf{e_{!a}}$ (send message a), $\mathsf{e_{?b}}$ (receive message b), and $\mathsf{e_{\#c}}$ (synchronize with x on action c) describe corresponding environment actions. The oclets $\mathsf{e_{!a}}$, $\mathsf{e_{?b}}$, and $\mathsf{e_{\#c}}$ actually describe the unrestricted environment E of x: the environment has a single state E and may initiate any interaction in each of its steps.

Each oclet specification O_{Spec} of a system Γ having oclets such as $\mathsf{o_1}$ has a canonical oclet specification O_E that describes the behavior of the unrestricted environment of Γ. The composed specification $\mathsf{O}_{Spec} \cup O_E$ describes the behavior of $\Gamma \oplus E$.

Synthesis with an unconstrained environment. By including O_E in the specification, we can describe the synthesis problem for an open system Γ (8.3) as a component synthesis problem of the closed system $\Gamma \oplus E$ (8.3). The synthesized system would contain a component E describing the behavior of the environment.

Though, our synthesis algorithm does not solve every instance of the problem. If Γ may receive a message a from E, then O_E contains oclet $\mathsf{e_{!a}}$ of Figure 8.14, which makes $O_{Spec} \cup O_E$ *unbounded*. Thus, we cannot apply our synthesis algorithm of

Section 8.3. Whether the synthesis problem is undecidable in this case, depends on whether unbounded oclet specifications have a finite complete prefix of some form.

If Γ interacts only *synchronously* with E, then O_E only contains oclets such as $\mathsf{e}_{\#\mathsf{c}}$ of Figure 8.14(a) and $O_{Spec} \cup O_E$ is bounded iff O_{Spec} is bounded. In this case, our synthesis algorithm can be applied. The corresponding problem in LSCs, to synthesize components which synchronously interact with an environment as specified, is undecidable [18].

Synthesis of a fast system. To also synthesize open systems with asynchronous communication, we have to adjust the setting. Our behavioral model of distributed systems is inherently asynchronous: the system Γ cannot react on every step of the environment E and receive each message as soon as it is available. Especially if E sends messages "faster" than Γ can receive them. Conversely, by assuming that Γ can receive messages "faster" than E sends new messages, the problem vanishes: under this assumption, E produces a new message a whenever Γ received the previous a message but not earlier. A similar assumption is made in LSCs regarding how the system reacts to environment inputs [51].

The oclets o_2 and o_3 of Figure 8.14(b) describe the interaction behavior of component x under the assumption that x reacts faster than its environment. Whenever x enables action $?\mathsf{a}$, the environment already provided a message a (maximality of the environment), and because there is no oclet $\mathsf{e}_{!\mathsf{a}}$ of the environment there will be no pending message a when x is not ready to receive it (x reacts faster than the environment).

An oclet specification O_{Spec} of a system Γ having oclets such as o_2 and o_3 describes the interaction behavior of Γ with a maximal environment E in which Γ reacts faster than E. Thus, O_{Spec} is bounded iff the interaction of the components of Γ is bounded. When synthesizing Γ from O_{Spec} according to Section 8.3, we obtain Petri net components as depicted in Figure 8.14(c). The implicit receiving and sending of messages (by transitions $?\mathsf{a}$ and $!\mathsf{b}$) can be made explicit with corresponding places that describe the message channels.

Synthesis with acceptance conditions. The synthesis from $O_{Spec} \cup O_E$ with explicit communication or O_{Spec} with implicit communication itself is of little interest, because an oclet specification O_{Spec} does not allow to express an *acceptance condition* of the system, for example, "component x *eventually* responds to each received message a with a message b." Such an acceptance condition naturally arises when a system designer specifies the interaction behavior of an open system [74]. To describe practically relevant acceptance conditions we would have to (1) extend the semantics of oclets, which would be an exercise in formal definitions, and (2) our synthesis algorithm likewise, which is a non-trivial problem. We propose a corresponding research programme as future work in Section 9.3.

9. Conclusion

This final chapter concludes this thesis. Section 9.1 summarizes the main contributions of our approach for relating scenario-based specifications to distributed systems. Section 9.2 discusses open problems. Finally, Section 9.3 sketches ideas how this research can be continued.

9.1. Contributions of this Thesis

The scenario-based approach is a technique for designing a distributed system consisting of several components. A *scenario* describes a "self-contained story" about how several system components interact to realize a specific goal. Technically, a scenario is partial order of actions and each action is associated to one or more components. A specification is a set of mutually related scenarios. A system implements a specification if its components interact as described in the scenarios. The main challenge for the scenario-based approach is to construct an implementation from a given specification — preferably automatically. More precisely, the *synthesis problem* is to define an algorithm that constructs for every scenario-based specification *Spec* system components $\Sigma_1, \ldots, \Sigma_r$ that are described in *Spec* so that the composed system $\Sigma_1 \oplus \ldots \oplus \Sigma_r$ is a minimal implementation of *Spec*.

The synthesis problem is difficult: the high expressive power of a scenario-based specification renders the synthesis problem undecidable whereas decidable subclasses of scenario-based specifications render the approach inflexible in practical settings.

Specifically, one of the most expressive and flexible scenario-based techniques are Live Sequence Charts (LSCs) [25]. LSCs draw their flexibility from the notion of a *history* of a scenario that describes when the scenario may occur in the system. Several scenarios may *overlap*. Mutual relations between scenarios emerge from the relation between a scenario's history and other scenarios. The synthesis problem from LSCs has no general solution [18]. In contrast, the synthesis of a Petri net from a scenario-based specification succeeds by applying results from Petri net theory to scenarios [13]. A drawback of existing complete solutions of the synthesis problem is a restriction on the input: scenarios are either finite or composition operators explicitly define mutual dependencies between scenarios.

The problem that we addressed in this thesis was to strike a balance between a flexible and sufficiently expressive scenario-specification technique on one hand and efficient synthesis and analysis algorithms on the other hand. We restricted ourselves to specifying, analyzing, and synthesizing the *control-flow* of distributed systems.

This thesis contributed a formal theory that positions scenarios as a binding element between system models and system behavior. The new theory combines the theory of Petri nets with the theory of history-dependent scenarios in the notion of an *oclet*. An oclet is an acyclic Petri net with a distinguished *history*. This notion subsumes several existing ones: an oclet with an empty history describes a distributed run, an oclet with a single transition and history of depth 1 generalizes a Petri net transition, and any other oclet describes a scenario. Each of these shapes of an oclet is obtained from the other shapes by composition and decomposition.

On one hand, oclets are a flexible specification technique for distributed systems in the style of LSCs. On the other hand, this thesis contributes an algorithm that solves the synthesis problem for oclets. The following sections review the contributions that follow from the theory of oclets, Figure 9.1 shows the overall picture.

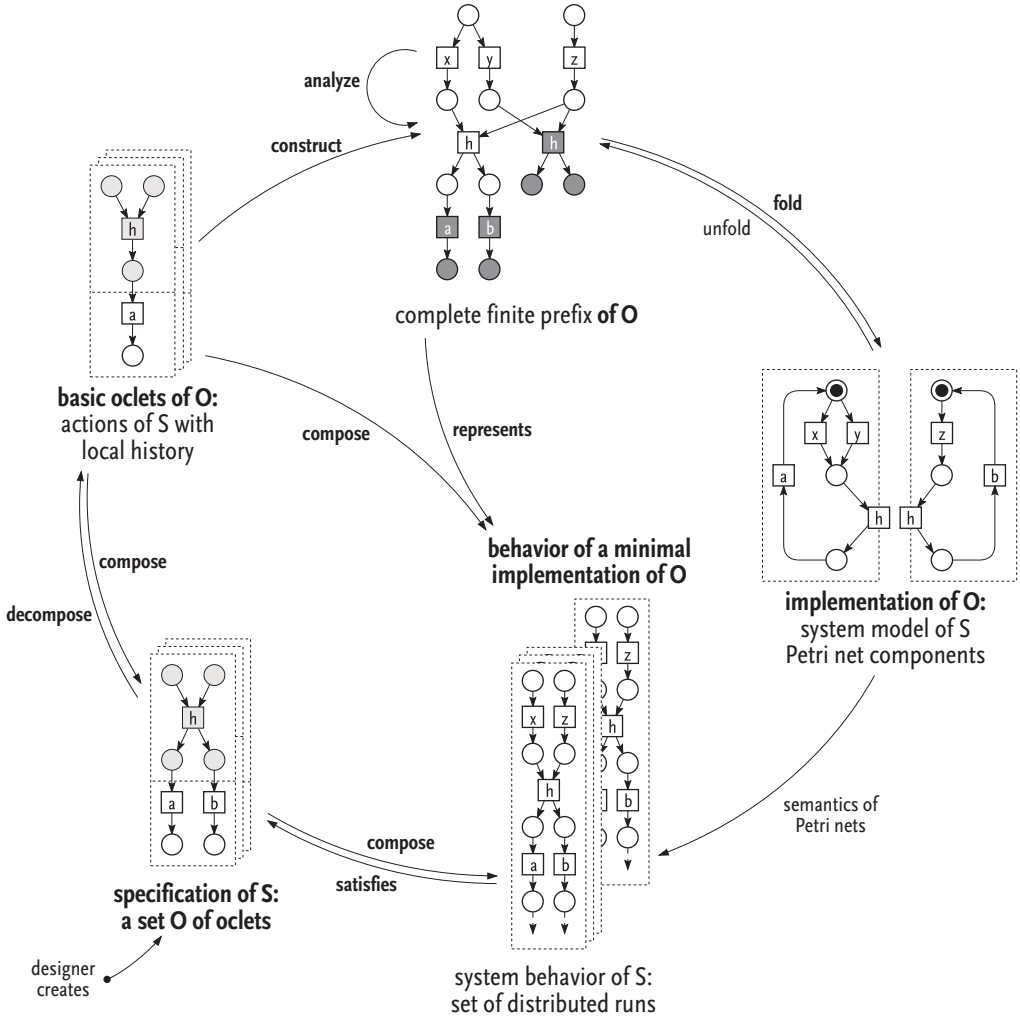


Figure 9.1. Overview on the results of this thesis.

9.1.1. Oclets describe scenarios

Oclets contribute a formal theory for relating scenario-based specifications to distributed systems. To this end, oclets inherit a few notions from scenarios and from Petri nets. An oclet formalizes a scenario of a system — a behavior that the system shall exhibit — as a partially ordered set of actions denoted by an acyclic Petri net. Corresponding to Live Sequence Charts (LSCs), an oclet distinguishes a *history* denoting *when* the system shall exhibit the oclet's behavior. Petri nets provide the formal syntax of oclets together with the notions of *local* actions and *local* states that are ordered by *causality*.

A kernel of scenario-based specifications. Chapter 4 defined a *declarative* semantics that positions oclets in the spectrum of scenario-based specification techniques. An *oclet specification* is a set of oclets. A set R of (distributed) runs *satisfies* an oclet o if every run of R that ends with o's history has a continuation with the entire oclet o. The runs R satisfy a specification O if R satisfies each oclet of O.

For composing specifications, oclets adapt a principle from logics. One oclet o denotes a behavioral property $R(o)$, i.e., the system behaviors that satisfy o. The semantics of an oclet specification $O = \{o_1, \ldots, o_n\}$ is the *intersection* of the semantics of its oclets, $R(O) = R(o_1) \cap \ldots \cap R(o_n)$. This composition principle corresponds to conjunction in logics.

The semantics of oclets formalizes a *kernel* of scenario-based specifications that we identified in Chapter 3. This kernel uses a *minimal set of notions* from existing techniques. It specifically allows for specifying system behavior in a flexible way using the notion of a *history*. Yet, oclets provide just enough expressive power for specifying *all* behaviors of a distributed system: every Petri net has an *equivalent* oclet specification. The semantics of oclets are based on *distributed* runs which makes oclets the first scenario-based technique with history having a true-concurrency semantics.

All other contributions of this thesis follow from the combination of distributed runs with histories formalized as Petri nets.

Composing and decomposing scenarios. Oclets provide canonical operators for composing and decomposing oclets. Two oclets o_1 and o_2 *compose* to a larger oclet $o_1 \rhd o_2$. Conversely, an oclet o *decomposes* into its single actions $o_1, \ldots, o_k$ with a local history; composing $o_1, \ldots, o_k$ yields o again.

Oclets generalize the semantic notions of Petri nets: an oclet with an empty history describes a distributed run of a Petri net; an oclet with a single action describes a Petri net transition; an oclet's history generalizes the enabling condition of a Petri net transition.

The composition and decomposition operators also establish a number of semantic results which we developed in Chapters 5 and 6. Every oclet specification O defines has a unique least set $\min \mathcal{R}(O)$ of distributed runs that satisfies O; the composition operator $\rhd$ constructs $\min \mathcal{R}(O)$. An implementation of O may inevitably exhibit more behavior than $\min \mathcal{R}(O)$. However, O defines the unique behavior $\hat{R}(O)$ of any *minimal implementation* of O. The composition operator $\rhd$ constructs the set $\hat{R}(O)$ of distributed runs from the single actions of O starting in the *least initial state* of O. $\hat{R}(O)$ contains $\min \mathcal{R}(O)$, the additional runs $\hat{R}(O) \setminus \min \mathcal{R}(O)$ are known as *implied* scenarios. The semantics of oclets reveals the difference between satisfying behavior $\min \mathcal{R}(O)$ and implied behavior $\hat{R}(O) \setminus \min \mathcal{R}(O)$ as the difference between global scenarios and their decomposition into single actions.

By these results, oclets transfer the benefits of scenario-based specifications to Petri nets, and vice versa.

9.1.2. Oclets model distributed systems

Operational semantics for scenarios. The composition and decomposition operators on oclets define *operational semantics* for oclet specifications O that are contributed in Chapter 6. The oclets of the specification canonically decompose into their single actions $o_1, \ldots, o_k$. The composition operator $\triangleright$ constructs the distributed runs of O action by action: an action o_i is *enabled* at a distributed run π if π ends with the local history of o_i; in this case o_i can *occur* which yields the continuation $\pi \triangleright o_i$. The operational semantics turn an oclet specification into a *model* of a distributed system.

The operational semantics of oclets adopt the idea of *scenario play-out* from LSCs in terms of Petri nets and their true-concurrency semantics. The operational semantics of oclets constructs for each oclet specification O the behavior $\hat{R}(O)$ that is also exhibited by any minimal implementation of O. This combination yields two contributions: (1) oclets allow modeling distributed systems in a flexible manner, and (2) efficient verification techniques from Petri nets are made available for scenario-based specifications.

Flexible system modeling. The operational semantics generalize the operational semantics of Petri nets by the notion of a *local history* of an action. On one hand, Petri nets are embedded into oclets and each Petri net can equivalently be translated into an oclet specification. On the other hand, a system designer can model system behavior by describing scenarios instead of a system model. Combining both ideas allows for a novel and flexible style of system modeling.

A system designer only has to describe the scenarios of the system. Each scenario describes a specific self-contained story of the system, two scenarios may *overlap*. The operational semantics derive the system's behavior without requiring the system modeler to structurally compose or integrate the scenarios. Consequently, the system designer can structure the system model *behaviorally*, for example, by distinguishing scenarios for standard behavior and for exceptional behavior. If the system model needs to be changed to implement a new behavioral requirement r, then a system designer may identify and change only those scenarios related to r, all other scenarios remain untouched.

Efficient verification of scenario-based models. Oclets allows for modeling complex system behavior in a structured approach. Yet, the modeled system's behavior may not satisfy all requirements a system designer has in mind. Such requirements may be formalized, for example in terms of temporal logics. It needs a verification technique for ensuring that the modeled system's behavior satisfies all formalized requirements.

The embedding of Petri nets into oclets allows to adapt efficient verification techniques from Petri nets to scenario-based models. Chapter 7 generalizes the technique of *finite complete prefixes* by McMillan to oclets. A finite complete prefix *Fin* of an oclet specification O represents the behavior $\hat{R}(O)$, which is constructed by O's operational semantics, in a finite structure — even if $\hat{R}(O)$ is infinite. This thesis defines an algorithm that constructs *Fin* by identifying all

reachable states of O in terms of the histories of the oclets in O. *Fin* represents each reachable state and each occurring action of O. In other words, *Fin* represents the state-space of O. The McMillan technique mitigates the state-space explosion problem by preserving concurrency of actions — instead of imitating concurrency by exponentially many interleavings.

Consequently, oclets provide a state-space construction algorithm to scenario-based models that exhibits polynomial running time for practical systems. Chapter 7 also presented results of an industrial case study in which oclet-based verification checked the correctness of industrial business process within a few milliseconds per process.

Thus, by the theory of oclets, McMillan's technique becomes available to scenario-based models. It is the first time that McMillan's technique has been applied in this domain.

