Data and Abstraction for Scenario-Based Modeling with Petri Nets

Dirk Fahland and Robert Prüfer

Eindhoven University of Technology, The Netherlands
Humboldt-Universität zu Berlin, Germany
d.fahland@tue.nl, pruefer@informatik.hu-berlin.de

Abstract. Scenario-based modeling is an approach for describing behaviors of a distributed system in terms of partial runs, called *scenarios*. Deriving an operational system from a set of scenarios is the main challenge that is typically addressed by either *synthesizing* system components or by providing *operational semantics*. Over the last years, several established scenario-based techniques have been adopted to Petri nets. Their adaptation allows for verifying scenario-based models and for synthesizing individual components from scenarios within one formal technique, by building on Petri net theory. However, current adaptations of scenarios face two limitations: a system modeler (1) cannot abstract from concrete behavior, and (2) cannot explicitly describe data in scenarios. This paper lifts these limitations for scenarios in the style of Live Sequence Charts (LSCs). We extend an existing model for scenarios, that features Petri net-based semantics, verification and synthesis techniques, and close the gap between LSCs and Petri nets further.

Keywords: scenario-based modeling, data, abstraction, Petri nets

1 Introduction

Designing and implementing a distributed system of multiple components is a complex task. Its complexity originates in the component interactions. Established *scenario-based methods* such as *(Hierarchical) Message Sequence Charts* ((H)MSCs) [26] and *Live Sequence Charts* (LSCs) [6] alleviate this complexity: A system designer *specifies* the system's behaviors as a *set of scenarios*. Each scenario is a self-contained, partial execution usually given in a graphical notation. Then system components are *synthesized* (preferably automatically) that together interact as described in the scenarios. Alternatively, a specification *becomes* a system model by equipping scenarios with *operational semantics*.

A significant drawback of established techniques is that components have to be synthesized in a different formal theory than the one in which the scenarios are given, e.g., HMSCs or LSCs are synthesized into Petri nets or statecharts [7, 19]. Also operational semantics for MSCs and LSCs require a translation into another formalism like automata [31], process algebras [30], or require involved formal techniques such as graph grammars [24] or model-checking [20]. Many HMSC and LSC specifications cannot be distributed into components but require centralized control [7, 4]. This renders

turning scenarios into systems surprisingly technical while scenarios appear to be very intuitive.

Approaches which express scenarios, system behaviors and system model in the same formal theory face less problems. In particular, approaches which describe scenarios in terms of Petri nets and their partially ordered runs, e.g., [9], have been successful. The approach in [2] presents a general solution for synthesizing a Petri net from HMSC-style specifications in a Petri net-based model. [8] shows how to compose complex system behaviors from single Petri net events with preconditions. The model of *oclets* [11, 12] adapts ideas from LSCs to Petri nets: a scenario is a partial run with a distinguished precondition; system behavior emerges from composing scenarios based on their preconditions. This idea allows oclets to adapt existing Petri net techniques for a general solution to the synthesis problem for LSC-style scenarios [12].

The Petri net-based scenario techniques [9, 2, 8, 11, 12] in their current form only describe control-flow and provide no means for abstracting behavior in a complex specification. Any practically applicable specification technique needs some notion of abstraction as well as some explicit notion of data, and means to describe several components of the same kind.

This paper addresses these problems of practical applicability of Petri-net based scenarios. We show for the model of oclets how to extend scenarios by abstract causal dependencies (abstracting from a number of possibly unknown actions between two dependent actions), and how to express data in scenarios by adapting notions of Algebraic Petri nets [32]. Our contribution is two-fold: First, abstraction and data are two key features of LSCs, so our extension of oclets closes the gap between LSCs and Petri nets further. Second, all our extensions are simple generalizations of existing concepts from Petri net theory, giving rise to the hope that existing verification and synthesis results, e.g., [12], can be transferred to the more expressive model proposed in this paper.

We proceed as follows. Section 2 recalls the scenario-based approach in more detail and explains the basic ideas of oclets by an example. In Sections 3 and 4, abstract causal dependencies and data are introduced into the model, respectively. Section 5 discusses the relation of oclets to Petri nets. We conclude and discuss related and future work in Section 6.

2 Specifying with Scenarios

This section recalls the scenario-based approach by the help of an example and discusses features and limitations of scenario-based specification techniques. Two of these limitations will be addressed in the remainder of this paper.

2.1 Running example and requirements for capturing it

Our running example is a gas station (adapted from [23]) that allows customers to refuel their cars using one of the available pumps as follows. When a customer arrives with his car at a pump, he asks the operator to activate that pump for a certain amount of fuel for which he pays in advance or after he finished pumping the gas. The customer can start an activated pump to refuel his car, pumping gas one unit at a time. The pump stops when

all requested gas has been pumped, or when stopped by the customer. The pump then signals the operator the pumped amount and the operator returns corresponding change to the customer. Each customer gets a free snack that he may pick up after starting the pump and before leaving the gas station.

A specification technique capable to express this gas station has to describe (R1) *distributed components* (e.g., pump, customer, operator), (R2) *interaction* between components, (R3) *sequential*, *independent*, and *alternative* ordering of actions, (R4) *pre-conditions* of actions (e.g., "when all requested gas has been pumped"), (R5) actions that depend on *data* (e.g., returned change, amount of pumped gas), (R6) *multiple instances* of the same kind of component (e.g., multiple customers and pumps). Furthermore, a specification technique also should allow a system designer to keep an overview of larger specifications by (R7) *means of abstraction*. Finally, the specification technique should allow to (R8) *derive the specified behavior in an intuitive* way, that is, the derived behavior should be "correct by construction" and not require additional verification.

2.2 Principles of scenario-based specifications

In the scenario-based approach, a system designer obtains a system model of a distributed system (e.g., our gas station example) in two steps. First she describes the system behavior as *component interactions*. One *scenario* describes how several components interact with each other in a particular situation; a *specification* is a set of scenarios. When she completed the specifications, components are *synthesized* (preferably automatically) such that all components together interact as described in the scenarios.

The most valued feature of this approach is that scenarios *tangibly* decompose complex system behavior into smaller, self-contained stories of component interactions (scenarios) which are easy to understand. Fig. 1 shows 3 scenarios (M0, M1, M2) of the running example in the well-established syntax of MSCs. In each MSC, a vertical *lifeline* describes one *component*, arrows between components describe *interactions*, boxes at components describe *local actions*; M0 is a special case as it describes the creation of a new instance of a customer. Established scenario-based techniques adhere to a few simple principles that allow to *derive system behavior from scenarios* in a comprehensible way as follows.

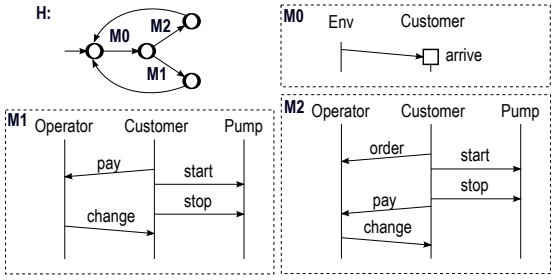

Fig. 1. An HMSC H describing compositions of MSCs M0, M1, M2.

S1 A scenario is *partial order* of actions, understood as a *partial run* of the system. A *specification* is a set of scenarios.
S2 System behavior follows from *composing* (appending) scenarios.
S3 Each scenario distinguishes a prefix as a *precondition* describing *when* the scenario can occur; the remainder of the scenario is called *contribution*.

S4 When a system run ends with a scenario's precondition, the run can continue by *appending* the scenario's contribution. Scenarios with the same precondition and different contributions lead to *alternative* runs.

S1 is the most generally agreed upon principle for scenarios and usually expressed in an MSC-like notation as in Fig. 1; other notations are possible [6, 9, 11, 2]. To specify practically relevant systems, more principles are needed. The probably most established scenario-based techniques – (H)MSCs, LSCs and UML Sequence Diagrams – realize these principles differently as we discuss next.

HMSCs proposed principle S2 first, where the order of scenario composition is described by a finite automaton [26]. For instance, HMSC H of Fig. 1 describes that M0 is followed by M1 *or alternatively* by M2, and then M0 can occur again. This way, HMSCs are capable to express the requirements R1-R3 of Sect. 2.1, but not R4. In the HMSC standard, notions of data (R5) are only provided on a syntactical, but not on a semantical level [26]. Also multiple instances of the same component cannot be expressed: the HMSC of Fig. 1 allows only one customer to be served at a time. Means of abstraction (R7) are provided by the possibility of nesting one MSC inside another MSC. UML Sequence Diagrams express scenario composition entirely by nesting scenarios in each other.

