Architecture of Information Systems

using the theory of Petri nets

lecture notes
for
Systeemmodelleren 1 (2M310)

K. van Hee
N. Sidorova
M. Voorhoeve
and
J. van der Woude

Department of Computing Science
Technische Universiteit Eindhoven

Table of Contents

1 Introduction

Information systems are complex artifacts like airplanes and power plants. Because society heavily depends on information systems, they have to be developed very carefully.

This course focuses on an essential part of the development of information systems: the architecture of the system. We describe architectures using high level Petri nets in combination with UML-notation and Z-notation. An architecture is used by potential users to understand the systems, by programmers who have to construct parts of it and by persons who have to deploy and maintain the system when it is built. An important role of an architecture is that it can be used to verify and validate properties of the system.

After an introduction the course starts with the theory of Petri nets. This process modelling technique was developed in 1962 and its theory and applications are still being developed further. We use only a small part of Petri net theory, however some parts we use are quite new. About half of the course is devoted to this theory and to modelling with Petri nets. The other half is devoted to architecture of information systems. An important part of an architecture can de described by the diagram techniques of (high level) Petri nets but we also need a data modelling notation and a notation for describing input-output relations. For data modelling we use UML's class models and for relations we use Z-notation.

At the end of the course the students should have a good knowledge of the Petri net theory that is presented, know what the role of an architecture is in the development of a system and last but not least they should be able to design and verify an architecture of a complex system based on an informal, verbal description.

The approach of this course is applicable to the software engineering project (2R690), specifically for the first three phases.

The course is based on material of the book "Information systems engineering, a formal approach" (K.M. van Hee, Cambridge University Press (1994)) but contains important new material. The software tool ExSpect can be used to make a part of the models that form an architecture. Models in ExSpect are executable and can be used as prototype. For verification of particular aspects we have Woflan. (ExSpect and Woflan are both developed at the TU/e).

2 Information Systems Engineering

Systems engineering is the scientific discipline focused on the creation of new systems designed to play a role in our society. In most cases systems have a limited scope and instead of a role in 'society' we often consider the role of a system in some business environment. Besides the development of totally new systems it becomes more and more important to consider the renewal or renovation of existing systems. This requires different approaches and introduces new challenges. One of the major issues is that the systems engineers have to understand the system that has to be renovated. An existing system puts extra constraints on the development of new parts which makes renovation different from development from scratch.

Information systems can be characterized by their functions: the generation, interpretation, presentation, transformation, transportation and storage of information. Information systems are becoming very complex and society is strongly dependent on them. This implies that it becomes more and more important to develop information systems in a systematic and controlled way. The information systems we focus on require many person-years of development (think of 20 or more) so the only way to develop these systems in a reasonable time is to develop them by team work. This requires a strong coordination mechanism which allows to divide a system into *components* that can be developed in isolation and that are relatively easy to assemble. Note that we always use already existing components like a database management system.

Today *component based development* is considered to be the most promising way of developing information systems. In order to make the development of information systems more efficient, there is a strong focus on *reusing* existing components. Although different business environments require different information systems, it turns out that many required systems functions are the same as in other systems, so the reuse of components is a big challenge. At the moment there are many vendors who sell software packages or components. These vendors try to make their products applicable in a wide range of environments and therefore they endow their systems with parameter mechanisms in order to tune their systems to the specific needs of a business environment by choosing the right parameter values. This process of parameter setting is called the *configuration* of components. Therefore the development of information systems has changed dramatically the last ten years: instead of constructing the software from scratch in some programming language, the *selection*, *configuration* and *assembly* of existing components is becoming the major part of the job. Of course there is always a need for new components and the assembly of components often requires programming as well, so the programming discipline will continue to be important.

A crucial factor in the systematic, controlled and coordinated development of systems is the ability to keep overview over the complexity of the system. The way to do this is to use models to describe parts or aspects of a system. A set of models that together define the essentials of a system is called the *architecture* of the system. The development of the architecture and the maintenance of it during the development process is one of the most important tasks in the development of complex information systems. The architecture of a system plays different roles, not only in the development process but also during the life cycle of a system. Therefore it IT-architects are becoming more and more important.

3 Information Systems Life cycle

Information systems have to play a role in some business environment, in particular they have to fulfill tasks in some *business process*, like the control of equipment or the administration of an insurance company.

In the life cycle of a system, we distinguish the following 15 activities:

1. requirements analysis

2. process architecture
3. functional architecture
4. software architecture
5. network architecture
6. selection of existing components
7. design of new components
8. construction of new components
9. configuration of components
10. assembly of components
11. testing the system
12. acceptance of the system
13. installation in the operational environment
14. migration of old to new system
15. operations: functional and technical maintenance

It is really a cycle because after some years of operation the system has to be renovated or renewed due to changing business goals and new technology (i.e. new hardware and software that is needed for our system) and then a new requirements analysis starts. There are many different ways to cluster the activities of an information systems lifecycle into phases. There are also many different ways to manage the lifecycle of systems development.

For example, the approach of software tool company *Rational*, which is popular in industry, distinguishes four phases: *inception*, *elaboration*, *construction*, and *transition*. (We will not go into detail here.)

As another example, in the standard lifecycle of ESA, the European Space Agency, which is used in our course on Software Engineering (2R690), the lifecycle has six phases, which cluster the activities mentioned above as follows:

	phase	*activities*
1	user requirements definition	requirements analysis
2	software requirements	process and functional architecture
3	architectural design	software and network architecture
		selection of existing components
		design of new components
4	detailed design and production	construction of new components
		configuration of components
		assembly of components
		testing
5	transfer	acceptance test
		installation
		migration
6	operations and maintenance	functional and technical maintenance

The *requirements analysis* has three goals: to understand the environment in which the system will operate, to define the functions the system is supposed to have and to list the constraints the system or the development process has to fulfill. Therefore, a description is made of the business process to be supported by the information system. At least, there should be a verbal description. This is important because all the *stakeholders* of the system should be able to verify this.

Stakeholders are persons for whom the system will play a role. Stakeholders may be users, system administrators or managers of an organization. Besides stakeholders, we distinguish *actors*. They are either persons who have interaction with the system, like users and administrators, or who have other software systems that interact with the system.

Besides the process the system has to support, we describe the functionality of the system by means of *use cases*. A use case is a "piece of functionality" of the system. A use case belongs to one or more actors; one of them is the initiator of the use case, and the others are participants in it.

At the level of requirements, the use cases are described in natural language, as a list of tasks the system has to fulfill.

We do not require more than verbal statements in this phase; formalizations are usually made in the architecture activities :

The second group of activities is called the *architecture*. The architecture is the answer by the architect to the question represented by the requirements. We distinguish four levels of architecture:

1. the *process architecture*, also called *business architecture*; a set of models that describes the environment of the system.

2. the *functional architecture*: a set of models that describe the logical structure of the system as a set of related logical components. Together, they cover the functionality required by the use cases; usually, one component handles one or more use cases;

3. the *software architecture*: a set of software components that realize the functional architecture. A software component is a "piece of code";

4. the *network architecture*: a set of computer devices and a communications network on which the software components are executing.

The architecture results in a set of components and the communication relationships between them. For each component we have either a *specification*, i.e. a description that tells us *what* the component is doing or a *decomposition* into a network of communicating sub-components. In many cases the components are already available and have to be searched for.

This is done in the *selection* activity. If they can be found, they have to be configured (i.e., parameters have to be set). This is done in the *configuration* activity. If they are not available, they have to be constructed where the models of the component are used as specification. Configuration of software packages is a relatively young profession. After the realization of components, the assembly of components has to be performed in the *assembly* activity. In the *test* activity we check if the system functions according to the requirements, both the functional and the non-functional requirements. If the acceptance test is successful, the system is installed in the environment where it should be operational; this is called *installation*. Before the system can be used it should be loaded with the business data. Often this requires conversion of data from the old to the new system; this is called *migration*. When this is complete, the system can be used, and the *operational* phase starts: the applications should be maintained; functional maintenance means updating user and system parameters, while technical maintenance includes the adaptation of the system to new infrastructure.

In this course we focus on the design of architectures and in particular of the process and functional architecture. As said before the architecture is not only used for the development of a system but also for the maintenance and renovation of the system. So the architecture itself should be maintained carefully.

1. Petri Nets

4 Preliminaries

4.1 Bags

$\mathbb{N}$ denotes the set of natural numbers. A *bag* (or a multi-set) over some alphabet A is a mapping from A into $\mathbb{N}$, which indicates a number of element occurrences in the bag. The set of bags over A is denoted by $\mathsf{bag}.A$. We denote bags by listing the elements between square brackets and we use superscripts for the multiplicity of the occurrences. So $[a^2, b^3, c]$ is the bag consisting of two occurrences of a, three of b and one of c.

The operations we use on bags are addition, comparison and subtraction. So we have $[a^2, b^3, c] + [a^3, c^2, d] = [a^5, b^3, c^3, d]$, $[a^2, b^3, c] < [a^2, b^3, c^2, d]$, and $[a^2, b^3, c^2, d] - [a^2, b^3, c] = [c, d]$. Sometimes we use a simplified notation, writing p, or i^k+p, instead of $[p]$ or $[i^k, p]$ respectively. Overloading the notation, we write $\emptyset$ for the empty bag. As for sets, we write $a \in X$ to indicate individual elements of a bag.

To make things precise, let $X, Y \in \mathsf{bag}.A$ and $a \in A$. Note that $X(a)$ denotes the number of appearances of a in bag X. Now we will formally define the operations $\in, +, -, \leq, <$ on bags.

$$\begin{aligned}
a \in X &\Leftrightarrow X(a) > 0 \\
(X + Y)(a) &= X(a) + Y(a) \\
(X - Y)(a) &= \max\{0, X(a) - Y(a)\} \\
X \leq Y &\Leftrightarrow \forall\, a \in A : X(a) \leq Y(a) \\
X < Y &\Leftrightarrow X \leq Y \wedge (\exists\, a \in A : X(a) < Y(a))
\end{aligned}$$

4.2 Equivalence relation

A binary relation R on a set A is an *equivalence relation* iff R is

– reflexive, i.e., $\forall\, a \in A : (a, a) \in R$;

– symmetric, i.e., $\forall\, a, b \in A : (a, b) \in R \Rightarrow (b, a) \in R$;

– transitive, i.e., $\forall a, b, c \in A : ((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$.

The symbol '$\sim$' is among the most commonly used for equivalence relations. An equivalence relation defines a partition of A into *equivalence classes*, i.e. the sets of the form $\{x \in A \mid x \sim a\}$.

4.3 Transition Systems

Definition 4.1. *A* transition system *is a* $\langle S, \mathcal{A}, T, s_0 \rangle$ *where*

– S *is a set of* states*;*

– $\mathcal{A}$ *is an* alphabet of actions*, i.e., a (finite) set of distinct labels*

– $T \subseteq S \times \mathcal{A} \times S$ *is a* transition relation*;*

– $s_0 \in S$ *is an* initial *state.*

Each transition is thus a triple (s_1, a, s_2) where s_1 is a source state, a is an action label and s_2 is a destination state. We denote it as $s_1 \overset{a}{\rightarrow} s_2$, and we say that a leads from s_1 to s_2.

If a transition system has a finite number of states it can be represented by a graph. Each state is represented by a node and each transition by a labeled edge.

We say that a state s' is *reachable* from state s if there exists a transition sequence σ (maybe empty) leading from s to s'. In this case we write $s \overset{*}{\rightarrow} s'$. We say that a state is *dead* with respect to the current state of the transition system, if it is not reachable from the current state. We say that a state s is *recurrent* iff $\forall s' : s \overset{*}{\rightarrow} s' \Rightarrow s' \overset{*}{\rightarrow} s$, and a state s is *transient* iff it is not recurrent. And finally, a state s is *deadlock* iff therei s no transition starting in s, and a state s is *livelock* iff all the transitions starting in s lead to s.

5 Petri Nets

5.1 Nets and markings

Definition 5.1 (Petri net). *A* Petri net *is a tuple* $N = \langle P, T, F \rangle$*, where:*

– P *and* T *are two disjoint non-empty finite sets of* places *and* transitions *respectively;*

– $F : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ *is a* weight function *(also called* flow relation*).*

We present nets with the usual graphical representation consisting of circles for places, boxes for transitions and arrows linking places and transitions related by F. When we do not want to distinguish between places and transitions, we refer to them as the nodes of a Petri net.

Definition 5.2 (marking). *Markings are configurations of a net. A* marking *M of N is a bag over P, i.e., a mapping from P into $\mathbb{N}$ where $M(p)$ denotes the number of tokens in p in marking M. We write (N, M) to denote a Petri net N with marking M.*

Given a transition $t \in T$, the *preset* ${}^\bullet t$ and the *postset* $t^\bullet$ of t are bags over P given by ${}^\bullet t(p) \overset{\text{def}}{=} F(p,t)$ and $t^\bullet(p) \overset{\text{def}}{=} F(t,p)$ for any $p \in P$. Similarly, for a given place $p \in P$, the *preset* ${}^\bullet p$ and the *postset* $p^\bullet$ of p are bags over T given by ${}^\bullet p(t) \overset{\text{def}}{=} F(t,p)$ and $p^\bullet(t) \overset{\text{def}}{=} F(p,t)$ for any $t \in T$.

Definition 5.3 (enabledness). *A transition $t \in T$ is* enabled *in marking M iff ${}^\bullet t \leq M$.*

Definition 5.4 (firing). *An enabled transition t may* fire*, thus performing action t. This results in a new marking M' defined by $M' \overset{\text{def}}{=} M - {}^\bullet t + t^\bullet$.*

We write $M \xrightarrow{t}_N M'$ to denote that $M \xrightarrow{t} M'$ is a step in net N. Derived notations are “$M \xrightarrow{t} M'$” when N is implicit, $M \xrightarrow{t}$ when M' is not relevant, and “$M \rightarrow M'$” when there is a transition t such that “$M \xrightarrow{t} M'$”.

Lemma 5.5. *Each Petri net determines a transition system.*

Proof. The transition system induced by a Petri net takes the markings as states and the transitions as action alphabet. The transition relation records the changes in the markings that result from the firing of the transitions. So we have $(\mathsf{bag}.P, T, U, m_0)$, where $m_0 \in \mathsf{bag}.P$ and $m_1 \xrightarrow{t} m_2$ in U iff ${}^\bullet t \leq m_1 \wedge {}^\bullet t + m_2 = m_1 + t^\bullet$

It should be noticed that several transitions in a Petri net can be enabled at the same time. They may be concurrently enabled or, otherwise, they are in conflict.

Definition 5.6 (concurrently enabled). *A subset of transitions $T' \subseteq T$ is* concurrently enabled *in a marking M iff $\sum\limits_{t \in T'} {}^\bullet t \leq M$.*

Definition 5.7 (conflict). *Subsets of transitions T_1 and T_2 with $T_1 \cap T_2 = \emptyset$ are* in conflict *if both T_1 and T_2 are concurrently enabled in M, but $T_1 \cup T_2$ is not concurrently enabled in M.*

According to Definition 5.4 enabled transitions may fire, but only one at a time. If several transitions are concurrently enabled, they can fire in any order. Such a view of concurrency corresponds to the *interleaving process semantics*. In the framework of the *step semantics*, we can allow transitions to fire simultaneously, i.e., if a subset T' of transitions is concurrently enabled, all transitions of T' may fire at once. Moreover, we could go one step further by considering bags of transitions to be enabled concurrently. It is possible, for example, that a marking M is equal to ${}^\bullet t + {}^\bullet t$ for some transition t. Then we could allow t to perform two firings at once. In the following, we will mainly work with the interleaving semantics of processes.

Definition 5.8 (firing sequence). *A sequence $\langle t_1, \dots, t_n \rangle$ of transitions is a firing sequence in (N, M) iff there exist markings $M_1, \dots, M_n$ such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$.*

We write "$M \xrightarrow{\sigma} M'$" when σ is a firing sequence leading from M to M', "$M \xrightarrow{*} M'$" if there exists a firing sequence (maybe empty) leading from M to M', and "$M \xrightarrow{+} M'$" if there exists a non-empty firing sequence leading from M to M'.

Definition 5.9 (reachability). *We say that a marking M' is* reachable *in (N, M) iff $M \xrightarrow{*} M'$.*

Definition 5.10 (Parikh vector). *Let $N = \langle P, T, F \rangle$ be a Petri net and σ a finite sequence of transitions. The Parikh vector $\overrightarrow{\sigma} : T \to \mathbb{N}$ of a transition sequence σ is the bag of σ.*

For example, if $T = \{t_0, \dots, t_6\}$, then the Parikh vector of the sequence $\langle t_3, t_5, t_3, t_4, t_2 \rangle$ is $(0, 0, 1, 2, 1, 1, 0)$. Note that the Parikh vector destroys the information on the order of the firings.

Lemma 5.11. *Let $M \xrightarrow{\sigma_1} M_1$, $M \xrightarrow{\sigma_2} M_2$, and $\overrightarrow{\sigma_1} = \overrightarrow{\sigma_2}$. Then $M_1 = M_2$.*

Lemma 5.12 (additivity of markings). *Let $M_1 \xrightarrow{\sigma_1} M_1'$ and $M_2 \xrightarrow{\sigma_2} M_2'$. Then $M_1 + M_2 \xrightarrow{\sigma} M_1' + M_2'$ where σ is an arbitrary merge (or interleaving) of σ_1 and σ_2.*

Definition 5.13 (liveness). *A transition t of a marked Petri net (N, M_0) is* live *iff for every reachable marking M there is a marking M' reachable from M and enabling t.*
A net N is live iff all its transitions are live.

Definition 5.14 (dead). *A transition t of a marked Petri net (N, M_0) is* dead *iff there is no marking M' reachable from M_0 which enables t.*
A marking M is a deadlock *in a Petri net N iff all transitions of (N, M) are dead.*

Lemma 5.15. *A live Petri net (N, M) does not have any dead transition.*

Note that the reverse is not true: a net which has no dead transitions may be not live. Take, for instance, a net where every transition can fire only once. Such a net is not dead, but it is not live as well.

Definition 5.16 (boundedness). *A place p of a Petri net (N, M_0) is* bounded *iff there is a natural number n such that for every reachable state the number of tokens in p is at most n.*
*A place p is k-*bounded *iff for every reachable state the number of tokens in p is at most k.*
*A net (N, M_0) is (k-)*bounded *iff all its places are (k-)bounded.*
A 1-bounded place or net is also called safe.

Definition 5.17 (conservation). *A marked Petri net (N, M) is a* conservative *net iff all the reachable markings have the same total number of tokens as M.*

Lemma 5.18. *A marked Petri net (N, M) is a conservative net iff for every non-dead transition t:* $| {}^{\bullet}t | = | t^{\bullet} |$.

Definition 5.19 (path). *Let $N = \langle P, T, F \rangle$ be a Petri net, and let $n_1, n_k \in (P \cup T)$. An* undirected path *C from a node n_1 to a node n_k, is a sequence $\langle n_1, n_2, \ldots, n_k \rangle$, where $n_j \in (P \cup T)$, for $j = 1, \ldots, k$, such that for every i with $1 \le i < k$, we have either $(n_i, n_{i+1}) \in F$ or $(n_{i+1}, n_i) \in F$.*
The path is directed *if $(n_i, n_{i+1}) \in F$ for all suitable i.*

Definition 5.20 (connectedness). *A Petri net $N = \langle P, T, F \rangle$ is* connected *iff for every pair of nodes x and y, $x, y \in (P \cup T)$, there is an undirected path leading from x to y.*
If for every node there is a directed path to any other node, the net is called strongly connected.

Definition 5.21 (acyclic net). *A Petri net $N = \langle P, T, F \rangle$ is an acyclic net iff there is no directed path from a node to itself.*

5.2 Place invariants

Let's consider a Petri net in Fig. 1.1. It is easy to see that whatever the initial marking of the net is, the sum of the tokens in places p_1, p_3, p_4 in any reachable marking is the same as in the initial one. The explanation of this phenomenon is very simple: with its firing, every transition of the net consumes from p_1, p_3, p_4 the same number of tokens as it puts into these places. We will say that $(1, 0, 1, 1, 0)$ (or $p_1 + p_3 + p_4$) is a place invariant of this net. The invariant indicates that the amount of particular resources stays unchanged during the work of the net. Often, we are interested in more complicated invariants of the form $I(p_1) \cdot M(p_1) + \ldots + I(p_n) \cdot M(p_n)$, or just its vector $(I(p_1), \ldots, I(p_n))$ of coefficients. The coefficient vector has length $| \; P \; |$ and we may express the invariant by way of the innerproduct: $I \bullet M$ thus opening up the toolbox of linear algebra. Although we won't treat the calculation of invariants here, it may be reassuring to know that simple linear algebraic techniques suffice to automatically calculate the invariants of a Petri net.

Definition 5.22 (place invariant). *Let $N = \langle P, T, F \rangle$ be a Petri net. A mapping $I : P \to \mathbb{Z}$ is a place invariant iff for every transition $t \in T$ the following holds:*

$$\sum_{p \in {}^{\bullet}t} I(p) = \sum_{p \in t^{\bullet}} I(p), \text{ or, equivalently, } I \bullet ({}^{\bullet}t) = I \bullet (t^{\bullet})$$

The equivalence above may be explained by

$$I \bullet ({}^{\bullet}t) = \sum_{p \in P} I(p) \cdot {}^{\bullet}t(p) = \sum_{p \in P} I(p) \cdot F(p, t) = \sum_{p \in {}^{\bullet}t} I(p)$$

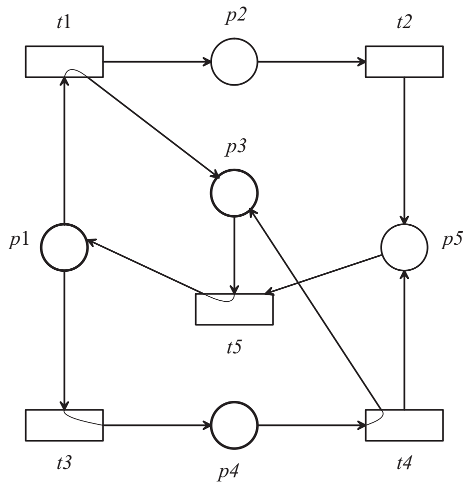


Fig. 1.1. *$M(p_1) + M(p_3) + M(p_4) = const.$*

and something similar for the right hand side.