9.1.3. Oclets synthesize distributed systems from scenarios

The main problem addressed in this thesis is to automatically synthesize from a scenario-based specification components of a distributed system that together implement the specification and exhibit as few additional behavior as possible. More precisely,

> to define an algorithm Syn that, given a scenario-based specification O and a distribution of the actions in O onto components $1, \ldots, r$, constructs components $\Sigma_1, \ldots, \Sigma_r$ that implement the respective actions of O s.t. the composed system $\Sigma_1 \oplus \ldots \oplus \Sigma_r$ is a minimal implementation of O.

The problem has no general solution and is undecidable — in standard techniques like Live Sequence Charts (LSCs) or High-Level Message Sequence Charts (HMSCs) as well as in oclets.

Chapter 8 contributes an efficient synthesis algorithm Syn for *bounded* specifications. In a bounded specification, a component sends only a bounded number of messages between any two messages which it receives. The algorithm is based on the finite complete prefix *Fin* of an oclet specification O which represents all behavior of O in a finite structure. The synthesis folds *Fin* into an equivalent Petri net Σ using a simple equivalence relation on the nodes of *Fin*. The constructed system Σ is a *minimal* implementation of O. Moreover, Σ implements exactly those local states and local actions specified in O. Consequently, Σ decomposes into any number of components given by any distribution of its actions.

Bounded specifications. The assumption of a bounded specification also renders HMSCs synthesis decidable. Synthesis succeeds also from some unbounded specifications which satisfy certain properties [26]. The disadvantage of HMSC-based synthesis is that its input already explicitly specifies the relations between scenarios by explicit composition operators. With the available standard composition operators, not every system can be specified [13].

The notion of a bounded specification is not directly applicable to LSCs. Clearly, if an LSC specification describes a finite state space, then synthesis is decidable. However, the undecidability proof in [18] only requires *synchronous communication* over a *fixed number* of components, i.e., synthesis from 1-bounded LSC specifications is undecidable. We are not aware of a sufficient criterion for synthesis from LSCs that characterizes finite state spaces similar to the notion of boundedness in oclets.

Contribution of oclets to synthesis. Within the domain of specifications that describe a finite space, oclets make the following contributions compared to LSCs and HMSCs.

Regarding the input, a system designer may specify any finite set of oclets. The relations between oclets follow from their histories and does not have have to be given explicitly like in HMSCs. Further, oclet-based synthesis also allows to specify any choice of components as synthesis result. In contrast, LSCs allow to decide whether a specification with finite state space can be implemented for a given choice of components, but the answer may be "no".

Regarding the output, the specification language of oclets is expressive enough to specify every Petri net. So, the synthesis algorithm may return every bounded Petri net. This completeness result does not hold for HMSCs [13]. In terms of expressive power, LSCs are known to specify sub-classes of Büchi automata [16, 18]; the class of synthesizable systems has not been determined yet.

The synthesis algorithm from oclets *preserves the causal relations* of the specification. So, all behavioral information of the specification is available during synthesis in its original form. Existing techniques either operate on the *structure* of the specification or on *sequential observations* of distributed behavior, as discussed in Section 8.4.

A major limitation of the synthesis from oclets is that oclets synthesize *closed systems*. We have shown in Section 8.4.3 how to specify the behavior of an *open system*. However, the limited expressive power of oclets do not allow to express interesting high-level properties of an open system as this can be done in LSCs [25].

Altogether, oclets establish a well-balanced theory for relating the flexible technique of scenario-based specifications to an adequate and well-researched formal model of *distributed systems*. The entire theory is formally grounded in basic notions of Petri nets which this thesis consistently generalized by the notion of a local history. The hope is that this generalization allows for transferring further results from the theory of Petri nets to scenario-based specifications.

9.1.4. Tool support

The results of this thesis have been implemented in the software tool prototype Greta. Greta is a plug-in based tool for which developed several plug-ins. In first place, Greta provides a graphical user interface for modeling oclet specifications. Three plug-ins extend this functionality. (1) An execution engine implements the operational semantics of oclets of Chapter 6 and allows a system designer to execute and validate an oclet specification. (2) A second plug-in implements

the algorithm for computing a finite complete prefix of an oclet specification O and for analyzing behavioral properties of O presented in Chapter 7. (3) A third plug-in implements the algorithm for synthesizing a minimal implementation from a given oclet specification developed in Chapter 8. GRETA is available at `http://service-technology.org/greta`.

We applied GRETA for modeling an actual disaster management process in our approach and evaluated its performance in an experiment using a large data sample from an industrial setting as presented in Section 7.7. There, a finite complete prefix of an oclet specification could be constructed requiring linear to polynomial running time and space in the size of the input. Constructing a finite complete prefix underlies our analysis and synthesis techniques. Thus, the experiment confirms that our solutions to analysis of scenario-based models and synthesis from scenarios can be applied in an industrial setting.

9.2. Open Problems

In this thesis, not all open problems for relating scenario-based specifications to Petri nets have been solved. This section discusses problems that would make the formal theory presented in this thesis more complete.

Structural size of the synthesized system

The synthesis algorithm defined in Chapter 8.2 constructs for every bounded oclet specification O a labeled Petri net system Σ that implements O and exhibits as few additional behaviors as possible. In other words, Σ is a behaviorally minimal implementation of O. But, Σ is not necessarily structurally minimal. The size of Σ depends on two factors: (1) the size of the finite complete prefix *Fin* constructed in Chapter 7, and (2) the equivalence relation used to fold *Fin* to Σ.

The size of *Fin* in turn depends on the *adequate order* that determines equivalent events in the construction algorithm. Several adequate orders are available usually balancing between computational complexity and size of the prefix [38, 24]. Thus, an open question is to determine adequate orders that return structurally minimal prefixes, or prefixes of a specific shape.

The equivalence relation for folding *Fin* is derived from the adequate order. An open question is whether this equivalence relation can be improved on any given finite complete prefix so that *Fin* folds to a structurally minimal implementation, or an implementation of a specific shape.

Synthesizing unlabeled Petri nets

Our synthesis algorithm returns for every bounded oclet specification O a Petri net system Σ that implements. In general, the transitions (places) of Σ are labeled s.t. there may be several transitions (places) with the same label. There are oclet specifications O which have no equivalent *unlabeled* Petri net system Σ with exactly the same set of distributed runs $R(\Sigma) = \hat{R}(O)$ as shown in Section 8.2.2. Any unlabeled Petri net system Σ that exhibits the runs $\hat{R}(O)$ would also exhibit

additional behavior $R(\Sigma) \setminus \hat{R}(O)$. Adding a new place p and arcs to Σ "in the right way" reduces this additional behavior. Technically, now p occurs in the runs of Σ; thus, the behavior of Σ and O has to be compared on their labeled partial orders of events only (see Section 2.3). Petri net *region theory* has been applied in similar settings for computing places that constrain the behavior of Σ, specifically in works by Desel et al. who synthesize Petri nets from a more restricted class of scenarios compared to oclets [12, 13].

The open research question is whether Petri net region theory can be generalized to synthesize a minimal, unlabeled Petri net that implements a given oclet specification and to characterize those classes of specifications where this is impossible.

Synthesis of unbounded Petri nets

The general synthesis problem has no solution because oclets are strictly more expressive than Petri nets. This thesis presented an algorithm that solves the bounded synthesis problem. Though, this result can be generalized by the following observation. Petri nets are embedded into oclets. Thus, even for each unbounded Petri net system Σ exists an equivalent unbounded oclet specification O. Desel et al. [29] define an algorithm for computing a complete finite prefix of an unbounded Petri net system. The open research question is whether this algorithm can be generalized to oclets to solve the synthesis for any oclet specification that can be implemented by a (possibly unbounded) Petri net system.

9.3. Further Research

This final section sketches ideas for extending the theory of oclets beyond the scope of this thesis. These ideas aim at the grand goal of systematic development of distributed systems with appropriate specification, modeling, and verification techniques.

Alleviating underlying assumptions

The formal model of oclets presented in this thesis made two assumptions about how a scenario-based specification describes behavior. First, *conditions* and events have been given equal importance in the formal model. Second, we excluded *invisible actions* from the model of oclets. In the following we discuss how these assumptions can be alleviated.

Conditions. Scenarios are a technique to specify behavior of distributed systems in early stages of design. Typically, a system's behavior is understood as a course of *events*. Knowledge about conditions, that is, visited states of the system, is mostly implicit. Put differently, a system designer is usually able to name and order events, but will face difficulties in determining whether a specific state is visited in a specific situation.

Most scenario-based techniques cater for this impartial knowledge by representing a scenario as partial order of *events* — with additional means of expression. For instance, the vertical lines that order a component's events in an MSC imply a condition between any two events. Formal semantics show that an MSC's implicit conditions can be made explicit while preserving the behavior [62, 33]. Oclets allow for a similarly implicit use of conditions. Several conditions of a specification may be labeled identically. For instance, every condition may be labeled with the name of a component as shown in Figure 4.2 on page 86. The ordering of events then follows from their histories wrt. *preceding events*. Whether two different conditions with the same label describe the same system state follows from their history. Here, the *standard ordering of events implies a canonical labeling of conditions*. When ordering of events does not follow from the linear order of a component or from message exchange, then a system designer expresses this explicitly. For instance in oclets, two post-conditions with different labels allow to express that an event has two concurrent post-events. For the same modeling task, other techniques also provide additional constructs like *co-regions* [65] or *inline scenarios* [95, 55].

The open research question is to develop *modeling practices* that support a system designer in using canonical labeling of conditions in oclets. These modeling practices may lead to a *richer syntax* for oclets, like that of MSCs. This richer syntax could hide conditions by providing syntactic sugar to express canonical ordering of events in a component, and additional constructs to express non-standard ordering.

Invisible actions. Invisible actions change the interpretation of an *arc* between two events. Now, an implementation may exhibit additional events and conditions between two specified events. In this case, a specification only describes *transitive causal dependencies* between events instead of *direct causal dependencies*. As a consequence, scenarios may overlap in more intricate ways than seen in this thesis.

We discussed extensively in Section 3.8 that invisible actions introduce *unobservable nondeterminism* to oclets. A specification could not distinguish implementations that are not branching bisimilar. Section 4.5.2 discussed how the semantics of oclets could be extended to invisible actions without introducing unobservable nondeterminism. The open research problem is the *synthesis under invisible actions*. That synthesis from LSCs is undecidable already holds for a very restricted class of LSCs: a fixed number of components communicates synchronously and *allows invisible actions* [18]. So, we also expect oclet-based synthesis to become harder under invisible actions.

A possible research programme is to generalize our synthesis algorithm to oclets that allow invisible actions. The critical aspects in this research are the following: (1) investigate whether a specification defines a unique minimal behavior when invisible actions are allowed, (2) define composition and decomposition operations that respect occurrences of invisible actions, (3) identify the notion of a reachable history under invisible actions, and (4) construct a finite complete prefix with (5) an equivalence relation that allows to fold the prefix into an implementation. These five steps constitute *a blue print for generalizing oclet-based synthesis to allow further means of expression.*

Towards the expressive power of LSCs

The formal model of oclets presented in this thesis established a minimal kernel of scenarios with history. This kernel is expressive enough to embed Petri net systems while allowing to synthesize an implementation from any bounded oclet specification. In contrast, Live Sequence Charts provide a much higher expressive power. Regarding control-flow constructs, LSCs provide notions for imperative scenarios, multi-modal scenarios, forbidden scenarios, and for distinguishing visible from invisible actions [25]. Compared to oclets, these notions allow a system designer to specify high-level system requirements while abstracting away much more of the details of the system.

More expressive scenarios. We already discussed how the distinction between visible and invisible actions can be included in the semantics of oclets and in the synthesis. Similar results regarding anti-oclets (which express forbidden scenarios) have been published in [39]. A possible research programme is to systematically formalize the notions from LSCs in the formal model of oclets. The aim is to extend the expressive power of oclets while preserving results on composing and analyzing oclets, and on synthesizing implementations to the possible extent.

This research programme can be pushed further by also considering notions of data and time. It is worth investigating how Colored Petri nets [66] allow to express the notions of data and symbolic instances of LSCs [55]. Likewise, various classes of time Petri nets [89] could formalize time-related notions of LSCs [55]. Both extensions of Petri nets allow the construction of finite complete prefixes [71, 23].

Synthesis of open systems. In this thesis, we addressed the synthesis of a *closed* system from a scenario-based specification. The synthesis of an *open* system Γ which interacts with an unrestricted environment E is significantly harder as shown in Section 8.4.3. We have shown how to reduce the synthesis of an open system Γ from an oclet specification O to a synthesis of the closed system $\Gamma \oplus E$ where E is an unrestricted environment.

The open issue here is that oclets are not expressive enough to describe interesting properties of the interaction behavior of Γ. Increasing the expressive power of oclets as proposed, makes the synthesis problem practically relevant.

The synthesis problem from a more expressive model of oclets is an open problem. A possible research programme has to systematically follow the steps (1)-(5), which we gave for invisible actions, also for all other means of expression introduced to oclets. If the semantics of oclets is generalized towards LSCs, then an approach by Esparza and Heljanko [36] provide a starting point for translating an oclet specification with higher expressive power to basic oclets. LSCs are known to specify a sub-class of Büchi automata [16, 18]. Esparza and Heljanko [36] show how to translate a stuttering-invariant LTL formula φ to an acceptor A_φ that corresponds to a Büchi automaton, and how to *asynchronously* compose A_φ with a Petri net N to check whether N satisfies φ. The open problems are (1) whether their approach also allows to translate an oclet o with higher expressive power to

an acceptor A_o, and (2) whether the acceptor A_o can be used to *synthesize* an open system Γ instead of verifying that Γ satisfies o.

Structuring specifications

Most of the scenario-based techniques discussed in this thesis structure a scenario-based specification. A typical approach is a *hierarchical* specification that relates scenarios to each other by a graph and composition operators, for example in Hierarchical MSCs [65]. Another approach is an *inline scenario* that embeds a scenario into another scenario as used in UML Sequence Diagrams [95] and in LSCs [56]. The results of this thesis show that these concepts are not required semantically for specifying system behavior. Yet, these techniques allow to structure complex specifications and to make relations between scenarios explicit. The model of oclets introduced in this thesis lacks such structuring techniques. A possible research programme is to provide structuring techniques for oclets.

Inline scenarios typically embed several scenarios into a larger scenario together with an operator, for example, to choose between two alternative sub-scenarios within a larger scenario; Figure 3.13 on page 66 depicts an example. A related research question is to which extent inline scenarios are mere "syntactic sugar". In other words, which oclets with inline scenarios can be decomposed or transformed into a set of oclets with the same semantics.

In contrast, hierarchical specifications provide a "bird's eye view" on scenarios by relating several scenarios of a specification to each other. Usually an edge from one scenario to another scenario expresses their sequential composition; Figure 3.10 on page 62 depicts an example. Consequently, hierarchical specifications as well as the history of a scenario both describe how scenarios relate to each other. A hierarchical specification over oclets must be consistent with the oclet's histories. This observation gives rise to a number of research question. (1) Assuming a hierarchical specification H_O over a set of oclets O to be given, do O and H_O specify the same behavior? (2) Is it possible to *derive* from a given oclet specification O a hierarchical specification H_O s.t. O and H_O specify the same behavior? (3) And vice versa? (4) Assuming a hierarchical specification H_O over a set of oclets O s.t. both consistently specify the same behavior; if now O (or H_O) is changed how to change H_O (or O) to maintain consistency? Ideally, the hierarchical specification H_O gives a system designer an overview on the scenario relations in O s.t. the specification can be changed consistently either on the level of single oclets in O or on the global level of H_O.

Graphical notation and tool support

All problems presented up to this point aim at letting a system designer write down scenarios of the system exactly in the way she has them in mind. Such an approach needs to be supported by a suitable graphical notation. One could either develop a high-level notation on top of oclets which hides their technical details, or define a translation from an established notation like LSCs to oclets. A corresponding tool would properly relate the high-level notation to a semantically equivalent oclet specification (or an extension of oclets).

System mining

A completely different approach to synthesizing a system from event-based descriptions is *process mining*. A mining algorithm synthesizes from a set T of execution traces a Petri net system Σ that exhibits these traces. The traces T are assumed to be *recorded observations* of an actual system. So, the behavioral information in T is assumed to be *incomplete* or contain information from different levels of abstraction which leads to *noise*.

Usually, Σ has to *generalize* the behavior described in T, which raises a question that we discussed in Chapters 3 and Chapters 4: does T describe *sample behaviors of the system* (i.e., the existential view of scenarios) or *all system behaviors* (i.e., the universal view of scenarios)? If the mined system model Σ reproduces *exactly* T, then it is *overfitting* given the incompleteness of T. For this reason, Σ is ususally generalized to exhibit more runs than T. Generalizing Σ may lead to *underfitting*, i.e., Σ exhibits behavior which is not related to T anymore [115]. Processing mining algorithm balance the fitness of Σ with respect to various optimization goals [118].