It has been repeatedly observed that this approach to scenarios requires a *global* understanding of the entire system as the ordering of scenarios is described in a global automaton [17, 12]. Moreover, as MSCs of a HMSC cannot overlap, a specification may have to be refactored when a new scenario shall be included [34] and specifications tend to consist of many small-scale scenarios composed in complex ways, which is counter-intuitive to the idea of one scenario describing a "self-contained story" of the system [38, 12].

LSCs extend MSCs in a different way to provide enough expressive power [6]. Altogether, LSCs provide notions for preconditions of scenarios, data, multiple instances and abstraction on component lifelines, satisfying R1-R7 of Sect. 2.1 [21]. LSCs first proposed principle S3 that a scenario is triggered by a precondition; this idea has been adopted in other approach as well [17, 35, 11]. Fig. 2 shows 3 LSCs corresponding to the MSCs of Fig. 1. However, LSCs specify system behavior not by scenario composition, but each LSC denotes *a linear-time temporal logic formula*: when a system run ends with an LSC's precondition, then the run *must* continue with the LSC's contribution, i.e., the given events occur in the run eventually in the given order.

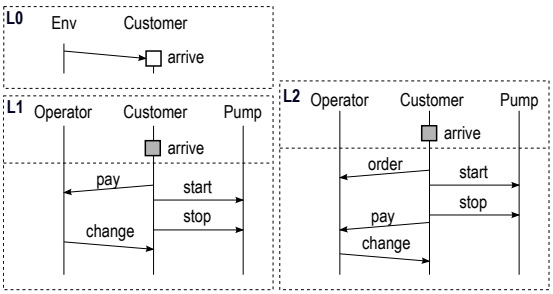

Fig. 2. Three LSCs of the gas station example.

For instance, L1 expresses that *after* arrive occurred (precondition), the customer pays, starts and stops the pump, and gets his change (contribution). Additionally, LSCs

specify particular events as *implicitly forbidden* at particular stages of an LSC. For instance, in L2, event pay is *forbidden to occur before* event stop. At the same time, L1 requires pay to occur right after arrived (before stop) and *forbids occurrences of* pay *after* stop. In contrast to intuition, L1 and L2 are *contradictory* and no system satisfies both LSCs. This contradiction arises because of the linear-time semantics of LSCs as both L1 and L2 have to occur in the same run. The contradiction vanishes when using a *branching-time semantics* for LSCs as proposed in the model of *epLSCs* [37]: when the pre-condition occurs, *some run* continues with the contribution (principle S4). However, also in this model, deriving system behavior from a specification is cumbersome: whether two epLSCs have to occur in different runs or may occur overlappingly in the same run still requires to *check* their temporal logic formulae, possibly requiring verification on large parts of the specified state-space [20].

Between LSCs and HMSCs. To summarize, HMSCs have a simple semantics that allows to derive specified behaviors by composing scenarios. Though, HMSCs suffer from the global automaton and that scenarios cannot overlap. LSCs allow for local preconditions and overlapping scenarios, but are based on an intricate semantics that makes it hard to understand behavior specified by a set of LSCs. That simplicity of semantics influences the way how components can be synthesized from scenarios can be seen when comparing available techniques. While synthesis is generally infeasible from both HMSCs and LSCs, synthesis from feasible subclasses of scenarios to Petri nets is straight forward for HMSCs [31, 2] yielding components that are *correct by construction*, whereas synthesis from LSCs [1, 28] requires to *verify the synthesis result* to ensure correctness.

In the following, we derive a scenario-based technique that inherits the advantages of HMSCs and LSCs without their disadvantages: a LSC-style syntax of scenarios with local precondition is given a HMSC-style semantics where system behavior follows from scenario composition. The hope is that a simpler semantic model allows to synthesize from LSC-style scenarios components that are correct by construction. Indeed, this already has been proven to be successful for a simple model of scenarios that we present next.

2.3 A simple model for scenarios based on Petri nets

We derive a simple semantic model for LSC-style scenarios by applying principles of Petri net theory. Petri net-based scenarios benefit from expressing scenarios, behavior, and system in the same formal model, which allows to create a Petri-net based operational semantics for scenarios and supports the crucial step from specification to system model. For HMSC-style specifications, corresponding semantics and synthesis techniques are already available [2]

For LSC-style scenarios, a simple model that adopts the semantics of epLSCs [37] to Petri nets has been proposed in the model of *oclets* [11, 12] that we recall next. Oclets realize all principles S1-S4 in the following way. Fig. 3 shows four scenarios of the gas station example of Sect. 2.1 in the notation of oclets. The partial order of actions is expressed as a so called *labeled causal net*. A transition (place) of a causal net is called event (condition); the flow relation defines a partial order over events and conditions

s.t. the net is conflict-free. The grey-filled (white-filled) nodes indicate the precondition (contribution) of an oclet.

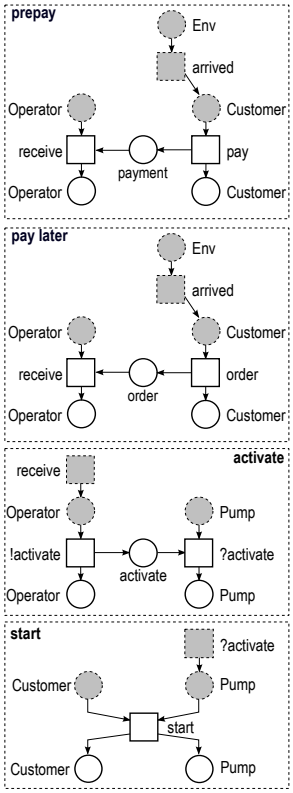

Fig. 3. Scenarios for the gas station example.

The four oclets describe some behavior of the gas station example. Oclet prepay: after a customer arrived at the gas station, he asks to activate a pump and prepays his gas (event pay). Alternatively (oclet pay later), the customer may just ask the operator to activate the pump (order). Oclet activate: after receiving the order, the operator activates the pump. Oclet start: When a pump is activated, the customer may start it.

Oclets generalize the semantics of Petri net transitions to scenarios. Whenever a run ends with an oclet's precondition, the oclet is *enabled* and the run can continue by appending the oclet's contribution. Two oclets with the same precondition and different contributions are alternatives and hence yield alternative continuations.

For example, consider the run π_0 indicated in Fig. 4. In π_0, oclets pay and pay later are enabled. Continuing π_0 with pay yields the run π_1 indicated in Fig. 4(left), that was obtained by appending pay's contribution to π_0. Appending use card yields the run π_1' of Fig. 4(right) that is alternative to π_1. In π_1, oclet activate is enabled; appending its contribution yields π_2. This way, oclets *derive* the specified behavior of the gas station by composition.

Properties and limitations. In their current form, oclets allow to express properties (R1-R4) of Sect. 2.1 and allow to analyze scenarios, and synthesize a system by reusing and extending Petri net techniques [12]. Yet, oclets cannot express data (e.g., how much gas to pump) or distinguish instances (e.g., two different pumps). Furthermore, the events in an oclet currently describe a "contiguous piece of behavior." In the worst case, the specification consists of many short scenarios, only. *Abstraction* would allow a system designer to also specify longer scenarios of corresponding, non-contiguous pieces of behavior. In the remainder of this paper, we show how to introduce abstraction and data to oclets, thus providing a scenario-based technique between HMSCs and LSCs. We stay close to the spirit of Petri nets and define an extension in terms of simpler, existing principles.

3 Adding Abstraction: Abstract Dependencies

In this section, we introduce means to abstract from behavior in a scenario. We sketch the idea by our running example before we present formal definitions.