Theorem 5.23 (property of place invariants). *Let (N, M_0) be a marked Petri net, and I a place invariant of N. For any reachable marking M in (N, M_0) we have $I \bullet M = I \bullet M_0$.*

Proof. First, note that $M \xrightarrow{t} M'$ implies that $M + {}^\bullet t = t^\bullet + M'$, so that after the innerproduct with the invariant, we have $I \bullet M + I \bullet ({}^\bullet t) = I \bullet (t^\bullet) + I \bullet M'$; hence, by 5.22, $I \bullet M = I \bullet M'$. Since a reachable marking results from M_0 after a finite sequence of firings, iteration of the above reasoning leads to the required result. □

5.3 Transition invariants

After defining place invariants, it seems logical to introduce a similar notion for transitions. Note, however, that we restrict the transition invariants to natural valued coefficients, as negative firings do not make much sense.

Definition 5.24 (transition invariant). *Let $N = \langle P, T, F \rangle$ be a Petri net. A mapping $J : T \to \mathbb{N}$ is a transition invariant iff for every place $p \in P$ the following holds:*

$$\sum_{t \in {}^\bullet p} J(t) = \sum_{t \in p^\bullet} J(t), \text{ or, equivalently, } J \bullet ({}^\bullet p) = J \bullet (p^\bullet)$$

Lemma 5.25 (property of transition invariants). *Let σ be a finite sequence of transitions of a net N which is enabled at a marking M. Then the Parikh vector $\overrightarrow{\sigma}$ is a transition invariant iff $M \xrightarrow{\sigma} M$ (i.e., iff the consecutive firings of transitions in σ reproduce the marking M).*

Proof. Let σ be a firing sequence such that $M \xrightarrow{\sigma} M'$. Because σ was assumed to be enabled in M, this is equivalent to $M + \sum\limits_{t \in \sigma} t^{\bullet} = \sum\limits_{t \in \sigma} {}^{\bullet}t + M'$.
Let $p \in P$ be arbitrary. Then $M(p) = M'(p)$ iff $(\sum\limits_{t \in \sigma} t^{\bullet})(p) = (\sum\limits_{t \in \sigma} {}^{\bullet}t)(p)$ or, equivalently with some rewriting using the Parikh vector, $\sum\limits_{t \in T} \vec{\sigma}(t) \cdot F(p,t) = \sum\limits_{t \in T} \vec{\sigma}(t) \cdot F(t,p)$. In terms of the inproduct this reads $\vec{\sigma} \bullet (p^{\bullet}) = \vec{\sigma} \bullet ({}^{\bullet}p)$. Thus $M = M'$ iff $\vec{\sigma}$ is a transition invariant. $\square$

Fig. 1.2 gives an example of transition invariants.

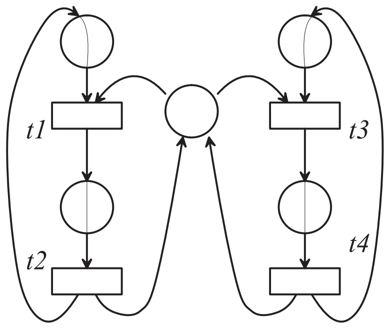


Fig. 1.2. $(1, 1, 0, 0)$ and $(0, 0, 1, 1)$, or in other words $[t_1, t_2]$ and $[t_3, t_4]$ are transition invariants.

6 Special types of Petri nets

6.1 Petri nets with inhibitor arcs

One of the main factors limiting the modeling power of Petri nets is the lack of possibility for zero testing, i.e. determining whether an unbounded place is empty. Let us suppose we are constructing a model for a system processing two kinds of tasks incoming to places p_1, p_2 respectively, with transitions t_1, t_2 processing each kind of tasks respectively (Fig. 1.3). Now, let the first kind of task have a priority over the second one, i.e. transition t_2 may fire iff transition t_1 is not enabled. This can not be modeled with classical Petri net.

To increase the modeling power of Petri nets, the *inhibitor* arcs will be introduced. While the normal arc leading from a place to a transition says that the transition is enabled only if a token is present in this place, the inhibitor arc says that the transition is enabled only if there is no token in this place. Graphically, we present the inhibitor arc as a line with a circle at the end. So we can model the priority in the system from Fig. 1.4 by introducing an inhibitor arc from place p_1 to transition t_2.

It was proved that a Petri net with inhibitor arcs produces a modeling scheme which can model any Turing machine. This, however, also means that Petri nets with inhibitor arcs belong to the class of systems for which most of the analysis problems, like reachability, boundedness, termination (absence of infinite firing sequence), are undecidable.

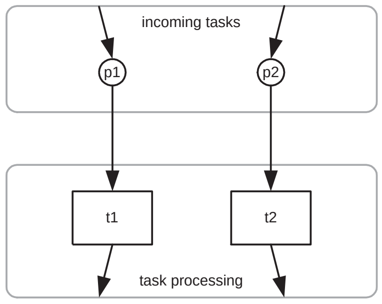


Fig. 1.3. Processing of two kinds of tasks.

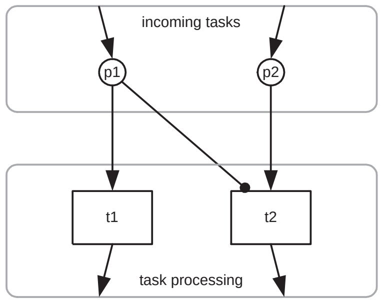


Fig. 1.4. Introducing a priority by means of the inhibitor arc.

6.2 Subclasses of Petri nets

Definition 6.1 (state machine). *Let $N = \langle P, T, F\rangle$ be a Petri net. N is a* state machine *(SM) iff*

$$\forall t \in T : \mid {}^\bullet t \mid \leq 1 \wedge \mid t^\bullet \mid \leq 1.$$

State machines are equivalent to finite automata.

Definition 6.2 (marked graph). *Let $N = \langle P, T, F\rangle$ be a Petri net. N is a* marked graph *(MG) iff*

$$\forall p \in P : \mid {}^\bullet p \mid \leq 1 \wedge \mid p^\bullet \mid \leq 1.$$

Marked graphs are dual to state machines in the graph-theoretic sense and from the modeling point of view. State machines can represent conflicts by a place with several output transitions, but they can not represent concurrency and synchronization. Marked graphs, on the other hand, can represent concurrency and synchronization, but cannot model conflicts or data-dependent decisions.

Definition 6.3 (free-choice Petri net). *Let $N = \langle P, T, F\rangle$ be a Petri net. N is a* free-choice Petri net *iff* $\forall t_1, t_2 \in T$, ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ *implies* ${}^\bullet t_1 = {}^\bullet t_2$.

Free-choice Petri nets allows both concurrency and conflicts but in more restricted manner then general Petri nets. By the definition of free-choice Petri nets, the input places for the transitions form a partition; so if a place is an input to several transitions (potential conflict), then input places for all these transitions coincide. Hence either all of these conflicting transitions are simultaneously enabled, or none of them is enabled. This allows the choice (conflict resolution) of the transition that will fire next, to be made freely; the presence of tokens in other places is not involved in this decision.

Definition 6.4 (workflow net). *A Petri net N is a* WF-net (Workflow net) *if and only if:*

– *N has two special places: i and f. Place i is a initial place: ${}^{\bullet}i = \emptyset$ and f is a final place: $f^{\bullet} = \emptyset$.*

– *If we add a* closing transition *t to N that connects place f with i (i.e., ${}^{\bullet}t = \{f\}$ and $t^{\bullet} = \{i\}$), then the resulting Petri net is strongly connected. Later on, we call this net the* closure *of WF-net N and denote $\overline{N}$ (cf. Fig. 1.5).*

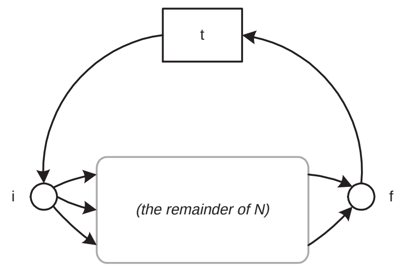


Fig. 1.5. The scheme of $\overline{N}$.

Given the definition of a workflow net, it is easy to derive the following structural properties of WF-nets:

Lemma 6.5. *For every WF-net $N = \langle P, T, F\rangle$*

– *the second requirement in the definition above is equivalent to the following requirement: 'Every node $x \in P \cup T$ is on a path from i to f';*

– *for any place $p \in P$ different from i, ${}^{\bullet}p \neq \emptyset$, i.e., i is the only initial place;*

– *for any place $p \in P$ different from f, $p^{\bullet} \neq \emptyset$, i.e., f is the only final place.*

Definition 6.6 (soundness). *A WF-net is k-*sound *iff for every marking M reachable from marking i^k, there exists a firing sequence leading from marking M to marking f^k. Formally:*

$$\forall M : (i^k \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} f^k).$$

A WF-net is sound *iff for every natural k, it is k-sound.*

For some classes of Petri nets (e.g., colored Workflow nets), it is often enough to require 1-soundness of a net. 1-soundness does not necessarily imply k-soundness. Fig. 1.6 gives an example of a net which is 1-sound but not 2 sound.

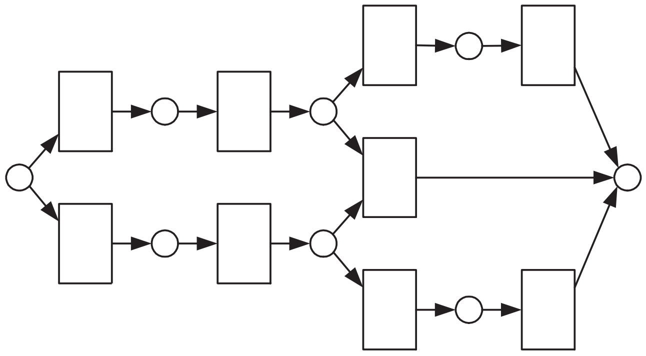


Fig. 1.6. An example of a net that is 1-sound but not 2-sound.

Lemma 6.7. *In k-sound WF-net the only marking reachable from i^k with at least k tokens in place f is f^k itself. Formally:*

$$\forall M : (i^k \xrightarrow{*} M \wedge M \geq f^k) \Rightarrow (M = f^k).$$

Proof. Let's assume that on the contrary, there exists a marking M such that $i^k \xrightarrow{*} M + f^k$. Since the net is k-sound, there exists a firing sequence σ such that $(M+f^k) \xrightarrow{\sigma} f^k$. As $f^\bullet = \emptyset$, and hence the tokens in f can not move to another place and they are not involved in any firing, we have $M \xrightarrow{\sigma} \emptyset$, which means that (due to the additivity of markings) the last transition in σ has an empty postset. This implies that there is no path from this transition to any other node of the net. This contradicts the requirement of strong connectivity in the definition of WF-net. □

Lemma 6.8. *If the closure of a WF-net is live and bounded then the net is 1-sound.*

Proof. Let $(\overline{N}, i)$ be live, in particular, the closing transition t is live. Since enabling markings for t are at least f it follows that

$$(\forall M : (i \xrightarrow{*} M) \; : \; (\exists M' :: (M \xrightarrow{*} f + M'))).$$

Now assume that N is not 1-sound and let μ be such that $i \xrightarrow{*} \mu$ but μ does not lead to f. By the above it follows that $\mu \xrightarrow{*} f + \mu'$ for some nonempty μ'. But this violates the boundedness of $\overline{N}$ since

$$i \xrightarrow{*} \mu \xrightarrow{*} f + \mu' \xrightarrow{t} i + \mu', \text{ hence } i \xrightarrow{*} i + k \cdot \mu' \text{ for all } k.$$

□

Lemma 6.9. *If a WF-net is 1-sound then its closure is bounded.*

Proof. Let's assume there exists N such that it is 1-sound and its closure is unbounded. Then $\overline{N}$ contains an unbounded place p. Hence, there exists an infinite sequence of reachable in $(\overline{N}, i)$ markings $p^{k_1} + M_1, p^{k_2} + M_2, \ldots, p^{k_n} + M_n, \ldots$ with $k_1 < k_2 < \ldots < k_n < \ldots$. Since the sequence is infinite, it contains some markings $p^{k_i} + M_i, p^{k_j} + M_j$ such that $p^{k_i} + M_i < p^{k_j} + M_j$ [1]. In other words, there exists $M' > \emptyset$ such that $p^{k_j} + M_j = (p^{k_i} + M_i) + M'$.

Since N is 1-sound, $[f]$ is the only marking with a token in place f that is reachable in (N, i) (Lemma 6.7). The closing transition t of $\overline{N}$ is the only transition in $\overline{N}$ from which i can get a token, and ${}^\bullet t = f$, hence i is the only marking with a token in i reachable in $\overline{N}$. This fact implies that every marking reachable in $(\overline{N}, i)$ is reachable in (N, i) as well. So $p^{k_i} + M_i, p^{k_j} + M_j$ are reachable in N. N is 1-sound, hence (Def. 6.6) there exists a firing sequence σ such that $p^{k_i} + M_i \xrightarrow{\sigma}_N f$. Then (Lemma 5.12) $i \xrightarrow{*} p^{k_j} + M_j \xrightarrow{\sigma}_N f + M'$, which contradicts the statement of Lemma 6.7. □

Definition 6.10 (t-workflow net). *A Petri net N is a* tWF-net (t-workflow net) *if and only if:*

- *N has two special transitions: t_i and t_f. Transition t_i is an initial transition: ${}^\bullet t = \emptyset$. Transition t_f is a stop transition: $f^\bullet = \emptyset$.*
- *If we add a place p to N which connects transitions t_f and t_i (i.e., ${}^\bullet p = t_f$ and $p^\bullet = t_i$), then the resulting Petri net is strongly connected.*

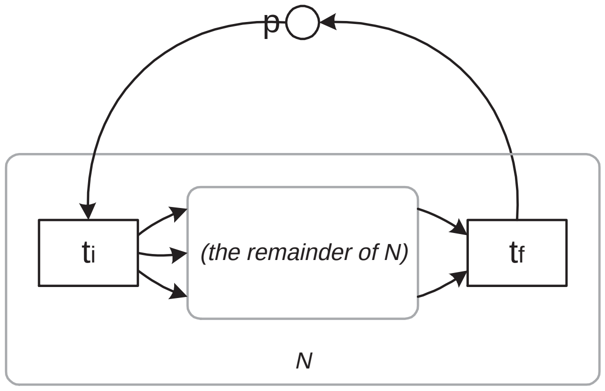


Fig. 1.7. The scheme of $\overline{N}$.

Definition 6.11 (extension and closure). *The extension of a tWF-net N is the net pN WF-net constructed by adding to N places i and f with ${}^\bullet i = \emptyset$, $i^\bullet = t_i$, ${}^\bullet f = t_f, f^\bullet = \emptyset$.*
The closure of N is the net obtained by adding one place p with ${}^\bullet p = t_f$ and $i^\bullet = t_i$. We will use the same notation $\overline{N}$ as in case of WF-nets.

[1] which is a consequence of Dixon's Lemma that can be interpreted as following: "any infinite subset of $\mathbb{N}^P$ contains an infinite increasing sequence".

Definition 6.12 (soundness). *A tWF-net $N = \langle P, T, F\rangle$ is k-*sound *iff the net pN, as defined in 6.11, is a k-sound WF-net.*

Lemmas 6.8, 6.9 immediately imply that the following statements hold:

Lemma 6.13. *If the closure of a tWF-net is live and bounded then the net is 1-sound.*

Lemma 6.14. *If a tWF-net is 1-sound then its closure is bounded.*

Definition 6.15 (SMWF). *N is a State Machine Workflow net (SMWF) iff N is a Workflow net and a state machine.*

Lemma 6.5 implies that for any SMWF the following holds:

$$\forall t \in T :\mid {}^{\bullet}t \mid = 1 \wedge \mid t^{\bullet} \mid = 1.$$

This immediately implies the following fact:

Lemma 6.16. *Any SMWF is a conservative net.*

Lemma 6.17. *Any SMWF is a sound workflow net.*

Proof. First, we prove that any SMWF is 1-sound. Consider a SMWF N with the initial marking i. Any reachable marking in (N, i) consists of 1 token (lemma 6.16). Let p be a marking consisting of the only token in an arbitrary place p. N is strongly connected, which means that there exists a path $p, t_1, p_1, t_2, p_2, \ldots, t_n, f$ leading from p to f. It is easy to see that $t_1, \ldots, t_n$ form a firing sequence σ such that $p \xrightarrow{\sigma} f$. Hence, N is a 1-sound net.

Now, let's prove that, for any natural k, N is k-sound. Consider an arbitrary marking M reachable from i^k. Lemma 6.16 implies that $\mid M \mid = k$, i.e., $M = p_1 + p_2 + \ldots + p_k$. From the proof above we know that there exist firing sequences $\sigma_1, \ldots, \sigma_k$ such that $p_1 \xrightarrow{\sigma_1} f, \ldots, p_k \xrightarrow{\sigma_k} f$. Then (lemma 5.12) $M \xrightarrow{\sigma_1 \ldots \sigma_k} f^k$, i.e., N is k-sound whatever k is taken. Hence, N is sound. □

Definition 6.18 (MGWF). *N is a* marked graph Workflow net *(MGWF) iff N is a WF-net and N is a marked graph.*

Lemma 6.5 implies that for any MGWF the following holds:

$$\forall p \in P \setminus \{i, f\} :\mid {}^{\bullet}p \mid = 1 \wedge \mid p^{\bullet} \mid = 1.$$

Lemma 6.19. *A MGWF is sound iff it has no cycle.*

Proof. Let N be sound and assume that $\sigma = \langle t_1, p_1, \ldots, t_n, p_n, t_1 \rangle$ is a cycle in N. Clearly, $p_1 + \ldots + p_n$ is an invariant, which is 0 since i doesn't occur in σ. Hence no p_i is reachable and all t_i are dead.
The net N is a WF net, so there is a directed path $\langle i, v_1, q_1, v_2, \ldots, v_n, q_n, t_1 \rangle$. Since $q_n^\bullet = [t_1]$ and t_1 is dead, q_n cannot lead to f, and q_n is hence not reachable. As ${}^\bullet q_n = [v_n]$, v_n is dead. Repeating this line of reasoning shows that v_1 is dead too, which can't be true since $i \xrightarrow{*} f$ and $i^\bullet = [v_1]$. So we have shown that a sound MGWF is acyclic.

In the other direction, let N have no cycles. We first consider N with the initial marking $[i]$ and show that N is 1-sound.

We introduce a recursively-defined ordering function on the nodes of N (see an example at Fig. 1.8):

$$ord(x) = \begin{cases} 0 & \text{if } {}^\bullet x = \emptyset \\ \max\limits_{y \in {}^\bullet x} ord(y) + 1 & \text{else.} \end{cases}$$

Since N is acyclic, the recursion is well-founded. Moreover, for any node $n : n \neq i \wedge n \neq f \Rightarrow ord(i) < ord(n) < ord(f)$.

Note that since $\forall\, p \in P : | p^\bullet | = 1$, there are no conflicts on a token between transitions, i.e., every enabled transition eventually fires. Therefore, we can consider a firing of enabled transitions in any convenient order without loss of generality.

In marking $[i]$, the transition with $ord(t) = 1$ is enabled. Its firing leads to a marking M such that every place p with $ord(p) = 2$ has one token. Hence all the transitions with $ord(t) = 3$ are enabled. In general, if all the transitions with $ord(t) < n-1$ have fired, we got a marking in which all the places with $ord(p) = n-1$ have a token, i.e., all the transitions with $ord(t) = n$ are enabled. Finally, we get to the point where the input transition of f is enabled, it fires and puts a token into f.

Now, we show that it is the only token in this marking. In our firing sequence, every transition fired exactly once, i.e., every place (besides i) got exactly one token during the execution run. On the other hand, every place, besides f is an input place for some transition, and since every transition has fired, every token, besides the one in f has been removed. We can conclude that N is 1-sound.

Now consider the run of the net $(N, [i^k])$. Since there are no conflicts on a token, the marking of the net can be considered as a superposition of markings obtained in k nets $(N, [i])$. So we immediately get that N is k-sound for any k, and, hence, sound. □

The other conclusions we can draw from the proof above are the following:

Lemma 6.20. *Let N be an acyclic MGWF. Then*

– $\langle N, [i] \rangle$ *is safe;*

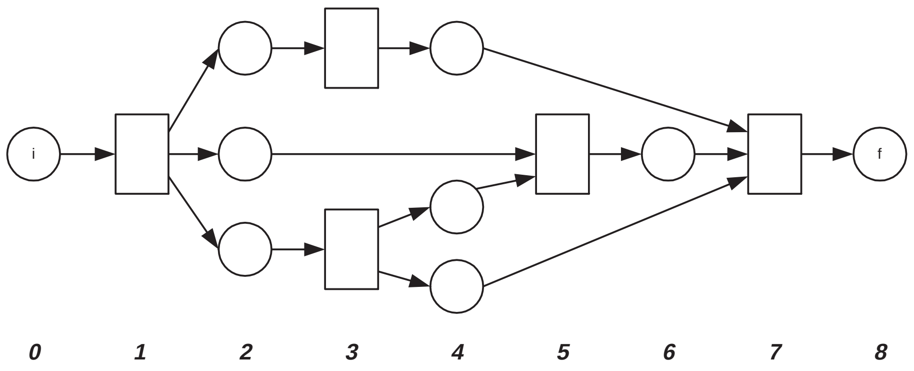


Fig. 1.8. Defining *ord*-function on a marked graph.