Particularly in early stages of system design, a scenario-based specification is immature. The specified behavior is incomplete and a system designer might specify behavior on different levels of abstraction. A classical synthesis algorithm like the one presented in this thesis is likely to either return no model at all or to overfit the intended system behavior. So, one open research question is how to construct a *generalized* implementation of a scenario-based specification. First works allow to synthesize systems in case of incomplete specifications [90]. However, the issue of generalization suggests to combine scenario-based synthesis with process mining techniques to synthesize an implementation wrt. specific optimization goals.

A complementary question is to *discover scenarios from observations of existing systems*. Current mining algorithms return a model of the entire system. Often these models are difficult to understand [115] and hence difficult to improve. A discovered scenario-based model might be easier to understand and to improve as each scenario has a simple structure. The inherent difficulty for mining scenarios is to discover "self-contained stories" that can be understood by a human system designer. The formal model of oclets forms a possible starting point for this research because oclets and mining are based on the same notions from Petri nets.

Altogether, these research questions strive towards a flexibly usable system design technique providing different views on the system model. These views allow a system designer to understand, think, and develop the system always in those concepts she finds most suitable for her task. She should be able to switch her view on the system whenever necessary; any change to the model should consistently propagates to all other views. The theory of oclets presented in this thesis is another small step in that direction.

Appendix

A.1. Basic Notation and Terminology

This section presents, as a reference, the basic formal notations used throughout all chapters of this thesis.

Sets

That standard notions from set algebra apply: $\cup$ (union), $\cap$ (intersection), $\setminus$ (set difference), $\emptyset$ (empty set). The following additional notions are used; let X and Y be sets.

$\uplus$	*disjoint* union of X and Y, $X \uplus Y = X \cup Y$ whenever $X \cap Y = \emptyset$.

Relations

Let $R \subseteq X \times Y$ be a relation.

$R\|_{U \times V}$	the *restriction* of R to U and V: $R\|_{U \times V} := R \cap (U \times V)$, for $U \subseteq X, V \subseteq Y$.
R^{-1}	the *inverse* of R: $(x, y) \in R^{-1}$ iff $(y, x) \in R$.

Let $R \subseteq X \times X$ be a relation.

R^+	the *transitive* closure of R: the least relation $R^+ \subseteq X \times X$ s.t. $R \subseteq R^+ \wedge \forall (x, y), (y, z) \in R^+ : (x, z) \in R^+$.
R^*	the *reflexive-transitive closure* of R: the least relation $R^* \subseteq X \times X$ s.t. $R^+ \subseteq R^* \wedge \forall x : (x, x) \in R^*$.

Functions

A function $f : X \to Y$ is a relation $f \subseteq X \times Y$ where for all $(x, y_1), (x, y_2) \in f$ holds $y_1 = y_2$. Notations: $f(x) = \perp$ iff there exists no $y \in Y$ with $(x, y) \in f$; $f(x) = y$ iff $(x, y) \in f$.

$f\|_U$	*restriction* of function f to arguments $U \subseteq X$ with $f\|_U(x) := f(x)$, for all $x \in U$.
$f \circ g$	*concatenation* of functions $g : X \to Y$ and $f : Y \to Z$ and with $(f \circ g)(x) := f(g(x))$, for all $x \in X$.

A.2. Lattices and Fixed Points

In Chap. 5 we construct system behavior from scenarios. The central Definition 5.17 repeatedly extends a set of runs by appending scenarios. The proof in Theorem 5.18 that the constructed set is well-defined and minimal builds on a result on fixed points in lattices. This section provides the corresponding definitions.

A *complete lattice* $L = (L, \sqsubseteq) = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partially ordered set $(L, \sqsubseteq)$ s.t. each $L' \subseteq L$ has

1. a least upper bound $l \in L$ with $\forall l' \in L : l' \sqsubseteq l$, denoted $\sqcup L' := l$, and
2. a least lower bound $l \in L$ with $\forall l' \in L : l \sqsubseteq l'$, denoted $\sqcap L' := l$.

Furthermore, $\bot = \sqcup\emptyset = \sqcap L$ is the *least element*, and $\top = \sqcap\emptyset = \sqcup L$ is the *greatest element*.

A function $f : L \to L$ is *monotone* iff $\forall l, l' \in L : l \sqsubseteq l' \Rightarrow f(l) \sqsubseteq f(l')$.

Theorem A.1 (Knaster and Tarski [43], originally in [109]). *Let $\mathcal{L} = (L, \sqsubseteq, \top, \bot)$ be a complete lattice. Let $f : L \to L$ be monotone. Then there is a least fixed point $\mathrm{LFP}(f)$ and a greatest fixed point $\mathrm{GFP}(f)$ of f. These are*

- $\mathrm{LFP}(f) = \inf\{x \in L \mid f(x) \sqsubseteq x\}$, *and*
- $\mathrm{GFP}(f) = \sup\{x \in L \mid x \sqsubseteq f(x)\}$. ⋆

Lemma A.2 (Inductive Computation of Fixed Point, Lemma 20.5 in [43]):
Let $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. Let $f : L \to L$ be monotone. Then $\mathrm{LFP}(f) = f^\lambda$ with λ being the least ordinal with $f^{\lambda+1} = f^\lambda$. ⋆

A.3. Basic Notions on Petri Nets

This section complements the introduction to Petri nets from Chapter. 2 by several basic notions on Petri nets.

Isomorphism on Petri Nets

Definition A.3 (Homomorphism, isomorphism on Petri nets). Let N and M be two Petri nets. A mapping $\alpha : X_N \to X_M$ is a *homomorphism* from N to M, denoted $\alpha : N \to M$, iff

1. $\forall p \in P_N : \alpha(p) \in P_M \wedge \forall t \in T_N : \alpha(t) \in T_N$ (α preserves sorts),
2. $\forall x \in X_N : \ell(x) = \ell(\alpha(x))$ (α preserves labels), and
3. $\forall (x, y) \in F_N : (\alpha(x), \alpha(y)) \in F_M$ (α preserves arcs).

A mapping $\alpha : X_N \to X_M$ is an *isomorphism* from N to M iff α is a homomorphism from N to M and α^{-1} is a homomorphism from M to N. In this case N and M are *isomorphic*.

Given a net N and an isomorphism α we write N^α as a short-hand to denote the Petri net that is isomorphic to N by isomorphism $\alpha : N \to N^\alpha$. By $[N]$ we denote the class of all Petri nets that are isomorphic to N; we write $N^\alpha \in [N]$ to denote a specific representative of this class. ⌟

A.4. Distributed Runs and Partially Ordered Sets of Events

This thesis builds largely on the behavioral model of distributed runs as introduced in Chapter 2. The formal definitions build on labeled causal nets (Def. 2.7). There are other formalizations of distributed runs, most notably labeled partial orders, to which we refer occasionally in this thesis. This section provides the corresponding formal definitions.

Definition A.4 (Partial order). Let X be an arbitrary set. A *partial order* over X is a binary relation $\leq \subseteq X \times X$ that is *reflexive* ($x \leq x$, for all $x \in X$), *transitive* ($x \leq y$ and $y \leq z$ imply $x \leq z$, for all $x, y, z \in X$), and *anti-symmetric* ($x \leq y$ and $y \leq x$ imply $x = y$, for all $x, y \in X$). ⌟

Let $\mathcal{L}$ be a set of labels including the dedicated label $\tau \in \mathcal{L}$.

Definition A.5 (Labeled partial order (LPO)). A *labeled partial order* (LPO) $r = (E, \leq, \ell)$ over $\mathcal{L}$ consists of a set E, a partial order $\leq \subseteq E \times E$ over E, and a labeling $\ell : E \to \mathcal{L}$. We write F_r to denote the *support* of r which is the least relation $F_r \subseteq E \times E$ with $F_r^* = \leq$. ⌟

We interpret E as a set of events as conceptualized in Sect. 2.3.2.

1. Let $\pi = (B, E, F, \ell)$ be a labeled causal net over $\mathcal{L}$; π induces the LPO $r_\pi := (E, (F^*)|_{E \times E}, \ell|_E)$.
2. Let $r = (E, \leq, \ell)$ be an LPO over $\mathcal{L} \setminus \{\tau\}$; r induces the causal net $\pi_r := (E, B, F, \ell')$ with $B := F_r$, $F := \{(e, b), (b, e') \mid (e, e') \in F_r, b = (e, e') \in B\}$ and $\ell'(e) := \ell(e)$, for all $e \in E$, and $\ell'(b) := \tau$, for all $b \in B$.

The following technical lemma is used in proving Lem. 5.5. It states that the set $x{\downarrow_\pi}$ of predecessors of x can be expressed as the union of a set Z with the predecessors of all pre-nodes of Z. By this property, we show in Lem. 5.5 that a distributed run remains finitely preceded under composition.

Lemma A.6: *Let π be a labeled causal net. Let $x \in X_\pi$ and let $Z \subseteq x{\downarrow_\pi}$ s.t.*

$$\forall z \in Z\ \forall y \in X_\pi : z \leq_\pi y \wedge y \leq_\pi x \Rightarrow y \in Z. \tag{A.1}$$

Then $x{\downarrow_\pi} = Z \cup \bigcup_{z \in {}^\bullet Z \setminus Z} (z{\downarrow_\pi})$. ⋆

Proof (by induction). We prove the proposition by induction on the size $n = |Z|$.

Let $n = 1$. Then $Z = \{x\}$ and ${}^\bullet Z \setminus Z = {}^\bullet x$.

$$\begin{array}{rcll} x{\downarrow_\pi} & = & \{y \in X_\pi \mid y \leq_\pi x\} & \textit{[by Def. 2.5]} \\ & = & \{y \in X_\pi \mid y = x \vee \exists z : y \leq_\pi z \wedge (z, x) \in F_\pi\} & \textit{[by } \leq_\pi = F_\pi^* \textit{]} \\ & = & \{x\} \cup \{y \in X_\pi \mid \exists z : y \leq_\pi z \wedge z \in {}^\bullet x\} & \\ & = & \{x\} \cup \bigcup_{z \in {}^\bullet x} (z{\downarrow_\pi}) & \textit{[by Def. 2.5]} \\ & = & Z \cup \bigcup_{z \in {}^\bullet Z} (z{\downarrow_\pi}). & \end{array}$$

Let $n > 1$. Let $y \in Z$ with ${}^\bullet y \cap Z = \emptyset$ (y exists because π is finitely preceded).

- (1) $y <_\pi x$. Assume $y = x$. Then for all $z \in Z \setminus \{x\}$ holds $z \not\leq_\pi x$ (because ${}^\bullet x \cap Z = {}^\bullet y \cap Z = \emptyset$). But $Z \subseteq x{\downarrow_\pi}$. Contradiction to Def. 2.5.
- (2) $Y := Z \setminus \{y\}$ satisfies (A.1) because there exists no $z \in Y$ with $z \leq_\pi y$.
- (3) $x{\downarrow_\pi} = Y \cup \bigcup_{z \in ({}^\bullet Y \setminus Y)} (z{\downarrow_\pi})$, by (2) and inductive assumption.
- (4) $y{\downarrow_\pi} = \{y\} \cup \bigcup_{z \in {}^\bullet y} (z{\downarrow_\pi})$, because $\{y\}$ satisfies (A.1) wrt. $y{\downarrow_\pi}$ and inductive assumption.
- (5) $y \in {}^\bullet Y \setminus Y$. Firstly, $y \notin Y$ by definition. Secondly, (1) implies that there ex. $z \in y^\bullet$ with $z \leq_\pi x$. Thus $z \in Z$ and $y \in {}^\bullet z \subseteq {}^\bullet Z$.
- (6) ${}^\bullet Z \setminus Z = {}^\bullet(Y \cup \{y\}) \setminus (Y \cup \{y\}) = ({}^\bullet Y \cup {}^\bullet y) \setminus (Y \cup \{y\}) = ({}^\bullet Y \setminus (Y \cup \{y\})) \cup ({}^\bullet y \setminus (Y \cup \{y\})) = ({}^\bullet Y \setminus Y) \setminus \{y\} \cup ({}^\bullet y \setminus (Y \cup \{y\})) = ({}^\bullet Y \setminus Y) \setminus \{y\} \cup {}^\bullet y$. The last equality holds because ${}^\bullet y \cap Z = \emptyset$ and $Y \subseteq Z$ imply ${}^\bullet y \cap Y = \emptyset$.

$$
\begin{array}{lcll}
x{\downarrow_\pi} & = & Y \cup \bigcup_{z \in ({}^\bullet Y \setminus Y)} (z{\downarrow_\pi}) & \textit{[by (3)]} \\
& = & Y \cup (y{\downarrow_\pi}) \cup \bigcup_{z \in (({}^\bullet Y \setminus Y) \setminus \{y\})} (z{\downarrow_\pi}) & \textit{[by (5)]} \\
& = & Y \cup \{y\} \cup \bigcup_{z \in {}^\bullet y} (z{\downarrow_\pi}) \cup \bigcup_{z \in (({}^\bullet Y \setminus Y) \setminus \{y\})} (z{\downarrow_\pi}) & \textit{[by (4)]} \\
& = & Z \cup \bigcup_{z \in (({}^\bullet Y \setminus Y) \setminus \{y\} \cup {}^\bullet y)} (z{\downarrow_\pi}) & \textit{[by (2) and set theory]} \\
& = & Z \cup \bigcup_{z \in {}^\bullet Z \setminus Z} (z{\downarrow_\pi}) & \textit{[by (6)]}
\end{array}
$$

Hence, the proposition holds. □

A.5. Properties of Distributed Runs

Proof of Lemma 2.11 from page 37. A prefix of a distributed run can be characterized in different ways.

Proof (of Lemma 2.11). Let π and ρ be distributed runs.

(1) Claim: $\pi \sqsubseteq_{\mathbf{t}} \rho$ iff $X_\pi \subseteq X_\rho$ and $F_\pi = F_\rho|_{X_\pi \times X_\pi}$ and $(y \in X_\pi \wedge (x,y) \in F_\rho) \Rightarrow x \in X_\pi$, for all $x, y \in X_\rho$.

By Def. 2.10 holds $\pi \sqsubseteq_{\mathbf{t}} \rho$ iff $X_\pi \subseteq X_\rho$, $\leq_\pi = \leq_\rho|_{(X_\pi \times X_\pi)}$, $\forall x \in X_\pi : x{\downarrow_\rho} \subseteq X_\pi$. We show the equivalence of the respective propositions.

1. $X_\pi \subseteq X_\rho$ holds in both cases by definition.
2. $\leq_\pi = \leq_\rho|_{(X_\pi \times X_\pi)}$ iff $F_\pi = F_\rho|_{X_\pi \times X_\pi}$.
   Holds by $\leq_\pi = F_\pi^*$.
3. $\forall x \in X_\pi : x{\downarrow_\rho} \subseteq X_\pi$ iff $(y \in X_\pi \wedge (x,y) \in F_\rho) \Rightarrow x \in X_\pi$, for all $x, y \in X_\rho$.
   ($\Rightarrow$) Assume $(x,y) \in F_\rho, x \notin X_\pi, y \in X_\pi$. Then $x \leq_\rho y$ which is equivalent to $x \in y{\downarrow_\rho}$ but $x \notin X_\pi$.
   ($\Leftarrow$) Assume $x \in y{\downarrow_\rho} \in F_\rho, x \notin X_\pi, y \in X_\pi$. W.l.o.g. $(x,y) \in F_p rocPrime$ (if not consider the path from x to y along F_ρ and take the first arc (x^*, y^*) on this path with $x^* \notin X_\pi, y^* \in X_\pi$). But then $(x,y) \in F_p rocPrime, x \notin X_\pi, y \in X_\pi$.

(2) Claim: $F_\pi = F_\rho|_{X_\pi \times X_\pi}$ and $(x \in X_\pi \wedge (x,y) \in F_\rho) \Rightarrow y \in X_\pi$, for all $x, y \in X_\rho$ iff $X_\pi \subseteq X_\rho$ and $F_\pi = F_\rho|_{X_\rho \times X_\pi}$.

It suffices to show that (2.1) $F_\rho \cap (X_\pi \times X_\pi) = F_\pi$ and (2.2) $(x \in X_\pi \wedge (x,y) \in F_\rho) \Rightarrow y \in X_\pi$ is equivalent to (2.3) $F_\pi = F_\rho \cap (X_\rho \times X_\pi)$.

($\Rightarrow$) Let π and ρ satisfy (2.1) and (2.2). Assume $F_\pi \neq F_\rho \cap (X_\rho \times X_\pi)$. By assumption (2.1) and $X_\pi \subseteq X_\rho$ then would hold $F_\pi \subset F_\rho \cap (X_\rho \times X_\pi)$. Thus there would exist $(x, y) \in F_\rho, x \in X_\rho \setminus X_\pi, y \in X_\pi$ which violates (2.2). Hence (2.3) holds.