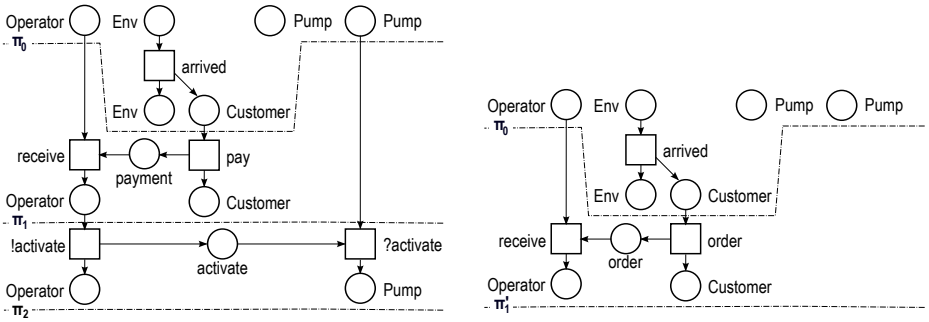

Fig. 4. Two alternative runs of the gas station built by composing scenarios of Fig. 3.

3.1 Abstracting causal dependencies

As stated in Sect. 2.3, the flow relation of an oclet as described in [11, 12] denotes direct causal dependencies which may restrict how a particular system behavior can be specified. *Abstract causal dependencies* allow other events to occur between two events of a scenario.

Fig. 5 shows examples of abstract dependencies; an abstract dependency is drawn as a dashed arrow. The oclet main describes the main interaction between customer and the gas station's operator and the pump (see Sect. 2.1). The oclet abstracts from other behavior taking place at the gas station, some of that behavior is needed to make the customer's interaction happen. For instance, oclet main only abstractly describes the dependency of the two Pump conditions. This allows events which are not depicted to occur between these conditions. In particular event start of oclet start of Fig. 3 can occur here. In other words, oclet start *refines* this abstract dependency of oclet main. There are further abstract dependencies in main that have to be detailed by other oclets. Yet, the main scenario clearly describes that once the pump has been activated, it eventually completes pumping, which will lead to the operator returning change to the customer. We complete the specification in Sect. 4.4.

Abstract dependencies are also useful in a scenario's precondition. Here, they allow to specify that an oclet is enabled if a specific behavior occurred "some time" in the past, instead of immediately. For instance, oclet main[completed] in Fig. 5 expresses that activate must have occurred some time in the past in order to enable event completed, including the possibility that other events occurred in between.

Introducing abstract dependencies in oclets comes at a price: we cannot continue a run with an enabled oclet by appending its contribution. The principle solution is to decompose an oclet into *basic* oclets such as main[completed] in Fig. 5(top right). Each basic oclet contributes exactly one event, its precondition consists of all transitive predecessors in the original oclet. This effectively moves abstract dependencies into preconditions and system behavior emerges from concatenating well-defined single events.

In the remainder of this section, we formalize these ideas by extending syntax and semantics of oclets [11, 12] with abstract dependencies. We assume the reader to be familiar with the basic concepts of Petri Nets and their distributed runs; see [33] for an introduction.

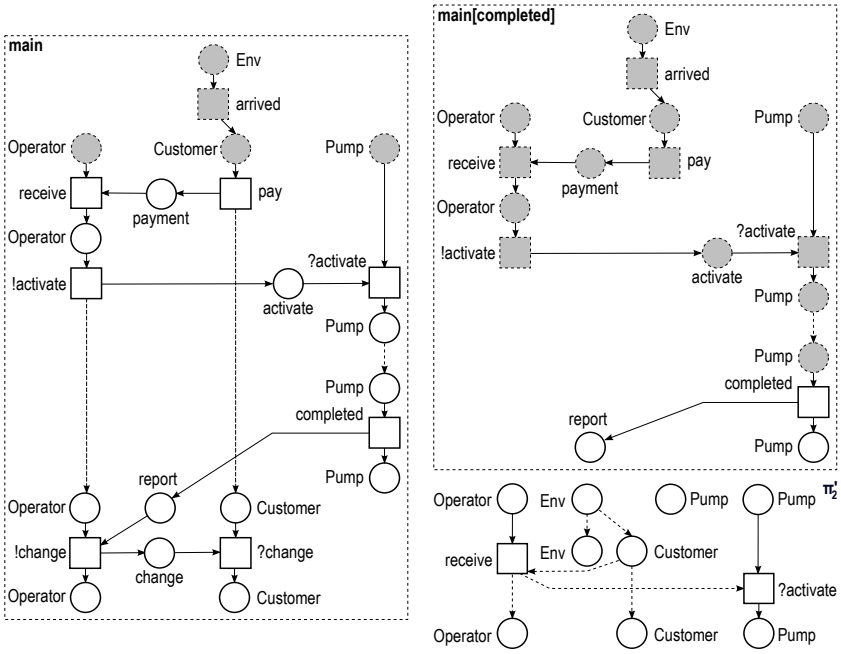

Fig. 5. Main scenario of the gas station example (left), abstract dependencies allow to abstract several details of the system behavior; a basic oclet (top right); an abstract run (bottom right).

3.2 Basic Notions

First, we recall some basic notation. A *partial order* over a set A is a binary relation $\leq \subseteq A \times A$ that is reflexive (i.e. $\forall a \in A : a \leq a$), transitive (i.e. $\forall a, a', a'' \in A : a \leq a' \wedge a' \leq a'' \Rightarrow a \leq a''$), and antisymmetric (i.e. $\forall a, a' \in A : a \leq a' \wedge a' \leq a \Rightarrow a = a'$). Let $a{\downarrow_{\leq}} := \{a' \in A \mid a' \leq a\}$ and $a{\uparrow_{\leq}} := \{a' \in A \mid a \leq a'\}$ denote the *transitive predecessors* and *successors* of $a \in A$, respectively. As usual, for a relation $R \subseteq (A \times A)$, R^+ and R^* denote the transitive, and reflexive-transitive closures of R.

We write a *Petri Net* as $N = (P, T, F)$, with places P, transitions T ($P \cap T = \emptyset$), and arcs $F \subseteq (P \times T) \cup (T \times P)$. Notation-wise, introducing N, N_1, N' implicitly introduces their components P_N, P_1, P' etc. For each *node* $x \in X_N := P \cup T$, ${}^\bullet x = \{y \in X \mid (y, x) \in F\}$ and $x^\bullet = \{y \in X \mid (x, y) \in F\}$ are the *pre-* and *post-set* of x, respectively. A *causal net* $\pi = (B, E, F)$ is a Petri net where (1) $\leq_\pi := F^*$ is a partial order over X_π, (2) for each $x \in X_\pi$, $x{\downarrow_{\leq_\pi}}$ is finite, and (3) for each $b \in B$, $|{}^\bullet b| \leq 1$ and $|b^\bullet| \leq 1$. An element of B (E) is called *condition* (*event*). The arcs of a causal net denote *direct causal dependencies*: x depends on y iff $x \leq y$, and x and y are *concurrent* iff neither $x \leq y$ nor $y \leq x$. We write $\min \pi = \{x \mid {}^\bullet x = \emptyset\}$ and $\max \pi = \{x \mid x^\bullet = \emptyset\}$ for the nodes without predecessor and successor, respectively.

In the following, we consider *labeled* causal nets $\pi = (B, E, F, \ell)$ where each node $x \in X_\pi$ is assigned a label $\ell(x) \in L$ from some given set L. We will interpret a labeled causal net as a *partially ordered run* (of a possibly unknown system); in analogy to Petri nets, an event e describes an occurrence of an action (or transition) $\ell(e)$, and a condition b describes an occurrence of a local state (a token on a place) $\ell(b)$.

3.3 Runs with abstract dependencies

We formally introduce *abstract dependencies* by generalizing the notion of a partially ordered run to an *abstract* run. This definition then canonically lifts to oclets with abstract dependencies.

Definition 1 (Partially ordered run). *An* abstract partially ordered run *(*run *for short)* $\pi = (B, E, F, A, \ell)$ *is a labeled causal net* (B, E, F, ℓ) *with* abstract dependencies $A \subseteq X_\pi \times X_\pi$ *s.t.* $\leq_\pi := (F \cup A)^*$ *is a partial order over the nodes* X_π. π *is* concrete *iff* $A = \emptyset$.

We write π^α for any run that is isomorphic to π by the isomorphism $\alpha : \pi \to \pi^\alpha$. Run π *occurs* in run ρ, written $\pi \subseteq \rho$, iff $B_\pi \subseteq B_\rho, E_\pi \subseteq E_\rho, F_\pi \subseteq F_\rho, A_\pi \subseteq A_\rho, \ell_\pi = \ell_\rho|_{X_\pi}$. We will work with two important relations on runs: prefixes and refinement.