– $\langle N, [i^k] \rangle$ *is k-bounded;*

– $\langle N, [i^k] \rangle$ *contains no dead transitions.*

Definition 6.21 (free-choice WF). *Let* $N = \langle P, T, F \rangle$ *be a free choice Petri net and a Workflow net. Then* N *is a* free-choice Workflow net.

Lemma 6.22. *A 1-sound free-choice WF-net is safe.*

(The proof of this lemma is beyond the scope of this course.)

7 Constructing Petri Nets

7.1 Transformations of classical Petri nets

Definition 7.1 (renaming places). *Let* $N = \langle P, T, F \rangle$ *be a Petri net and* M *be a marking of* N*. A renaming function for places is a function* $\rho : P \to P'$*. Renaming* (N, M) *according to* ρ *gives us a new net* $\langle P', T, F' \rangle$ *and marking* M'*, denoted by* N/ρ *and* M/ρ *respectively, where* F' *and* M' *are defined by:*

$$F'(p', t) = \sum_{\rho(p)=p'} F(p, t),\ F'(t, p') = \sum_{\rho(p)=p'} F(t, p) \text{ and } M'(p') = \sum_{\rho(p)=p'} M(p).$$

In applying the renaming operation to N we reduce N by *fusing* all places with the same ρ-image into a single place. Those new places are in fact equivalence classes under the equivalence relation $\{(p, q) \mid \rho(p) = \rho(q)\}$ on P induced by ρ. (And the renaming function is the quotient map for that relation.) The equivalence classes will be written as p/ρ, which explains the notation N/ρ and M/ρ for the new net and marking.
The transitions are untouched but the arcs between places and transitions follow the fusion process: if $^\bullet t$ is $[p_1, \ldots, p_k]$, then in N/ρ, $^\bullet t$ is $[p_1/\rho, \ldots, p_k/\rho]$. A marking M in N is fused into a

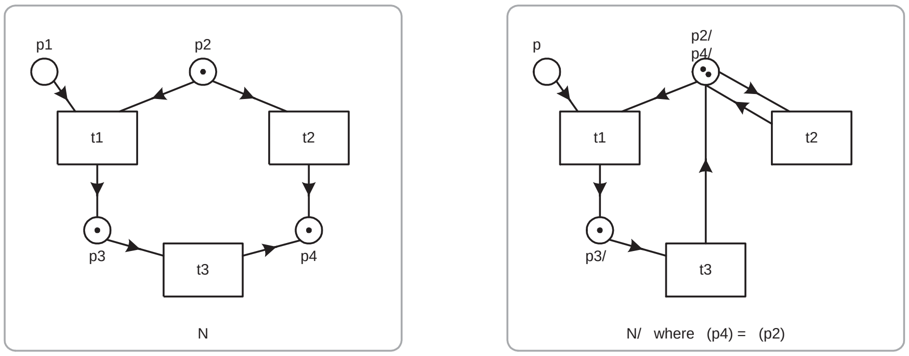


Fig. 1.9. Place fusion

marking $m \overset{\text{def}}{=} M/\rho$ in N/ρ. It is easy to see that m and M have the same number of tokens. Fig. 1.9 gives an example where p_4 is renamed to p_2. In cases where only a few places of a net are renamed while the other places are left untouched, we will say that we rename places $p, q, s, \ldots$ to $x, y, z, \ldots$ correspondingly. So we can see that renaming places is a way of expressing their fusion.

Definition 7.2 (renaming transitions). *Let $N = \langle P, T, F\rangle$ be a Petri net and M be a marking of N. A renaming function for transitions is a function $\rho : T \to T'$. Renaming (N, M) according to ρ gives us a new net $N/\rho = \langle P, T', F'\rangle$ where F is defined by:*

$$F'(p, t') = \sum_{\rho(t)=t'} F(p, t) \text{ and } F'(t', p) = \sum_{\rho(t)=t'} F(t, p).$$

The marking remains the same, as the places don't change.

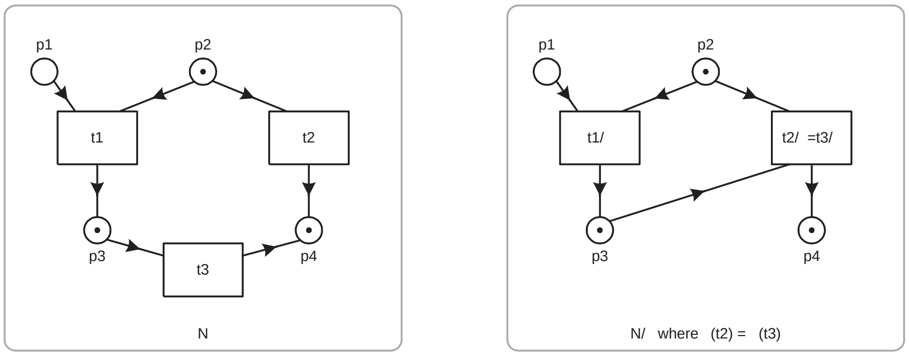


Fig. 1.10. Transition fusion

As was the case for place renaming, the renaming of transitions is a kind of fusion; see figure 1.10 for an example. The net N is reduced by fusing transitions with the same ρ-image into a single transition. We write t/ρ for the equivalence class of t (and thus for the fused place in N/ρ). The places are untouched but the arcs between places and transitions follow the fusion process: if $^\bullet p$ is $[t_1, \ldots, t_k]$, then in N/ρ, $^\bullet p$ is $[t_1/\rho, \ldots, t_k/\rho]$.

Definition 7.3 (net union). *Let $N_1 = \langle P_1, T_1, F_1 \rangle$ and $N_2 = \langle P_2, T_2, F_2 \rangle$ be two Petri nets. The union $N_1 \cup N_2$ of these nets is a net $\langle P_1 \cup P_2, T_1 \cup T_2, \tilde{F_1} + \tilde{F_2} \rangle$, where the zero complementing domain extensions $\tilde{F_i}$ of F_i are defined by*

$$\tilde{F}_i(x,y) = \begin{cases} F_i(x,y) & x,y \in (P_i \cup T_i) \\ 0 & \textit{else.} \end{cases}$$

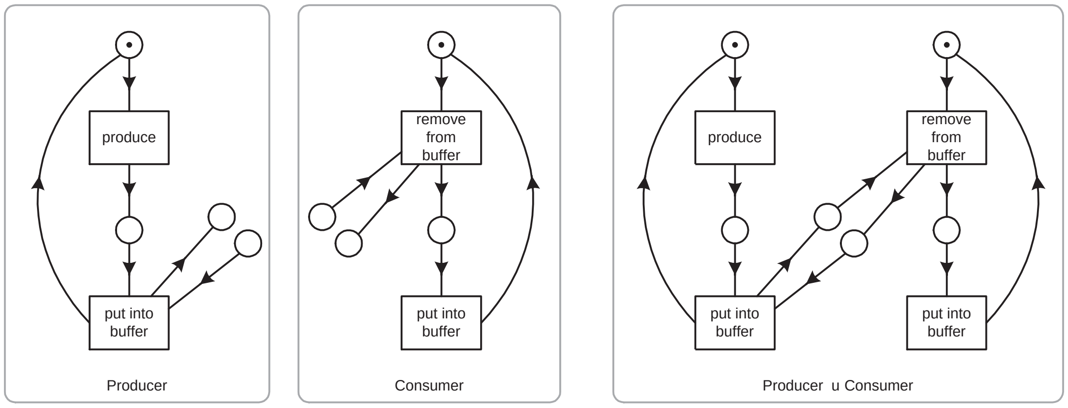


Fig. 1.11. The union of the Producer and the Consumer nets.

Net union may be used as a weak alternative to the fusion by renaming. The difference between the renaming and the union is that the renaming is applied on a single net, while the union allows us to "fuse" places and transitions of one net with places and transitions of the other net (only in pairs). Fig. 1.11 gives an example of the union of two nets.

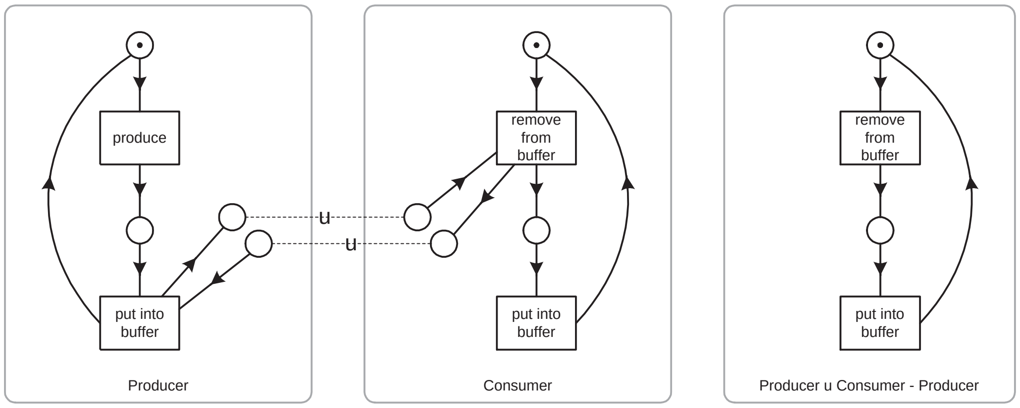


Fig. 1.12. The difference operation illustrated.

Definition 7.4 (net difference). *Let $N_1 = \langle P_1, T_1, F_1 \rangle$ and $N_2 = \langle P_2, T_2, F_2 \rangle$ be two Petri nets. The difference N_1/N_2 of these nets is a net $\langle P_1 \backslash P_2, T_1 \backslash T_2, F \rangle$, where*

$$F(x,y) = max\{0, \tilde{F_1}(x,y) - \tilde{F_2}(x,y)\},$$

and the $\tilde{F_i}$'s as defined in Def. 7.3.

7.2 Connecting Workflow Nets

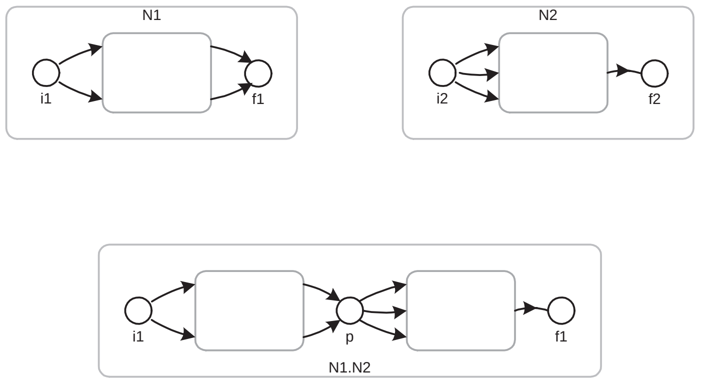


Fig. 1.13. Sequential composition of WF-nets.

Definition 7.5 (sequential composition). *Let $N_1 = \langle P_1, T_1, F_1\rangle$, $N_2 = \langle P_2, T_2, F_2\rangle$ be two Workflow nets with start places i_1, i_2 and final places f_1,f_2 respectively. The* sequential composition $N = N_1 \cdot N_2$ *of N_1, N_2, is a net obtained in the following way (Fig. 1.13):*

1. *bijectively rename N_1 such that i_1 and f_1 are renamed to i and p respectively, while f doesn't occur, resulting in N_1';*
2. *bijectively rename N_2 such that i_2 and f_2 are renamed to p and f, respectively, while the other names of nodes differ from those of N_1', resulting in N_2';*
3. $N \overset{def}{=} N_1' \cup N_2'$.

Proposition 7.6. *Let N_1, N_2 be two k-sound Workflow nets. Then $N_1 \cdot N_2$ is a k-sound Workflow net.*

Definition 7.7 (sequential composition – t-Workflow nets). *Let $N_1 = \langle P_1, T_1, F_1\rangle$, $N_2 = \langle P_2, T_2, F_2\rangle$ be two t-Workflow nets with start transitions i_1, i_2 and stop transitions f_1, f_2 respectively. The* sequential composition *of N_1, N_2, is a net $N = N_1 \cdot N_2$ obtained in the following way (Fig. 1.13):*

1. *rename t_{f1} in N_1 to t, where $t \notin (T_1 \cup T_2)$, resulting in N_1';*
2. *rename t_{f2} in N_2 to t, resulting in N_2';*

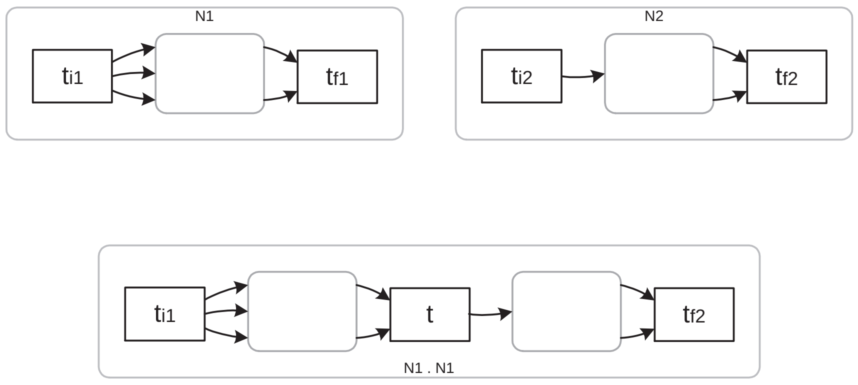


Fig. 1.14. Sequential composition of tWF-nets.

3. $N \overset{def}{=} N_1' \cup N_2'$.

Proposition 7.8. *If N_1, N_2 are two k-sound t-Workflow nets then $N_1 \cdot N_2$ is a k-sound t-Workflow net.*

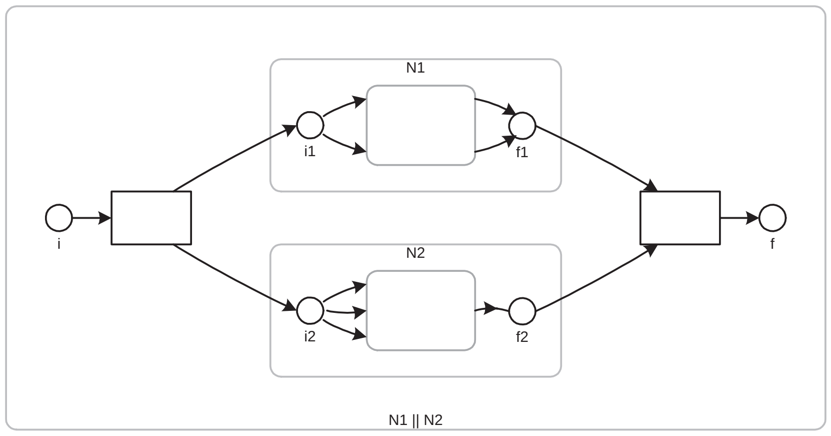


Fig. 1.15. Parallel composition of WF-nets.

Definition 7.9 (parallel composition). *Let $N_1 = \langle P_1, T_1, F_1\rangle, N_2 = \langle P_2, T_2, F_2\rangle$ be two Workflow nets with start places i_1, i_2 and stop-places f_1, f_2 respectively. Assume that their node sets are disjoint and don't contain members of $\{i, f, t_p, t_s\}$. The* parallel composition *$N_1 \parallel N_2$ of N_1, N_2, is a net $N = \langle P, T, F\rangle$ such that (Fig. 1.15)*

$$P = P_1 \cup P_2 \cup \{i, f\},\ T = T_1 \cup T_2 \cup \{t_p, t_s\} \text{ and } F = F_1 \cup F_2 \cup G$$

where G is zero outside the domains of the F_i except for

$$^\bullet i = \emptyset, i^\bullet = [t_p] \text{ and } {}^\bullet f = [t_s], f^\bullet = \emptyset,$$
$$^\bullet t_p = i, t_p^\bullet = [i_1, i_2] \text{ and } {}^\bullet t_s = [f_1, f_2], t_s^\bullet = [f].$$

Proposition 7.10. *If N_1, N_2 are two k-sound Workflow nets then $N_1 \parallel N_2$ is a k-sound Workflow net.*

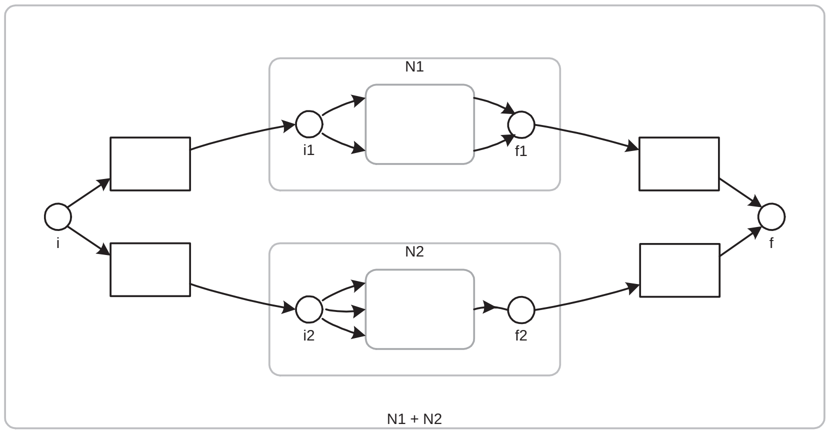


Fig. 1.16. Alternative composition of WF-nets.

Definition 7.11 (alternative composition). *Let $N_1 = \langle P_1, T_1, F_1\rangle, N_2 = \langle P_2, T_2, F_2\rangle$ be two Workflow nets with start places i_1, i_2 and stop-places f_1, f_2 respectively. Assume that their node sets are disjoint and don't contain members of $\{i, f, t_{i_1}, t_{i_2}, t_{f_1}, t_{f_2}\}$. The* alternative composition *of N_1, N_2, $N_1 + N_2$, is a net $N = \langle P, T, F\rangle$ such that (Fig. 1.16)*

$$P = P_1 \cup P_2 \cup \{i, f\},\ T = T_1 \cup T_2 \cup \{t_{i_1}, t_{i_2}, t_{f_1}, t_{f_2}\} \text{ and } F = F_1 \cup F_2 \cup G$$

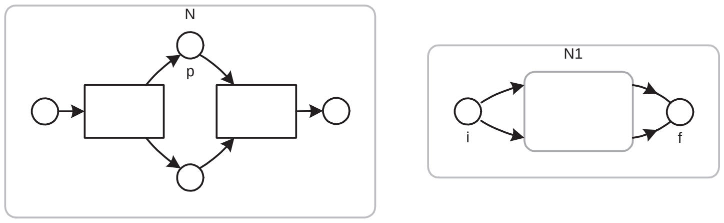
where G is zero outside the domains of the F_i except for

$$\begin{gathered} {}^\bullet i = \emptyset, i^\bullet = [t_{i_1}, t_{i_2}] \text{ and } {}^\bullet f = [t_{f_1}, t_{f_2}], f^\bullet = \emptyset, \\ {}^\bullet t_{i_1} = [i], t_{i_1}^\bullet = [i_1] \text{ and } {}^\bullet t_{i_2} = [i], t_{i_2}^\bullet = [i_2], \\ {}^\bullet t_{f_1} = [f_1], t_{f_1}^\bullet = [f] \text{ and } {}^\bullet t_{f_2} = [f_2], t_{f_2}^\bullet = [f]. \end{gathered}$$

Proposition 7.12. *If N_1, N_2 are two up-to-k-sound Workflow nets then $N_1 + N_2$ is an up-to-k-sound Workflow net.*

Definition 7.13 (place-expansion). *Let N be a Petri net with a distinguished place p and N_1 be a WF-net with initial place i and final place f, such that the nodes are disjoint. The p-expansion of net N with net N_1 is a net $N' = \langle P \cup P_1 \setminus \{p\}, T \cup T_1, G\rangle$, where G is the obvious modification of the union of the weight functions such that ${}^\bullet i = {}^\bullet p$ and $f^\bullet = p^\bullet$. (Fig. 1.17)*

Lemma 7.14. *Let N be an n-sound WF-net with a k-bounded place p and N_1 an up-to-k sound WF-net. Then p-expansion of net N with net N_1 is n-sound.*



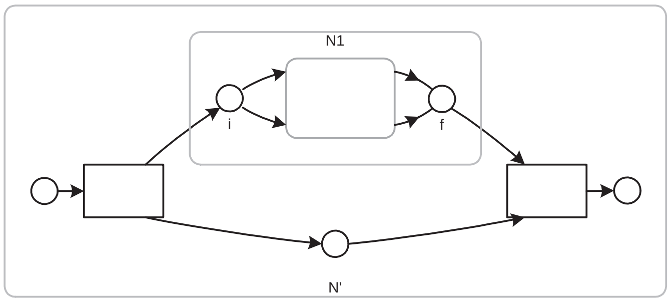


Fig. 1.17. Place expansion of WF-nets.

Definition 7.15 (transition expansion). *Let N be a Petri net with a distinguished transition t and N_1 be a tWF-net with start transition t_i and stop transition t_f, such that the nodes are disjoint. The t-expansion of net N with net N_1 is a net $N' = \langle P \cup P_1, T \cup T_1 \setminus \{t\}, G\rangle$, where G is the obvious modification of the union of the weight functions such that ${}^\bullet t_i = {}^\bullet t, t_f^\bullet = t^\bullet$. (Fig. 1.18)*

Lemma 7.16. *Let N be an n-sound WF-net with a transition t such that the places from ${}^\bullet t$ are k-bounded, and N_1 be a tWF-net which is m-sound for any $m \leq k$. Then p-expansion of net N with net N_1 is n-sound.*

Definition 7.17 (loop addition). *Let $N = \langle P, T, F\rangle$ be a WF-net with a place $q \in P$ and a transition $t \notin T$. Then the* loop extension of N is the net $N' = \langle P, T \cup \{t'\}, F'\rangle$ where $F'(q,t) = f'(t,q) = 1$ and $F'(x,y) = F(x,y)$ for any $x, y \in (P \times T) \cup (T \times P)$.