($\Leftarrow$) Let π and ρ satisfy (2.3). Then (2.1) $F_\rho \cap (X_\pi \times X_\pi) = F_\pi$ holds because $F_\pi \subseteq X_\pi \times X_\pi$. Further, by (2.3) there exists no $(x, y) \in F_\rho$ with $x \in X_\rho$ and $y \in X_\pi$ (because otherwise $(x, y) \notin F_\pi = F_\rho|_{X_\rho \times X_\pi}$). But this is equivalent to (2.2). □

Properties of induced prefixes A set Y of nodes of a distributed run $p\pi$ induces the prefix $\pi[Y]$ of π that consists of Y and all predecessors of Y. Induced prefixes are specifically interesting if Y is a cut of π.

Lemma A.7: *Let ρ a distributed run, let $Y \subseteq X_\rho$, let $\pi := \rho[Y]$. Then:*

1. *$\pi = \rho[X_\pi]$, and if π is finite, then $\pi = \rho[\max \pi]$.*
2. *$\pi \sqsubseteq_{\mathbf{t}} \rho$.*
3. *$\forall x \in X_\pi : post_\pi(x) = \emptyset \Rightarrow x \in Y$.*
4. *If Y is cut of ρ, then $\rho[Y] \sqsubseteq \rho$.*
5. *Let $Z \subseteq X_\rho$. If $Z \leq_\rho Y$, then $\rho[Z] = \pi[Z]$.*
6. *Let π^* be a finite distributed run. If $\pi^* \sqsubseteq_{\mathbf{t}} \rho$ and $\max \pi^* \leq_\pi Y$, then $\pi^* \sqsubseteq_{\mathbf{t}} \rho[Y]$. If additionally $\pi^* \sqsubseteq \rho$, then $\pi^* \sqsubseteq \rho[Y]$.* ⋆

Proof. (1) By definition of $y{\downarrow_\rho}$ (Def. 2.5) holds for each $y \in Y$: if $y{\downarrow_\rho} \subseteq X_\pi$, then for each $x \in y{\downarrow_\rho}$ holds $x{\downarrow_\rho} \subseteq X_\pi$ and $x \in x{\downarrow_\rho}$. Thus $\bigcup_{x \in X_\pi} x{\downarrow_\rho} = X_\pi = \bigcup_{y \in Y} y{\downarrow_\rho}$. Thus, $\rho[X_\pi] = \rho[Y]$ by Def. 2.14. Let π be finite. Claim: $x \in X_\pi$ iff there ex. $y \in \max \pi$ with $x \leq_\pi y$. ($\Leftarrow$) Holds by Def. 2.14. ($\Rightarrow$) Let $x \in X_\pi$. The set of transitive predecessors $x{\uparrow_\pi} = \{y \mid x \leq_\pi y\}$ is finite because π is finite. Thus there ex. $y \in x{\uparrow_\pi}$ with $post_\pi(y) = \emptyset$, by $\leq_\pi$ being anti-symmetric. Hence $y \in \max \pi$ and $x \leq_\pi y$.

(2) $\pi \sqsubseteq_{\mathbf{t}} \rho$. By construction: (2.a) $X_\pi \subseteq X_\rho$. Further, $F_\pi = F_\rho|_{X_\pi \times X_\pi}$ implies (2.b) $\leq_\pi = \leq_\rho |_{X_\pi \times X_\pi}$. Finally, by definition of $\pi = \rho[Y]$ holds: for each $x \in X_\pi$ exists $y \in Y$ with $x \in y{\downarrow_\rho}$; this implies (2.c) $x{\downarrow_\pi} \subseteq x{\downarrow_\pi} \subseteq X_\pi$. Properties (2.a), (2.b), and (2.c) are equivalent to $\pi \sqsubseteq_{\mathbf{t}} \rho$ by Def. 2.10.

(3) By construction of π holds: for each $x \notin Y$ exists $y \in Y$ with $x \leq_\pi y$. Thus $x \notin Y$ implies $post_\pi(x) \neq \emptyset$. By contraposition holds the proposition $post_\pi(x) = \emptyset \Rightarrow x \in Y$.

(4) Let B be a cut of ρ, let $\pi := \rho[B]$. By (2) holds (4.a) $\pi \sqsubseteq_{\mathbf{t}} \rho$. Let $e \in E_\pi$. To show: $post_\pi(e) = post_\rho(e)$. By construction holds $post_\pi(e) \subseteq post_\rho(e)$. Assume there exists $b^* \in post_\rho(e) \setminus post_\pi(e)$; then $b^* \notin X_\pi$. Thus for all $b \in B$, $b^* \notin b{\downarrow_\rho}$, hence $b^* \,||_\rho\, b$. Thus B is not a maximal co-set of conditions; contradiction to B being a cut of ρ. Thus (4.b) $post_\pi(e) = post_\rho(e)$. Properties (4.a) and (4.b) are equivalent to $\pi \sqsubseteq \rho$ by Def. 2.10.

(5) Let $Z \subseteq X_\rho$ s.t. $\forall z \in Z\ \exists y \in Y : z \leq_\rho y$. We have to show $\rho[Z] = \pi[Z]$ where $\pi = \rho[Y]$.

- (5.1) Firstly, $Z \subseteq X_\pi$ holds: $\bigcup_{y \in Y} y{\downarrow_\rho} = \{x \in X_\rho \mid \exists y \in Y : x \leq_\rho y\} = X_\pi$, by Def. 2.14. Thus $Z \subseteq X_\pi$ by the assumption $Z \leq_\pi Y$.
- (5.2) Thus, $\pi_Z := \pi[Z]$ is defined.
- (5.3) Secondly, $F_\pi = F_\rho|_{X_\pi \times X_\pi}$ by Def. 2.14. Thus, $X_Z = \bigcup_{z \in Z} y{\downarrow_\pi} = \bigcup_{z \in Z} z{\downarrow_\rho}$.
- (5.4) By Def. 2.14 holds: $F_Z = F_\pi|_{X_Z \times X_Z} = (F_\rho|_{X_\pi \times X_\pi})|_{X_Z \times X_Z} = F_\rho|_{X_Z \times X_Z}$; the last equality holds by $X_Z \subseteq X_\pi$.
- (5.5) From (5.3) and (5.4) follows $\rho[Z] = \pi_Z = \pi[Z]$.

(6) Let $\pi^* \sqsubseteq_{\mathbf{t}} \rho$, π^* finite, s.t. for each $x \in \max \pi$ exist $y \in Y$ with $x \leq_\rho y$. From π^* being finite follows by (1) $\pi^* = \rho[\max \pi^*]$. From (5) follows $\rho[\max \pi^*] = \pi[\max \pi^*]$; from (2) follows $\pi[\max \pi^*] \sqsubseteq_{\mathbf{t}} \pi$. Thus $\pi^* \sqsubseteq_{\mathbf{t}} \pi = \rho[Y]$.

Assume $\pi^* \sqsubseteq \rho$. Let $e \in E^*$; to show: $post^*(e) = post_\pi(e)$. From $\pi^* \sqsubseteq_{\mathbf{t}} \pi$ follows $\pi^* \subseteq \pi$; thus $post^*(e) \subseteq post_\pi(e)$ for all $e \in E^*$. By Def. 2.14 holds $post_\pi(e) \subseteq post_\rho(e)$ and $post_\rho(e) = post^*(e)$ holds by $\pi^* \sqsubseteq \rho$ (Def. 2.10). Thus $post_\pi(e) = post^*(e)$, hence $\pi^* \sqsubseteq \pi = \rho[Y]$. □

A.6. Implementing an Oclet Specification wrt. Visible Actions

This section propose how to generalize Definition 4.10 to allow that an implementation of an oclet specification O has invisible actions which are not mentioned O.

We have seen in our discussion in Section 3.8 and in the concluding example in Section 3.9, that partial occurrences of scenarios are potentially dangerous for branching time semantics. Specifically, two non-branching bisimilar systems may satisfy the same specification. The following refined definition of when a Petri net system Σ implements an oclet specification O ensures Σ exhibits only those behaviors specified O *up to branching bisimulation*.

The semantics of an oclet specification O is the set $\mathcal{R}(O)$ of all system behaviors that satisfy each of the oclets in O. A Petri net system Σ implements an oclet specification iff the runs of Σ satisfy O, i.e., $R(\Sigma) = R \in \mathcal{R}(O)$ (Def. 4.10). To allow for occurrences of invisible actions in Σ, we no longer require equality $R(\Sigma) = R$ but only an equivalence $R(\Sigma) \sim R$ wrt. visible actions; this equivalence has to preserve the branching behavior of R. Vogler adapts in [125] the classical notion of branching bisimulation [119] and defines when two event structures are *history-preserving bisimilar* wrt. a set $Vis \subseteq \mathcal{L}$ of visible actions.

Vogler's notion [125, Def.6.3.6] can be adapted to our setting as follows. The critical part in this definition is the equivalence of runs wrt. their visible events. Assume a set $Vis \subseteq \mathcal{L}$ of visible names to be given. Let π be a distributed run; the visible events in π are $E_{Vis} := \{e \in E_\pi \mid \ell(e) \in Vis\}$; these visible events induce the following labeled partial order over π: $\pi|_{Vis} = (E_{Vis}, \leq_\pi |_{E_{Vis} \times E_{Vis}}, \ell_\pi|_{E_{Vis}})$. Two distributed runs π_1 and π_2 are equivalent wrt. Vis iff $\pi_1|_{Vis}$ and $\pi_2|_{Vis}$ are isomorphic.

History preserving branching bisimulation extends the equivalence wrt. Vis to sets of distributed runs by also preserving the moments of choice as follows. Let

R_1 and R_2 be prefix-closed sets of distributed runs. Intuitively, R_1 and R_2 are *history-preserving bisimilar* wrt. *Vis* iff there exists an equivalence relation $\sim$ between the prefixes in R_1 and R_2 s.t.

1. $\varepsilon \sim \varepsilon$,
2. for every $\pi_1 \in R_1$ and $\pi_2 \in R_2$, with $\pi_1 \sim \pi_2$ and every continuation $\rho_1 \in R_1$, $\pi_1 \sqsubseteq \rho_1$
   - exists a continuation $\rho_2 \in R_2$, $\pi_2 \sqsubseteq \rho_2$ with $\rho_1 \sim \rho_2$, s.t.
   - for the respective extensions $\Delta\pi_1 := (\rho_1 - \pi_1)$ and $\Delta\pi_2 := (\rho_2 - \pi_2)$, their induced LPOs $\Delta\pi_1|_{Vis}$ and $\Delta\pi_2|_{Vis}$ are isomorphic, and
3. vice versa for every continuation of π_2.

This notion allows, for instance, the system behavior R_1 to have additional invisible actions $a \notin Vis$ compared to R_2 as long as for each run in R_1 and exists a corresponding run in R_2 that is isormphic wrt. visible actions. Moreover, each choice between two runs in R_1 must have a corresponding choice between two equivalent runs in R_2, and vice versa.

The notion of history-preserving bisimulation presented here entirely ignores the conditions of distributed runs; Vogler provides in [125] a bisimulation definition for safe Petri nets that also reflects markings.

A.7. Composing Oclets

Chapter 5 introduces a composition operator $\triangleright$ to compose oclets, Def. 5.2. The soundness of this operator and its applications relies on a number of technical properties. This section presents proofs of properties related to the composition.

Lemma 5.5: Causal nets are closed under composition with oclets. Lemma 5.5 states that the composition $o_1 \triangleright o_2$ yields a distributed run. The corresponding formal proof follows from the definition of a causal net (Def. 2.6).

Proof (of Lemma 5.5). We have to show that $\rho := \pi_1 \cup \pi_2$ (for composing oclets $o_1 = (\pi_1, hist_1)$ and $o_2 = (\pi_2, hist_2)$ is a causal net. First of all, ρ is a Petri net by Def. 2.4. Thus, we have to show by Def. 2.6 that (1) F_ρ^* is a partial order over X_ρ, (2) F_ρ^* is finitely preceded, and (3) $|{}^\bullet b| \leq 1 \wedge |b^\bullet| \leq 1$, for all $b \in B_\rho$.

(1) $F_\rho^* \subseteq X_\rho \times X_\rho$ and F_ρ^* is reflexive and transitive by construction. We have to show that F_ρ^* is anti-symmetric. By construction, $F_\rho = F_1 \cup F_2$. And by assumption, F_1^* and F_2^* are partial orders. Further, $hist_2 \subseteq \pi_1$ which implies $F_{hist_2}^* \subseteq F_1^*$.

Assume by contradiction that F_ρ contains a cycle, i.e., that there are nodes $x, y \in X_\rho, x \neq y$ with $(x, y) \in F_\rho^*$ and $(y, x) \in F_\rho^*$. Then there are arcs $(x_1, y_1), (y_2, x_2) \in F_\rho$ with $x_1, x_2 \in X_1$ and $y_1, y_2 \in X_{con_2}$ (otherwise either F_1 or F_2 would be anti-symmetric which contradicts π_1 and π_2 being causal nets). From $y_1, y_2 \in X_{con_2}$ follows $(x_1, y_1), (y_2, x_2) \in F_2 \subseteq F_\rho$. From $\pi_2 \cap con_1 = \emptyset$ follows $x_1, x_2 \in X_{hist_2}$.

But $hist_2 \sqsubseteq \pi_2$ (by Def. 4.1). Thus, $(y_2, x_2) \in F_2 \wedge x_2 \in X_{hist_2} \Rightarrow y_2 \in X_{hist_2}$ (by Lem. 2.11); contradiction. Hence, F_ρ^* is anti-symmetric.

(2) We show that F_ρ^* is finitely preceded by showing that each $x \in X_\rho$ has only finitely many predecessors $x{\downarrow_\rho}$ (Def. 2.5). We distinguish two cases:

1. $x \in X_1$. Then holds $x{\downarrow_\rho} = x{\downarrow_1}$ as follows. Assume $x{\downarrow_1} \neq x{\downarrow_\rho}$. By $\pi_1 \subseteq \rho$ holds $x{\downarrow_1} \subseteq x{\downarrow_\rho}$. Thus there exists $z \in x{\downarrow_\rho} \setminus x{\downarrow_1}$. W.l.o.g. $(z, y) \in F_\rho \setminus F_1$ with $y \in x{\downarrow_1} \subseteq X_1$. By $F_\rho = F_1 \cup F_2$ holds $(z, y) \in F_2$. There are two cases:
   a) If $z \in X_1$, then $(z, y) \in F_{hist_2}$ because $con_2 \cap \pi_1 = \emptyset$. But $hist_2 \subseteq \pi_1$ by o_2 being enabled at o_1, which implies $(z, y) \in F_1$. Contradiction.
   b) Thus $z \notin X_1$ and $(z, y) \in F_2$. Thus $z \notin X_{hist_2}$ and $y \in X_{hist_2}$ because $hist_2 \subseteq \pi_1$ by o_2 being enabled at o_1. But $y \in X_{hist_2}$ requires $z \in X_{hist_2}$ because $hist_2 \sqsubseteq \pi_2$ (definition of prefix, Def. 2.10). Contradiction.

   Thus, $x{\downarrow_\rho} = x{\downarrow_1}$. Hence x has finitely many predecessors because π_1 is a causal net.

2. $x \notin X_1$. Then $x \in X_{con_2}$. Let $Z := x{\downarrow_\rho} \cap X_{con_2}$.
   a) Claim: $\forall z \in Z\ \forall y \in X_\rho : z \leq_\rho y \wedge y \leq_\rho x \Rightarrow y \in Z$.
   Assume the opposite: there exists $z \in Z$ and $y \in X_\rho$ with $z \leq_\rho y \wedge y \leq_\rho x \wedge y \notin Z$. Then $y \notin X_{con_2}$. Hence $y \in X_1$ by $\rho = \pi_1 \cup \pi_2$ and $con_2 \cap \pi_1 = \emptyset$. W.l.o.g. exists $u \in Z \subseteq X_{con_2}$ and $(u, y) \in F_\rho$. But $y \in X_1$ requires $z \in X_1$ because $hist_2 \sqsubseteq \pi_2$ (definition of prefix, Def. 2.10). Contradiction.
   b) Thus by Lem. A.6 holds: $x{\downarrow_\rho} = Z \cup \bigcup_{z \in {}^\bullet Z \setminus Z} z{\downarrow_\rho}$.
   c) For each $z \in {}^\bullet Z \setminus Z$ holds $z \notin X_{con_2}$ which is equivalent to $z \in X_1$. Thus $x{\downarrow_\rho} = Z \cup \bigcup_{z \in {}^\bullet Z \setminus Z} z{\downarrow_1}$ by the first case.
   d) $Z \subseteq X_{con_2} \subseteq X_2$ is finite because o_2 is finite by Def. 4.1. Hence ${}^\bullet Z \setminus Z$ is finite because ${}^\bullet z$ is finite for each $z \in Z$. From π_1 being a causal net follows that $z{\downarrow_1}$ is finite for each $z \in {}^\bullet Z \setminus Z$. Thus the finite union $\bigcup_{z \in {}^\bullet Z \setminus Z} z{\downarrow_1}$ is finite.