Definition 2 (Prefix). *A run* π *is a* prefix *of a run* ρ*, written* $\pi \sqsubseteq \rho$*, iff (1)* $\min \rho \subseteq X_\pi \subseteq X_\rho$ *(2)* $F_\pi = F_\rho|_{X_\rho \times X_\pi}$, $A_\pi = A_\rho|_{X_\rho \times X_\pi}$ *(*π *contains* all *predecessors), (3) for each* $e \in E_\rho$ *and all* $(e, b) \in F_\rho$ *holds* $(e, b) \in F_\pi$ *(events have all post-conditions). The set of all prefixes of* ρ *is* $Pre(\rho) := \{\pi \mid \pi \sqsubseteq \rho\}$.

Fig. 5(bottom right) shows an abstract run; Fig. 4 shows concrete runs; π_1 is a prefix of π_2; π_1' is not a prefix of π_1. Each abstract run describes a set of concrete runs (without abstract dependencies) that *refine* the abstract run. Intuitively, an abstract dependency in a run π can be refined by a number of nodes that respect the partial order of π. The refinement can *exclude* nodes with a particular label, which we need for oclet semantics.

Definition 3 (Refine an abstract run). *Let* π *and* ρ *be abstract distributed runs. Let* $\kappa : A_\pi \to 2^L$ *assign each abstract dependency in* π *a (possibly empty) set of forbidden labels.* ρ refines π *w.r.t.* κ*, written* $\rho \preceq_\kappa \pi$ *iff*

- $B_\pi \subseteq B_\rho, E_\pi \subseteq E_\rho, \forall x \in X_\pi : \ell_\pi(x) = \ell_\rho(x)$,
- $F_\pi \subseteq F_\rho$ *and* $(F_\pi \cup A_\pi)^+ \subseteq (F_\rho \cup A_\rho)^+$, *and*
- $\forall (x, y) \in A_\pi \nexists e \in E_\rho : x \leq_\rho e \leq_\rho y \wedge \ell_\rho(e) \in \kappa(x, y)$.

We write $\pi \preceq \rho$ *if* $\kappa(x, y) = \emptyset$ *for all* $(x, y) \in A_\pi$. *Every run* π *describes the set* $[\![\pi]\!] := \{\rho \mid \rho \preceq \pi, A_\rho = \emptyset\}$ *of all concrete runs that refine* ρ.

Run π_2 of Fig. 4 refines the abstract run π_2' of Fig. 5.

3.4 Scenarios with abstract dependencies

Syntax. Abstract dependencies canonically lift from abstract runs to oclets. As already sketched in Sect. 3.1, an oclet is an abstract run with a distinguished prefix.

Definition 4 (Oclet). *An oclet* $o = (\pi, pre)$ *consists of an abstract distributed run* π *and a prefix* $pre \sqsubseteq \pi$ *of* π.

Fig. 5 shows oclet main. Def. 4 generalizes "classical" oclets as introduced in [11, 12] by abstract dependencies of the underlying run. We call *pre* the *precondition* of *o* and the suffix $con(o) := (B_\pi \setminus B_{pre}, E_\pi \setminus E_{pre}, F_\pi \setminus F_{pre}, A_\pi \setminus A_{pre}, \ell_\pi|_{X_{con(o)}})$ its *contribution*; technically $con(o)$ is not a net as it contains arcs adjacent to nodes of the precondition.

A specification is a set of oclets together with an *initial run* that describes how system behavior starts.

Definition 5 (Specification). *A* specification $\Omega = (O, \pi_0)$ *is a set O of oclets together with an abstract run π_0 called* initial *run.*

A specification usually consists of a *finite* set of oclets; we allow the infinite case for technical reasons. This is all syntax that we need.

Semantics. The semantics of oclets is straight forward. An oclet o describes a scenario with a necessary precondition: the contribution of o can occur *whenever* its precondition occurred. Then, we call *o enabled*.

Definition 6 (Enabled Oclet). *Let $o = (\pi, pre)$ be an oclet and let π be a run. Each $(x, y) \in A_\pi$ defines the post-events of y as* forbidden*, i.e., $\kappa(x, y) = \{\ell(e) \mid (y, e) \in F_o \cup A_o, e \in E_o\}$. Oclet o is* enabled in π *iff there exists a refinement $pre' \preceq_\kappa pre$ s.t. (1) $pre' \subseteq \pi$, (2) $\max pre' \subseteq \max \pi$, and (3) $X_{con(o)} \cap X_\pi = \emptyset$.*

An oclet is enabled in a run π if the complete precondition occurs at end of π, i.e., "just happened." The forbidden events κ restrict which events may occur in place of an abstract dependency of *pre* (see Def. 3); this ensures enabling only at a "very recent" occurrence of the precondition. The same model is applied in LSCs [6].

An oclet o can be enabled at several different locations in π (whenever we find *pre* several times at the end of π). We say that o is enabled in π *at location α* iff o^α is enabled in π. For technical reasons, o's contribution is assumed to be disjoint from π so that it can be appended to π.

Definition 7 (Continue a run with an oclet). *Let o be an oclet and let π be a distributed run. If o is enabled in π, then the* composition *of π and o is defined as $\pi \triangleright o := (\pi \cup \pi_o) = (B_\pi \cup B_o, E_\pi \cup E_o, F_\pi \cup F_o, \ell', A_\pi \cup A_o)$ with $\ell'(x) = \ell_\pi(x)$, for all $x \in X_\pi$, $\ell'(x) = \ell_o(x)$, for all $x \in X_o$.*

A specification Ω describes a set $R(\Omega)$ of abstract runs: that is, the *prefixes* of all runs that can be constructed by repeatedly appending enabled oclets of Ω to the initial run. The concrete system behaviors specified by Ω are the concrete runs that refine $R(\Omega)$.

Definition 8 (Semantics of a specification). *Let $\Omega = (O, \pi_0)$ be a specification. The abstract runs of Ω are the least set $R(\Omega)$ of runs s.t.*

1. *$Pre(\pi_0) \subseteq R(\Omega)$, and*
2. *for all $\pi \in R(\Omega), o \in O$, if o is enabled in π at α then $Pre(\pi \triangleright o^\alpha) \subseteq R(\Omega)$.*

A set R of concrete runs satisfies *Ω iff for each $\pi \in R(\Omega)$, $[\![\pi]\!] \cap R \neq \emptyset$.*

3.5 Operational semantics

A system designer can use oclets to specify the behaviors of a distributed system. Harel et al. [20] suggested to turn a specification into an executable system model by providing *operational semantics* for scenarios.

Operational semantics describe system behavior as occurrences of single events. Each event has a *local* precondition, if the precondition holds, the event can occur by being appended to the run. These principles are naturally captured by *basic* oclets that

contribute just a single event; we define operational semantics of oclets by *decomposing* complex oclets into basic oclets.

We call an event e of a run π *concrete* iff e has no abstract dependencies, i.e., $\forall (x, y) \in A_\pi : x \neq e \neq y$. Oclet o is *basic* iff its contribution consists of exactly one concrete event e (with post-conditions). If o is not basic, then it can be decomposed into basic oclets. Each concrete event e of o's contribution induces the basic oclet $o[e]$ that contributes e and e's post-set and has as precondition all transitive predecessors $e \downarrow_{\leq_o}$ of e in o.

Definition 9 (Decomposition into basic oclets). *Let $o = (\pi, pre)$ be a basic oclet. A concrete event $e \in E_{con(o)}$ induces the basic oclet $o[e] = (\pi', pre')$ with $X' = e \downarrow_{\leq_o} \cup e^\bullet$, $F' = F|_{X' \times X'}$, $A' = A|_{X' \times X'}$, $\ell' = \ell|_{X'}$, and $pre' \sqsubseteq \pi'$ s.t. $X_{pre'} = e \downarrow_{\leq_o} \setminus \{e\}$. The basic oclets of o are $\hat{o} = \{o[e] \mid e \in E_{con(o)}, e \text{ is concrete}\}$.*

There may be specifications where a particular action a has no corresponding concrete event $e, \ell(e) = a$, i.e., it is always adjacent to some abstract dependency. In this case, the specification provides no information on how to refine these abstract dependencies. We found it useful for concise specifications, that in this case, a non-concrete event e of an oclet $o = (\pi, pre)$ also induces the basic oclet $o[e]$ where the abstract dependencies between e and some condition b are turned into direct dependencies, i.e., replace in o each abstract dependency $(b, e) \in A_o, b \in B_o$ by an arc $(b, e) \in F_o$ and each $(e, b) \in A_o, b \in B_o$ by an arc $(e, b) \in F_o$, and then compute $o[e]$ as in Def. 9.

