Life-cycle inheritance : a Petri-Net-Based approach

Eindhoven University of Technology
Department of Mathematics and Computing Science

Life-Cycle Inheritance
A Petri-Net-Based Approach

by

W.M.P. van der Aalst and T. Basten

96/06

ISSN 0926-4515

editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Report 96/06
Eindhoven, March 1996

Life-Cycle Inheritance

A Petri-Net-Based Approach

W.M.P. van der Aalst and T. Basten

Department of Mathematics and Computing Science
Eindhoven University of Technology, The Netherlands
email: {wsinwa,tbasten}@win.tue.nl

Abstract. Inheritance is one of the key issues of object-orientation. The inheritance mechanism allows for the definition of a subclass which inherits the features of a specific superclass. This means that methods and attributes defined for the superclass, are also available for objects of the subclass. Existing methods for object-oriented modeling and design, abstract from the dynamic behavior of objects when defining inheritance. Nevertheless, it would be useful to have a mechanism which allows for the inheritance of dynamic behavior. This paper describes a Petri-net-based approach to the formal specification and verification of this type of inheritance. We use Petri nets to specify the dynamics of an object class. The Petri-net formalism allows for a graphical representation of the life cycle of objects which belong to a specific object class. Four possible inheritance relations are defined. These inheritance relations can be verified automatically. Moreover, four powerful transformation rules which preserve specific inheritance relations are given.

Keywords: Object orientation, Petri nets, Inheritance, Object life cycle.

1 Introduction

Although object-oriented design is a relatively young practice, it is considered to be the most promising approach to software development. Within a few years the two leading object-oriented methodologies, OMT [13] and OOD [5], have conquered the world of software engineering. Both methodologies use state-transition diagrams for specifying the dynamic behavior of objects. Typically, for each object class, one state-transition diagram is specified. Such a state-transition diagram shows the state space of a class and the methods that cause a transition from one state to another. In this paper, we use Petri nets (See for example [12]) for specifying the dynamics of an object class. There are several reasons for using Petri nets. First of all, Petri nets provide a graphical description technique which is easy to understand and close to state-transition diagrams. Second, parallelism, concurrency and synchronization are easy to model in terms of a Petri net. Third, many techniques and software tools are available for the analysis of Petri nets. Finally, Petri nets have been extended with color, time and hierarchy [10, 11]. The extension with color allows for the modeling of object attributes and methods. The extension with time allows for the quantification of the dynamic behavior of an object. The hierarchy concept can be used to structure the dynamics of an object class.

In this paper, we use the term *object life cycle* to refer to a Petri net specifying the dynamics of an object class. Figure 1 shows two object life cycles. Object life cycle N_0 specifies the dynamics

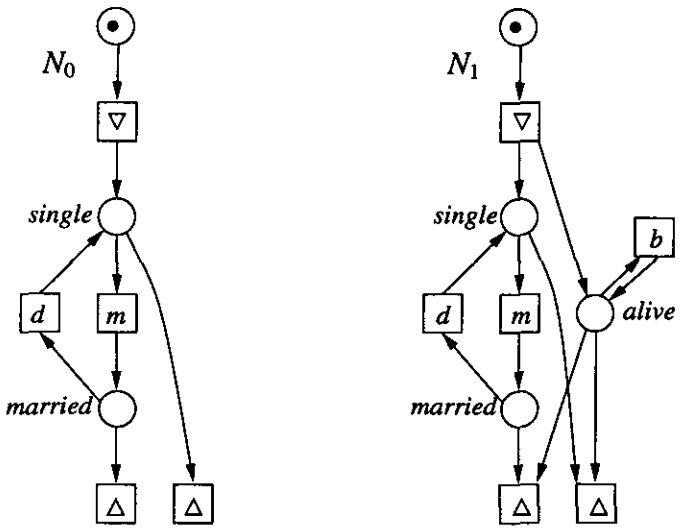

Figure 1: Two object life cycles.

of an object of the class *person*. The creation of an object is modeled by a transition with a ∇ label. Firing this transition corresponds to the birth of a person. Firing one of the two transitions with a Δ label results in the termination of the object, i.e., the death of a person. An object of the class *person* is either in state *single* or in state *married*. Each of these states corresponds to a place in the object life cycle N_0. Firing the transition with label m corresponds to the marriage of a person and results in the transfer of a token from *single* to *married*. Firing the transition with label d corresponds to the divorce of a person. As a result, a token is transferred from *married* to *single*. Object life cycle N_1 specifies the dynamics of the class *another_person*. Compared to the original object life cycle, the transition labeled b and the place *alive* have been added. Firing this additional transition corresponds to the birthday of a person. Note that in this object life cycle the state of a person is represented by two tokens. As a result, it is possible to have concurrency within the same object.