   Hence x has only finitely many predecessors in ρ.

(3) $|pre_\rho(b)| \leq 1 \wedge |post_\rho(b)| \leq 1$, for all $b \in B_\rho$. For each $b \in B_1$ holds $|pre_1(b)| \leq 1 \wedge |post_1(b)| \leq 1$ and for each $b \in B_2$ holds $|pre_2(b)| \leq 1 \wedge |post_2(b)| \leq 1$. Thus the number of pre- and post-events could only differ for $b \in (B_1 \cap B_2) = B_{hist_2}$.

1. $pre_2(b) = \emptyset$. Then $|pre_\rho(b)| = |pre_1(b)| + |pre_2(b)| \leq 1 + 0$ by π_1 being a causal net.
2. $pre_2(b) = \{e\}$. Then o_2 enabled at o_1 implies $\{e\} = pre_1(b)$. Thus $pre_\rho(b) = pre_1(b) \cup pre_2(b) = \{e\}$.
3. $post_2(b) = \emptyset$. Then $|post_\rho(b)| = |post_1(b)| + |post_2(b)| \leq 1 + 0$ by π_1 being a causal net.
4. $post_2(b) = \{e\}$.
   If $e \in E_{hist_2}$, then $\{e\} = post_1(b)$ by o_2 enabled at o_1 (Def. 5.1 and 4.4) and π_1 being a causal net. Hence, $post_\rho(b) = post_1(b) \cup post_2(b) = \{e\}$.
   If $e \notin E_{hist_2}$, then $e \in E_{con_2}$. Hence $b \in \max hist_2$ (otherwise $b \in B_{hist_2}$ has

no post-event $e \in E_{con_2}$). Thus $b \in \max \pi_1$ by o_2 enabled at o_1 (Def. 5.1 and 4.4), which implies $post_1(b) = \emptyset$. Hence, $post_\rho(b) = post_1(b) \cup post_2(b) = \{e\} \cup \emptyset$.

Thus $|pre_\rho(b)| \leq 1 \wedge |post_\rho(b)| \leq 1$, for all $b \in B_\rho$. Taking (1), (2), and (3) together, we have shown that $\pi_1 \cup \pi_2$ is a causal net (Def. 2.6). □

Lemma 5.6: Appending an oclet yields a continuation. Lemma 5.6 states that in the composition $o_1 \rhd o_2 = (\pi_1 \cup \pi_2, hist_1)$, π_1 is a prefix of $\pi_1 \cup \pi_2$. The proof follows from the definition of prefix in Def. 2.10.

Proof (of Lemma 5.6). Let $\rho := \pi_1 \cup \pi_2$ where $o_1 = (\pi_1, hist_2)$, $o_2 = (\pi_2, hist_2)$ are oclets s.t. o_2 is enabled at o_1. We first show that π_1 is prefix of ρ and the prove the completeness condition.

By Lem. 5.5, ρ is a distributed run. Thus $\pi_1 \sqsubseteq \rho$ iff (1) $X_1 \subseteq X_\rho$, (2) $F_1 = F_\rho|_{(X_1 \times X_1)}$, (3) $\forall (x, y) \in F_\rho : y \in X_1 \Rightarrow x \in X_1$, and (4) $\forall e \in E_1 : post_1(e) = post_\rho(e)$ by Def. 2.10.

(1) $X_1 \subseteq X_\rho$ holds by $\rho = \pi_1 \cup \pi_2$, Def. 2.4.

(2) $F_\rho|_{(X_1 \times X_1)} = (F_1 \cup F_2)|_{(X_1 \times X_1)} = F_1 \cup F_2|_{(X_1 \times X_1)} = F_1 \cup F_2|_{(X_{hist_2} \times X_{hist_2})} = F_1 \cup F_{hist_2} = F_1$.

- 1st equality holds by $\rho = \pi_1 \cup \pi_2$.
- 3rd equality holds by $X_1 \cap X_2 = X_{hist_2}$, as follows: $X_1 \cap X_2 = X_1 \cap (X_{hist_2} \cup X_{con_2}) = (X_1 \cap X_{hist_2}) \cup (X_1 \cap X_{con_2}) = X_{hist_2}$ by Def. 4.1. $X_{hist_2} \subseteq X_1$ (by $hist_2 \subseteq \pi_1$ because o_2 enabled at o_1, Def.5.1), and $X_{con_2} \cap X_1 = \emptyset$ (by $con_2 \cap \pi_1 = \emptyset$, by assumption on o_2 in Def. 5.2) hold. Hence the proposition.
- 4th equality holds by $hist_2 \sqsubseteq \pi_2$.
- 5th equality holds by $hist_2 \subseteq \pi_1$ (by o_2 enabled at o_1, Def. 5.1).

(3) Let $(x, y) \in F_\rho$ and let $y \in X_1$. $F_\rho = F_1 \cup F_2$ by $\rho = \pi_1 \cup \pi_2$. Thus, we distinguish two cases:

1. $(x, y) \in F_1$. Then $x \in X_1$ by $F_1 \subseteq X_1 \times X_1$ by definition of Petri nets, Def. 2.1.
2. $(x, y) \in F_2$. Then $y \in X_{hist_2} \subseteq X_1$ by $hist_2 \subseteq \pi_1$ and $con_2 \cap \pi_1 = \emptyset$. Thus, $hist_2 \sqsubseteq \pi_2$ implies: $\forall (x, y) \in F_2 : y \in X_{hist_2} \Rightarrow x \in X_{hist_2}$ (Def. 2.10). Thus $x \in X_{hist_2} \subseteq X_1$.

(4) It remains to show that for each event $e \in E_1$ holds $post_1(e) = post_\rho(e)$ (by Def. 2.10). By $\rho = \pi_1 \cup \pi_2$ we have for each $e \in E_1$: $post_\rho(e) = post_1(e) \cup post_2(e)$. If $e \notin E_2$ then $post_\rho(e) = post_1(e) \cup post_2(e) = post_1(e)$ by $post_2(e) = \emptyset$. If $e \in E_2$ then $e \in E_{hist_2}$ by assumption; moreover $hist_2 \subseteq \pi_1$ by assumption. Thus $post_1(e) \subseteq post_2(e)$, hence $post_\rho(e) = post_1(e) \cup post_2(e) = post_1(e)$. Altogether, $\pi_1 \sqsubseteq \rho = \pi_1 \cup \pi_2$. □

A.8. Branching Processes

Every oclet system has a unique maximal branching process

Section 7.2.4 introduces the set of all branching processes of an oclet system Ω. Theorem 7.12 claims that Ω has a unique maximal branching process $\beta(\Omega)$ if the oclets of Ω are label consistent. The proof of this theorem follows from Lemmas 7.14 and 7.13 which are proven formally in this section.

Extending branching processes is confluent. Lemma 7.14 states that a branching process β can be composed with possible extensions in any order; but only if the possible extensions are label consistent.

Proof (of Lemma 7.14). Let $\Omega = (O, \pi_0)$ be an oclet system with label consistent basic oclets O. Let $[o_1], [o_2] \in O$. Let β be a branching process of Ω and w.l.o.g. let o_1 and o_2 be possible extensions of β at cuts B_1 and B_2, respectively. There are two cases.

(1) The events of o_1 and o_2 have different labels. W.l.o.g $con_1 \cap con_2 = \emptyset$. Then it follows from the respective definitions that B_2 is a cut of $(bp \rhd o_1)$ and o_2 is a possible extension of $(bp \rhd o_1)$ at B_2; correspondingly for B_1 and o_1 in $(\beta \rhd o_2)$. Thus, $(\beta \rhd o_1) \rhd o_2 = (\beta \cup \pi_1) \cup \pi_2 = (\beta \cup \pi_2) \cup \pi_1 = (\beta \rhd o_2) \rhd o_1$.

(2) The events of o_1 and o_2 have the same label. If the respective events $e_1 = event(o)$ and $e_2 = event(o_2)$ are appended to different conditions ${}^\bullet e_1 \neq {}^\bullet e_2$ of β, then the arguments of case (1) hold and $(\beta \rhd o_1) \rhd o_2 = (\beta \rhd o_2) \rhd o_1$. If ${}^\bullet e_1 = {}^\bullet e_2$, then label consistency of O implies that con_1 and con_2 are isomorphic. Thus, $\beta \rhd o_1 = \beta \rhd o_2$. □

Unique maximal branching process of oclet systems.

Proof (of Lemma 7.13). Let $\Omega = (O, \pi_0)$ be an oclet system with label consistent basic oclets O. We prove the existence of a unique, maximal branching process $\beta(\Omega)$ by relating each branching process of Ω to a set R of distributed runs of Ω. The largest set $R(\Omega)$ of runs of Ω defines $\beta(\Omega)$.

(1) Let β be a branching process of Ω and let C be a configuration of β. The distributed run $\beta[C]$ is a distributed run of Ω; this can be proven directly by induction on the number of events in C.

(2) Thus, the set $R(\beta)$ of runs of β is a set of runs of Ω by definition of $R(\beta)$ (Def. 7.5). Moreover, $R(\beta) \subseteq R(\Omega)$ because $R(\Omega)$ is the set of all distributed runs of Ω.

(3) Let $\beta^{(1)} \sqsubseteq \beta^{(2)} \sqsubseteq \beta^{(3)} \sqsubseteq \ldots$ be a monotone sequence of branching processes of Ω s.t. $\beta^{(i)} \neq \beta^{(i+1)}$, for all $i \geq 1$. So, Corollary 7.6 implies $R(\beta^{(1)}) \subset R(\beta^{(2)}) \subset R(\beta^{(3)}) \subset \ldots \subseteq R(\Omega)$. By definition, the runs in $R(\beta^{(i)})$ correspond bijectively to the configurations in $\beta^{(i)}$. Thus the sequence contains a maximal branching process β^* with $\beta^{(i)} \sqsubseteq \beta^*$ and $R(\beta^{(i)}) \subset R(\beta^*) \subseteq R(\Omega)$, for all $i \geq 1$.

We now show that every such monotone sequence eventually converges to the unique maximal branching process $\beta(\Omega)$ with $R(\beta(\Omega)) = R(\Omega)$.

(4) Let $\beta_1^{(1)} \sqsubseteq \beta_1^{(2)} \sqsubseteq \beta_1^{(3)} \sqsubseteq \ldots \sqsubseteq \beta_1^*$ and $\beta_2^{(1)} \sqsubseteq \beta_2^{(2)} \sqsubseteq \beta_2^{(3)} \sqsubseteq \ldots \sqsubseteq \beta_2^*$ be two maximal sequences of branching processes s.t. each $\beta_i^{(j)}$ has a possible extension $(o_i^{(j)}, B_i^{(j)})$ with $\beta_i^{(j)} \rhd o_i^{(j)} = \beta_i^{(j+1)}$ for $j > 1, i = 1, 2$. The maximality of both sequences implies that there exists no branching process β^* of Ω s.t. $\beta_i^* \sqsubseteq \beta^*$ and $\beta_1^* \neq \beta_2^*$, for $i = 1, 2$.

Claim: $\beta_1^* = \beta_2^*$. Assume by contradiction that $\beta_1^* \neq \beta_2^*$. There are two cases:

1. $\beta_1^* \subseteq \beta_2^*$. Then by construction β_1^* is a prefix of β_2^*. But this contradicts the maximality of β_1^*. Correspondingly for $\beta_2^* \subseteq \beta_1^*$.
2. β_1^* contains events D_1 that do not occur in β_2^*, and β_2^* contains events D_2 that do not occur in β_1^*.
   Let $e \in D_1$ be minimal wrt. $<$, i.e., the events $C := \{e' \mid e' < e\}$ are events of β_1^* and β_2^*. By construction, C is a configuration of β_1^* and β_2^* and the distributed runs induced by C are identical: $\beta_1^*[C] = \beta_2^*[C]$. Because e is an event of β_1^* there exists a possible extension o of β_1^*. From the definition of possible extensions follows that the oclet o is also enabled at $\beta_2^*[C]$ and $\beta_2^*[C] \rhd o$ can be constructed.
   We now show that also $\beta_2^* \rhd o$ can be constructed. Because of the confluence of $\rhd$ (Lemma 7.14) we may assume w.l.o.g. that there exists an index $j > 0$ s.t. $\beta_2^*[C] = \beta_2^{(j)}$.
   Thus, $o_2^{(j)}$ and o are possible extensions of $\beta_2^{(j)}$ and $\beta_2^{(j)} \rhd o \neq \beta_2^{(j)} \rhd o_2^{(j)}$ by assumption on o. The confluence of $\rhd$ implies that o is a possible extension of $\beta_2^{(j)} \rhd o_2^{(j)} = \beta_2^{(j+1)}$. This arguments applies to all successors of $\beta_2^{(j+1)}$ in the sequence. Thus, o is a possible extension of β_2^* and $\beta_2^* \rhd o$ is a branching process of Ω. But $\beta_2^* \rhd o$ contains β_2^* as a strict prefix which contradicts the maximality of β_2^*. The same argument applies to the events D_2 of β_2^* not contained in β_1^*.

Thus both sequences converge to $\beta_1^* = \beta_2^*$.

(5) Let β^* be the unique largest element of a monotone sequence of branching processes of Ω. Claim: $R(\beta^*) = R(\Omega)$. By contradiction, let $\rho \in R(\Omega)$ be a least run of Ω s.t. there exists no configuration C of β^* with $\beta^*[C] = \rho$.

By construction of β^*, ρ is not the initial run π_0 of Ω because π_0 is prefix of every branching process of Ω and hence $\pi_0 \in R(\beta^*)$.

Let $\pi \sqsubseteq \rho$ and $o \in O$ s.t. $\rho = \pi \rhd o \in R(\Omega)$. We show that o is a possible extension of β^* by applying Definition 7.8.

1. Because ρ is minimal, it holds $\pi \in R(\beta^*)$. Thus, there exists a configuration C_π s.t. $\beta^*[C_\pi] = \pi$; o is enabled at $\beta^*[C_\pi]$.
2. From $\pi \rhd o \notin R(\Omega)$ follows $con_o \cap \beta^* = \emptyset$.
3. We have to show that β^* contains no event f having the same label and the same pre-conditions as $e := event(o)$.

Assume by contradiction that f is an event of β^* with ${}^\bullet f = {}^\bullet e \subseteq Cut(C_\pi)$. Then $C \cup \{f\}$ is a configuration of β^* and $\beta^*[C_\pi \cup \{f\}] = \pi \rhd o$ because O is label consistent. But this implies $\rho = \pi \rhd o = \beta^*[C_\pi \cup \{f\}] \in R(\beta^*)$ which contradicts that ρ is the least run of Ω not represented in β^*.

Thus o is a possible extension of β^* and we may construct $\beta^* \rhd o$ which contradicts the maximality of β^*.

Thus every sequence $\beta_1 \sqsubseteq \beta_2 \sqsubseteq \beta_3 \sqsubseteq \ldots$ of branching processes of Ω constructed according to Definition 7.10 eventually reaches the unique fixed point β^* that represents all runs of Ω, $R(\beta^*) = R(\Omega)$. □

A.9. Relation Between Oclets and Petri Nets

Undecidability for oclets

Figure 8.7 in Section 8.2.2 shows an example of an oclet specification which cannot be implemented equivalently by an unlabeled Petri net. Theorem 8.1 states that the difference between Petri nets and oclets is non-trivial: whether a marking is reachable is decidable for Petri nets but undecidable for oclet specifications.

Proof of Theorem 8.1

Proof. The undecidability of oclets follows from a reduction to *Post's Correspondence Problem* (PCP) [100]. A PCP instance P is given by a finite alphabet Γ and a finite set of pairs $(x_i, y_i), i = 1, \ldots, n$ of finite words $x_i, y_i \in \Gamma^+$. The problem is to decide whether there exists a finite sequence $i_1, i_2, \ldots, i_k$ of indices between 1 and n s.t. $x_{i_1} x_{i_2} \ldots x_{i_k} = y_{i_1} y_{i_2} \ldots y_{i_k}$; this problem is known to be undecidable.

The proof that reachability in oclets is undecidable shows that every PCP instance P can be translated to an oclet specification O s.t. P has a solution iff a specific marking is reachable in O.