In both cases of Def. 9 and with additional basic oclets, the operational semantics of an oclet specification follows from its basic oclets.

Definition 10 (Operational semantics). *Let $\Omega = (O, \pi_0)$ be a specification. $\hat{\Omega} = (\bigcup_{o \in O} \hat{o}, \pi_0)$ is the* basic *specification induced by Ω. It defines the* operational semantics *of Ω as $R(\hat{\Omega})$. Ω is* operational *iff $R(\hat{\Omega})$ satisfies Ω.*

We call $R(\hat{\Omega})$ the operational semantics of Ω because $\hat{\Omega}$ describes a set of single events. Each event is enabled when its local precondition holds; an enabled event can occur (by appending it to the run). Not every specification has operational semantics that satisfy the specification. A non-operational specification needs to be refined to become operational. Yet, we can characterize operational specifications.

Theorem 1. *Let $\Omega = (O, \pi_0)$ be a specification s.t. π_0 is concrete and each event in the contribution of each oclet in O is concrete. Then Ω is operational.*

Proof. This theorem has been proven for oclets without abstract dependencies in [11]: by induction on the prefixes of an oclet's contribution, an oclet's contribution can be reconstructed from its basic oclets. In particular, whenever oclet o is enabled in π, also each basic oclet of o is enabled in π or in a continuation $\pi \rhd o[e_1] \rhd \ldots \rhd o[e_k]$. This reasoning applies also when o has abstract dependencies in its precondition (still, each basic oclet $o[e]$ gets enabled whenever o is enabled).

Theorem 1 states a rather strict sufficient condition for operational specifications (abstract dependencies only in preconditions). Next, we present a more general sufficient condition: Ω is operational if all abstract dependencies of Ω can be refined by its basic oclets.

Refining abstract runs lifts to oclets: oclet $o_2 = (\pi_2, pre_2)$ refines oclet $o_1 = (\pi_1, pre_1)$, written $o_2 \preceq o_1$, iff $\pi_2 \preceq \pi_1$ and $pre_2 = pre_1$, i.e., we may only refine contributions. An oclet's refinement can be justified by another oclet.

Definition 11 (Justified by oclet). *Let o_1, o_2 be oclets s.t. o_2 refines o_1. The refinement from o_1 to o_2 is* justified *by an oclet o iff $X_{o_2} \setminus X_{o_1} \subseteq X_{con(o)}, F_{o_2} \setminus F_{o_1} \subseteq F_{con(o)}, A_{o_2} \setminus A_{o_1} \subseteq A_{con(o)}$.*

Let $\Omega = (O, \pi_0)$ be an oclet specification. The refinement from o_1 to o_2 is justified by Ω *iff there is a sequence of refinements from o_1 to o_2 s.t. each refinement is justified by a basic oclet $o \in \hat{O}$ (or by an isomorphic copy o^α of o).*

Theorem 2. *Let $\Omega = (O, \pi_0)$ be a specification. If each oclet $o_1 \in O$ can be refined to an oclet o_2 justified by Ω s.t. all events of o_2's contribution are concrete, then Ω is operational.*

Lemma 1. *Let $\Omega = (O, \pi_0)$ be a specification. Let o be an oclet that can be refined into an oclet o', justified by Ω, s.t. each $e \in E_{con(o')}$ is concrete. Let π be a run s.t. o is enabled in π. Then there exists a sequence $o_1, \ldots, o_n \in \hat{O} \cup \hat{o}$ of basic oclets s.t. $(\pi \rhd o_1 \rhd \ldots \rhd o_n) = (\pi \rhd o') \preceq (\pi \rhd o)$.*

Proof (Lem. 1). Proof by the number $n = |E_{con(o')}|$. For $n = 0$, the proposition holds trivially. For $n > 0$, let $e \in E_{con(o')}$ be a maximal (no other event of o succeeds e).

Case 1: $e \in E_{con(o')}$ and e is concrete. Obtain o_{-e}, o'_{-e} by removing e and $e^\bullet$ from o, o'. $o'_{-e} \preceq o_{-e}$ justified by Ω and $\rho := \pi \rhd o_1 \rhd \ldots \rhd o_{n-1} = \pi \rhd o'_{-e} \preceq \pi \rhd o_{-e}$ by inductive assumption. From e being complete follows $o[e] \in \hat{o}$ and $o[e]$ enabled in ρ. Thus $(\rho \rhd o[e]) = (\pi \rhd o'_{-e} \rhd o[e]) = (\pi \rhd o') \preceq (\pi \rhd o)$.

Case 2: $e \in E_{con(o')}$ and e is not complete, or for some $b \in e^\bullet$, $b \in \max \pi_o$ (s.t. e is added by refining $(x, b) \in A_o$, see Def. 3). By Def. 11, a basic oclet $\tilde{o}$ of Ω with $\widetilde{pre} \subseteq \pi'$ justifies $e \in E_{o'} \cap E_{\tilde{o}}$ and all $(x, e), (e, y) \in F_{o'} \setminus F_o \subseteq F_{con(\tilde{o})}$. Thus, there ex. oclet o'' s.t. $o' \preceq o'' \preceq o$ where $o' \preceq o''$ is justified by $\tilde{o}$ (and the rest by Ω). All events of o'' (except e) are concrete. Obtain $o_{-e}, o'_{-e}, o''_{-e}$ by removing e and $e^\bullet$ from o, o', o''. By construction holds $o''_{-e} = o'_{-e}$ and $\max \widetilde{pre} \subseteq \max \pi'_{-e}$. Refinement $o''_{-e} \preceq o_{-e}$ is justified by Ω, thus $\rho := (\pi \rhd o_1 \rhd \ldots \rhd o_{n-1}) = (\pi \rhd o'_{-e}) \preceq (\pi \rhd o_{-e})$ holds by inductive assumption. Further, $\widetilde{pre} \subseteq \pi'_{-e} \subseteq \rho$ and $\max \widetilde{pre} \subseteq \max \pi'_{-e} \subseteq \max \rho$ holds. Thus, $\tilde{o}$ is enabled in ρ and $(\rho \rhd \tilde{o}) = (\pi \rhd o'_{-e} \rhd \tilde{o}) = (\pi \rhd o') \preceq (\pi \rhd o)$. □

Proof (Thm. 2). By induction on the semantics of $\Omega = (O, \pi_0)$. Let $\hat{\Omega} = (\hat{O}, \pi_0)$. Base: By Def. 8, $\pi_0 \in R(\Omega)$ and $\pi_0 \in R(\hat{\Omega})$. Step: Show for $\pi \rhd o \in R(\Omega)$ (o enabled in π) that there ex. $\rho \in R(\hat{\Omega})$ s.t. $\rho \preceq \pi \rhd o$. By assumption there ex. a refinement $o' \preceq o$ justified by Ω. As $\hat{o} \subseteq \hat{O}$, Lem. 1 implies that $\pi \rhd o' \in R(\hat{\Omega})$ and $(\pi \rhd o') \preceq (\pi \rhd o)$. □

4 Adding Data: Σ-Oclets

In the current model of oclets, conditions and events can be labeled with specific data values. But the language of oclets does not allow to *concisely* describe manipulation of data values and data-dependent enabling of events. In this section, we extend oclets with notions of data. As we adopt techniques that are well-established in several Petri net formalisms, we just describe our approach in an informal manner; for technical details, see [13]. Similar to Sect. 3, we start with a general description of how to specify data.

4.1 Specifying Data

We propose to incorporate data into oclets in the same way as data has been introduced in Place/Transition nets (*P/T nets*) by several classes of *high level* Petri Nets, e.g., in Coloured Petri nets (CPNs) [27]. Recall that in CPNs, a marking distributes concrete values (from one or several domains) on places; expressions on arcs describe which values are consumed or produced by an occurrence of a transition; a *guard* expression at the transition may restrict consumable and producible values further. The method-wise relevant property of CPNs is that each colored net can be *unfolded* w.r.t. all possible interpretations of its expressions into an equivalent P/T net. Then, a (black) token on a P/T net place (p, v) denotes value v on colored place p; likewise each colored transition unfolds to several P/T net transitions defined by the consumed and produced concrete values. A special class of CPNs are Algebraic Petri nets [32] where expressions and values are defined by a *Σ-algebra* with a *signature Σ*.