Lemma 7.18. *The loop extension of a k-sound WF-net is k-sound.*

Definition 7.19 (iteration). *Let $N = \langle P, T, F\rangle$ be a WF-net. The* iteration *of N is a WF-net $N' = \langle P \cup \{p_i, p_f\}, T \cup \{t_i, t_f, t_+\}, G\rangle$, where G contains the additions to F such that (Fig. 1.19)*

$$
\begin{gathered}
{}^\bullet t_+ = [f], t_+^\bullet = [i] \\
{}^\bullet p_i = \emptyset, p_i^\bullet = [t_i] \textit{ and } {}^\bullet p_f = [t_f], p_f^\bullet = \emptyset, \\
{}^\bullet t_i = [p_i], t_i^\bullet = [i] \textit{ and} \\
{}^\bullet t_f = [f], t_f^\bullet = [p_f].
\end{gathered}
$$

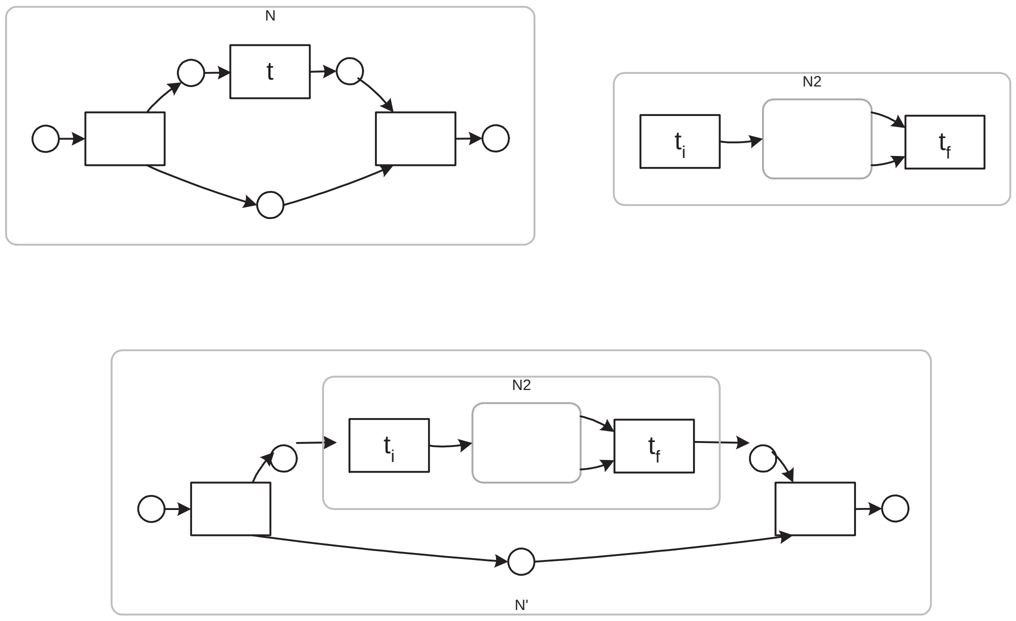


Fig. 1.18. Transition expansion of WF-nets.

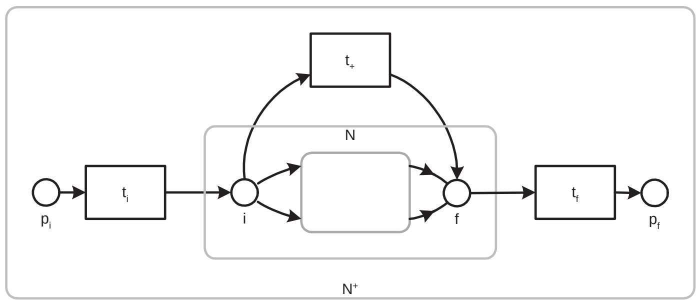


Fig. 1.19. Iteration of a WF-net process.

Lemma 7.20. *If N is a k-sound WF-net, then its iteration is a k-sound WF-net.*

Definition 7.21 (State-Machine Pattern net). *A workflow net $N = \langle P, T, F\rangle$ has a* State-Machine Pattern net *(SM-pattern net) iff there exist a state machine $\tilde{N} = \langle \tilde{P}, \tilde{T}, \tilde{F}\rangle$, a set W of disjoint sound t-WF nets with elements $w = \langle P_w, T_w, F_w\rangle$ and a bijective function $w : \tilde{T} \to W$ such that expanding every transition $t \in \tilde{T}$ with $w(t) \in W$, we obtain N (see Fig. 1.20).*
Function w is called the expansion function.

Note that

$$P = \tilde{P} + \bigcup_{w \in W} P_w; \quad T = \bigcup_{w \in W} T_w.$$

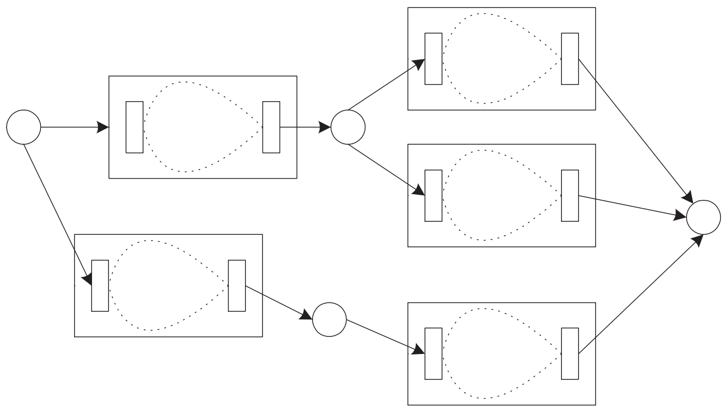


Fig. 1.20. An example of a SM-pattern net

Lemma 7.22. *Let N be a SM-pattern net as defined above. Then for any marking M reachable from $[i]$ one of the following conditions holds:*
(i) *there is $p \in \tilde{P}$ such that $M = [p]$;*
(ii) *there is $w \in W$ such that $M \in \mathsf{bag}.P_w$ and $M \xrightarrow{*} q$ for some $q \in \tilde{P}$.*

Proof. For the initial marking $[i]$, condition (i) holds. Now suppose we have a marking $[p]$ with $p \in \tilde{P}$. Then one of the initial transitions of some t-workflow net w from W can fire, resulting in a marking $M \in \mathsf{bag}.P_w$. So condition (ii) holds. Since w is a sound t-workflow net and it is connected with N only via initial and final transitions, tokens will stay in w until the final transition of w fires, resulting in some marking $[q]$ where $q \in \tilde{P}$ (condition (i) holds). □

Lemma 7.23. *Let N be a workflow net with an SM-pattern. Then N is sound.*

Proof. Since N is a workflow net with a SM-pattern, N can be obtained as a result of transition expansions applied to a state machine. Any state machine is sound (lemma 6.17), and applying a transition expansion to a sound workflow net, one obtains a sound workflow net (lemma 7.15). Hence, N is sound. □

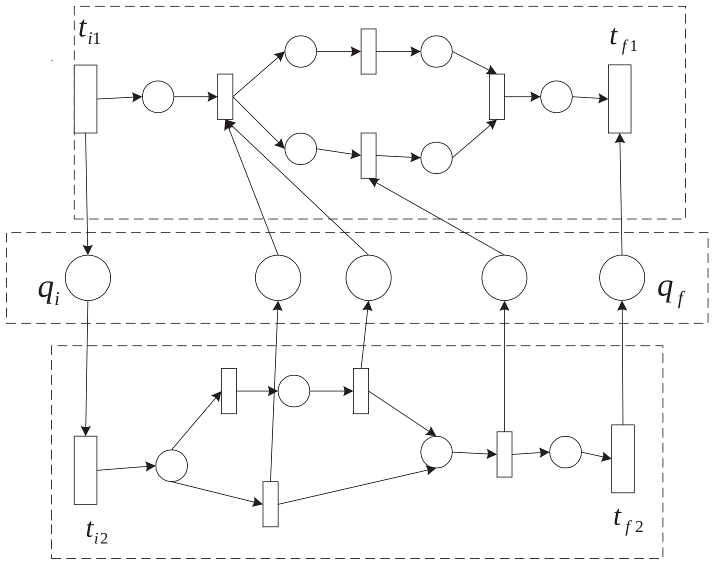


Fig. 1.21. An example of a CS-coupling (not sound!)

7.3 Client-Server coupling

Definition 7.24 (Client-Server Coupling). *Let $N_1 = \langle P_1, T_1, F_1\rangle, N_2 = \langle P_2, T_2, F_2\rangle$ be two disjoint t-workflow nets with initial and final transitions $t_{i1}, t_{i2}, t_{f1}, t_{f2}$ respectively, Q be a set of places disjoint with P_1 and P_2, and $cs : Q \to ((T_1 \times T_2) \cup (T_2 \times T_1))$ be an injective function such that there exist $q_i, q_f \in Q$ with $cs(q_i) = (t_{i1}, t_{i2}),\ cs(q_f) = (t_{f2}, t_{f1})$. We write $cs_t(q)$ for $\pi_1 \circ cs(q)$ — the projection of cs on the first component (transmitter), and $cs_r(q)$ for $\pi_2 \circ cs(q)$ — the projection of cs on the second component (receiver).*

A Client-Server Coupling *(CS-coupling) $CS_N(N_1, N_2, Q, cs)$ is a t-workflow net $N = \langle P_1 \cup P_2 \cup Q, T_1 \cup T_2, F\rangle$ with the initial transition t_{i1} and the final transition t_{f1} (see Fig. 1.21), where*

$$F(x,y) = \begin{cases} F_i(x,y) & \text{if } x, y \in (P_i \cup T_i), i = 1,2; \\ 1 & \text{if } y \in Q \text{ and } cs_t(y) = x; \\ 1 & \text{if } x \in Q \text{ and } cs_r(x) = y; \\ 0 & \text{otherwise.} \end{cases}$$

Note that in $CS_N(N_1, N_2, Q, cs)$ N_1 is the *client* and N_2 is the *server*.
We say that the CS-coupling is sound iff the resulting t-workflow net is sound.

Lemma 7.25. *Let N_1, N_2 be MGWF-nets and $N = CS_N(N_1, N_2, Q, cs)$ be an acyclic CS-coupling of them. Then N is a sound MGWF.*

Proof. First, N is a MGWF, since every place in N has exactly one input and one output position (check it with Def. 7.24). Since any acyclic MGWF is a sound workflow net (Lemma 6.19), N is a sound workflow net. □

Definition 7.26 (isomorphic nets). *Two Petri nets $N_1 = \langle P_1, T_1, F_1\rangle, N_2 = \langle P_2, T_2, F_2\rangle$ are called* isomorphic *if there exist bijective functions $r_p : P_1 \to P_2,\ r_t : T_1 \to T_2$ defining a renaming of places and transitions of N_1 which results in N_2.*

Note that the renaming functions for the workflow nets can map initial places to initial places and final places to final places only.

Theorem 7.27 (isomorphic coupling). *Let N_1, N_2 be disjoint SM-pattern nets with corresponding state machines $\tilde{N}_1, \tilde{N}_2$ and w_1, w_2 as expansion functions, and moreover, $\tilde{N}_1, \tilde{N}_2$ are isomorphic nets with renaming functions r_p, r_t.*
Let N be a parallel composition of N_1, N_2 where for every transition $u \in \tilde{T}$ either a CS-coupling $CS_N(w_1(u), w_2(r_t(u)), Q_u, cs_u)$ or $CS_N(w_2(r(u)), w_1(u), Q_u, cs_u)$ is applied such that for every $u, v \in \tilde{T}$ with $^\bullet u = {}^\bullet v$ the coupling of pairs $(w_1(u), w_2(r_t(u)))$, $(w_1(v), w_2(r_t(v)))$ goes in the same direction (either client-server for both, or server-client), and moreover, every such a coupling forms a sound t-workflow net (see Fig.1.22). Then N is a sound workflow net.

Proof. Any marking M of N reachable from $[i]$ (except for the ones with tokens in $[f]$) can be represented as $M_1 + M_2 + M_3$ where $M_1 \in \mathsf{bag}.P_1$, $M_2 \in \mathsf{bag}.P_2$ and $M_3 \in \mathsf{bag}.(\bigcup_{t\in T_1} Q_t)$. The only marking directly reachable from $[i]$ is $[i_1, i_2]$, i.e., $[i_1, r_p(i_1)]$.

Let us consider an arbitrary marking $[p, r_p(p)]$, where $p \in (P_1 \cup P_2)$ and the output t-workflows of p serve as servers (see Fig. 1.23). Without loss of generality we can assume that $p \in P_1$. We'll refer to such a marking as a marking of type I. In any such a marking, only the output transitions of p are enabled (since every output transition of $r_p(p)$ has also an input place from Q and, thus, not enabled). Every output transition of p is an input transition of one of the transition workflow nets from N_1. Now let one of the output transitions of p fire, say the input transition t_{iu} of a workflow net $w_1(u)$. Then we obtain a marking of the form $M_1 + M_2 + M_3$ where $M_1 \in \mathsf{bag}.P_{w_1(u)}$ for some $u \in p^\bullet$ ($u \in \tilde{N}$), $M_2 = [r_p(p)]$, $M_3 = [q_{iu}] + M'_3$ with $M'_3 \in \mathsf{bag}.(Q_u \setminus \{q_{iu}, q_{fu}\})$ (type II). The only transitions enabled in this marking are the input transition $t_{ir(u)}$ of the workflow $w_2(r_t(u))$ and possibly transitions of $w_1(u)$. So after a firing we can obtain either another marking of type II, or a marking consisting of a marking of the workflow net $CS_N(w_1(u), w_2(r_t(u)), Q_u, cs_u)$ and a one-token marking $[r_p(p)]$, which just indicates that $w_2(r_t(u))$ can start working (type III).

The only transitions of $CS_N(w_1(u), w_2(r_t(u)), Q_u, cs_u)$ that communicate with the rest of the net are t_{fu} and $t_{fr(u)}$. So until $w_1(u)$ or $w_2(u)$ terminates, the firings in N are the firings in $CS_N(w_1(u), w_2(r_t(u)), Q_u, cs_u)$ only. Moreover, by definition of CS-coupling, $t_{fr(u)}$ fires before t_{fu} does. Since $CS_N(w_1(u), w_2(r_t(u)), Q_u, cs_u)$ is a sound coupling, $t_{fr(u)}$ will eventually fire leading to a marking consisting of a marking of the workflow net $CS_N(w_1(u), w_2(r_t(u)), Q_u, cs_u)$ plus a one-token marking $[r_p(q)]$, where $r_p(q)$ is the output place of $t_{fr(u)}$ (type IV).

In the marking of type IV the following things can happen. First, the termination of the workflow $w_1(u)$ after the firing of some transition sequence. After that we obtain a marking $[q, r(q)]$ of type I. (Here we use the fact that the coupling is isomorphic.) And second, if the t-workflow net with input place $r(q)$ served as clients in the corresponding CS-couplings, they are enabled by a token in $r(q)$ and may fire. Let a transition $t_{ir(w)}$ fire consuming a token from $r(q)$. Then we obtain a marking $M_1+M_2+M_3$ where $M_1 \in \mathsf{bag}.P_{w_1(u)}$, $M_2 \in \mathsf{bag}.P_{w_2(r(w))}$, and $M_3 = [q_{fu}, q_{iw}]+M'_3$ with

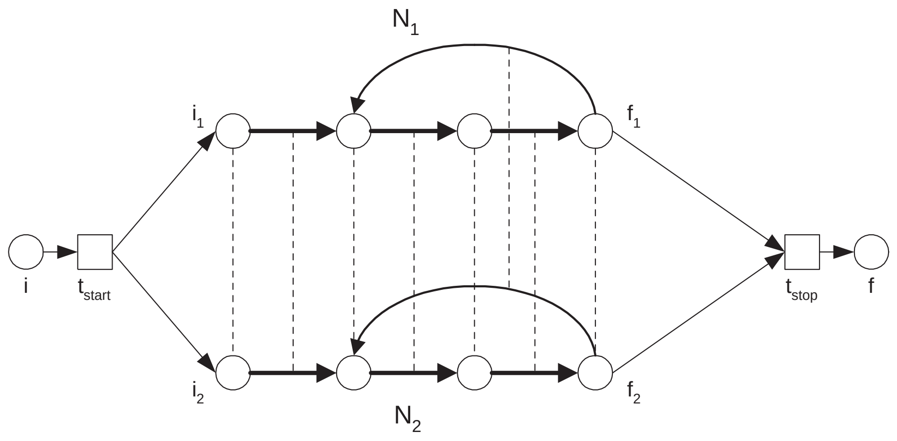


Fig. 1.22. CS-coupling

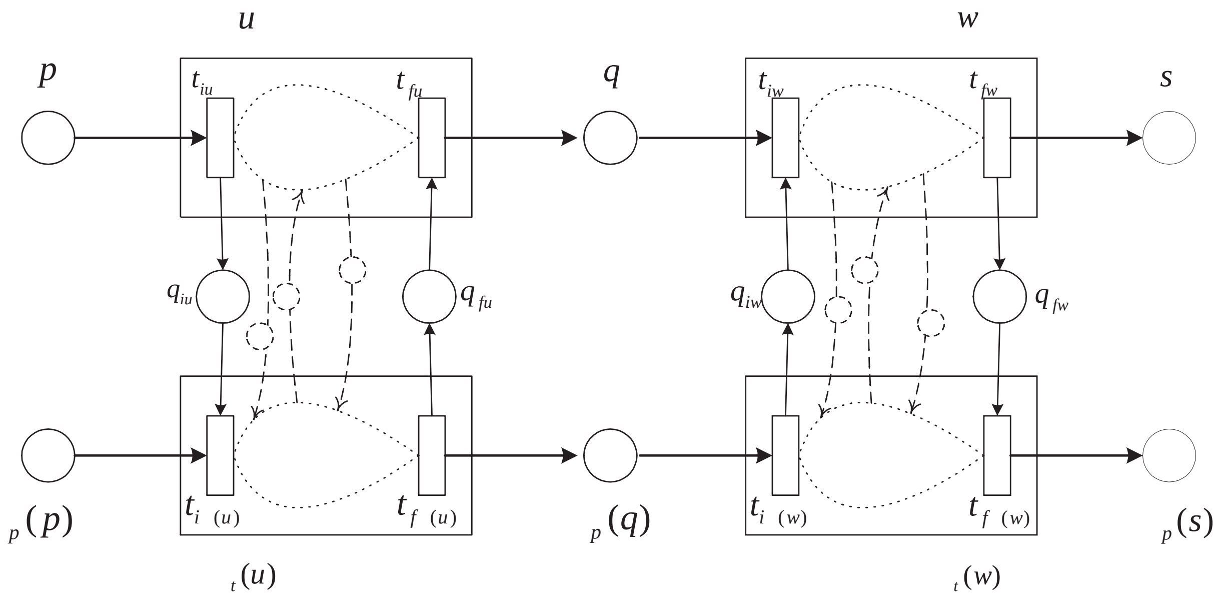


Fig. 1.23. Scheme for Theorem 7.27

$$\begin{tikzcd}
type1 \arrow[r] & type2 \arrow[r] \arrow[loop above] & type3 \arrow[r] \arrow[loop above] & type4 \arrow[r] \arrow[loop above] \arrow[lll, bend left] & type5 \arrow[loop above] \arrow[lll, bend right]
\end{tikzcd}$$

Fig. 1.24. Scheme for Theorem 7.27

$M'_3 \in \mathsf{bag}.((Q_u \setminus \{q_{iw}, q_{fw}\}) \cup (Q_w \setminus \{q_{iw}, q_{fw}\}))$ (type V). t-workflow $w_2(r(w))$ cannot terminate until t-workflow $w_1(w)$ puts a token into q_{fw}. Therefore, from a marking of type V we can obtain either a marking of type V, or, in case $w_1(u)$ terminates, a making of type II.

So we constructed an abstract transition system representing the behaviour of N (see Figure 1.24).

The only thing left to prove is that we can always reach the marking $[f_1, f_2]$ (see Fig. 1.22). It is easy to see that in a marking of any of the five types we can define a sequence of firings leading to f. Namely, for markings of type I we take a corresponding place p, for markings of types II,III, and IV — place q, and for markings of type V — place s. Since N_1, N_2 are sound isomorphic marked graphs, there is a path in N_1 leading from this chosen place to f_1 and the corresponding path through the renamed places and transitions of N_2 to f_2. We can always make our choices for firings in N according to this path. So the marking $[f_1, f_2]$ will be reached, and t_{stop} will eventually fire yielding the marking f, which means that N is sound. □

8 Timed Petri nets

Petri nets can be extended with a concept of *time*. Timed Petri nets can be used to investigate:

– the correctness of the desired functionality, absence of deadlocks, etc.

– performance of the modeled system, which allows to remove bottlenecks, predict mean waiting times and average throughput, compare different strategies.

In timed Petri nets each token gets a time stamp telling when this token can be used. We can specify either fixed or interval delays (i.e., random delays where values are taken from some interval) for transitions. Let $\mathbb{R}^+$ be the set of all non-negative real numbers. For the *time set* we will write $\mathbb{Q}$, which is normally either $\mathbb{R}^+$ or $\mathbb{N}$.

Definition 8.1 (timed Petri net). *A* timed Petri net *is a tuple $N = \langle P, T, F, \Delta_{min}, \Delta_{max} \rangle$, where:*

– *P and T are two disjoint non-empty finite sets of* places *and* transitions *respectively;*

– *$F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a* weight function*;*

– *$\Delta_{min}, \Delta_{max} : T \to \mathbb{Q}$ are functions for the* minimal *and the* maximal delays *of transitions, satisfying $\Delta_{min}(t) \leq \Delta_{max}(t)$ for any $t \in T$.*

Definition 8.2 (marking). *A* marking *M of a timed Petri net N is a bag over $P \times \mathbb{Q}$.*

Thus, a token is a pair (p, τ) where p stands for a place and τ for a time stamp. In order to extract a place or a time stamp out of the token we use a projection operation. So $\pi_1(p, t) = p$, and $\pi_2(p, t) = t$. We define $\pi_1(M)$ as a bag of places corresponding to the tokens of M.