The idea for the reduction is to translate each pair (x_i, y_i) to an oclet o_i defining two concurrent sequences of events x_i and y_i. Let $x_i = a_i^1 a_i^2 \ldots a_i^{n_i}$ and $y_i = b_i^1 b_i^2 \ldots b_i^{m_i}$ for $a_i^j, b_i^k \in \Gamma, 1 \leq i \leq n_i, 1 \leq j \leq m_i$. The oclet $\mathsf{rule}(i)$ of Figure A.2 represents the pair (x_i, y_i). There exists a corresponding oclet for each $i = 1, \ldots, n$. The oclets $\mathsf{rule}(i), i = 1, \ldots, n$ can be composed in any order to a finite run. Any sequence $i_1, i_2, \ldots, i_k$ is equivalently expressed by the finite run

$$\pi := \mathsf{init} \rhd \mathsf{start} \rhd \mathsf{rule}(i_1) \rhd \mathsf{rule}(i_2) \rhd \ldots \rhd \mathsf{rule}(i_k).$$

The maximal conditions of π equivalently represent the sequences $x = x_{i_1} x_{i_2} \ldots x_{i_k}$ and $y = y_{i_1} y_{i_2} \ldots y_{i_k}$. By composing $\pi := \pi \rhd \mathsf{check}$, the sequence $i_1 \ldots i_k$ is closed and cannot be extended by any of the $\mathsf{rule}(i)$ oclets.

To check whether x and y are identical in π, a family of check oclets is defined. Checking begins by checking whether the first two letters of $x = a_1 a_2 \ldots a_r$ and $y = b_1 b_2 \ldots b_s$ are identical. So, for each letter $a \in \Gamma$ define the oclet $\mathsf{check}(\mathsf{init}, a)$ of Figure A.2. $\mathsf{check}(\mathsf{init}, a)$ is enabled at π iff the very first events in π both produced a post-condition with label a. This is the case iff $a_1 = b_1 = a \in \Gamma$. Then,

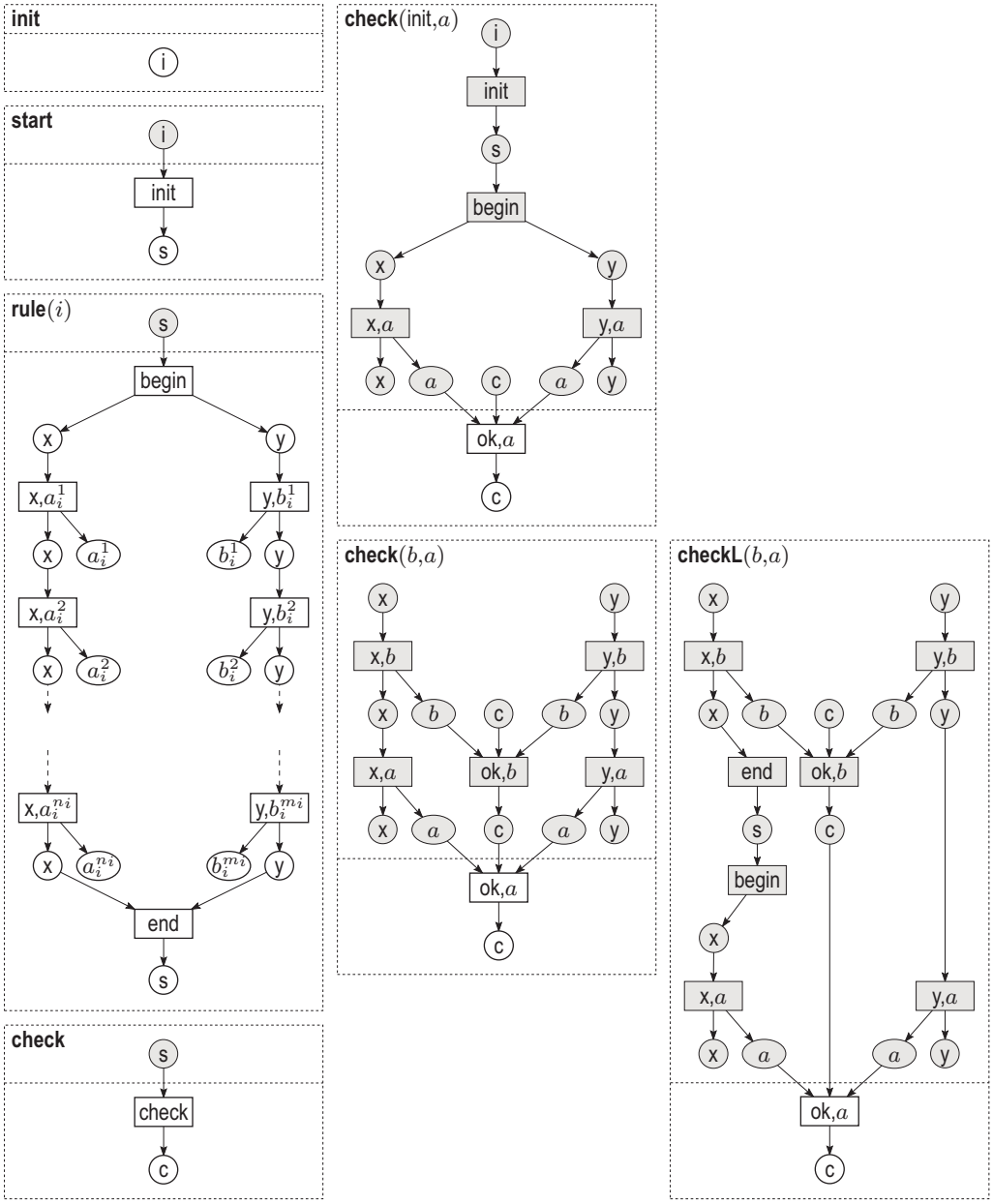


Figure A.2. Oclets to reduce PCP to reachability of the state [c].

extending $\pi := \pi \rhd \mathsf{check}(\mathsf{init}, a)$ removes both post-conditions from the maximal conditions of π.

Now, a_2 and b_2 can be compared. This is done by the oclets $\mathsf{check}(b, a)$ for any $b, a \in \Gamma$ shown in Figure A.2. Oclet $\mathsf{check}(b, a)$ is enabled at π iff x and y both have the letter b at index i *and* both have the letter a at index $i + 1$. The increment of the index i follows from the structure of π and $\mathsf{check}(b, a)$ and is not denoted explicitly in the oclets. By induction, the oclets $\mathsf{check}(b, a), b, a \in \Gamma$ advance from one event to the next in x and y simultaneously. In case one of the words, say x, reaches the end of a rule (x_i, y_i) and begins a new rule (x_{i+1}, y_{i+1}) then events end and begin occur in π. This special case is handled by the oclet $\mathsf{checkL}(b, a)$ of Figure A.2. There exists a symmetric oclet $\mathsf{checkR}(b, a)$ in case y reaches end and begin, and a corresponding oclet $\mathsf{checkLR}(b, a)$ in case both words encounter that situation.

Thus, by extending $\pi \rhd \mathsf{check}(a_1, a_2) \rhd \mathsf{check}(a_2, a_3) \rhd \ldots$, the specification advances to the largest index up to which $x = a_1 a_2 \ldots a_r$ and $y = b_1 b_2 \ldots b_s$ match. If $r = s$ and $a_i = b_i$, for all $1 \leq i \leq r$, then all maximal conditions of π that have been labeled with a letter of Γ were "consumed" by the $\mathsf{check}(.,.)$ oclets. The remaining maximal condition has the label c. In this situation, no oclet is enabled. Because Γ is finite, we only need finitely many oclets to construct and to check any instance of P. It follows from the definitions that P has a solution iff the oclet specification contains a run π that reaches a cut labeled exactly $[\mathsf{c}]$. So, reachability is undecidable in oclets because PCP is undecidable. □

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] C. B. Achour, C. Rolland, C. Souveyet, and N. A. Maiden. Guiding Use Case Authoring: Results of an Empirical Study. In *RE '99*, pages 36–43, Washington, DC, USA, 1999. IEEE Computer Society.

[3] A. V. Aho, S. Gallagher, N. D. Griffeth, C. Schell, and D. Swayne. SCF3/Sculptor with Chisel: Requirements Engineering for Communications Services. In K. Kimbler and W. Bouma, editors, *FIW*, pages 45–63. IOS Press, 1998.

[4] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[5] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

[6] R. Alur, G. J. Holzmann, and D. Peled. An Analyser for Mesage Sequence Charts. In T. Margaria and B. Steffen, editors, *TACAS*, volume 1055 of *LNCS*, pages 35–48. Springer, 1996.

[7] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In J. C. M. Baeten and S. Mauw, editors, *CONCUR*, volume 1664 of *LNCS*, pages 114–129. Springer, 1999.

[8] D. Amyot and A. Eberlein. An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunication Systems*, 24(1):61–94, 2003.

[9] Y. Atir, D. Harel, A. Kleinbort, and S. Maoz. Object Composition in Scenario-Based Programming. In J. L. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *LNCS*, pages 301–316. Springer, 2008.

[10] P. Baldan, N. Busi, A. Corradini, and G. M. Pinna. Domain and event structure semantics for Petri nets with read and inhibitor arcs. *Theoretical Computer Science*, 323(1-3):129–189, 2004.

[11] H. Ben-Abdallah and S. Leue. Timing Constraints in Message Sequence Chart Specifications. In *FORTE X PSTV XVII '97*, pages 91–106, London, UK, UK, 1998. Chapman & Hall, Ltd.

[12] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. *Fundamenta Informaticae*, 88(4):437–468, 2008.

[13] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri nets from Infinite Partial Languages. In J. Billington, Z. Duan, and M. Koutny, editors, *ACSD*, pages 170–179. IEEE, 2008.

[14] E. Best and C. Fernández. *Nonsequential Processes - A Petri Net View*, volume 13 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

[15] A. W. Biermann and R. Krishnaswamy. Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, 1976.

[16] Y. Bontemps and P. Heymans. Turning High-Level Live Sequence Charts into Automata. In *SCESM'02*, May 2002.

[17] Y. Bontemps and P. Heymans. As Fast As Sound (Lightweight Formal Scenario Synthesis and Verification). In H. Giese and I. Krüger, editors, *SCESM'04*, pages 27–34, Edinburgh, May 2004. IEE.

[18] Y. Bontemps and P.-Y. Schobbens. The computational complexity of scenario-based agent verification and design. *Journal of Applied Logic*, 5(2):252 – 276, 2007. Logic-Based Agent Verification.

[19] Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundamenta Informaticae*, 62(2):139–169, 2004.

[20] R. V. Book. On the structure of context-sensitive grammars. *International Journal of Parallel Programming*, 2(2):129–139, June 1973.

[21] F. Bordeleau, J. P. Corriveau, and B. Selic. A Scenario-Based Approach to Hierarchical State Machine Design. In *ISORC '00*, page 78, Washington, DC, USA, 2000. IEEE Computer Society.

[22] R. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, 24:1131–1155, 1998.

[23] T. Chatain and C. Jard. Complete Finite Prefixes of Symbolic Unfoldings of Safe Time Petri Nets. In S. Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *LNCS*, pages 125–145. Springer, 2006.

[24] T. Chatain and V. Khomenko. On the Well-Foundedness of Adequate Orders Used for Construction of Complete Unfolding Prefixes. *Information Processing Letters*, 104(4):129–136, 2007.

[25] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.

[26] P. Darondeau. Unbounded Petri Net Synthesis. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 413–438. Springer, 2003.

[27] J. Desel. Validation of Process Models by Construction of Process Nets. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management*, volume 1806 of *LNCS*, pages 110–128. Springer, 2000.

[28] J. Desel and J. Esparza. *Free choice Petri nets*. Cambridge University Press, New York, NY, USA, 1995.

[29] J. Desel, G. Juhás, and C. Neumair. Finite Unfoldings of Unbounded Petri Nets. In J. Cortadella and W. Reisig, editors, *ICATPN*, volume 3099 of *LNCS*, pages 157–176. Springer, 2004.

[30] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[31] S. S. R. Dssouli and J. Vaucher. Toward an Automation of Requirements Engineering using Scenarios. *Journal of Computing and Information*, 2:1110–1132, 1996.

[32] A. P.-G. Eberlein. *Requirements Acquisition and Specification for Telecommunication Services*. PhD thesis, University of Wales, Swansea, UK, Nov 1997.

[33] C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In Prinz et al. [101], pages 133–148.

[34] M. Elkoutbi and R. K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *ICATPN*, pages 166–186, 2000.

[35] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28(6):575–591, 1991.

[36] J. Esparza and K. Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Springer-Verlag, 2008.

[37] J. Esparza and M. Nielsen. Decidability Issues for Petri Nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[38] J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

[39] D. Fahland. Oclets - Scenario-Based Modeling with Petri Nets. In G. Franceschinis and K. Wolf, editors, *Petri Nets*, volume 5606 of *LNCS*, pages 223–242. Springer, 2009.

[40] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf. Instantaneous Soundness Checking of Industrial Business Process Models. In U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, editors, *BPM*, volume 5701 of *LNCS*, pages 278–293. Springer, 2009.

[41] D. Fahland, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative vs. Imperative Process Modeling Languages: The Issue of Maintainability. In B. Mutschler, R. Wieringa, and J. Recker, editors, *ER-BPM'09*, pages 65–76, Ulm, Germany, Sept. 2009. (LNBIP to appear).

[42] D. Fahland and H. Woith. Towards Process Models for Disaster Response. In D. Ardagna, M. Mecella, and J. Yang, editors, *Business Process Management Workshops*, volume 17 of *LNBIP*, pages 254–265. Springer, 2008.

[43] C. Fritz. Some Fixed Point Basics. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*, pages 359–364. Springer, 2001.

[44] T. Gazagnaire, B. Genest, L. Hélouët, P. S. Thiagarajan, and S. Yang. Causal Message Sequence Charts. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 166–180. Springer, 2007.

[45] B. Genest, M. Minea, A. Muscholl, and D. Peled. Specifying and Verifying Partial Order Properties Using Template MSCs. In I. Walukiewicz, editor, *FoSSaCS*, volume 2987 of *LNCS*, pages 195–210. Springer, 2004.

[46] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In W. Schäfer and P. Botella, editors, *ESEC*, volume 989 of *LNCS*, pages 254–271. Springer, 1995.

[47] U. Goltz and W. Reisig. Processes of Place/Transition-Nets. In J. Díaz, editor, *ICALP*, volume 154 of *LNCS*, pages 264–277. Springer, 1983.

[48] Y. Gurevich. *Specification and validation methods*, chapter Evolving algebras 1993: Lipari guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[49] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. In *CIAA '00*, volume 2088 of *LNCS*, pages 1–33, London, UK, 2001. Springer-Verlag.

[50] D. Harel, H. Kugler, S. Maoz, and I. Segall. Accelerating Smart Play-Out. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM*, volume 5901 of *LNCS*, pages 477–488. Springer, 2010.

[51] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-out of Behavioral Requirements. In *FMCAD'02*, pages 378–398, London, UK, 2002. Springer-Verlag.

[52] D. Harel, H. Kugler, and A. Pnueli. Smart Play-Out Extended: Time and Forbidden Elements. In *QSIC*, pages 2–10. IEEE Computer Society, 2004.

[53] D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 309–324. Springer, 2005.

[54] D. Harel, S. Maoz, and I. Segall. Some Results on the Expressive Power and Complexity of LSCs. In A. Avron, N. Dershowitz, and A. Rabinovich, editors, *Pillars of Computer Science*, volume 4800 of *LNCS*, pages 351–366. Springer, 2008.

[55] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[56] D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: the Play-in/Play-out Approach. *Software and System Modeling*, 2(2):82–107, 2003.

[57] D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 485–499. Springer, 2007.

[58] D. Harel and P. S. Thiagarajan. *UML for real: design of embedded real-time systems*, chapter Message Sequence Charts, pages 77–105. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[59] L. Hélouët, C. Jard, and B. Caillaud. An event structure based semantics for High-Level Message Sequence Charts. *Mathematical. Structures in Comp. Sci.*, 12(4):377–402, 2002.

[60] J. G. Henriksen, M. Mukund, K. N. Kumar, and P. S. Thiagarajan. Regular Collections of Message Sequence Charts. In M. Nielsen and B. Rovan, editors, *MFCS*, volume 1893 of *LNCS*, pages 405–414. Springer, 2000.

[61] S. Heymer. A Non-Interleaving Semantics for MSC. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*. Humboldt-Universität zu Berlin, 1998.

[62] S. Heymer. A Semantics for MSC Based on Petri Net Components. In E. Sherratt, editor, *SAM*, pages 262–275. VERIMAG, IRISA, SDL Forum, 2000.

[63] A. W. Holt. *A Mathematical Model of Continuous Discrete Behavior*. Massachusettes Computer Associates, Inc., Nov. 1980.

[64] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal Approach to Scenario Analysis. *IEEE Software*, 11:33–41, 1994.

[65] ITU-T. Message Sequence Chart (MSC). Recommendation Z.120, International Telecommunication Union, Geneva, 2004.

[66] K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modeling and Validation of Concurrent Systems*. Springer, 2009.

[67] J.-P. Katoen and L. Lambert. Pomsets for MSC. In H. König and P. Langendörfer, editors, *FBT*, pages 197–207. Verlag Shaker, 1998.