We adapt the idea of Algebraic Petri nets to oclets and introduce *Σ-oclets*. In a run, a place p carrying the value v is labeled (p, v). In a Σ-oclet, each condition is labeled with a pair (p, t) where p is a name (e.g., of a component) and t is a term (over function symbols and variables of Σ) describing possible values on p. Events are labeled with names of actions as before; an event may carry an additional guard expression (defined over Σ). A system designer can use different terms in the pre- and post-sets of an event to describe how values change by an occurrence of an event.

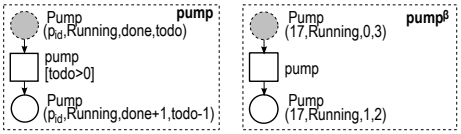

Fig. 6. Σ-oclet pump of the gas station example (left) and an unfolding pump^{β} (right) by assigning each variable a concrete value.

Fig. 6 shows Σ-oclet pump of the gas station example; the complete specification is given in Sect. 4.4. Oclet pump describes how the pump at the gas station refuels the customer's car by one unit of fuel and updates its internal records about the provided and the remaining amount of fuel. A Pump's internal record is represented as a 4-tuple. An occurrence of event pump increases done by 1 (3rd entry) and decreases todo by 1 (4th entry). The guard restricts occurrences of pump to those cases where todo > 0. Technically, $\mathsf{p_{id}}$, todo and done are variables and Running is a constant of Σ. As in Petri nets, the semantics of a Σ-oclet o can be understood by unfolding o into a "low-level" oclet that has no terms and variables. The basic idea is to assign a value to each variable in o, and then to replace each term t in o by the value obtained by evaluating t. For example, for Σ-oclet pump, the assignment

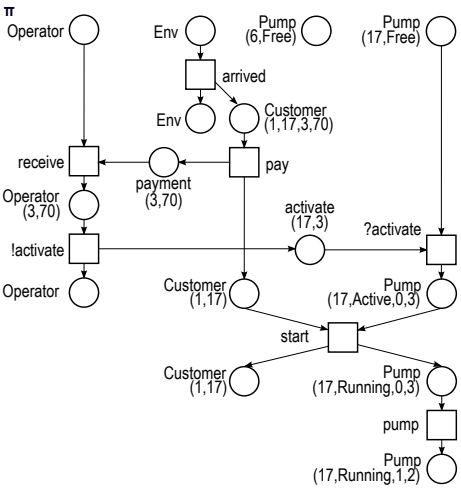

Fig. 7. A distributed run with data.

$\beta : \mathsf{p_{id}} \mapsto 17, \mathsf{done} \mapsto 0, \mathsf{todo} \mapsto 3$ yields the classical oclet pump^β shown in Fig. 6. An occurrence of pump^β can be understood as an occurrence of pump in mode β. Other assignments yield other low-level oclets. A Σ-oclet can only be unfolded if all guards evaluate to *true*.

This way an entire set of Σ-oclets O can be unfolded to a (possibly infinite) low-level oclet specification O'. O describes the distributed runs that are described by O', that is, can be constructed from the unfolded oclets. For example, pump^β is enabled in run π of Fig. 7, i.e., oclet pump is enabled for $\mathsf{p_{id}} \mapsto 17$, etc. We can continue π by appending the contribution of pump^β which updates the internal record of the pump. Now pump is enabled for the assignment $\mathsf{todo} \mapsto 2$. Thus, pump can occur two more times, i.e., until the assignment $\mathsf{todo} \mapsto 0$ violates the guard expression.

This notion of data now allows to express *data-dependent behavior*: event pump *repeats* as often as specified by todo. Further, we are now able to distinguish different *instances* of a component. Unlike in run π_2 of Fig. 4, we can now distinguish the two pumps run in π of Fig. 7. This permits to activate exactly the pump that was chosen by the customer.

4.2 Formalization

The formalization of Σ-oclets is straight-forward as it follows exactly the principles of Coloured Petri nets. First, a specification defines an algebraic signature Σ providing sorts (of values), variables, constants and function symbols. We usually assume sorts *Bool* and the usual Boolean operators be given that are interpreted in the usual way. The signature permits to build sorted terms over its symbols and variables. The model of oclets is then extended to Σ-oclets as follows:

1. Label each event and each condition of an abstract distributed run with a pair (a, t) of a name a and a term t over Σ so that the term of an event is of type *Bool* (this terms *guards* the event); such a run is called a *Σ-run*.
2. A *Σ-oclet* is a Σ-run with a distinguished prefix.
3. A *Σ-specification* is a set of Σ-oclets together with an initial run that is assumed to be variable free.

The semantics of Σ-oclets is defined by *unfolding* each Σ-oclet into a set of "low-level" oclets according to Def. 4. Fix a Σ-algebra $\mathcal{A}$ to interpret all sorts, constants, and function symbols. For each oclet o, find an assignment of its variables such that all guards of all events evaluate to *true*, and then replace each term t by its value in $\mathcal{A}$ under this assignment (see Fig. 6). Depending on $\mathcal{A}$, an oclet o can unfold into infinitely many different oclets (each representing a different value). The semantics of the unfolded specification defines the semantics of the Σ-specification. The technical details of this construction are given in [13].

4.3 Operational semantics

In principle, Σ-oclets gain operational semantics by the operational semantics of their unfolding. However, unfolding a Σ-specification Ω into an (infinite) set of oclets seems

impractical to operationalize a Σ-specification. A more practical approach is to directly decompose a Σ-oclet into its basic Σ-oclets by lifting Def. 9 to preserve each node's terms. Then, the operational semantics of Ω are the runs of its basic Σ-specification $\hat{\Omega}$.

While $\hat{\Omega}$ yields semantics based on occurrences of single events, it may introduce spurious behavior (not specified by Ω). This spurious behavior arises if a basic Σ-oclet $o[e]$ of o unfolds to some basic low-level oclet that is not defined by the unfolding of entire o. For instance, consider a Σ-oclet o with two events, where event e_1 carries the guard $(x > 0)$, event e_2 carries the guard $(x < 5)$, and $e_1 \leq e_2$. If x is assigned 17, then $o[e_1]$ may occur while $o[e_2]$ may not – the operational semantics would "get stuck" in the middle of o after e_1. To avoid such behavior, we demand that each Σ-oclet $o \in \Omega$ is *data-consistent*. To this end, o should exhibit two properties: (1) two distant nodes only carry variables that also occur in their joint predecessors, and (2) two guards of events of o do not contradict each other. For Σ-specifications that contain only data-consistent oclets, the operational semantics can be implemented; details are given in [13].

For such specifications, it is sufficient to check for a basic Σ-oclet $o[e]$ first, whether its pre-condition occurs at the end of a run π (ignoring the particular values of π), and then to check whether the variables in $o[e]$ can be bound in a way that the terms in $o[e]$ match the values in π.

4.4 Complete Gas Pump Example

Fig. 8 shows the complete set O of Σ-oclets for the gas pump example including its initial run init. The signature Σ for the Σ-specification $\Omega = (O, \mathsf{init}, \mathcal{A})$ is Boolean and also contains the theory of integers with its usual interpretation, and additionally defines constant symbols Price, 6, 17 of sort integer. Further, it has a sort containing the constant symbols Free, Active and Running. The variables are $V = V_{Nat} = \{\$, \mathsf{change}, \mathsf{c_{id}}, \mathsf{done}, \mathsf{p_{id}}, \mathsf{todo}, \mathsf{left}\}$. The initial run init specifies an environment (Env), the operator, and two inactive pumps with ids 6 and 17.

Oclets prepay (when the customer pays in advance) and pay later (when the customer pays at the end) describe the system behavior at the most abstract level. All other oclets but env justify a refinement of main. The specification is data-consistent, and the run of Fig. 7 is a run of this specification.

Oclet env describes the arrival and leaving of a new customer; technically a new *instance* of Customer is created with id ($\mathsf{c_{id}}$), the amount the customer wants to refuel (todo), the id of the pump he wants to use ($\mathsf{p_{id}}$) and his payment ($). Only customers who can pay their desired amount of gas may participate. A customer instance is destroyed after the customer received his change.

When the customer paid or ordered his amount, the operator activates the pump (see prepay and pay later). The $\mathsf{p_{id}}$ assures to activate the pump that the customer wants to use. The amount that has to be pumped is passed to the pump.