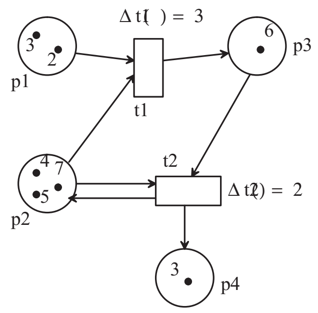


Fig. 1.25. A timed Petri net.

Now, consider a net N with a marking M and a transition t of this net. We will say that the transition is *potentially enabled* in M iff ${}^\bullet t \leq \pi_1(M)$. In case if $\pi_1(M) = {}^\bullet t$, it is clear that t can fire at the moment of time $\tau_1 = \max\limits_{o \in M} \pi_2(o)$, i.e. at the moment when the token with the maximal time stamp becomes available. However, it is possible that ${}^\bullet t < \pi_1(M)$, i.e., we could chose several different subsets X of M which can be involved into the firing of t. In this case, we chose the subset which allows the earliest firing of t. So we define

$$\tau(t, M) \quad = \quad \begin{cases} \min\ X : X \leq M \wedge \pi_1(X) = {}^\bullet t : \max\limits_{o \in X} \pi_2(o) & {}^\bullet t \leq \pi_1(M) \\ \infty & \text{else} \end{cases}$$

$\tau(t, M)$ gives us the earliest time moment when t can fire in M. We assign ∞ to the transitions that are not enabled in M to establish an order relation on the firing times. In case if $\tau(t, M) < \infty$, we also define

$$m(t, M) = \{X \mid X \leq M, \pi_1(X) = {}^\bullet t, \max_{o \in X} \pi_2(o) = \tau(t, M)\}.$$

$m(t, M)$ is a set of the markings that can be involved into the earliest firing of t.

Suppose we calculated $\tau(t, M)$ for every transition t of N. Now we need to decide which transition is to fire. We choose to pick the one with minimal firing time, which equals $\min\limits_{t \in T} \tau(t, M)$. If several transitions have this firing time, one of them is chosen nondeterministically. The result of the firing of t is the marking

$$M' = M - M_1 + M_2,$$

where $M_1 \in m(t, M)$ and M_2 is such that $\pi_1(M_2) = t^\bullet$ and the time stamps of all tokens in M_2 are $\tau(t, M) + \delta$, $\Delta_{min}(t) \leq \delta \leq \Delta_{max}(t)$.

Let's consider the simple net at Fig. 1.25, which is a net with fixed transitions delays ($\Delta_{min} = \Delta_{max} = \Delta$). The marking of this net is $M = [(p_1, 3), (p_1, 2), (p_2, 4), (p_2, 5), (p_2, 7), (p_3, 6), (p_4, 3)]$. Both transition t_1 and transition t_2 are enabled in M since $\pi_1(M) = [p_1^2, p_2^3, p_3, p_4]$, ${}^\bullet t_1 = [p_1, p_2]$, ${}^\bullet t_2 = [p_2, p_3]$, and thus ${}^\bullet t_1 \leq M$ and ${}^\bullet t_2 \leq M$.

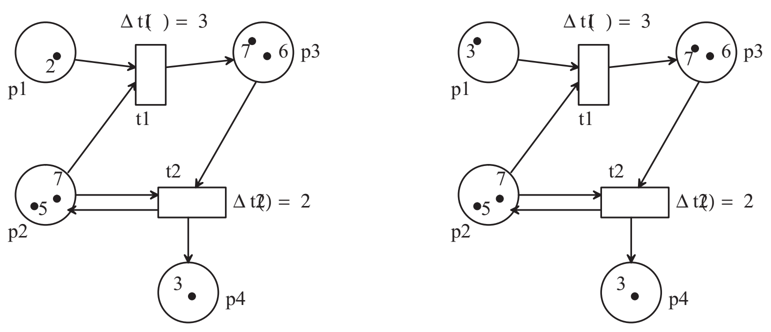


Fig. 1.26. Firing results.

Now we calculate $\tau(t_1, M)$. Submarkings of M that satisfy the condition $\pi_1(X) = {}^{\bullet}t$ are the markings consisting of one token in place p_1 and one in place p_2. We define maximal time stamp for every such a pair of tokens, and then chose the minimum of these maxima. So we get that $\tau(t_1, M) = 4$. This firing time corresponds to the markings $[(p_1, 2), (p_2, 4)]$ and $[(p_1, 3), (p_2, 4)]$ which form the set of markings $m(t, M)$.

Similarly, we calculate that $\tau(t_2, M) = 6$. So $\min\limits_{\tau \in T} \tau(t, M) = 4$ and the transition that is to fire is t_1. Its firing will consume either tokens $[(p_1, 2), (p_2, 4)]$ or tokens $[(p_1, 3), (p_2, 4)]$ and produce one token with the time delay $4 + 3 = 7$ for p_3. So the firing of t leads nondeterministically to one of the markings at Fig. 1.26.

Note that the delays of transitions do not assume that the firing of the transition takes some time. On the contrary, all the firings are instantaneous. The delay is a way to postpone the availability of the produced tokens. If, for instance, two customers come to a shop to develop their films on Tuesday, that means that both of them will get the developed films on Wednesday (Fig. 1.27). So, the transition fires there twice at the same moment of time, which is in this case Tuesday.

However, if we try to model the work of the film-developing machine under the assumption that the machine processes film sequentially, we would need a different way of the delay modelling. In Fig. 1.28, we introduce an auxiliary place *idle* indicating when the device is idle (if *idle* has a token) or busy (no token). This net can be reduced to the net in Fig. 1.29 showing the same behaviour.

9 Petri nets with hierarchy

Most software systems can be regarded as components, which are built from other (smaller) components. Every component has an interface stating how other components can communicate with it. It is only at the highest level that we have components that do not communicate with other components. These components can be seen as "ordinary" Petri nets. Component technology induces hierarchical modeling, since the components that we use are themselves built from (sub)components and so on. *Petri nets with hierarchy* is the formalism that we shall use for modeling components. Here we will limit ourselves to hierarchical classical nets; in the sections to come we will meet "colored" nets with hierarchy. These nets are used to define components.

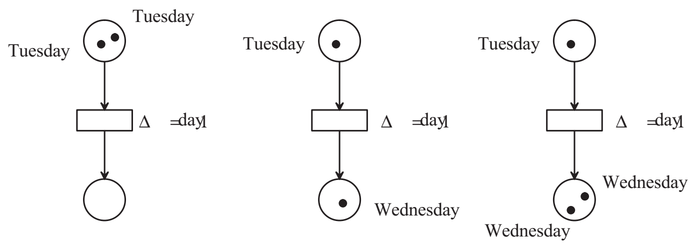


Fig. 1.27. Developing films 1.

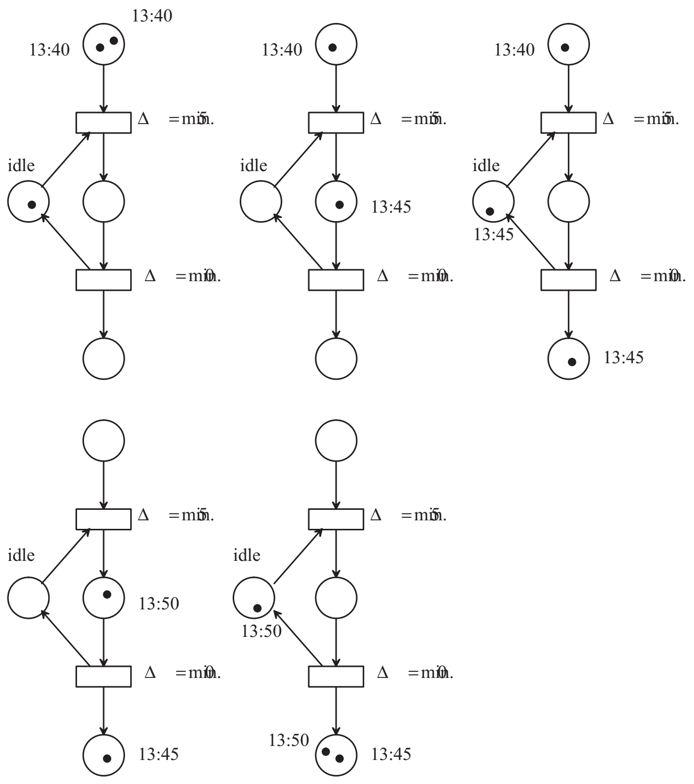


Fig. 1.28. Developing films 2.

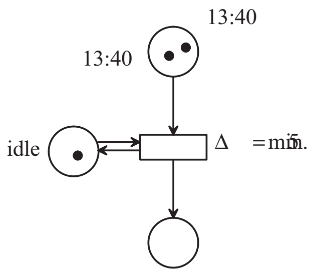


Fig. 1.29. Developing films 3.

The interface of a hierarchical net will consist of a set $E = e_1, \ldots, e_k$ of *pins*, which can be connected to places in the net's environment to allow communication. We shall distinguish input, output, store and inhibitor pins. A hierarchical net is constructed from subnets and places. Every subnet D within a hierarchical net C must be *installed* by connecting the pins from the interface of D to either a place p of C or a pin e of C. By connecting pins of different subnets to the same place of C, subnets can influence one another. By connecting a pin e of a subnet D of C to a pin of C, we say that e becomes a part of the interface of C.

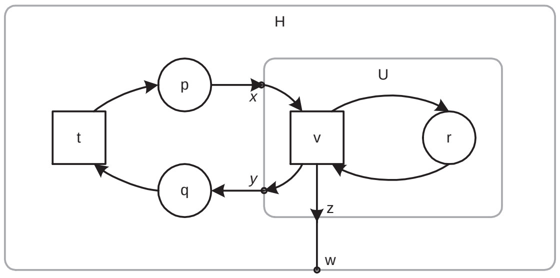


Fig. 1.30. A hierarchical net

A hierarchical net H in figure 1.30 contains the subnet U with pins x, y and z that has been installed into H by connecting pin x to place p, pin y to place q and pin z to outer pin w.

Now we give a formal definition for classical Petri nets with hierarchy. We presuppose a set Π of pins.

Definition 9.1 (hierarchical net). *A* hierarchical net *C is a pair (E, R), where $E \subseteq \Pi$ is its interface and a 4-tuple $R = \langle D, P, f, M \rangle$ is the* decomposition, *where D is a set of subnets, P a set of places, $f \in \bigcup(E_d \mid d \in D) \to (P \cup E_C)$* [2] *a function telling which pin of which subnet is connected to which place or pin of C and $M \in \mathsf{bag}..P$ an initial marking.*

We assume the sets of pins of the subnets and of C to be disjoint. In case we have pins with the same names, we rename them. Similarly, we assume that the set of places P and the sets of places of the subnets are disjoint.

The set E of a net's pins is decomposed into $E^i + E^o$, which are input and output pins respectively [3]. Note that a pin of a subnet can be connected to a pin of C only if the pins are of the same kind, so e.g. if $x \in E^o_d$ and $f(x) \in \Pi$, then $f(x) \in E^o_C$.

We call atomic hierarchical nets *processors*. A processor possesses an interface E and a decomposition consisting of empty sets, functions and bags only.

A hierarchical net C is called *flat* iff all its subnets D are processors. We can *unfold* non-flat nets by replacing nonatomic subnets by their decompositions. The unfolding operators Φ_d has as domain the nets $C = (E_C, (D, P, f, M))$ where $d \in D$. $\Phi_d(C)$ has the same interface E_C as C. The subnets

[2] By E_d, E_C we denote the interface (E part) of the subnet d, C respectively
[3] the $+$ operation is a union of disjoint sets

of $\Phi_d(C)$ are found by replacing d in D by the subnets of d. The places of $\Phi_d(C)$ are found by adding the places of d to P. The markings are obtained in a similar way. The connection function of $\Phi_d(C)$ is obtained by adding the connection function of d to f transferring the connections to E_d by means of f.

Definition 9.2. *Let $C = (E, (D, P, f, M))$ be a hierarchical net and let $d = (E_d, (D', P', f', M')) \in D$ be a nonatomic subnet. Then $\Phi_d(C)$ is the net $(E, (\Delta, P \cup P', g, M \cup M'))$, where $\Delta = (D \setminus \{d\}) \cup D'$ and $g \in \bigcup\{E_c \mid c \in \Delta\} \to (P \cup P' \cup E_C)$ defined as follows:*

$$g(e) \quad = \quad \begin{cases} f(e) & \text{if } \exists\, c \in D \setminus \{d\} :: e \in E_c \\ f'(e) & \text{if } \exists\, c \in D' :: e \in E_c \wedge f'(e) \in P' \\ f(f'(e)) & \text{if } \exists\, c \in D' :: e \in E_c \wedge f'(e) \in E_d \end{cases}$$

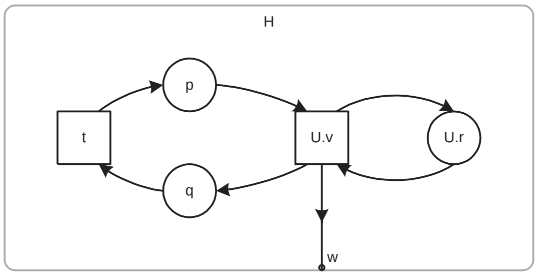


Fig. 1.31. Unfolding a hierarchical net

In figure 1.31 we see the unfolding of subnet U from figure 1.30. The indirect connection from $U.v$ to p via pin x has been made direct, and so has the connection from q to $U.v$ via y. The pin z of U has been connected to the pin w of $\Phi(H, U)$.

We define a hierarchical net C and its unfolding $\Phi_d(C)$ as equivalent: they have the same interface and behavior w.r.t. the interface, although they have different decompositions. A net C can be *flattened* iff there exists a sequence of unfoldings $\Phi_{d1}, \Phi_{d2}, \ldots, \Phi_{dn}$ such that the net $\Phi_{d1}(\Phi_{d2}(\ldots(\Phi_{dn}(C))\ldots))$ is flat. There are nets that cannot be flattened according to the definition above, e.g. *recursive* nets that contain (a copy of) themselves as a subnet.

A hierarchical net is called *closed* if has an empty interface. Closed flat hierarchical nets define marked Petri nets.

Definition 9.3. *Let $C = (\emptyset, \langle D, P, f, M\rangle)$ be a closed flat hierarchical net. Then C defines the marked Petri net (N_C, M), where $N_C = \langle P, D, \phi\rangle$, and ϕ is defined for $d \in D, p \in P$ by*
$\phi(d, p) = \mid \{e \mid e \in E_d^o \wedge f(e) = p\} \mid$ and $\phi(p, d) = \mid \{e \mid e \in E_d^i \wedge f(e) = p\} \mid$.

Closed hierarchical nets that can be flattened thus define marked Petri nets, that correspond to transition systems. If a net is not closed, we can use it to construct closed nets; if such a net can be flattened, such a closed hierarchical net can be flattened too and thus defines a Petri net.

10 Colored Petri Nets

In classical Petri nets, we consider flows of tangible and information objects, but we abstract from their actual properties. If we want to consider the latter, we have to extend our model, thus arriving at *colored* nets. In colored nets, tokens have values (called colors for historical reasons).

The token values may have several types (e.g. numbers, strings, booleans). We assume the existence of a set $\mathbb{T}$ of types (note that $\mathbb{T}$ is thus a set of sets of values). We will *type* places, i.e. connect types to them, so a place p typed with $\mathcal{A}$ will only contain tokens of type $\mathcal{A}$. Our set $\mathbb{T}$ also contains the one-point type $\mathbf{1}$, which has the value $*$ as its only element. A classical Petri net is a colored net with all places typed with $\mathbf{1}$.

We will start by recollecting some standard notions. Then we describe flat colored nets and finally hierarchical colored nets.

10.1 Preliminaries

We define the dependent product $\Pi\sigma$ of a function σ with a set of types as codomain.

Definition 10.1. *Let A be a collection of labels and $\sigma : A \to \mathbb{T}$ an assignment of types to labels. The* dependent product *$\Pi\sigma$ is defined by*

$$\Pi\sigma = \{f : A \to \bigcup \mathbb{T} \mid \forall a : a \in A : f.a \in \sigma.a\}$$

So, if σ has domain $\{a, b\}$ and $\sigma.a = \mathcal{A}, \sigma.b = \mathcal{B}$, then $\Pi\sigma = \{(a, x), (b, y) \mid x \in \mathcal{A} \wedge y \in \mathcal{B}\}$. The dependent product can be seen as the tuple (record) type constructor.

The *bagification function* β converts sequences to bags in the obvious way. For instance $\beta.\langle a, b, c, a, b, a\rangle = [a^3 b^2 c]$. Note the correspondence to the Parikh vector.

10.2 Colored flat nets

The first step is assigning types to places and transitions.
Let $\langle P, T, F\rangle$ be a Petri net and let $\tau : P \to \mathbb{T}$ be an assignment of types to places, then τ induces a typing of the transitions in T. This typing of the transitions will also be denoted by τ. If a transition t has an input place p of type $\mathcal{A}$ and an output place q of type $\mathcal{B}$, then we assign to t the type $\tau.t = \mathcal{A} \leftrightarrow \mathcal{B}$ which is the set of relations with domain $\mathcal{A}$ and codomain $\mathcal{B}$. If there are more input and output places, with weights attached to them, we take the weighted cartesian product of the inputs as domain and the weighted cartesian product of the outputs as codomain.

This notion is formalized as follows. The type $\tau.t$ of t is $\mathsf{domtype}.t \leftrightarrow \mathsf{codtype}.t$, with

$$\mathsf{domtype}.t = \Pi\delta \text{ and } \mathsf{codtype}.t = \Pi\gamma, \text{ where } \delta, \gamma : P \to \mathbb{T} \text{ are given by}$$
$$\delta.p = (\tau.p)^{F(p,t)} \text{ and } \gamma.p = (\tau.p)^{F(t,p)}.$$

So a transition is typed as relation between its input and output. If ${}^{\bullet}t = [a^2b]$, $t^{\bullet} = [ab^2]$, and $\tau.a = \mathcal{A}, \tau.b = \mathcal{B}$, then $\tau.t = \Pi\{(a, \mathcal{A}^2), (b, \mathcal{B})\} \leftrightarrow \Pi\{(a, \mathcal{A}), (b, \mathcal{B}^2)\}$, which is isomorphic to $(\mathcal{A}^2 \times \mathcal{B}) \leftrightarrow (\mathcal{A} \times \mathcal{B}^2)$. Note that the exponent 2 in $[a^2b]$ denotes the bag multiplicity and in $\mathcal{A}^2$ denotes the cartesian product of a set with itself.
(By defining $\mathcal{A}^0 = \mathbf{1}$, we do not need to limit ourselves to positive weights, since $\Pi(f \cup \{(a, \mathcal{A}^0)\})$ is isomorphic to Πf.)

For such a typed transition t a relation, say $\rho.t$ will be given that has the required type. I.e.

$$\mathsf{dom}.(\rho.t) \subseteq \mathsf{domtype}.t \text{ and } \mathsf{cod}.(\rho.t) \subseteq \mathsf{codtype}.t$$

Note that the relation $\rho.t$ does not have to be total: the domain of the relation may be a proper subset of the domain type and similarly for the codomain versus the codomain type.
We will use the mathematical language Z to specify the input-output behaviour of the relation $\rho.t$ that belongs to the transition t.

Definition 10.2 (simple flat colored net). *A* simple flat colored net *is a classical Petri net with a typing $\tau : P \to \mathbb{T}$ of the places and a type correct assignment ρ of relations to the transitions.*

A marking for a colored net, like the marking for classical nets, records the tokens in the places. A token in place, say, p posesses a value of type $\tau.p$. A place is marked with several tokens, so the marking of a single place is an element of $\mathsf{bag}.(\tau.p)$. Finally, the marking of the whole net can be modeled as an element of a dependent product.

Definition 10.3 (marking). *A* marking *for a simple flat colored net with typing $\tau : P \to \mathbb{T}$ is an element of $\Pi\ \mu$ where $\mu : P \to \mathbb{T}$ is defined by $\mu\ .p = \mathsf{bag}.(\tau.p)$.*

We thus have defined the state space; in order to define the state transitions, we extend our partial order $\leq$ on bags to $\Pi\ \mu$ in the obvious way: if $m, M \in \Pi\ \mu$, then $m \leq M$ iff for every $p \in \mathsf{dom}.M$ we have $m.p \leq M.p$. We define addition and subtraction on $\Pi\ \mu$ in the same way.

Essentially the firing rules in the colored Petri net would be defined as follows. We say that a transition $t \in T$ is enabled in a marking $M \in \Pi\ \mu$ of the net if the marking contains (in the sense of bags) an element $x \in \mathsf{dom}.(\rho.t)$, i.e. $x \leq M$.
For a pair $(x, y) \in \rho.t$, we have a successor marking $M' = M - x + y$ such that $M \xrightarrow{t} M'$. However, this definition schema is not correct. It is not even type-correct since for $x \in \mathsf{dom}.(\rho.t)$ we have that $x.p \in (\tau.p)^{F(p,t)}$ while $M.p \in \mathsf{bag}.(\tau.p)$. So the first is a tuple and the second is a bag. The conversion is made by a bagification step (see β above):

$$\beta.(x.p) \leq M.p \text{ for every } p \in P, \text{ i.e. } \beta \circ x \leq M.$$
$$M'.p = M.p - \beta.(x.p) + \beta.(y.p) \text{ for every } p \in P, \text{ i.e. } M' = M - \beta \circ x + \beta \circ y.$$

We thus arrived at the following definition.