[68] I. Khriss, M. Elkoutbi, and R. K. Keller. Automatic Synthesis Of Behavioral Object Specifications From Scenarios. *Journal of Integrated Design & Process Science*, 5(3):53–77, 2001.

[69] O. Kluge. Modelling a Railway Crossing with Message Sequence Charts and Petri Nets. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *LNCS*, pages 197–218. Springer, 2003.

[70] K. Koskimies, T. Systä, J. Tuomi, and T. Männistö. Automated Support for Modeling OO Software. *IEEE Software*, 15:87–94, 1998.

[71] V. E. Kozura. Unfoldings of Coloured Petri Nets. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *LNCS*, pages 268–278. Springer, 2001.

[72] I. Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In M. Pezzè, editor, *FASE*, volume 2621 of *LNCS*, pages 387–402. Springer, 2003.

[73] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *DIPES '98*, pages 61–71, Norwell, MA, USA, 1999. Kluwer Academic Publishers.

[74] O. Kupferman and M. Vardi. *Advances in Temporal Logic*, chapter Synthesis with incomplete information, pages 109–127. Kluwer Academic, 2000.

[75] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *LICS*, 2001.

[76] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[77] J. Landgren and S. Jul, editors. *Proceeding of the 6th International ISCRAM Conference*, Gothenburg, Sweden, May 2009.

[78] D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 264–277, London, UK, 1981. Springer-Verlag.

[79] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM Models from Message Sequence Chart Specifications. Technical report, Dept. of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, 1998.

[80] H. Liang, J. Dingel, and Z. Diskin. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In *SCESM '06*, pages 5–12, New York, NY, USA, 2006. ACM.

[81] P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In D. Sangiorgi and R. de Simone, editors, *CONCUR*, volume 1466 of *LNCS*, pages 18–33. Springer, 1998.

[82] P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *ICALP*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.

[83] T. Maier and A. Zündorf. The Fujaba Statechart Synthesis Approach. In *SCESM'03*, Oregon, Portland, USA, May 2003.

[84] E. Mäkinen and T. Systä. MAS — an Interactive Synthesizer to Support Behavioral Modelling in UML. In *ICSE '01*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[85] N. Mansurov. Automatic Synthesis of SDL from MSC and its Applications in Forward and Reverse Engineering. *Computer Languages*, 27(1-3):115 – 136, 2001.

[86] H. N. C. Martínez. Synthesizing State-Machine Behaviour from UML Collaborations and Use Case Maps. In Prinz et al. [101], pages 339–359.

[87] S. Mauw and M. A. Reniers. Operational Semantics for MSC'96. *Computer Networks*, 31(17):1785–1799, 1999.

[88] K. L. McMillan. A Technique of State Space Search Based on Unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[89] P. M. Merlin and D. J. Farber. Recoverability of Communication Protocols: Implications of a Theoretical Study. *IEEE Transactions on Communication*, 24(9):1036–1043, Sep 1976.

[90] B. Mitchell, R. Thomson, and P. Bristow. Scenario Synthesis from Imprecise Requirements. In D. Amyot and A. W. Williams, editors, *SAM*, volume 3319 of *LNCS*, pages 122–137. Springer, 2004.

[91] U. Montanari. True Concurrency: Theory and Practice. In R. S. Bird, C. Morgan, and J. Woodcock, editors, *MPC*, volume 669 of *LNCS*, pages 14–17. Springer, 1992.

[92] U. Montanari and M. Pistore. An Introduction to History Dependent Automata. *Electronic Notes in Theoretical Computer Science*, 10:170 – 188, 1998. HOOTS II, Second Workshop on Higher-Order Operational Techniques in Semantics.

[93] M. Mukund, K. N. Kumar, and P. S. Thiagarajan. Netcharts: Bridging the gap between HMSCs and executable specifications. In R. M. Amadio and D. Lugiez, editors, *CONCUR*, volume 2761 of *LNCS*, pages 293–307. Springer, 2003.

[94] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part I. *Theoretical Compututer Science*, 13:85–108, 1981.

[95] Object Management Group. UML Superstructure, v2.2, formal/09-02-02. Standard, 2009.

[96] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst. Constraint-Based Workflow Models: Change Made Easy. In R. Meersman and Z. Tari, editors, *OTM Conferences (1)*, volume 4803 of *LNCS*, pages 77–94. Springer, 2007.

[97] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[98] A. Pneuli and R. Rosner. Distributed reactive systems are hard to synthesize. In *SFCS '90*, pages 746–757 vol.2, Washington, DC, USA, 1990. IEEE Computer Society.

[99] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, pages 652–671, 1989.

[100] E. L. Post. A Variant of a Recursively Unsolvable Problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

[101] A. Prinz, R. Reed, and J. Reed, editors. *SDL 2005*, volume 3530 of *LNCS*. Springer, 2005.

[102] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[103] W. Reisig. Modelling and Verification of Distributed Algorithms. In U. Montanari and V. Sassone, editors, *CONCUR*, volume 1119 of *LNCS*, pages 579–595. Springer, 1996.

[104] W. Reisig. *Elements Of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer-Verlag, Sept. 1998.

[105] S. Ren, K. Rui, and G. Butler. Refactoring the Scenario Specification: A Message Sequence Chart Approach. In D. Konstantas, M. Léonard, Y. Pigneur, and S. Patel, editors, *OOIS*, volume 2817 of *LNCS*, pages 294–298. Springer, 2003.

[106] C. Rolland and C. B. Achour. Guiding the Construction of Textual Use Case Specifications. *Data & Knowledge Engineering*, 25(1-2):125–160, 1998.

[107] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. *IEEE Transaction on Software Engineering*, 32(8):587–607, 2006.

[108] M. Sgroi, A. Kondratyev, Y. Watanabe, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of Petri Nets from Message Sequence Charts Specifications for Protocol Design. In *DASD '04*, Washington DC, USA, April 2004.

[109] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[110] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Transactions on Software Engineering*, 35(3):384–406, 2009.

[111] S. Uchitel, J. Kramer, and J. Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. *SIGSOFT Software Engineering Notes*, 26(5):74–82, 2001.

[112] S. Uchitel, J. Kramer, and J. Magee. Implied Scenario Detection in the Presence of Behaviour Constraints. *Electronic Notes in Theoretical Computer Science*, 65(7):65 – 84, 2002.

[113] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29:99–115, 2003.

[114] A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Petri Nets*, volume 1491 of *LNCS*, pages 429–528. Springer, 1996.

[115] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software and System Modeling*, 9(1):87–111, 2010.

[116] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[117] W. M. P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[118] B. F. van Dongen, A. K. A. de Medeiros, and L. Wen. Process Mining: Overview and Outlook of Petri Net Discovery Algorithms. *Transactions on Petri Nets and Other Models of Concurrency*, 2:225–242, 2009.

[119] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[120] K. M. van Hee, A. Serebrenik, N. Sidorova, and W. M. P. van der Aalst. History-Dependent Petri Nets. In J. Kleijn and A. Yakovlev, editors, *ICATPN*, volume 4546 of *LNCS*, pages 164–183. Springer, 2007.

[121] K. M. van Hee, A. Serebrenik, N. Sidorova, and W. M. P. van der Aalst. Working with the Past: Integrating History in Petri Nets. *Fundamenta Informaticae*, 88(3):387–409, 2008.

[122] K. M. van Hee, A. Serebrenik, N. Sidorova, M. Voorhoeve, and J. M. E. M. van der Werf. Modelling with History-Dependent Petri Nets. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM*, volume 4714 of *LNCS*, pages 320–327. Springer, 2007.

[123] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In B. J. Krämer, K.-J. Lin, and P. Narasimhan, editors, *ICSOC*, volume 4749 of *LNCS*, pages 43–55. Springer, 2007.

[124] H. M. W. E. Verbeek, T. Basten, and W. M. P. v. d. Aalst. Diagnosing Workflow Processes using Woflan. *Computer Journal*, 44(4):246–279, 2001.

[125] W. Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *LNCS*. Springer, 1992.

[126] J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *ICSE '00*, pages 314–323, New York, NY, USA, 2000. ACM.

[127] G. Winskel. Event Structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.

[128] M. Wolf. Erstellung einer modellbasierten Laufzeitumgebung für adaptive Prozesse. Diplomarbeit, Humboldt-Universität zu Berlin, Sept. 2008.

[129] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Revisiting Statechart Synthesis with an Algebraic Approach. In *ICSE '04*, pages 242–251, Washington, DC, USA, 2004. IEEE Computer Society.

Index

From Scenarios To Components – Summary

Scenario-based modeling has evolved as an accepted paradigm for developing complex systems of various kinds. Its main purpose is to ensure that a system provides desired behavior to its users. A *scenario* is generally understood as a behavioral *requirement*, denoting a course of actions that shall *occur* in the system. A typical notation of a scenario is a Message Sequence Chart or, more general, a finite partial order of actions. A *specification* is a set of scenarios. Intuitively, a system *implements* a specification if all scenarios of the specification can occur in the system. The main challenge in this approach is to systematically *synthesize* from a given scenario-based specification state-based components which together implement the specification; preferably to be achieved automatically. A further challenge is to *analyze* scenarios to avoid erroneous specifications.

Existing scenario-based techniques exhibit a conceptual and formal gap between a scenario-based specification on the one hand and a state-based implementation on the other hand. This gap often renders synthesis surprisingly complex, and obscures the relationship between a specification and its implementation. Additionally, existing techniques for analyzing implementations cannot immediately be reused for analyzing specifications, and vice versa.

In this thesis, we introduce a semantic model for scenarios that seamlessly integrates scenario-based specifications and state-based implementations. We focus on modeling and analyzing the *control-flow* of systems. Technically, we use Petri nets together with the well established notion of *distributed runs* for (1) describing the semantics of scenarios, for (2) systematically constructing and analyzing a specification, and for (3) synthesizing an implementation from a given specification.

Our first contribution is to identify a minimal set of notions to specify the behavior of distributed systems with scenarios. We formalize these notions in a novel semantic model for scenarios, called *oclets*. Oclets combine formal notions from Petri net theory with formal notions from scenario-based techniques in a unified way. We define a classical *declarative semantics* for scenario-based specifications, which defines when a given set of runs *satisfies* a given specification. These semantics are *compositional*: a set of runs satisfies a composition of two specifications if and only if it satisfies each of the specifications. We then provide *composition* and *decomposition operators* on oclets and relations for comparing oclets. Using these notions, we systematically derive for each scenario-based specification S the behavior exhibited by a *minimal implementation* of S.

The second contribution of this thesis aims at bridging the conceptual and methodological gap between scenario-based specifications and state-based system models. In our approach, the semantics of scenarios and the semantics of systems both employ the same notions from Petri nets. We provide *operational semantics* for scenario-based specifications. On the basis of these operational semantics, we consider the problems of *analyzing* behavioral properties of a specification and of *synthesizing* components that implement the specification. We show that these problems are undecidable in general and present a sufficient property for the decidable case. Based on this property, we present algorithms for analysis and

synthesis. Our results generalize existing techniques from Petri nets to the domain of scenario-based specifications.

We implemented our algorithms for simulating, analyzing, and synthesizing from scenario-based specifications in our tool GRETA. An industrial case study shows the feasibility of our techniques.

From Scenarios To Components – Zusammenfassung

Szenario-basiertes Modellieren hat sich zu einer akzeptierten Entwurfstechnik für komplexe Systeme verschiedenster Art entwickelt. Mit ihr soll sichergestellt werden, dass ein System sich wie gewünscht gegenüber seinen Nutzern verhält. Ein *Szenario* beschreibt im Allgemeinen eine *Anforderung* an das Verhalten des Systems als eine Folge von Aktionen, die im System *vorkommen* soll. Häufig wird ein Szenario als ein Message Sequence Chart oder allgemein als halbgeordnete Menge von Aktionen notiert. Eine *Spezifikation* ist eine Menge von Szenarien. Intuitiv *implementiert* ein System eine Spezifikation wenn alle Szenarien der Spezifikation im System vorkommen können. Die Hauptschwierigkeit dieses Ansatzes besteht darin, aus einer szenario-basierten Spezifikation systematisch zustands-basierte *Komponenten* zu *synthetisieren*, die die Spezifikation gemeinsam implementieren; idealerweise gelingt dies automatisch. Eine weitere Herausforderung ist die *Analyse* von Szenarien, um fehlerhafte Spezifikationen zu vermeiden.

Existierende szenario-basierte Techniken weisen eine konzeptuelle Lücke zwischen einer szenario-basierten Spezifikationen und einer zustands-basierten Implementation auf. Aufgrund dieser Lücke ist die Synthese überraschend komplex und die Beziehung zwischen Spezifikation und Implementation nur schwer nachvollziehbar. Darüberhinaus können Analysetechniken für eine Implementation nicht unmittelbar zur Analyse einer Spezifikation wiederverwendet werden und umgekehrt.

In diese Arbeit führen wir ein Verhaltensmodell für Szenarien ein, das szenario-basierte Spezifikationen und zustands-basierte Implementationen nahtlos integriert. Wir konzentrieren uns hierbei auf die Modellierung und die Analyse des *Kontrollflusses* von Systemen. Auf der technischen Ebene verwenden wir Petrinetze und ihre *verteilten Abläufe*, um (a) die Semantik von Szenarien zu beschreiben, (b) eine Spezifikation systematisch zu konstruieren und zu analysieren, sowie (c) eine Implementation aus einer Spezifikation zu synthetisieren.

Zunächst identifizieren wir eine minimale Menge von Begriffen mit denen Systemverhalten mittels Szenarien spezifiziert werden kann. Wir formalisieren diese Begriffe in einem neuen Verhaltensmodell für Szenarien, die wir *Oclets* nennen. Dabei vereinheitlichen Oclets formale Begriffe von Petrinetzen und szenario-basierter Techniken in einem Modell. Wir definieren eine klassische *deklarative Semantik* für szenario-basierte Spezifikationen, die festlegt wann eine gegebene Menge von Abläufen eine Spezifikation *erfüllt*. Diese Semantik ist *kompositional*: eine Menge von Abläufen erfüllt eine Komposition zweier Spezifikationen genau dann, wenn sie jede der Spezifikationen erfüllt. Anschließend führen wir *Kompositions-* und *Dekompositionsoperatoren* auf Oclets ein sowie Relationen, um Oclets miteinander zu vergleichen. Mit diesen Begriffen lässt sich für jede szenario-basierte Spezifikation S das Verhalten einer *minimalen Implementation* von S systematisch ableiten.

Der zweite Beitrag dieser Arbeit hat zum Ziel, die konzeptuelle Lücke zwischen szenario-basierten Spezifikationen und zustands-basierten Systemmodellen zu

schließen. In unserem Ansatz basieren die Semantik von Szenarien und die Semantik von Systemen auf den selben Begriffen der Petrinetztheorie. Darauf aufbauend entwickeln wir eine *operationelle Semantik* für szenario-basierte Spezifikationen. Anhand dieser Semantik gehen wir die Probleme an, Verhaltenseigenschaften von Spezifikationen zu *analysieren* sowie Komponenten zu *synthetisieren*, die die Spezifikation implementieren. Wir zeigen, dass diese Probleme im Allgemeinen unentscheidbar sind und stellen ein hinreichendes Kriterium für den entscheidbaren Fall vor. Dann entwickeln wir Algorithmen zur Analyse und Synthese. Wir erhalten diese Ergebnisse, in dem wir bestehende Techniken der Petrinetzetheorie auf szenario-basierte Spezifikationen verallgemeinern.

Die vorgestellten Algorithmen zur Simulation, Analyse und Synthese von szenario-basierten Spezifikationen sind in unserem Werkzeug Greta implementiert. Wir stellen Ergebnisse aus einer industriellen Fallstudie vor, die die Tauglichkeit unserer Techniken in der Praxis belegen.

Acknowledgements

When I began my studies in informatics, I had thought of a career in industry, because this was where I was good at already. When Wolfgang Reisig asked me to join his group in Berlin in 2002, I was not sure whether I would feel comfortable in research. Since then he kept pushing the bar higher and higher, making me run my own research, getting scholar ships, going abroad, and coming into contact with an international research community. And somewhere in between, the last doubt about starting a PhD project vanished.

Ever since, he gave me the freedom and the resources to explore my ideas, meet the right people, and initiated the B.E.S.T programme from which I benefited tremendously during the project. The most important lesson he taught me in that time was to never be satisfied with what I have achieved, to strive for a better, more beautiful result, and to always look at the big picture by which others will understand what you are doing. I hope this thesis convinces him that I did learn something.

I would not have finished this thesis without the help of various people. In the following, I will express my gratitude to them. My co-promotor Karsten Wolf could envision where my research ideas would lead me to long before I understood what I was working on. He helped me structuring my ideas when they were immature, to scope my project, and to think beyond the boundaries that I had set for myself. Discussions with him were always fruitful and encouraging, and I enjoyed the hospitality in his group and at his home.