In general, the state of an object is represented by a configuration of tokens over places. Transitions represent state changes. The label of a transition refers to the *method* being executed when the transition fires. Note that a method is the implementation of an operation that can be executed for specific objects. There are three reserved method labels, ∇ (object creation), Δ (object termination) and τ (internal method). In Figure 1 there are no τ-labeled transitions. However, it turns out to be useful to distinguish internal methods from external methods. Internal methods can only be activated by the object itself. External methods can also be activated by other objects.

The two object life cycles shown in Figure 1 have a lot in common. In fact, N_1 comprizes N_0. Moreover, it appears that life cycle N_1 incorporates, or *inherits*, all properties of life cycle N_0 and adds its own unique properties. *Inheritance* is one of the key issues in object-orientation. Unfortunately, inheritance is often limited to sharing attributes and methods among object classes. Until now, a good concept for life-cycle inheritance was lacking (See for example [2]). Existing object-oriented methodologies such as OMT [13] and OOD [5] do not give a clear definition of inheritance with respect to the dynamics of an object class. In this paper, we tackle the problem of deciding whether the object life cycle of one class inherits the life cycle of another class.

In another paper ([4]), we have investigated this problem using a simple ACP-like process al-

gebra. In that paper, it is shown that *encapsulation* and *abstraction* turn out to be important concepts for the characterization of life-cycle inheritance. Based on these concepts, four inheritance relations have been defined. The process-algebraic characterization of life-cycle inheritance in [4] is rather straightforward because encapsulation and abstraction are well investigated and, in contrast to state-transition diagrams and Petri nets, states are not represented explicitly. Using the rich theory of ACP it is quite easy to show that each of the four inheritance relations has a number of desirable properties. Unfortunately, an algebraic characterization of inheritance is difficult to handle by people not familiar with process algebra. Most of these people prefer state-transition diagrams because of their graphical nature, simplicity and the fact that states are represented explicitly. These features do not apply to process algebra, but are essential to the success of existing object-oriented methodologies. Therefore, we resort to Petri nets for the specification of object life cycles. Petri nets provide a graphical formalism which is much closer to existing methodologies such as OMT and OOD.

In this paper, we show that it is possible to formalize the four inheritance relations in a Petri-net context. Moreover, we extend some of the results presented in [4]. For example, in contrast to the approach in [4], we are able to handle object life cycles with recursion. Although we allow recursion, it is possible to verify each of the four inheritance relations automatically. Moreover, a number of transformation rules which preserve specific forms of inheritance are presented. These transformation rules show how the object life cycle of a superclass may be extended for a subclass while preserving life-cycle inheritance.

2 Labeled Petri Nets

In this paper, we use standard Petri nets extended with a labeling function. Let A be some universe of action labels. Action labels can be thought of as method identifiers, i.e., firing a transition with label $a \in A$ corresponds to the execution of method a.

Definition 2.1. (Labeled Petri net) An A-labeled Petri net is a tuple $N = (P, T, F, \ell)$ where

i) P is a finite set of places;

ii) T is a finite set of transitions such that $P \cap T = \emptyset$;

iii) $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the flow relation;

iv) $\ell : T \to A$ is a labeling function.

A place p is called an *input place* of a transition t if and only if there exists a directed arc from p to t. Place p is called an *output place* of transition t if and only if there exists a directed arc from t to p. We use ${}^\bullet t$ to denote the set of input places for a transition t. The notations $t^\bullet$, ${}^\bullet p$, and $p^\bullet$ have similar meanings. For example, $p^\bullet$ is the set of transitions sharing p as an input place.

Places of a Petri net may contain zero or more *tokens*. The *state* of a Petri net, often referred to as marking, is the distribution of tokens over the places. Hence, the state of a net can be represented by a finite multi-set, or bag, of places. The following definitions and notations for bags are used.