Definition 10.4 (enabling and firing).
Transition t is enabled in marking M iff $(\exists\, x : x \in \mathsf{dom}.(\rho.t) : \beta \circ x \leq M)$.
The firing results in a marking $M' \in \{M - \beta \circ x + \beta \circ y \mid (x,y) \in \rho.t \text{ and } t \text{ enabled in } x\}$.
In other words $M \xrightarrow{t} M'$ iff $\beta \circ x + M' = M + \beta \circ y$ for x, y such that t is enabled in x and $(x,y) \in \rho.t$.

The role of inhibitor arcs is to prevent a transition from firing, thus further restricting the domain of input-output relation of the inhibited transition. Questions related to the inhibitor arcs in colored Petri nets will be treated in the section on the specification language.

In colored Petri nets the concept of special places called *stores* is introduced and widely used. A store is a place initially containing a single token. Transitions that consume a token from a store must necessarily produce a token back to this store. Both consumption and production must occur with weight 1. This concept makes little sense for classical nets, but for colored nets it can be compared to a (possibly structured) variable in programming.

10.3 Combining color and hierarchy

In the preceding section on hierarchy, the central concept was the hierarchical net. Places occurred inside hierarchical nets, while processors were the "leaves" of the hierarchy. In order to lift the coloring of Petri nets to the level of hierarchical Petri nets it is sufficient to define the coloring concepts for hierarchical nets in such a way that their flattening is colored according to the discussed coloring above.

We shall color or type the hierarchical nets inductively. A colored hierarchical net is a tuple $\langle E, \tau_E, \langle D, P, \tau_P, f, M\rangle\rangle$, where E is a set of pins, $\tau_E \in E \to \mathbb{T}$, D a set of colored subnets, P a set of places, $\tau_P \in P \to \mathbb{T}$, $f \in \bigcup(E_d \mid d \in D) \to (P \cup E_C)$ and $M \in \Pi\, \mu$, where $\mu : P \to \mathbb{T}$ is defined by $\mu\,.p = \mathsf{bag}.(\tau_P.p)$.

We assume again that all pins and places of a net and its subnets are disjoint, so that we can use the function τ as the disjoint union of all the τ_E's and τ_P's of a net and its subnets (and subsubnets etcetera).

The gluing function that joins the pins of subnets in D with pins in E and places in P should conform with the typing. So, for the type correctness, we should have $\tau.x = \tau.(f.x)$ for all pins x of subnets in D.

We extend the subsets of pins with inhibitor pins E^h and store pins E^s. So for any hierarchical net, its interface E can be written as $E = E^i + E^o + E^s + E^h$. The set of places P contains a subset P^s of stores. A store is a place that is always marked with a singleton bag, so for $s \in P^s$ we must have $M.s = [x]$ for some $x \in \tau.s$. If $f.e_d \in E_C$, the pins e_d and $f.e_d$ must be of the same kind. If $f.e_d \notin E_C$ and e_d is a store pin, then $f.e_d$ must be a store.

The basis of the coloring is the assignment of type correct relations to processors, the transitions of the corresponding flat Petri net. Again, processors are hierarchical nets with an empty decomposition.

Let $t = \langle E, \tau_E, \langle \emptyset, \emptyset, \emptyset, \emptyset, 0 \rangle \rangle$ be a processor. The set E of pins consists of four subsets, i.e. $E = E^i + E^o + E^s + E^h$. The inhibitor pins are irrelevant for the typing, and the store pins are bidirectional, so they count for the domain type as well as for the codomain type. Let δ be defined by $\delta.e = \tau_E.e$ for $e \in (E^i + E^s)$ and γ by $\gamma.e = \tau_E.e$ for $e \in (E^o + E^s)$, then $\mathsf{domtype}.t = \Pi\delta$ and $\mathsf{codtype}.t = \Pi\gamma$.

This settles the coloring of the hierarchical Petri nets. For the details of the definition, or rather specification, of the relations we refer to the section on Z.

2. Architecture

11 Architecture

We define an architecture of a system as a *set of related models* that describe the essentials of a system. The variety of models describes different components (parts) and different views (aspects) of the system. Components are building blocks: a system can be constructed by gluing together the components according to some rules. Views differ from components in the sense that they do not occur as a system on their own. To illustrate this we consider the building industry. Each floor of a building can be seen as a different component and the water supply system or the electricity system are examples of views. The latter systems are realized in parts that belong to components. In information systems the database (sub)system and user-interface (sub)system are examples of components and the structure of the communication and data are two examples of views. Usually we distinguish several standard views of a system: a business view, a functional view and a technical view. Each of these views can be split into parts. The technical view is split into a software view and a network view. So we divide the architecture of an information system into four levels:

1. *business architecture*. Business processes and the object classes that play a role considered from the perspective of the information system

2. *functional architecture*. The logical decomposition of the system into (logical) components and the assignment of processes and object classes to these components

3. *software architecture*. Software components that realize the functional architecture, e.g the database management system, the workflow engine and the connectivity software (middleware)

4. *network architecture*. A computer and communications network together with their operating systems

In case of a system that is implemented on a stand alone computer the network architecture is trivial: just one node. In our life cycle we recommend that all architecture is designed in the second phase. In some other life cycle models the business architecture is considered to be part of the requirements analysis.

Information systems are *discrete dynamic systems*, which means that these systems can be viewed as transition systems that have a (finite or countably infinite) state space and that make transitions through that state space at discrete points in time. We distinguish between *static* models and *dynamic* models. Static models describe the structure of a system or the state space (i.e. the set of states the systems might be in). Dynamic models describe the behavior of the system, i.e. the possible sequences of transitions a system can make.

The models an architecture consists of may be of different type. A verbal description can be a model, all sorts of diagrams can be models and a set of mathematical or logical formulas can be a model. We distinguish *informal* and *formal* models. Informal models have a (often sloppy) syntax and an intuitive semantics, like verbal models and many diagram techniques. Formal models have besides a syntax a formal defined semantics. The combination of a modelling syntax and a formal semantics is called a *modelling framework*. Mathematical and logical formulas are examples of formally models. Informal models are used in communications with stakeholders of a system (persons who have some say in the development of the system, e.g. potential users). Formal models are meant for systems engineers, programmers and for software tools that are able to analyze or interpret the models. There are modelling tools that can generate a simulation model from a description of the modelling framework. The simulation model is supposed to behave as the modelled system.

An architecture should have two important properties: *consistency* and *completeness*. "Consistency" means that all models are consistent internally and that they do not imply any contradiction or conflict when they are put together. Internal consistency should be defined in terms of the modelling framework. One simple but important consistency property is that a model is syntactically correct. A more advanced consistency property that is often required is that the systems are free of deadlocks. With “completeness” we mean that all models together provide sufficient information for *constructing* a system with the same functionality as the modelled system. A practical test for completeness is that from the set of models the external behavior of the system is fully defined and that a *simulation model* of the system can be generated and tested in the environment where the real system should operate. Note that in general the simulation model is not the same as the real system because the simulation model captures only the functional requirements of a system and not necessarily the non-functional requirements, such as the response times and scalability.

For the *verification* of the consistency and completeness we have several formal methods. Although the quality and scope of these methods are increasing rapidly there are many properties that can not be verified in a formal way. These properties should be checked by testing methods, i.e. by means of experiments with the system itself or with a model of the system. An example of a property that can not be verified by formal methods is "user friendliness" of a system. We call the checking of properties by means of testing: *validation*. It is well-known that the cost of correction of errors increases with the discovery in later stages of development. Therefore it is important to try to verify and validate a system as much as possible in the design stage. The architecture is a base for verification. In this course we focus on a particular way of verification: *correctness by construction*. In this approach we construct models using predefined patterns from which it is known that they imply correct behavior. This method is not a panacea for all verification issues. However a nice feature of this approach is that it does not require to search the state space but only the diagrams of the models. A system architecture is the base for the development of the software and for the maintenance of a system.

12 Components

The functional and software architecture are modelled by means of *components*. Components are subsystems that communicate with each other and with the environment. The functional components are conceptual or virtual in the sense that they only exist as mathematical models. In the software architecture the components represent real pieces of software. In theory there can be a one-to-one mapping between these different sets of components, however in practice the two component models are different and it is possible that one component of the functional model is realized by two or more components in the software architecture and that two components of the functional architecture are partly realized by the same software component. Although we focus here on the functional architecture the *component framework* is also applicable to the other area's of architecture.

The framework consists of two parts: the *conceptual framework*, i.e. the set of concepts we use in natural language to describe an architecture and the *formal framework* in which the concepts are translated. In general a formal framework consists of three parts: a *language* (textual or graphical) with a defined syntax, a *semantics* that gives a meaning for the language constructs in terms of mathematical or logical notions and a *theory*, i.e. a set of theorems that express properties of the models described in the language.

12.1 Conceptual framework

The framework consists of seven concepts.

– *Component*
  A *component* is an open system that is able to communicate with other components. Components are triggered by their environment to perform a *service*.

– *Workflow*
  For each kind of service a *workflow* is defined, i.e. a partially ordered set of *tasks* that has to be executed. The workflow is triggered from the environment of the component and it may trigger other components to perform other services as part of its own. So a component may divide the work it has to do into parts that can be "outsourced" to other components. The first workflow is called the "client" and the second one is called the "server" workflow.

– *Case*
  The work that is done in a workflow is called a "case", a "job" or a "transaction". We consider these terms as synonyms and we will choose the term *case*. So the function of a component is "case handling" . An example of a case is the manufacturing of a product in a factory or the fulfilment of a supply order in a warehouse.(In general we only consider the information aspects of such cases.) At some moment during the performance of a service the work in progress is in some stage, we call this the state of the case. Each case has its own, unique, *identity*. A case that belongs to a server workflow has its own identity and and has a method to determine the identity of the case in the client workflow that invoked it.

– *Task*
  A *task* is considered as an atomic unit of work in a workflow, which means that it has to be performed as one indivisible action. (Like the t-workflow expansion it is possible to expand a task into a non-atomic t-workflow). The smallest possible workflow consists of one task. A task transforms one or more inputs into zero or more outputs. Each task has an input-output relation, often specified by pre and post conditions. In many cases this is a functional relation so that the output is functionally dependent of the input. The input and output are *messages* or *objects*.

– *Object store*
  A component may have one or more *object stores* that store *objects*. A task may store, update, copy, delete or send an object to some other component. An object store stores objects from only one type. Objects have a unique *object identity*. There may be different object stores of the same type. A specific object resides in only one object store. (A copy of an object,with a different identity, may reside in another store.)

– *Subcomponent*
  A component may have *subcomponents*, so components can be *nested*. Subcomponents are useful if the workflow of a component can be divided into one or more specific sub-workflows that can be considered as "black boxes".

– *Consistency properties*
  There are several consistency properties. One is the requirement that each case terminates properly, which is expressed by the soundness property. Another one is that constraints that should hold for objects in the object stores remain valid after the workflows have finished.

12.2 Formal framework

We *model* components with two formal frameworks: *Petri nets* and *class models* belonging to UML (Universal Modelling Language. UML is not yet a formal framework according to our definition but it is a de facto standard). The Petri net framework is described in part I. The part of *class models* we are using is very similar to the entity relationship framework that is introduced in the ISO-course.

In this section we map the concepts to the elements of these formal frameworks. The formal framework also restricts the modelling process and the communication with the stakeholders.

– *Component net*
  Each *component* has a *component net*, i.e. a hierachical, colored Petri net with input and output pins that may be connected to places outside the component net. Component nets may be nested so inside a component there may be a *subcomponent* net (Fig. 2.1). The pins of a component net are inside connected to the pins of transitions within the component: input to input pins and output to output pins. We also have *inhibitor* pins and *store* pins. Pins may be connected to the pins of the same kind (i.e. input, output, inhibitor, object store) of the surrounding component. It is allowed to connect more than one pin to the same place or store in the surrounding component.

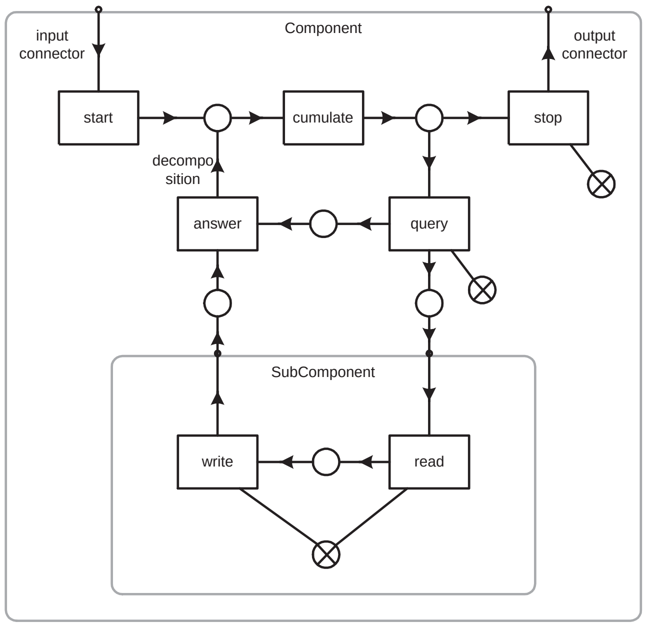


Fig. 2.1. Example of a component net

– *Workflow nets*
A component net has a workflow. The *workflow* of a component is modelled by a set of *t-workflow nets*: for each case type there is one specific t-workflow net. The *tasks* of a workflow are modelled by the *processors* of the t-workflow nets. Note that the smallest t-workflow net consists of one transition. Processors have input and output *pins*. Some of them are connected to places in the t-workflow net they belong to. The other pins may be connected to *interface places* that connect them to subcomponents. Alternatively the pins may be connected to the pins of the surrounding component. The t-workflow nets in one component are not connected to each other, but they may be connected to the same subcomponent or they may share object stores (see below).

– *Case tokens.*
A *case* is modelled as the set of tokens in a t-workflow net that is generated by one firing of the initial transition of the t-workflow net. We call them the *case tokens*. The state of a case is the marking of the t-workflow net. Remember that a case is always invoked by some external trigger or another case. We call this last case an ancestor of the first case. All cases carry the following properties:

– an identity

– a method to determine their ancestor

– a set of object identities for one or more classes.

– some data.

Transitions within a t-workflow net have built-in preconditions that guarantee that they only consume case tokens with the same identity as input and that they only produce case tokens with this identity. So cases are treated as if they move through the t-workflow net in isolation. The mechanism that takes care of the identity book keeping is not relevant for modelling, because we only have to know that a new case obtains a unique identity. Although not relevant for modelling it is nice to know at least one way to model such a mechanism as a "proof of concept". A way to deal with identities in the framework is to extend each t-workflow net with one extra place, the "identity" place (fig. 2.3), which is an input place for the initial transition and an output place for the final transition of the t-workflow net. The identity place contains all free identities and as soon as a case is started by an outside trigger, a free identity token is taken from the identity place and it is returned as soon as the case has finished. We assume the identity place has enough tokens and that each identity token is updated by the initial transition of the t-workflow net: so the identity consist of two parts: a fixed and unique label and a version number that indicates the number of times the identity token is used. Together they form the *case identity* and this is always a unique number. In this conceptual model we assume that each case token has an identity list containing the ancestor identities in chronological order. The initial transition of a workflow extends the identity list of the invoking token with the identity of the newly created case. During the processing of the workflow and the communication with its ancestor workflows the identity list remains unchanged. When the final transition of a t-workflow fires and it returns a token to the invoking workflow, the identity of the case belonging to this workflow is deleted from the entity list. In Fig. 2.2 we see a Message Sequence Chart (MSC) with three communicating t-workflows.

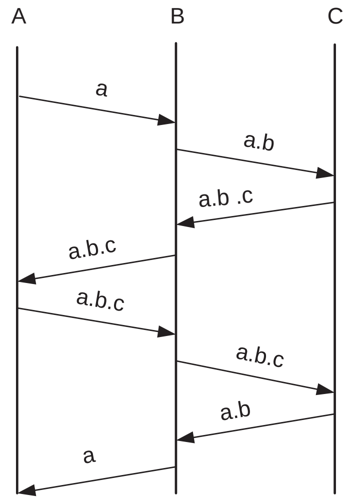


Fig. 2.2. Identity list

Typically we do not model this construction explicitly because it is assumed for all t-workflows.

– *Class model*
The *class model* is used to model the type of objects in an object store. A class model is made for the whole system that we wish to consider. In this framework we have classes that determine an object type. Objects have *attributes* and *methods* also called operations or functions. Each

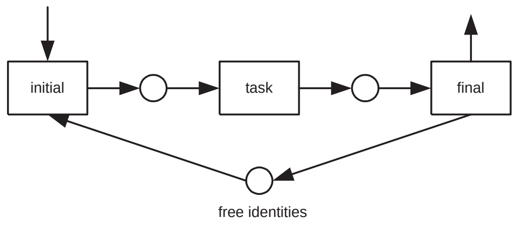


Fig. 2.3. Identities

attribute may have a *value* that belongs to a *datatype*. Classes may be associated with each other by means of *relationships*, each characterized by its *cardinality*. We use UML-notation (explained later). Note that we only use a subset of the class modelling notation of UML. We assume that all objects "know" the object identities of the objects they are associated with, so the objects have methods that manipulate and retrieve the object identities of the associated class. In the class model we may use *inheritance* relationships which facilitate the reuse of class definitions in modelling. For a class model we may define *constraints*: i.e. properties that each instance of the class should fulfill. In fact the cardinalities of the relationships are also constraints. An example of a constraint for two classes A with an attribute a and B with an attribute b that are associated to each other by a relationship r is: "the value a of an object x of class A is equal to the sum of the values b of objects y of class B that are related to x by r". Such a constraint is expressed in some language, for instance predicate logic.

– *Object store*
  We model object stores in a component net as special places, called *stores* (we use a circle with an x-cross in it as an icon for an object store.) An object store contains always a (possible empty) set of objects of one class. In Petri net terms this set of objects is considered as one token in the place. The class to which the objects of an object store belong may be involved in one or more relationships. We assume that the objects can access the objects they are related to by these relationships, provided that these other objects reside in the same component. So the t-workflow nets of a component can be linked by one or more common object stores. On the other hand an object store may be linked to more than one t-workflow net. Often we do not draw the connections of a t-workflow net to its "own" object store because it is assumed that the transitions have access to them. In most applications it is a requirement that t-workflow nets keep the constraints *invariant*. So during the execution of a t-workflow net a constraint may be violated, but if all t-workflow nets that an object is involved in have finished, the constraints should be valid again (this can be seen as a "weak invariant").

– *Scope*
  Components may be *nested*, but they are not allowed to overlap in another way. So an object store, a transition, a place or a subcomponent may belong to more than one component. We call the smallest component that includes this entity *scope* and the union of all components that include the entity the *telescope* of it. A transition may access an objects store if it is in the telescope of the object store. Similarly objects from one object store may *access* objects of another objects store that are in its telescope for retrieval purposes only, if and only if there exists a relationship between the classes of the object stores.

- *Consistency*
  We distinguish two important consistency properties.

  - The t-workflow nets (discarding the pins of transitions outside the t-workflow net) are *sound*. If they are connected to other (internal or external) components then the flattened Petri net should be sound.

  - The constraints that are required for the objects in the object stores should be valid if all t-workflows in the component they reside have finished. This is also called *transaction integrity* in the database world.

  Other consistency properties are often expressed with place invariants (for instance the book keeping of object identities) or with reachability, i.e. certain markings may never occur.

All the objects in the object stores represent *persistent* data, i.e. data that may still be in a component (in an object store) if all the cases (or t-worklow nets) have finished. The case tokens, messages and triggers that are modelled as data values and represent *volatile* data, i.e. data that is gone as soon as the cases have finished.
On the one hand a system is modelled as a (hierarchical) Petri net and on the other hand modelled as a class model. Both models are linked since all object stores belong to component and to a class. We model this link by means of the so-called *CRUD-matrix*. In this matrix the rows represent classes and the columns transitions. An entry in the matrix is either blank, which denotes that the transition has nothing to do with objects of the class or it contains one or more of the characters *C*, *R*, *U*, or *D*. These characters stand for: create, retrieve, update and delete one object of the class. We may also build a *CRUD-matrix* with classes replaced by object stores.

13 Road-map for Architecture

Here, we define the phases of the design of the architecture. Although our methods are applicable to all levels of architecture we concentrate on the business architecture and the functional architecture. So we cover a part of 'design of architecture' phase of the information systems life cycle. Besides the hierachical, timed and colored Petri nets and the UML-class model we use useful other techniques: the CRUD-matrix, Message Sequence Charts (MCS) and Z-schema's. This phase has the *requirements* as input. We assume the requirements in the form of a document describing the system. The architecture phase is further refined into the following five steps:

1. class model
2. life cycle model
3. component interaction model
4. specification of transitions and datatypes
5. verification of consistency properties

Although it looks as if this is a *sequential* process, it usually is an *iterative* process. In a next phase one finds errors or improvements for the former phases. In the end all parts have to be consistent, in particular the requirements document has to be updated frequently. In most practical cases the verbal description of the start differs fundamentally from the final version, which proves that the modelling activities give us new insights and ideas.
In the description of the modelling steps we describe *what* has to be done and we give some guidelines *how* things can be viewed or done.

13.1 Step 1: Class model

- We start with modelling the *dynamical classes* of a system, i.e. the objects that may change during the lifetime of the system. Objects can be stored in object stores. So the classes can be considered as the types of the object stores. In many applications it helps us to distinguish three kinds of objects and therefore their classes:

  - *Case* classes.
    A case object has a life cycle. A system or component is constructed to handle cases. The life time of a case can be a couple of milliseconds if the case is handled fully automatically, e.g. a billing transaction in a mobile telephone network. But it may take months if human intervention is needed, e.g. a claim handling of an insurance company.

  - *Resource* classes.
    Resource objects represent objects with a relatively long lifetime in the system. They often play a role in the handling of a case as a "resource" for the cases. Examples of resources are: employees, machines, computer systems, contracts, rulings, clients and suppliers.

  - *Event* classes.
    Event objects do not have a life cycle, they are created at some moment in time and they are used as message or trigger. They are created in a transition that is performed in a t-workflow net. Often we model triggers and messages just as data values, which means that they do not have an object identity and that they will not be stored explicitly in an object store.