Wil van der Aalst was one of the first to give me positive feedback on my ideas. His critics and suggestions always broadened and sharpened my view on my thesis and the problems that I was dealing with. Wil's feedback was most valuable during the writing of the thesis, as he could show me where the small and the large did not fit together, helping me to write a readable text from my pile of ideas. I very much enjoyed my many stays at his group in Eindhoven within the B.E.S.T programme in a working atmosphere in which I often made substantial progress in my research.

I am grateful to David Harel, Holger Schlingloff, Kees van Hee, Hajo Reijers, and Arjeh Cohen for their service as committee members. Especially, David Harel whose talk at the Petri net Summer School 2003 in Eichstätt inspired me to think about system design in unusual ways. The discussions with his group have been very fruitful for my work, and his feedback on synthesis of systems helped me to position my results.

This thesis would not have come to a successful end without the encouraging discussions with Christian Stahl. His advice was most valuable when I had to frame what would be part of this thesis, and made me discover and name the strengths of my work. He proof-read many parts of the thesis.

I am greatly indebted to Amir Kantor with whom I had the most challenging, most thought provoking, and most fundamental discussions about my research. He instantly understood my ideas, and always provided a very inspiring perspective on them. Together, we worked out many of the underlying assumptions and subtle

details of my research. Chapter 3 documents these insights at large, and the main result of Chapter 5 would look different without his contribution.

I also thank many other fellows. Peter Massuthe had endless patience when we together discussed the many technical details of my work. He also reminded me again of how to present my ideas to an audience. Niels Lohmann made me work on the tools that efficiently implement the results of this thesis. Without the tricks and insights which he shared, the tools would not have reached this level of maturity. I am greatly indebted to Manja Wolf. She gave me Greta which allowed me to test my ideas when they were yet immature, and to give my results a face to show to other people. I thank Christian Gierds, Jan Suermeli, Richard Müller, and Johannes Fichte for the many small and long discussions about the tiny problems that constantly arise when writing a PhD thesis, especially, Johannes who pointed me at Knaster-Tarski.

I owe much to Achim Fischer who always supported my research during the entire project. This thesis would not have been written without P.S. Thiagarjan, who taught me many of the techniques I applied in this thesis. I thank Wolfgang Thomas for an inspiring talk in Berlin which made me discover the expressive power of oclets. I would not have been able to demonstrate the feasibility of my results in practice without the cooperation with Jana Koehler and Hagen Völzer on the analysis of industrial business processes. Likewise, the scenic presentations by Heiko Woith about his practical experiences in disaster situations gave me the understanding of complex processes and allowed me to test my ideas on a real challenge.

Special thanks go to the "Kaffeerunde" in Berlin, my fellows from METRIK, and the members of the AIS group in Eindhoven. Everywhere, I enjoyed a warm and pleasant working atmosphere in scientific and not-so scientific discussions. Many thanks I would like to address to the secretaries in Berlin and Eindhoven, Birgit Heene, Riet van Buul, and Ine van der Ligt, for their help and support in so many administrative things, and in mastering the surprises that life gives us.

Most of all, I thank my family and my friends. My family, who never complained when I had to work even on weekends, and who supported me with everything I needed, whenever I needed it. Thank you so much! My friends, who never stopped asking me out for a little bit of distraction and relaxation, who understood that I often had to work, and who tried to understand what I was actually doing in the least years. Their support and interest was a constant source of energy. Very special thanks go to Annette for the most leveraging discussions and hints how to organize complex work, to Katja for her detailed proof reading of several chapters, to Andrea and Conny for her wise advices and the cheerful moments, to Svenno for making sure that I did not grow onto my desk, and to Manne for taking care of everything I could not take care of in the last years.

Dirk Fahland
Berlin, Juli 2010

Erklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift “From Scenarios to Components” selbständig und ohne unerlaubte Hilfe angefertigt sowie nur die angegebene Literatur verwendet habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder ich einen solchen besitze und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin (veröffentlicht im Amtlichen Mitteilungsblatt Nr. 34/2006) bekannt ist.

Dirk Fahland
Berlin, den 30. Juli 2010

Tabellarischer Lebenslauf

8.12.1980	geboren in Berlin
09/1987-08/1989	Kurt-Ritter-Schule Berlin-Friedrichshain
09/1989-09/1990	Edgar-André-Oberschule mit erweitertem Sprachunterricht (Russisch)
10/1990-06/2000	Umwandlung in Erich-Fried-Gymnasium, Berlin- Friedrichshain, Abitur
07/2000-06/2001	Zivildienst in der Sozialstation des Johannischen Sozialwerks in Berlin-Friedrichshain
07/2001-09/2001	Praktikum bei ArgusSoft GmbH, Bernau b. Berlin
10/2001-07/2006	Studium der Informatik an der Humboldt-Universität zu Berlin mit Nebenfach Steuerungs- und Regelungstechnik an der TU Berlin; Studienschwerpunkt: Spezifikation und Verifikation verteilter Systeme, theoretische Grundlagen der Informatik; Abschluss als Diplom-Informatiker
04/2002-07/2005	Studentische Hilfskraft am Lehrstuhl "Theorie der Programmierung" (Prof. Wolfgang Reisig), Institut für Informatik, Humboldt-Universität zu Berlin; Eigenverantwortliche Mitarbeit an Projekten des Lehrstuhls
05/2003-07/2006	Stipendiat der Studienstiftung des deutschen Volkes
08/2005-05/2006	Auslandsjahr an der National University of Singapore, unterstützt mit einem Jahresstipendium des Deutschen Akademischen Austauschdienstes (DAAD)
08/2006-07/2009	Promotionsstudium im DFG-Graduiertenkolleg METRIK (GRK 1324) betreut von Prof. Wolfgang Reisig, Forschungsschwerpunkt auf Spezifikation und Verifikation verteilter Systeme mittels formaler Methoden, insbes. Petrinetzen, temporaler Logik und szenariobasierter Techniken
seit 09/2006	Mitglied im B.E.S.T-Projekt (Berlin-Rostock-Eindhoven Service Technology) mit regelmäSSigen Aufenthalten bei der "Architecture of Information Systems Group" (Prof. Kees van Hee/Prof. Wil van der Aalst) der TU Eindhoven
09/2007-03/2009	Ideelle Promotionsförderung der Studienstiftung des deutschen Volkes
seit 08/2009	Wissenschaftlicher Mitarbeiter am Lehrstuhl "Theorie der Programmierung" (Prof. Wolfgang Reisig), Institut für Informatik, Humboldt-Universität zu Berlin

Curriculum Vitæ

8.12.1980	born in Berlin
09/1987-08/1989	primary school *Kurt Ritter* Berlin-Friedrichshain

09/1989-09/1990	high school *Edgar André* with extended language courses (Russian)
10/1990-06/2000	transformation of high school into a grammar school (*Erich-Fried-Gymnasium, Berlin-Friedrichshain*), university-entrance diploma (Abitur)
07/2000-06/2001	Civil service at the *Johannische Sozialwerk in Berlin-Friedrichshain*
07/2001-09/2001	Internship at *ArgusSoft GmbH*, Bernau b. Berlin
10/2001-07/2006	studies of informatics at the *Humboldt-Universität zu Berlin* with minor subject control engineering at *TU Berlin*; specialization in specification and verification of distributed systems, theoretical foundations of informatics; degree *Diplom-Informatiker*
04/2002-07/2005	student assisstant in the "Theory of Programming" group (Prof. Wolfgang Reisig), department of computer science, Humboldt-Universität zu Berlin
05/2003-07/2006	scholarship by the German National Academic Foundation (*Studienstiftung des deutschen Volkes*)
08/2005-05/2006	student exchange year at the *National University of Singapore*, supported by a 1-year scholarship of the German Academic Exchange Service (DAAD)
08/2006-07/2009	PhD graduate school *METRIK* at the Humboldt-Universität zu Berlin, supervised by Prof. Wolfgang Reisig and Prof. Wil van der Aalst, research focus on specification and verification of distributed systems with formal methods, esp. Petri nets, temporal logics, and scenario-based techniques
since 09/2006	member of the B.E.S.T-Projekt (Berlin-Rostock-Eindhoven Service Technology) with regular research visits and workshops at the "Architecture of Information Systems Group" (Prof. Kees van Hee/Prof. Wil van der Aalst) at the Technical University Eindhoven
09/2007-03/2009	PhD Fellowship by the German National Academic Foundation (*Studienstiftung des deutschen Volkes*)
since 08/2009	research assisstant in the "Theory of Programming" group (Prof. Wolfgang Reisig), department of computer science, Humboldt-Universität zu Berlin

SIKS Dissertatiereeks

1998

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects
1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information
1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspectives
1998-4 Dennis Breuker (UM) Memory versus Search in Games
1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets
1999-3 Don Beal (UM) The Nature of Minimax Search
1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects
1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems
1999-7 David Spelt (UT) Verification support for object database design
1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance
2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management
2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design
2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.
2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication
2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management
2000-8 Veerle Coupâ (EUR) Sensitivity Analyis of Decision-Theoretic Networks
2000-9 Florian Waas (CWI) Principles of Probabilistic Query Optimization
2000-10 Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture
2000-11 Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks
2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models
2001-3 Maarten van Someren (UvA) Learning as problem solving
2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
2001-6 Martijn van Welie (VU) Task-based User Interface Design
2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9 Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10 Maarten Sierhuis (UvA) Modeling and Simulating Work Practice; BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11 Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU) Architecture-Level Modifiability Analysis
2002-02 Roelof van Zwol (UT) Modelling and searching web-based document collections
2002-03 Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval
2002-04 Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05 Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07 Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09 Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems
2002-10 Brian Sheppard (UM) Towards Perfect Play of Scrabble
2002-11 Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems
2002-13 Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications
2002-14 Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15 Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16 Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications
2002-17 Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments
2003-02 Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems
2003-03 Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04 Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology
2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach
2003-06 Boris van Schooten (UT) Development and specification of virtual environments
2003-07 Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks
2003-08 Yongping Ran (UM) Repair Based Scheduling
2003-09 Rens Kortmann (UM) The resolution of visually guided behaviour
2003-10 Andreas Lincke (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
2003-11 Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12 Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval
2003-13 Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models
2003-14 Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15 Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems
2003-16 Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
2003-17 David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing
2003-18 Levente Kocsis (UM) Learning Search Decisions

2004

2004-01 Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02 Lai Xu (UvT) Monitoring Multi-party Contracts for E-business
2004-03 Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04 Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures
2004-05 Viara Popova (EUR) Knowledge discovery and monotonicity
2004-06 Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques
2004-07 Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08 Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiäle gegevensuitwisseling en digitale expertise
2004-09 Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning
2004-10 Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects
2004-11 Michel Klein (VU) Change Management for Distributed Ontologies
2004-12 The Duy Bui (UT) Creating emotions and facial expressions for embodied agents
2004-13 Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play
2004-14 Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU) Multi-Relational Data Mining
2004-16 Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning
2004-17 Mark Winands (UM) Informed Search in Complex Games
2004-18 Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models
2004-19 Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2004-20 Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications
2005-02 Erik van der Werf (UM)) AI techniques for the game of Go
2005-03 Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language
2005-04 Nirvana Meratnia (UT) Towards Database Support for Moving Object data
2005-05 Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06 Pieter Spronck (UM) Adaptive Game AI
2005-07 Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems
2005-08 Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications
2005-09 Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages
2005-10 Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments
2005-11 Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
2005-12 Csaba Boer (EUR) Distributed Simulation in Industry
2005-13 Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
2005-14 Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
2005-15 Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes
2005-16 Joris Graaumans (UU) Usability of XML Query Languages
2005-17 Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components
2005-18 Danielle Sent (UU) Test-selection strategies for probabilistic networks
2005-19 Michel van Dartel (UM) Situated Representation
2005-20 Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives
2005-21 Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 Samuil Angelov (TUE) Foundations of B2B Electronic Contracting
2006-02 Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations
2006-03 Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems
2006-04 Marta Sabou (VU) Building Web Service Ontologies
2006-05 Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines
2006-06 Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling
2006-07 Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering
2006-08 Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web
2006-09 Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion
2006-10 Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems
2006-11 Joeri van Ruth (UT) Flattening Queries over Nested Data Types
2006-12 Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts
2006-13 Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents
2006-14 Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change
2006-15 Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain
2006-16 Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks
2006-17 Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device
2006-18 Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing
2006-19 Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach
2006-20 Marina Velikova (UvT) Monotone models for prediction in data mining
2006-21 Bas van Gils (RUN) Aptness on the Web
2006-22 Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation
2006-23 Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web
2006-24 Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC
2006-26 Vojkan Mihajlovic (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval
2006-27 Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories
2006-28 Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007

2007-01 Kees Leune (UvT) Access Control and Service-Oriented Architectures
2007-02 Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach
2007-03 Peter Mika (VU) Social Networks and the Semantic Web
2007-04 Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach
2007-05 Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance
2007-06 Gilad Mishne (UVA) Applied Text Analytics for Blogs
2007-07 Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings
2007-08 Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations
2007-09 David Mobach (VU) Agent-Based Mediated Service Negotiation
2007-10 Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols
2007-11 Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System
2007-12 Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty
2007-13 Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology
2007-14 Niek Bergboer (UM) Context-Based Image Analysis
2007-15 Joyca Lacroix (UM) NIM: a Situated Computational Memory Model
2007-16 Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems
2007-17 Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice
2007-18 Bart Orriens (UvT) On the development an management of adaptive business collaborations
2007-19 David Levy (UM) Intimate relationships with artificial partners
2007-20 Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network
2007-21 Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005
2007-22 Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns
2007-23 Peter Barna (TUE) Specification of Application Logic in Web Information Systems
2007-24 Georgina Ramírez Camps (CWI) Structural Features in XML Retrieval
2007-25 Joost Schalken (VU) Empirical Investigations in Software Process Improvement

2008

2008-01 Katalin Boer-Sorbán (EUR) Agent-Based Simulation of Financial Markets: A modular, continuous-time approach
2008-02 Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations
2008-03 Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach
2008-04 Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration
2008-05 Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective
2008-06 Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
2008-07 Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning
2008-08 Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference
2008-09 Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective
2008-10 Wauter Bosma (UT) Discourse oriented summarization
2008-11 Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach
2008-12 Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation
2008-13 Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks
2008-14 Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort
2008-15 Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.
2008-16 Henriette van Vugt (VU) Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises
2008-18 Guido de Croon (UM) Adaptive Active Vision
2008-19 Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
2008-20 Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.
2008-21 Krisztian Balog (UVA) People Search in the Enterprise
2008-22 Henk Koning (UU) Communication of IT-Architecture
2008-23 Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia
2008-24 Zharko Aleksovski (VU) Using background knowledge in ontology matching
2008-25 Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
2008-26 Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled
2008-27 Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design
2008-28 Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks
2008-29 Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
2008-30 Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
2008-31 Loes Braun (UM) Pro-Active Medical Information Retrieval
2008-32 Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
2008-33 Frank Terpstra (UVA) Scientific Workflow Design; theoretical and practical issues
2008-34 Jeroen de Knijf (UU) Studies in Frequent Tree Mining
2008-35 Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009

2009-01 Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models
2009-02 Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques
2009-03 Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT
2009-04 Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering
2009-05 Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
2009-06 Muhammad Subianto (UU) Understanding Classification
2009-07 Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion
2009-08 Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments
2009-09 Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems
2009-10 Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications
2009-11 Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web
2009-12 Peter Massuthe (TUE, Humboldt-Universität zu Berlin) Operating Guidelines for Services
2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems
2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16 Fritz Reul (UvT) New Architectures in Computer Chess
2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data
2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System
2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making
2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification
2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence
2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment
2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations
2009-25 Alex van Ballegooij (CWI) RAM: Array Database Management through Relational Mapping
2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services
2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web
2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models
2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications
2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage
2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors
2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?
2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach
2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling
2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks
2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks
2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context
2009-39 Christian Stahl (TUE, Humboldt-Universität zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets
2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language
2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings
2009-42 Toine Bogers (UvT) Recommender Systems for Social Bookmarking
2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients
2009-44 Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations
2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful
2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter
2010-02 Ingo Wassink (UT) Work flows in Life Science
2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents
2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments
2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems
2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI
2010-07 Wim Fikkert (UT) Gesture interaction at a Distance
2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments
2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging
2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children
2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning
2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis
2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques
2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration
2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models
2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice
2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications
2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation
2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems
2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative
2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation
2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data
2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions
2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies
2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective
2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
2010-27 Marten Voulon (UL) Automatisch contracteren
2010-28 Arne Koopman (UU) Characteristic Relational Patterns
2010-29 Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels
2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval
2010-31 Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web
2010-32 Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-33 Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval

2010-34 Teduh Dirgahayu (UT) Interaction Design in Service Compositions
2010-35 Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval
2010-36 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification
2010-37 Niels Lohmann (TUE) Correctness of services and their composition
2010-38 Dirk Fahland (TUE) From Scenarios to Components