Afterwards, the pump can be started by the customer (see get gas). Then, the pump starts pumping (see pump). The customer may stop at any time, at the latest when all of the amount requested by him was pumped. When the pump finished, the actual amount that was pumped is passed to the operator. A customer who did not pay in the beginning can pay now (see pay later). Then, the change is calculated (see prepay and pay later). Note that because $\mathsf{c_{id}}$, $\mathsf{p_{id}}$, todo and $ occur in prepay's precondition, it is not necessary

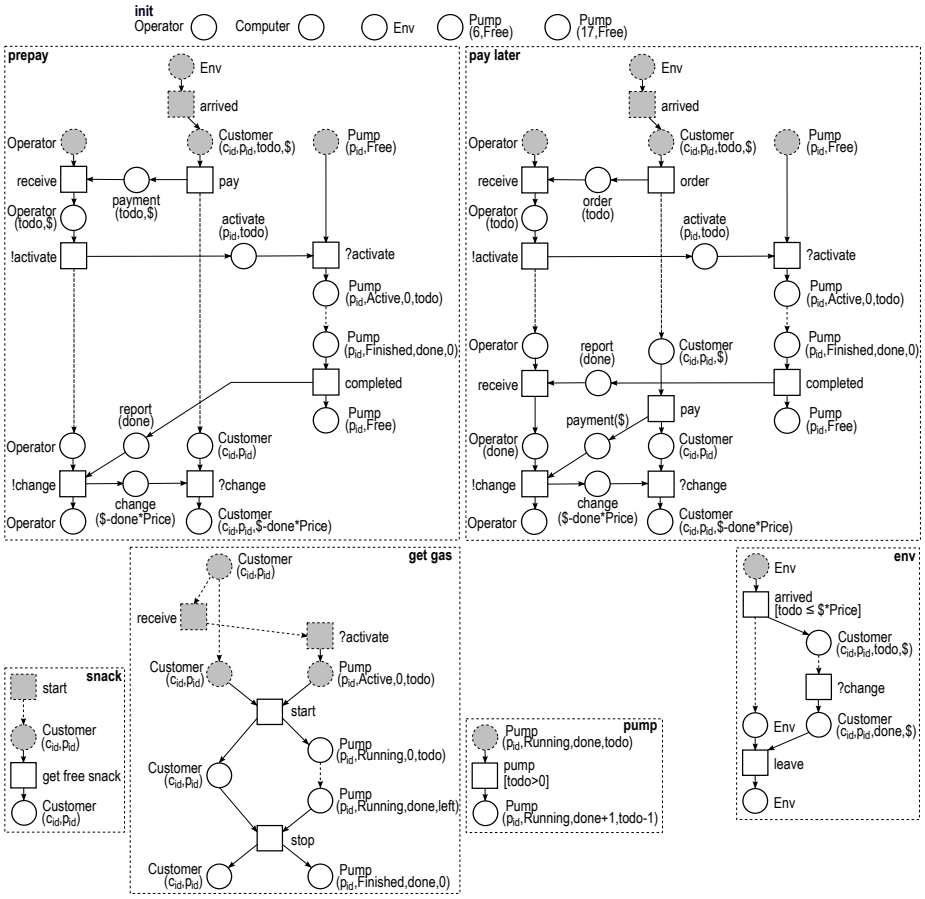

Fig. 8. Complete specification of the gas station example with Σ-oclets.

that the operator continuously carries any of these values up to the occurrence of change. Thus, he may start serving a second customer before the first one gets his change back.

Between starting the pump and leaving the gas station, the customer can get one free snack. Oclet snack can occur only once as an occurrence of get free snack after start prevents oclet snack from being enabled.

Tool support. The approach presented here is implemented in the Eclipse-based tool *Greta* [14]. Greta provides a graphical editor for oclets and animated execution via a simulation engine which implements the operational semantics of oclets including abstract dependencies and Σ-oclets. Greta finds variable assignments and evaluates terms using the simulation engine of CPN Tools via Access/CPN [40]. Additionally, Greta allows to *verify* whether a Petri net implements a given specification, and to *synthesize* a minimal labeled Petri net that implements a specification — both techniques operate on McMillan-prefixes for oclets [12]. Greta is available at `www.service-technology.org/greta`.

5 On the Relation of Oclets to Petri Nets

This section discusses the relation between oclets and Petri nets. As above, oclets without abstraction and data representation are called "classical."

Classical oclets vs. P/T nets. Classical oclets contain P/T-nets: for each P/T net N exists a specification Ω s.t. their behaviors are identical: $R(\hat{\Omega}) = R(N)$ [11]. The converse does not hold. Even classical oclets can mimic a Turing machine by their preconditions [12]. However, if Ω is *bounded* (i.e., there exists a k s.t. no run $\pi \in R(\hat{\Omega})$ of Ω has more than k maximal conditions with the same label), then there is a *labeled* Petri net N with the same behavior: $R(N) = R(\hat{\Omega})$. N can be *synthesized* automatically from Ω by first building a McMillan-prefix [10] for Ω that finitely represents $R(\hat{\Omega})$ and then folding that prefix to N [12].

Σ-oclets vs. P/T nets. Σ-oclets *extend* classical oclets. Clearly, a Σ-specification Ω with an infinite domain has no equivalent finite P/T net. If Ω has only finite domains, it unfolds to a finite low-level specification $val(\Omega)$. If abstract dependencies in $val(\Omega)$ can be refined s.t. they only occur in pre-conditions, then semantics of classical oclets carries over (Thm. 2). Thus not every Σ-specification has a P/T net with the same behavior. Yet it seems plausible that every *bounded* Σ-specification with finite domains has a net with the same behavior: finitely many oclets will allow to continue markings of finite size only in finitely many ways. This suggests that the synthesis from oclets [12] can also be generalized to Σ-oclets. If abstract dependencies in Ω cannot be refined, the synthesis of a P/T net also has to find a refinement of the abstract dependencies.

Σ-oclets vs. Algebraic Petri nets. As synthesizing Algebraic Petri nets from Σ-oclets is out of the scope of this paper, we just sketch some basic observations here. First, every Algebraic net N with term-inscribed arcs and term-inscribed transition guards has an equivalent Σ-specification: translate each transition and its pre- and post-places to a basic oclet by moving arc inscriptions to the respective pre- and post-condition. The reverse direction is more difficult. Term annotations are not an issue as both models are based on the same concepts. But enabledness of an event of a Σ-oclet depends on a "history" in the execution while enabledness of a transition depends on the current marking only. Hee et al. [22] have shown how to express history of tokens in a data-structure of an Algebraic Petri net, and how to use them in guards. Whether all data-dependencies expressible in Σ-oclets can be expressed this way is an open question. Yet, token histories of [22] have drawbacks regarding analysis. Thus, synthesizing a net without an explicit and complete recording of token histories is an interesting, open problem as well. In any case, the signature Σ of a specification has to be extended to allow remembering behavior in the past.

6 Conclusion and Future Work

In this paper, we proposed a formal model for scenarios that combines the advantages of HMSCs (simple semantics by composition of scenarios) with the advantages of LSCs (intuitive notion of scenarios with local preconditions and a flexible style of specifying systems). Our model called *oclets* is based on Petri nets and their distributed runs. The

basic semantic notions of composing LSC-style scenarios to distributed runs have been proposed earlier; in this paper we have shown how the basic model of oclets can be lifted to the expressive means of LSCs by introducing a notion of abstract dependencies and adopted concepts from Algebraic Petri nets to represent data. All of our extensions are constructed such that (1) they generalize and embed the "classical" oclet concept and (2) their semantics can be described in terms of classical oclets. Existing operational semantics for oclets canonically lift to our extended model. Oclets deviate from LSCs where their semantics turns problematic for the aim of deriving specified behavior by composing scenarios (see Sect. 2). Our approach is implemented in the tool Greta and was validated on a number of elaborate examples. Finally, composition, decomposition, abstraction, refinement, and unfolding suggest oclets to be an interesting model for a *calculus* of scenarios. The contribution of this model of scenarios becomes obvious when considering existing works that relate scenario-based techniques to Petri nets.