  Note that all case tokens may refer to a case object or a resource object. In modern java development methods the same distinction is made. There these classes are called respectively: "session beans", "entity beans" and "session beans without a state".

- Next we model *relationships* between the classes and we will classify these relationships (for instance: "composed_of" and "uses relationships"). We indicate the *cardinality constraints* for the relationships.

- Now we express the other constraints. Note that each constraint introduces the obligation to prove that the t-workflow nets that manipulate these relationships keep these constraints invariant.

- We usually do not model *attributes, methods* or *non-dynamical classes* in this step. Non-dynamical classes are used for objects that play a role in computations but that do not change during the lifetime of the systems or the changes of which are left out of consideration, e.g. the postal code table. We define all these class properties when we specify the transitions that need them. We make one exception to this rule if attributes, methods or non-dynamical objects are essential in the formulation of the constraints.

13.2 Step 2: Life cycle model

– For each *case class* we define one or more t-workflow nets each one expressing a different work-flows the objects of the class may be involved. We assume that each transition in a t-workflow net may access the object store of that class.

– For each *resource class* we define one or more t-workflow nets. Each of these nets usually has a very simple structure consisting of one transition. So we have one for the creation of a new object, one for deletion of an object, one for retrieval of one object or the whole set of objects and one for updating an object. Slightly more complicated t-workflow nets are for updates where the update is done by some other component. In that situation the t-workflow net has two transitions. (Fig. 2.4) All these transitions have access to the object store.

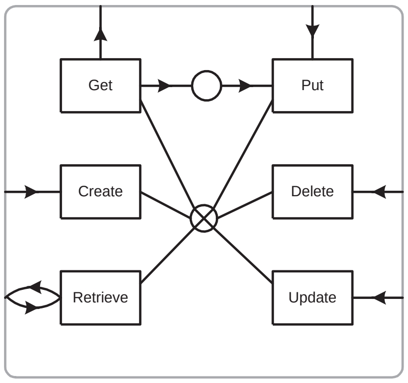


Fig. 2.4. Life cycle of a resource class

– Event classes do not have a life cycle.

– The tasks (i.e., processors) have a unique name and an informal description; transitions are not specified yet.

– If one class *inherits* from another then the life cycles may do it as well. Life cycle inheritance is defined for workflow nets by the p-expansion, the t-expansion or loop extension, i.e. life cycle *A* is a specialization of life cycle *B* if *A* can be derived from *B* by one of these transformations. At least soundness is a property that is inherited in this way. (There are more advanced forms of life cycle inheritance but they are out of the scope of this course).

– A CRUD matrix is produced: for each task, it is determined what objects are involved and in which way (C, R, U or D). Note that a task in the life cycle A of a class may use another class B.

– For each constraint of the class we have to prove that after the t-workflow nets have finished the constraint is valid.

Note that a t-workflow net is in fact a timed and colored workflow net. However we abstract from time and color in this phase.

13.3 Step 3: Component interaction model

– First, we define for each case and resource class a *component* consisting of a life cycle, an object store and an interface with the environment: messages created by tasks and messages triggering tasks.

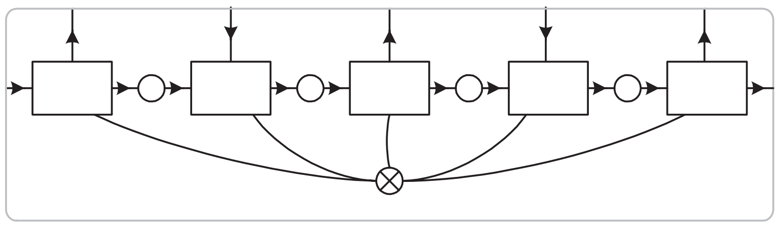


Fig. 2.5. Component for a dynamic class

– Next, we consider the *interaction* between tasks of different life cycles of the components. There are two possibilities: either two tasks have to *fuse* to become one task, or the tasks *communicate* by message passing. The first form is *synchronous* communication, the second form is *asynchronous*. The following rule has to be obeyed: if at least two communicating tasks of two life cycles communicate synchronously the life cycles are *fused* into one (Fig. 2.7) and become one component". In case they communicate asynchronously, they become two components and they are connected by means of places (Fig. 2.6). So one component may have more than one t-workflow net and more than one object store.

  The communication between two or more components can be defined or represented by means of Message Sequence Charts (Fig. 2.8). Note that a standard MSC only represents some fixed ordering of messages, while two components may have several different orders of message passing. So more than one MSC might be needed to define the communication between twe components. Components are always connected by means of interface places. (Fig. 2.9).

– For components and object stores we have some freedom in the *nesting*. A good rule is: "nest them as deep as possible". This means that they are hidden as much as possible and can be considered as black boxes. Consider for example (Fig. 2.10). Component D can only communicate (asynchronously) with the t-workflow net of C and D is hidden for the outside world. The same semantics would hold if D had been placed outside C, but then D would be visible for other components and it could be connected to them. For stores the level is determined by the telescope of the transitions that use it.

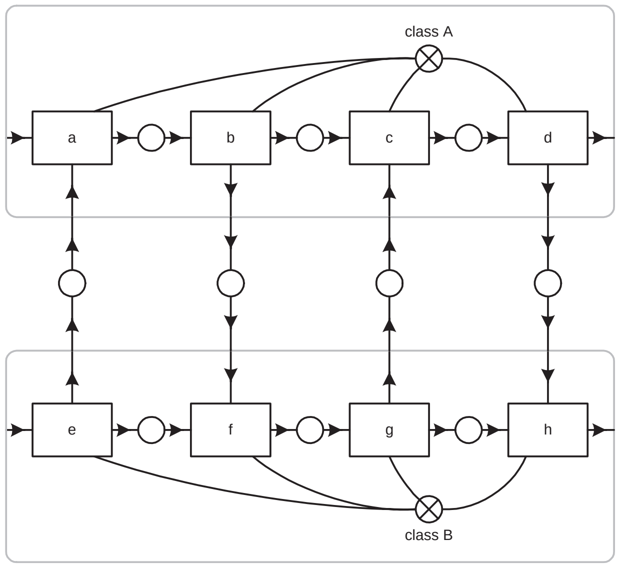


Fig. 2.6. Asynchronous-communication case

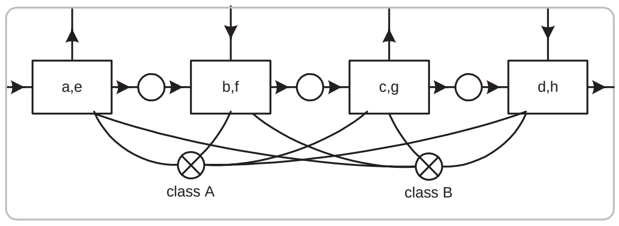


Fig. 2.7. Synchronous-communication case

– Finally a check has to be performed on the CRUD-matrix of step 3: "Are the tasks and the classes rightly coupled?".

13.4 Step 4: Specification of transitions and datatypes

If the components we need exist already then we do not have to specify them in much detail. Some form of specification is always necessary to find them and to check if a found component could fit. In practice we have often already an idea of available components. (Note that if we have found a suitable existing component we still have to configure it by setting parameters in a later stage.) Here we use a notation for specification of *data types* and for *input-output relations*. The first is necessary for places and pins, the latter for transitions. We will use the language Z, see the next section.

– The non-dynamical classes are specified.

– The attributes for the object classes are specified as well as their data types.

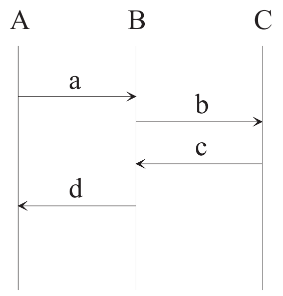


Fig. 2.8. A Message Sequence Chart

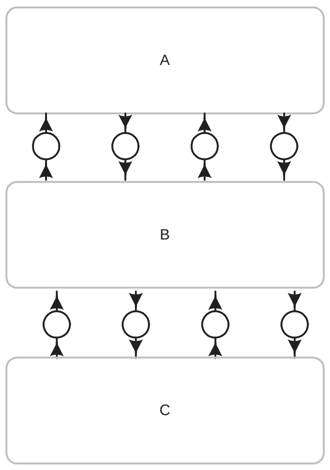


Fig. 2.9. Connected Components

- Specify the methods for classes.
- Data types for places within the t-workflow nets are specified.
- Data types for the pins and interface places are specified.

Now, we are ready to define the input-output relations of the transitions (cf. Part I, Colored Petri nets). All *pins* (input, output, inhibitor and store) of the transition have a unique identifier and a datatype. The input-output relation is specified by means of a *precondition* and a *postcondition* in which only the pin and store identifiers are free variables. The methods can be used in the pre and post conditions as functions. For example in Fig. 2.11 we have input pins x_1, x_2, x_3, output pins y_1, y_2 and store pin s.

$$\begin{aligned} \text{precondition:} &\quad f(x_1) = g(x_1, x_2) \wedge x_1 \in s \\ \text{postcondition:} &\quad y_1 = h(x_1, x_2) \wedge y_2 = k(x_2, x_3) \wedge s' = s \cup \{x_1, x_3\} \end{aligned}$$

In this example f, g, h and k are arbitrary functions. A store identifier occurs in two forms: one with and one without prime (cf. s and s'). The one with the prime denotes the old state of the store

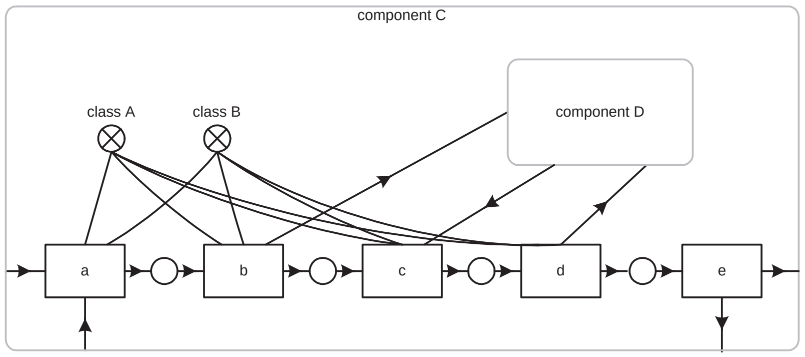


Fig. 2.10. Nested Components

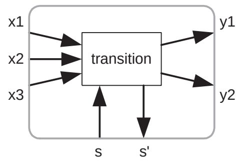


Fig. 2.11. Specification of a transition

and primed one denotes the new state. We refer to attributes of objects or tuples using the dot notation, so reference to the attribute *a* of object *s* is done by *s.a*.

13.5 Step 5: Verification of consistency

Verification of soundness can be proved by analyzing the reachability graph of the t-workflow nets. There are software tools for this check (cf. Woflan). Another approach is based on "correctness by construction": start with a t-workflow net that is known to be sound and use construction rules for which it is proved that they maintain soundness.
Verifying the invariance of constraints on the objects can sometimes be proved using place or transition invariants. In other cases we have to transform predicates using the pre- and post-conditions of the transitions.

14 Specification in coloured nets, Z and UML

14.1 Introduction

Specification of Petri nets consists roughly of two phases, the architecture of the net and the description of the details in the actions and the typing thereof. In the architectural phase the conceptual static modelling has been performed, here we are merely concerned with the mapping of that abstract typing onto the elements in the Petri net model. Such a specification and typing should allow for verification both on the level of human insight and that of rigourous mathematics. This calls for a formal but clear language with sufficient automated support as well as notational and generic freedom. However, there is also a need for general acceptance and adherance to common practices. (Of course these requirements will turn out to necessitate compromises.)

The choice is made for the state-based formal specification language Z ([0]) as a vehicle for specification as finegrained coloured modelling. Z is a matured and popular notation that grew out of collaboration of Oxford University with several industrial partners, and it accomodates imperative, object-oriented and functional styles. (An ISO standard is being developed and a committee proposal is available on the web ([1])).

As may be expected from a device with an extensive installed base there are ample tools available for editing, type checking and even theorem proving, e.g. Z/EVES ([2]), the documentation of which was very useful for the content and the typesetting of this section.

In the next subsections we give a brief and necessarily incomplete introduction to Z. The intention is to show how we can put elements of Z into good use in the specification tasks we encounter. We do, however, take our freedom in the choice of elements to be presented and in the emphasis they will be given. The notational divergence from the standard is rather small. The reader is encouraged to browse through some books on Z (e.g. [3], [4]) and to find and use Z-tools (e.g. [2])to get aquainted to the precision and possibilities of full-fledged Z.

14.2 Z

The specification language Z (pronounciation “zed”) is based on the Zermelo-Fränkel set theory and first order predicate logic. The focus of Z is on modelling systems based on (values of) state variables and the changes thereof. In fact, with the models and the operations on them, abstract data types and/or objects are described. The language also allows for purely functional descriptions (without states).

The systems and their operations are both described as a constrained collection of records (or a subset of a dependent product), so there is not really a difference between a system and an operation. For the system descriptions Z uses the notion of schema.

In order to express the records, some kind of typing is necessary, while the description of the constraints asks for expressions in terms of the members of the types and the operations on those elements. This means that in Z types and members thereof as well as operations on those types are defined by way of type definitions, set constructions and axiomatic definitions. Types and sets are closely related in Z and the difference is not really relevant for our purposes other than the difference in role. The following rule of thumb (though formally not correct) suffices: A set is a type when it is used in declarations of variables (and functions) and a type is a set when used as an “object” in value expressions.

In the sections to follow we will discuss the types and sets and their constructions, the expressions and operations on members within the types, the axiomatic definitions and the schemata. We will apply them to the specification of transitions or processors in coloured Petri-nets.

14.3 Type definitions

The only type built in in Z is the integer type $\mathbb{Z}$. In addition there are four ways to introduce new types. Here we shall discuss three of them and hint at the forth way that will be treated further in the subsection 14.5 on schemata.

Given type. New types may be introduced by just declaring the names of the new types in a list, the given type declaration (which is a simpel schema with only declarations of types).

$$[GNAME] \text{ or, for a bunch of types, } [GNAME_0, ..., GNAME_n]$$

Those newly defined types may be viewed as conceptual, abstract types without any reference to construction of the new type from old ones and without any clue on the shape of its members.

Basic constructions. There are several basic constructions for types. Since we hardly distinguish between types and sets we defer this discussion to the discussion of constructions within set theory. But it may be worth while to note that types are closed under cartesian product and powerset constructions.

Free type. A new type may be constructed as a disjoint sum of types each of which is embedded in the new type by a constructor, similar to the data definition in Haskell, for example

$$FNAME ::= constr_0 \mid constr_1 \langle\langle X \rangle\rangle \mid constr_2 \langle\langle Y \times Z \rangle\rangle$$

which defines the type *FNAME* as the disjoint sum of the types **1** (which is, as usual, suppressed in the $constr_0$-part of the right hand side), X and $Y \times Z$, using the constructors $constr_i$ as embeddings. This allows for the introduction of enumerated types like

$$BOOL ::= false \mid true$$

as well as inductive types. Recursion is allowed, so that for instance binary trees over A may be defined by

$$ATree ::= empty \mid leaf \langle\langle A \rangle\rangle \mid fork \langle\langle ATree \times ATree \rangle\rangle$$

Set constructions. The flexibility of Z depends on the flexibility of its users. All kinds of standard sets from mathematics may be used, like $\mathbb{N}$, $\mathbb{R}$, $\mathbb{B}$, *String* and *Char*. They may be considered to be known, but tools like Z/EVES do not always support them.
Type (set) constructors allow for other types (sets) to be introduced via set theoretic expressions. The following table gives the most important set constructions in Z without further ado.

name	shape
Empty set	$\emptyset$
Subrange	$m \mathrel{..} n$
Set comprehension	$\{n : \mathbb{N} \mid n < 481 \bullet 3 * n\}$
Set enumeration	$\{t_0, ..., t_n\}$
Set union, intersection and difference	$X \cup Y$, $X \cap Y$ and $X \setminus Y$
Power set	$\mathbb{P}\, X$
Finite subsets	$\mathbb{F}\, X$
Cartesian product	$X_0 \times ... \times X_n$
Relations	$X \leftrightarrow Y$
Total (partial) functions	$X \rightarrow Y$ $(X \nrightarrow Y)$
Total (partial) injective functions	$X \rightarrowtail Y$ $(X \rightarrowtail\!\!\!\!\!\!\!\;\;\nrightarrow Y)$
Finite sequences	seq.X
Bags	bag.X

The collection of bags over X may also considered to be the function space $X \nrightarrow \mathbb{N}$.

Record type. The dependent product (or the record with named fields) does exist in Z, but the definition uses a schema which also allows for restrictions on the tuples (or records). Discussion of those types will be deferred to a future subsection on schemata.
The following notation will be used as a shortcut

$$[label_0 : T_0;\ ...;\ label_n : T_n] \text{ with elements}$$
$$[label_0 \mapsto t_0, ..., label_n \mapsto t_n] \text{ provided } \forall\, i : \mathbb{N} \mid 0 \leq i \leq n \bullet t_i \in T_i$$

Note that the notation for elements of record types uses brackets instead of the curly braces that might have been expected (from the function notation to come). This stresses the fact that the images of the labels do not belong to one image type (the image type depends on the label: "dependent product").
The usual cartesian product is a record type too with default label names chosen to be a prefix of the positive naturals, while the ordinary parenthesis for tuples are used. I.e. (x, y) is just $[1 \mapsto x, 2 \mapsto y]$.

Note that the bracket notation above may be easy but it is **not** the Z standard. In Z (and thus in Z/EVES) the following delimiters are used $\langle\!|$ and $|\!\rangle$.

Type synonyms. Finally there is a set (or type) synonym construction in order to introduce shorthands or suggestive nomenclature, e.g.

$$\text{SHORTSUGGESTIVENAMEFORPRIMES} == \{k : \mathbb{N};\ l : \mathbb{N} \mid 1 < k \land 1 < l \bullet k * l\}$$

Be careful with meaningful identifiers!

14.4 Operations and expressions on elements

Most of the usual operations and expressions in mathematics are available in Z, but the notations may differ. In this section we mention only a few typical conceptional and notational elements, the major part of standard elements is considered understood.

- The logical and arithmatical "language" is standard with the following slight modification for the notation of the quantifications: a quantor is followed by a declaration which is a list of typed dummies, separated by semi-colons, closed with a bar; between the bar and the bullet we find the domain predicate and finally to the right of the bullet the term to be quantified is placed:

  $$(\exists\, x : X;\ y : Y \mid x \leq y \bullet x * y > 481) \text{ and } (\forall\, z : \mathbb{N} \mid \bullet\ z \text{mod} 2 = 0)$$

  So the bar and the bullet replace the two colons in the quantified expressions in predicate calculus. In particular, the domain predicate is omitted if it is understood or true as in the rightmost example.
  There are two shortcuts that are used often: if the domain predicate is true also the bar is removed and if the term is just the dummy (mostly in set expressions), the bullet and the term are removed as in:

  $$\{(x, y) \in X \times Y \mid x \leq y\ \wedge\ x * y > 481\} \text{ and } (\forall\, z : \mathbb{N} \bullet z \text{mod} 2 = 0)$$

  In the sequel we shall stick to the ternary format for quantifications.

- The λ-term as a means to express functions is considered to be a quantification too and follows the ternary notation and the shortcuts, like

  $$(\lambda\, x : \mathbb{Z};\ y : \mathbb{Z} \mid x \geq 0 \wedge y \geq 0 \bullet (x, y)) \text{ and } (\lambda\, x : \mathbb{N};\ y : \mathbb{N} \bullet (x, y))$$

  for the pair former on naturals.

- For ordered pairs (x, y) an alternative notation is $x \mapsto y$ which is called a *maplet*.

- Projections on cartesian products to certain factors are denoted by $\pi_{factorlabel(s)}$, e.g.

  $$\pi_{2,3}(a, b, c) = (b, c) \text{ and } \pi_{linne}[linne \mapsto molinae, eng \mapsto greencheek] = molinae$$

  The usual record notation is also allowed as in

  $$(a, b, c).1 = a \text{ and } [linne \mapsto molinae, eng \mapsto greencheek].linne = molinae$$

- Function application of f to n may also be denoted by $f\ n$.

- Functions may be denoted by enumerating their constituing pairs as maplets.

$$\{arg_0 \mapsto x_0, \ldots, arg_n \mapsto x_n\}$$

Note that the ordered pairs and maplet notation does not correspond to the common direction in application. The maplet has its argument on the left hand side and the result on the right, while function application usually is denoted with the argument on the right hand side of the function symbol.

- For $f, g : X \rightarrowtriangle Y$ the partial function $f \oplus g$ (f overridden by g) is defined by

$$\mathrm{dom}(f \oplus g) = \mathrm{dom} f \cup \mathrm{dom}\, g \text{ and } (f \oplus g)(a) = \textsf{ if } a \in \mathrm{dom}\, g \textsf{ then } g(a) \textsf{ else } f(a) \textsf{ fi}$$

- Relations are directed with the domain on the left and the codomain on the right (like the maplets for functions). The set of values in the codomain that may result from application of the relation to domain elements for which the relation is defined is called the range of the relation. For $R : A \leftrightarrow B$

$$\mathrm{dom} R = \{a : A \mid (\exists\, b : B \mid \bullet\ a\underline{R}b) \bullet a\} \text{ and } \mathrm{ran} R = \{b : B \mid (\exists\, a : A \mid \bullet\ a\underline{R}b) \bullet b\}$$

Note that we underline the relation if we use it in an infix notation. We might also have used the set notation $(a, b) \in R$ in stead of $a\underline{R}b$.

- Relational composition uses the semi-colon $R \fatsemi S$ in the usual meaning, don't use the functional composition symbol!

- The relational converse of R is denoted by $R^{\sim}$.

- The domain and range restrictions of relation R to part X of the domain type and to part Y of the codomain type respectively are

$$X \lhd R = \{r : R \mid r.1 \in X \bullet r\} \text{ and } R \rhd Y = \{r : R \mid r.2 \in Y \bullet r\}$$

Note that the type of a restriction is still the type of the original relation.

- The relational image is defined by $R(\!| S |\!) = \{r : R \mid r.1 \in S \bullet r.2\}$

- Sequences are enumerated between "angles" as in $\langle 4, 8, 1 \rangle$ or in $\langle\ \rangle$.

- Catenation of s and t is denoted by $s \frown t$. Moreover a repeated catenation is provided by $\frown/\, s$ which catenates all members of the sequence s of sequences.

- The proposed standard defines sequences to be functions on $\mathbb{N}$ with a domain starting from 1. So $head(s) = s(1)$ and $tail(s)(n) = s(n+1)$. On the other side of the sequence we have $last(s) = s(\#s)$ and $front(s)(n) = s(n)$ while $\#front(s) = \#s - 1$.

- The empty bag over any set is denoted by $[\![\,]\!]$, membership and bag inclusion are given by "x in B" and $A \sqsubseteq B$, while the multiplicity of the occurrences is represented by the size as infix operator, $B \sharp x$.

– Bag union and bag difference are denoted by $A + B$ and $A - B$. (In the Z-standard $A \uplus B$ and $A \uminus B$ are proposed).

14.5 Schemata

Next to the mathematical language we encountered in the former subsections, Z has the schema as a characteristic notational construct. Even more, the schema plays a central role in the Z-concepts. Roughly, a schema consists of a declaration part and a predicate part separated by a horizontal line. We may break up the predicate in two parts and consider the schema in a ternary format, like we have for quantifications. The declarations and predicates are to be expressed in terms of the mathematical language treated before.
Schemata may be used to define states and actions and they can be combined to specify classes in an object-oriented setting. However, these are all special cases of the constrained record type definition the schema really stands for.

Axiomatic definition. The axiomatic definition is an anonymous schema which is used for the definition of variables (and constants, functions or relations) with explicit constraints. We won't bother with scope rules as we don't treat the general Z paragraphs anyway.
A typical example for constraining variables is

$$\begin{array}{|l}
year : YEAR \\
febday : \mathbb{N} \\
\hline
1 \leq febday \leq 28 + (0 \max (1 - year \bmod 4))
\end{array}$$

A newline denotes a conjunction, but in order to save space the lines in the declaration part may be glued into one line with a semi-colon as separator, while in the predicate part the usual logical conjunction is to be used to combine the lines.
A typical function definition is

$$\begin{array}{|l}
pred : \mathbb{N} \nrightarrow \mathbb{N} \\
\hline
\operatorname{dom} pred = \mathbb{N} \setminus \{0\} \\
\forall n : \mathbb{N} \mid n \in \operatorname{dom} pred \bullet pred(n) = n - 1
\end{array}$$

State schema. The state schema is a named schema. It models a system consisting of state variables that may have certain values. It has a name, a declaration part for the typed state variables and a decription part for the constraint that is expressed via a predicate.

$$\begin{array}{|l|}
\hline
DMY \\
\hline
day : 1 \mathrel{..} 31 \\
month : 1 \mathrel{..} 12 \\
year : \mathbb{N} \\
\hline
month = 2 \Rightarrow 1 \leq day \leq 28 + (0 \max (1 - year \bmod 4)) \\
\mid 15 - 2 * month \mid \bmod 4 \neq 1 \Rightarrow day \neq 31 \\
\hline
\end{array}$$

What is meant with a DMY-system? In fact it is a member of a constrained record type (or dependent product); the state variables are the field names (or attributes or labels) and the constraint is given in terms of the "generic" field names. So for a member $x : DMY$ the constraint should be valid where the attributes *day*, *month* and *year* are replaced by the x-attributes $x.day$, $x.month$ and $x.year$. The non-mentioned member of DMY in the defining schema may also be referred to as θDMY which is pronounced as *the* DMY, or just *this* or *self* as in object orientation; thus the generic field name *day* stands for $\theta DMY.day$.

The state schema is the fourth way to define types in Z after given types, free types and the standard constructions in set theory. The state schema, is a kind of macro for instance definitions, and thus a vehicle for type definitions.
A horizontal notation, with an unmentioned name, is allowed too (but in most practical cases the lines are to short), as in:

$$[day : 1..31;\ month : \mathbb{N};\ year : \mathbb{N} \mid 1 \leq month \leq 12]$$

If the constraint in a schema is just true, the predicate part may be omitted, thus arriving at the notation for the record type we encountered in the section on type definitions.

$$[day : 1..31;\ month : \mathbb{N};\ year : \mathbb{N}]$$

Schema type refinement. Schemata may occur as types of declared fields in the declaration part of a schema, but they may also appear as a "supertype" in the declaration part, e.g. in the case below where a date is enriched with a name and year of birth in order to form a "birthday".

BIRTHDAY

DMY
$name : NAME$
$year : \mathbb{N}$

$year < \theta DMY.year$

The name clash is circumvented by using the self of DMY. The meaning of the importation of DMY is the following flattened schema with the unfolded occurrence of DMY:

FLATBIRTHDAY

$Dday : 1\,..\,31;\ \ Dmonth : 1\,..\,12;\ \ Dyear : \mathbb{N}$
$name : NAME;\ \ year : \mathbb{N}$

$Dmonth = 2 \Rightarrow 1 \leq Dday \leq 28 + (0 \max (1 - Dyear \bmod 4))$
$\mid 15 - 2 * Dmonth \mid \bmod 4 \neq 1 \Rightarrow Dday \neq 31$
$year < Dyear$

The name clash is resolved by renaming the fields of the flattened schema, a kind of α-conversion.

Operation schemata. Operations change state variables and/or construct output based on the values of the state variables and inputs to start with. In order to take care of the two values of the state variables (initial and final) two instantiations of the system have to be declared. By convention the result instantiation has the same name as the initial one but it is marked with a prime ($'$). There are also conventions that tell inputs from outputs: an input variable is postfixed with a question mark (?), while an output is postfixed with an exclamation mark (!).

changemonth

$x, x' : DMY$
$m? : \mathbb{N}$
$oldnew! : \mathbb{N} \times \mathbb{N}$

$1 \leq m? \leq 12$
$x'.month = m? \wedge x'.day = x.day \wedge x'.year = x.year$
$oldnew! = (x.month, m?)$

A few remarks are in order here:

- The condition on $m?$ in the first line of the predicate part is a precondition for the execution of *changemonth*. The rest of the predicate part may be viewed as the postcondition of the execution. Just for clarity's sake we allow for an extra field in the operation schemata: the precondition field, which as a domain predicate comes in between the declaration part and the postcondition part (the term).
  This is **not** standard in Z and cannot be found in Z/EVES. It is only used by us to stress the role of the preconditions. To turn it back in ordinary Z it suffices to remove the horizontal line between the precondition part and the predicate part.

- Instead of the declaration of variables for the states to be changed, we may import the system DMY twice, where the second is primed.

So the above schema may also be given by:

changemonth

DMY, DMY'
$m? : \mathbb{N}; \quad oldnew! : \mathbb{N} \times \mathbb{N}$

$1 \leq m? \leq 12$

$day' = day \wedge month' = m? \wedge year' = year \wedge oldnew! = (month, m?)$

Instead of explicitely importing two instantiations for the "before" and the "after" states, we may indicate the variability of the state variables by prefixing (the instantiation of) the system with Δ. If the state variables are only inspected and not altered, this is mentioned by prefixing (the instantiation of) the system with Ξ, in that case the equalities between primed and unprimed components are not mentioned anymore, e.g.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{changemonthandyear}} \\
\hline
\Delta DMY \\
\Xi x : DMY \\
m? : \mathbb{N};\quad oldnew! : \mathbb{N} \times \mathbb{N} \\
\hline
1 \leq m? \leq 12 \\
\hline
month' = m? \wedge year' = x.year \wedge oldnew! = (month, m?) \\
\hline
\end{array}$$

In the above schema the name *DMY* is used in two different ways. In the second line it is used as a type in a declaration and x is an invariant member of that type. In the first line it is a system (which may be interpreted as an unnamed member of the type *DMY*), this system is allowed to be changed.

Finally, note that an operation schema not really differs from a state schema. Again a record type is defined in which the state is built from the before and after states of the systems involved and the inputs and outputs. The precondition gives the applicability of the operation as a constraint on the argument part of the state space, and the postcondition gives the action as a relation between the argument and the result part of the state space.
If the postcondition is deterministic, the operation schema defines a *functional* record type, i.e. the result and output fields in the record are functionally dependent on the argument and input fields.

14.6 Application to coloured Petri nets

The road-map for architecture mentions six steps that lead to the design of the architecture. In the step that takes care of the modelling of the so called *dynamical* classes, the attributes, the methods and the constraints are mentioned. In a later stage, the specification of transitions and datatypes, the *non-dynamical* classes are to be defined and the attributes, methods and constraints need to be specified and/or refined in order for them to be exploited in the definitions of the transitions. In the last stage the consistency properties need to be verified, which can only be done if the system is specified in a sufficiently formal way. Hence, first the application of the Z specification language will be in determining, naming and defining the types to be used in the dynamical classes as well as in the non-dynamical ones. This is carried through in the typing of the places and the pins of the components on all levels. Afterwards the methods need specification and the necessary variables are subjected to the appropriate constraints. Finally, all these elements are to be used in the specification of the transitions.

Types, attributes and methods. The non-dynamical types are mainly the types to be used in the attributes of dynamical classes, the parameters of the methods and the (possibly persistent) data stores. (Conceptual modelling also comprises the classes as types, they will not be considered here.)

The “given” type definitions and the “free” type definitions are used to lay the fundaments of the types to be used in the specification process. For the sake of clarity type synonyms may be used in order to distinguish between conceptually different types and to have shorthands for types with complex type expressions. It may be interesting to define complex record structures using the state schema as dependent product with contraint. This may very well apply to the definition of the store types.

The declaration of variables and (enumerated) relations and functions with their constraints, their preconditions and their defining specifications can be given by way of the axiomatic definition schemata.

$DAY ::= mon \mid tue \mid wed \mid thu \mid fri \mid sat \mid sun$

$$\begin{array}{|l}
numday : \mathbb{N} \leftrightarrow DAY \\
\hline
numday \;=\; \{0 \mapsto mon,\ 1 \mapsto tue,\ 2 \mapsto wed,\ 3 \mapsto thu \\
\qquad\qquad\quad 4 \mapsto fri,\ 5 \mapsto sat,\ 6 \mapsto sun\}
\end{array}$$

$$\begin{array}{|l}
prevworkday : DAY \nrightarrow DAY \\
\hline
\forall\, a \in DAY;\ x \in \mathbb{N} \mid x\ \underline{numday}\ a \wedge x \leq 4 \\
\qquad\qquad \bullet\ prevworkday(a) \in numday(\!|\{((x-1) \bmod 7) \min 4\}|\!)
\end{array}$$

(The complification of using a relation as an intermediate means to express the function is constructed just for the sake of illustration of expressions. This applies to all odd definitions and expressions, they are not to be interpreted as being bright, beautiful or otherwise amenable, just as not impossible!)
In the above function definition for *prevworkday* the definition per argument is given via a quantification. We could have given an enumeration too, and even a λ-expression is allowed. Moreover, we could have chosen to give the definition via the operation schema, as follows

$$\begin{array}{|l}
pred : \mathbb{N} \leftrightarrow \mathbb{N} \\
\hline
0\ \underline{pred}\ 4 \\
\forall\, x \in \mathbb{N} \mid x > 0 \bullet x\ \underline{pred}\ (x-1)
\end{array}$$

$$\begin{array}{|l}
\text{--- } prevworkday \text{ ---} \\
\Delta x : DAY \\
\hline
numday(\!|\{x\}|\!) \cap \{5,6\} = \emptyset \\
\hline
x\ \underline{numday^{\sim} \mathbin{;} pred \mathbin{;} numday}\ x' \\
\hline
\end{array}$$

A function definition that has a shape in between the state altering schema and the axiomatic definition uses inputs and outputs without a system declaration.

prevworkday

$$x? : DAY$$
$$y! : DAY$$

$$x? \; \underline{numday \fatsemi <} \; 5$$

$$y! \neq fri \lor 0 \in numday(\!| \{x?\} |\!)$$
$$y! = fri \lor (\forall m, n : \mathbb{N} \mid m \, \underline{numday} \, x? \land n \, \underline{numday} \, y! \bullet m + 1 = n)$$

Processors. Next to expressing types, constants, variables, relations and functions we will use Z extensively in defining the actions performed by the transitions in coloured Petri nets. Transitions will be specified by way of the operation schemata in a way similar to the schema example for the definition of *prevworkday*. In fact, it is an ordinary relation definition. The typing of the relation, like in the mentioned example, is implicit in the declaration part of the schema; even stronger, the order of the factors doesn't matter, which means that the typing always employs the dependent products.

Consider a processor $\langle E, \langle \emptyset, \emptyset, \emptyset, 0 \rangle \rangle$ (i.e. a hierarchical net with empty decomposition). As mentioned in the section on colour and hierarchy, the set E of outer pins determines the typing of the transition. The real situation is slightly more complex than sketched there. The stores may be divided into two sets, the fixed stores whose values are to be unchanged and the changeble ones. The fixed stores only contribute to the domain type, not to the codomain type (unless the invariance is taken to be a constraint). Inhibitors play a role in the defining expressions, to be more precise, they are used in the precondition, the condition for firing. Inhibitor pins are connected to places, and places usually provide tokens on an individual basis and this is reflected in the typing in the declaration part of the operator schema, where the type of a "place"-pin is the type of an individual token. However, the inhibitor uses all available tokens in the connected place! A typical use of an inhibitor is to prevent a transition from firing as long as certain values are present in the inhibiting place, or even as long as there are tokens at all in the inhibiting place. These are properties of the collection of all tokens in that place instead of properties of individual tokens as is the case for normal coloured token games. This causes a problem in the comparison of the typing of the pins with the typing of the Z-schema gouverning the transition relation. The inhibitor pin has the same type as the place it is connected to, say X, but the declaration of the pin in the schema is chosen to be of the corresponding set type $\mathbb{P}\,X$ (since nets are finite we might even have said $\mathbb{F}\,X$). This results in the following general form of the operation schema definition of the transition relation.

Let $i : I, o : O$ and $h : H$ be the typed input, output and inhibitor pins of the transition $TRANS$, in particular, $E^i = \{i\}, E^o = \{o\}$ and $E^h = \{h\}$. The store pins are divided in two : $sc : SC$ and $sf : SF$.

TRANS

$i? : I$
$o! : O$
$\Delta\, sc : SC$
$\Xi\, sf : SF$
$\Xi\, h : \mathbb{P}\, H$

Precondition in terms of $i?, sc, sf, h$

Expressions giving values for $o!, sc'$ but *not* for sf', h

14.7 Specification of components

We have treated the specification of processors. We shall now treat the specification of components. A component is specified by an UML class diagram and a hierarchical colored net. The class diagram defines objects classes (with their attributes and methods) and the relations between them. The net defines the places, object stores and subnets. Eventually the net can be flattened, resulting in a flat net with processors, places and stores.

The UML class diagram defines schemata defining classes and an axiomatic definition of the relations between classes. The names of the schemata are the class names, which must be unique. The names of the relations must be unique too. The class diagram lists a set of attribute names per class.

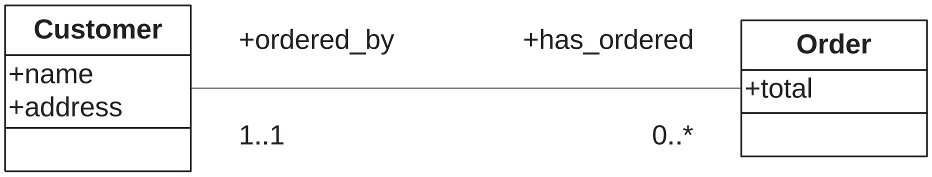


Fig. 2.12. A UML class model

In Figure 2.12, a class diagram is defined. This class diagram implicitly defines the following Z schemata and definitions.

Customer

$name : string$
$address : string$

Order

$total : \mathbb{N}$

$$\begin{array}{|l}
ordered\_by : Order \leftrightarrow Customer \\
has\_ordered : Customer \leftrightarrow Order \\
\hline
has\_ordered = ordered\_by^{\sim} \\
\forall x : Customer \mid\bullet\ 0 \leq has\_ordered(\!| \{x\} |\!) \leq \infty \\
\forall x : Order \mid\bullet\ 1 \leq ordered\_by(\!| \{x\} |\!) \leq 1
\end{array}$$

The two relations, *ordered_by* and *has_ordered* will be implemented by adding data about the orders made in *Customer* objects and/or data about the ordering customer in the *Order* objects. While specifying, we do not want to fix an implementation. Since the relations are inverses of one another, an update of one relation will automatically lead to an update of the other.

The processors in a component possess the implicit schemata and definitions from the class model. A processor is connected to input and output places and object stores. These object stores are of the type $\mathbb{P}\,.A$, with A some class name. The places may have any type, even a class. However, places typed with a class do not contain the true objects, which reside in the object stores. Instead, places may contain "value" copies of objects.

We present an example of a component definition. The component uses the class diagram in Figure 2.12, and thus the derived schemata and definitions. We add the object stores K for *Customer* and B for *Order* objects.

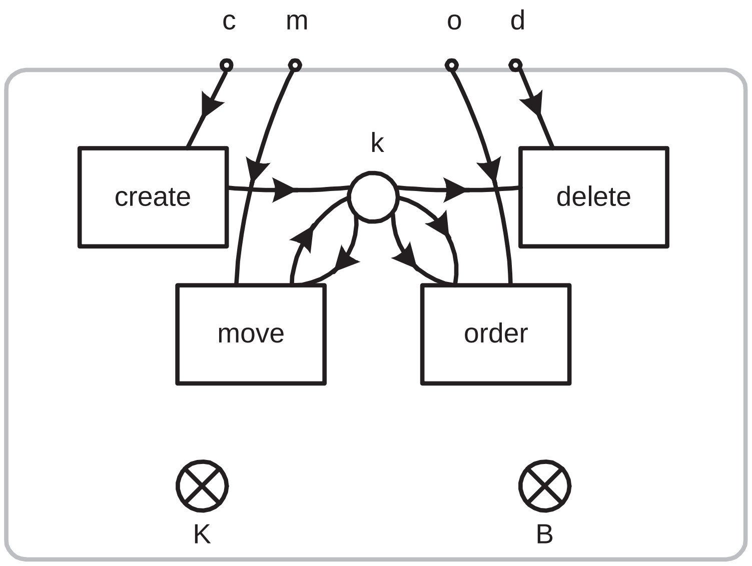


Fig. 2.13. A component

The component contains four processors; they can be specified as follows. The *create* processor receives customer information and creates a new customer, both as object and as token. Note that the equation $K' = K \cup \{k!\}$ causes the creation of an object in the object store K with *value* equal to $k!$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{create}} \\ \hline
c? : [\mathit{name} : \mathit{string}, \mathit{address} : \mathit{string}] \\
k! : \mathit{Customer} \\
\Delta K : \mathbb{P}\,\mathit{Customer} \\ \hline
k! = c? \\
K' = K \cup \{k!\} \\ \hline
\end{array}$$

The processor *move* receives a customer name and address and updates both a customer token and object, by altering the address to the given address for the customer with the given name. The equation $K' = K \setminus \{k?\} \cup \{k!\}$ denotes the *value* change occurring in the object store. This value change will be implemented by updating an object in K with value $k?$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{move}} \\ \hline
m? : [\mathit{name} : \mathit{string}, \mathit{address} : \mathit{string}] \\
k?, k! : \mathit{Customer} \\
\Delta K : \mathbb{P}\,\mathit{Customer} \\ \hline
k? \in K \\
k?.\mathit{name} = m?.\mathit{name} \\ \hline
k! = k? \oplus [\mathit{address} : m?.\mathit{address}] \\
K' = K \setminus \{k?\} \cup \{k!\} \\ \hline
\end{array}$$

The *order* processor receives a customer name and an amount ordered. It does not update the customer, but it creates a new order object. The *ordered_by* relation is augmented with a customer/order pair; the customer must have the given name and the order is the newly-created order. This relation update must be implemented by updating the concerned objects in the stores K and/or B. Note that we can update a relation iff we access object stores for both object classes. Since the processor *order* accesses both K and B, the relation *ordered_by* may be updated.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{order}} \\ \hline
o? : [\mathit{name} : \mathit{string}, \mathit{tot} : \mathbb{N}] \\
k?, k! : \mathit{Customer} \\
\Xi K : \mathbb{P}\,\mathit{Customer} \\
\Delta B : \mathbb{P}\,\mathit{Order} \\ \hline
k? \in K \\
k?.\mathit{name} = o?.\mathit{name} \\ \hline
n = [\mathit{tot} : o?.\mathit{tot}] \\
B' = B \cup \{n\} \\
\mathit{ordered\_by}' = \mathit{ordered\_by} \cup \{n \mapsto k?\} \\ \hline
\end{array}$$

The *delete* processor removes a customer with the given name, both as object and as token.

delete

$d? : [name : string]$
$k? : Customer$
$\Delta K : \mathbb{P}\, Customer$

$k? \in K$
$k?.name = d?.name$

$K' = K \setminus \{k?\}$

References

[0] The Z notation, http://www.afm.sbu.ac.uk/z/

[1] Formal specification - Z notation - syntax, type and semantics, Consensus working draft 2.6, http://www-users.cs.york.ac.uk/~ian/zstan/cd.html

[2] Documentation for Z/eves, Z/eves information: http://www.ora.on.ca/z-eves

[3] The way of Z, J. Jacky, Cambridge University Press 1997

[4] Formal object-oriented specification using object-Z, R. Duke and G. Rose, Cornerstones of computing, McMillan Press ltd, 2000