From Scenarios to Petri nets. To bridge the gap between scenario-based specifications and Petri nets, several approaches have been proposed. Methods to transform UML sequence diagrams to Coloured Petri nets (CPNs) are described in [5], [16], and [41]; the latter approach is used in [15] to model a variant of the gas station example used in this paper. Further, there exist approaches to provide Petri net semantics for MSCs [25, 29] and to synthesize a Petri net out of a MSC specification [36]. While these approaches are straight-forward, the scenario languages lack expressive power (MSCs, UML sequence diagrams) or tend to yield complex specifications in practice (HMSCs, see Sect. 2).

Approaches to transform a LSC specification to a CPN have been described in [1] and [28]. As two LSCs of a specification may be *contradictory* [20], both synthesis approaches need to generate the state space of the LSC and the synthesized CPN to check equivalency of both models. Also *operational semantics* of LSCs [20] requires model checking to find a correct play-out step. We have shown in this paper that Σ-oclets do not require model checking for operational semantics. The gives rise to the hope that components can be synthesized from Σ-oclets as correct by construction (as in the case of HMSCs).

To the best of our knowledge, no other Petri net-based model for scenarios features data and abstract causal dependencies. The notion of history-dependent behavior in Petri nets was introduced in [22]. Oclets particulary relate to *Token History Petri nets* where each token records its "traveling" through the net; LTL-past guards at transitions restrict enabledness to tokens with a particular history. The scenario-based approach of oclets provides a graphical syntax for a subclass of these guards. Particularly, Token History Petri nets might allow to synthesize components from an oclet specification. There are numerous refinement and abstraction techniques for Petri net system models, e.g., by modular refinement [39] or using rules [3]. Refinement of actions of a distributed run has been studied in [18]; we think these results can be lifted to refine abstract dependencies in oclets in a systematic way.

Future work. A next step for research work is to develop *symbolic* semantics for Σ-oclets allowing to concisely describe infinitely many behaviors. This should support solving the main challenge of scenarios: to synthesize high level Petri net components from a Σ-specification without unfolding into a concrete low-level model.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 257593 (ACSI).

References

1. Amorim, L., Maciel, P.R.M., Jr., M.N.N., Barreto, R.S., Tavares, E.: Mapping live sequence chart to coloured petri nets for analysis and verification of embedded systems. ACM SIGSOFT Software Engineering Notes 31(3), 1–25 (2006)
2. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. Fundam. Inform. 95(1), 187–217 (2009)
3. Berthelot, G.: Checking properties of nets using transformation. In: ATPN'85. LNCS, vol. 222, pp. 19–40. Springer (1985)
4. Bontemps, Y., Schobbens, P.Y.: The computational complexity of scenario-based agent verification and design. Journal of Applied Logic 5(2), 252 – 276 (2007)
5. Bowles, J., Meedeniya, D.: Formal transformation from sequence diagrams to coloured petri nets. In: Han, J., Thu, T.D. (eds.) APSEC. pp. 216–225. IEEE Computer Society (2010)
6. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Form. Methods Syst. Des. 19(1), 45–80 (2001)
7. Darondeau, P.: Unbounded Petri Net Synthesis. In: Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 413–438. Springer (2003)
8. Desel, J., Erwin, T.: Hybrid specifications: looking at workflows from a run-time perspective. Computer Systems Science and Engineering 15(5), 291–302 (2000)
9. Desel, J.: Validation of process models by construction of process nets. In: Business Process Management. pp. 110–128 (2000)
10. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. Springer-Verlag (2008)
11. Fahland, D.: Oclets - scenario-based modeling with Petri nets. In: Petri Nets 2009. LNCS, vol. 5606, pp. 223–242. Springer-Verlag, Paris, France (Jun 2009)
12. Fahland, D.: From Scenarios To Components. Ph.D. thesis, Humboldt-Universität zu Berlin (2010), `http://repository.tue.nl/685341`
13. Fahland, D., Prüfer, R.: Data and Abstraction for Scenario-Based Modeling with Petri Nets. Technical Report 12-07, Eindhoven University of Technology (2012)
14. Fahland, D., Weidlich, M.: Scenario-based process modeling with Greta. In: BPM Demos 2010. CEUR-WS.org, vol. 615, pp. 52–57 (2010)
15. Fernandes, J.M., Tjell, S., Jørgensen, J.B.: Requirements engineering for reactive systems with coloured petri nets: the gas pump controller example. In: Jensen, K. (ed.) 8th Workshop on Practical Use of Coloured Petri Nets and the CPN Tools. Aarhus (October 2007)
16. Fernandes, J.M., Tjell, S., Jørgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In: Proceedings of the Sixth International Workshop on Scenarios and State Machines. pp. 2–. SCESM '07, IEEE Computer Society, Washington, DC, USA (2007), `http://dx.doi.org/10.1109/SCESM.2007.1`
17. Genest, B., Minea, M., Muscholl, A., Peled, D.: Specifying and verifying partial order properties using template mscs. In: Walukiewicz, I. (ed.) FoSSaCS. Lecture Notes in Computer Science, vol. 2987, pp. 195–210. Springer (2004)
18. van Glabbeek, R.J., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Inf. 37(4/5), 229–327 (2001)
19. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. In: CIAA '00. LNCS, vol. 2088, pp. 1–33 (2001)

20. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: FMCAD'02. pp. 378–398. LNCS (2002)
21. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
22. van Hee, K.M., Serebrenik, A., Sidorova, N., van der Aalst, W.M.P.: Working with the past: Integrating history in petri nets. Fundam. Inform. 88(3), 387–409 (2008)
23. Heimbold, D., Luckham, D.: Debugging Ada Tasking Programs. IEEE Softw. 2, 47–57 (March 1985)
24. Hélouët, L., Jard, C., Caillaud, B.: An event structure based semantics for high-level message sequence charts. Math. Structures in Comp. Sci. 12(4), 377–402 (2002)
25. Heymer, S.: A semantics for msc based on petri net components. In: Sherratt, E. (ed.) SAM. VERIMAG, IRISA, SDL Forum (2000)
26. ITU-T: Message Sequence Chart (MSC). Recommendation Z.120, International Telecommunication Union, Geneva (2004)
27. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
28. Khadka, B., Mikolajczak, B.: Transformation from live sequence charts to colored petri nets. In: Wainer, G.A. (ed.) SCSC. pp. 673–680. Simulation Councils, Inc. (2007)
29. Kluge, O.: Modelling a railway crossing with message sequence charts and petri nets. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. Lecture Notes in Computer Science, vol. 2472, pp. 197–218. Springer (2003)
30. Mauw, S., Reniers, M.A.: An algebraic semantics of Basic Message Sequence Charts. The Computer Journal 37, 269–277 (1994)
31. Mukund, M., Kumar, K.N., Thiagarajan, P.S.: Netcharts: Bridging the gap between HMSCs and executable specifications. In: CONCUR. LNCS, vol. 2761, pp. 293–307 (2003)
32. Reisig, W.: Petri Nets and Algebraic Specifications. Theor. Comput. Sci. 80(1), 1–34 (1991)
33. Reisig, W.: Elements of distributed algorithms: modeling and analysis with Petri nets. Springer (1998)
34. Ren, S., Rui, K., Butler, G.: Refactoring the Scenario Specification: A Message Sequence Chart Approach. In: OOIS'03. LNCS, vol. 2817, pp. 294–298 (2003)
35. Sengupta, B., Cleaveland, R.: Triggered message sequence charts. IEEE Trans. Software Eng. 32(8), 587–607 (2006)
36. Sgroi, M., Kondratyev, A., Watanabe, Y., Lavagno, L., Sangiovanni-Vincentelli, A.: Synthesis of petri nets from message sequence charts specifications for protocol design. In: DASD '04 (2004)
37. Sibay, G., Uchitel, S., Braberman, V.A.: Existential live sequence charts revisited. In: ICSE'08. pp. 41–50. ACM (2008)
38. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Transactions on Software Engineering 29, 99–115 (2003)
39. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets, Lecture Notes in Computer Science, vol. 625. Springer (1992)
40. Westergaard, M.: Access/CPN 2.0: A High-Level Interface to Coloured Petri Net Models. In: Petri Nets'11. LNCS, vol. 6709, pp. 328–337. Springer (2011)
41. Yang, N., Yu, H., Sun, H., Qian, Z.: Modeling uml sequence diagrams using extended petri nets. Telecommunication Systems 48, 1–12 (2011), http://dx.doi.org/10.1007/s11235-011-9424-5, 10.1007/s11235-011-9424-5