A bag of elements from some alphabet A can be considered as a function from A to the natural numbers $\mathbb{N}$. That is, for some bag X over alphabet A and $a \in A$, $X(a)$ denotes the number of

occurrences of a. The empty bag is denoted $\mathbf{0}$. For the explicit enumeration of a bag, a notation similar to the notation for sets is used, but using square brackets instead of curly brackets and using superscripts to denote the cardinality of the elements. For example, $[a^2, b, c^3]$ denotes the bag with two elements a, one b, and three elements c; the bag $[a^2 \mid a \in A \wedge P(a)]$ contains two elements a for every $a \in A$ such that $P(a)$ holds, where P is some predicate on symbols of the alphabet. To denote individual elements of a bag, the same symbol "$\in$" is used as for sets. The union of two bags X and Y, denoted $X \uplus Y$, is defined as $[a^n \mid a \in A \wedge n = X(a) + Y(a)]$. The difference of X and Y, denoted $X - Y$, is defined as $[a^n \mid a \in A \wedge n = (X(a) - Y(a)) \max 0]$. The restriction of some bag X to some domain $D \subseteq A$, denoted $X \restriction D$, is defined as $[a^{X(a)} \mid a \in D]$. The notion of subbags is defined as expected: bag X is a subbag of Y, denoted $X \leq Y$, if and only if for all $a \in A$, $X(a) \leq Y(a)$.

Definition 2.2. (Marked Petri net) A *marked* Petri net is a pair (N, s) where $N = (P, T, F, \ell)$ is a labeled Petri net and where s is a bag over P denoting the *state* or *marking* of the net.

Marked Petri nets have a dynamic behavior which is defined by the following *firing rule*.

Definition 2.3. (Firing rule) Let (N, s) be some marked Petri net with $N = (P, T, F, \ell)$. A transition $t \in T$ is *enabled*, denoted $(N, s)[t\rangle$, if and only if each input place contains at least one token. That is, $(N, s)[t\rangle \Leftrightarrow {}^\bullet t \leq s$. An enabled transition can *fire*. If a transition t fires, then it *consumes* one token from each of its input places; it *produces* one token for each of its output places. The visible effect of a firing is the *label* of the transition. Formally, $(N, s)\ [\ell(t)\rangle\ (N, s - {}^\bullet t \uplus t^\bullet)$.

Based on the firing rule, the notion of reachability can be formalized.

Definition 2.4. (Reachability) Let (N, s) be a marked A-labeled Petri net. State s' is reachable from s, denoted $(N, s)[*\rangle(N, s')$, if and only if s' equals s or if for some $n \in \mathbb{N}$ there exist $a_0, a_1, \ldots a_n \in A$ and markings $s_1, \ldots, s_n$ such that $(N, s)\ [a_0\rangle\ (N, s_1)\ [a_1\rangle \ldots [a_n\rangle\ (N, s')$.

In this paper, we we want to be able to compare the behavior of objects which are specified by marked Petri nets. Therefore, an equivalence on Petri nets is needed. The equivalence should distinguish Petri nets whose behaviors have different moments of choice, because the moment of choice may influence the order in which methods are allowed to be executed. In addition, Petri nets with the same external behavior, but with possibly different internal behavior must be considered equal. Given these two requirements, branching bisimulation seems to be a suitable equivalence [7].

Let $\mathcal{N}$ be the set of marked $\mathcal{A}$-labeled Petri nets where $\mathcal{A}$ is equal to $A \cup \{\tau\}$. Recall that a τ-labeled transition corresponds to an internal method. The following auxiliary relation expresses that a marked Petri net can evolve into another marked net by firing a sequence of τ-labeled transitions.

Definition 2.5. The relation $\_[\ \rangle\rangle\ \_ : \mathcal{P}(\mathcal{N} \times \mathcal{N})$ is the smallest relation satisfying, for any $n, n', n'' \in \mathcal{N}$, $n\ [\ \rangle\rangle\ n$ and $n\ [\ \rangle\rangle\ n' \wedge n'\ [\tau\rangle\ n'' \Rightarrow n\ [\ \rangle\rangle\ n''$.

Let, for any $n, n' \in \mathcal{N}$ and $\alpha \in \mathcal{A}$, $n\ [(\alpha)\rangle\ n'$ be an abbreviation of $n\ [\alpha\rangle\ n' \vee (\alpha = \tau \wedge n = n')$. That is, $n\ [(\tau)\rangle\ n'$ means zero or one τ steps and, for any $a \in A$, $n\ [(a)\rangle\ n'$ is simply $n\ [a\rangle\ n'$. To define branching bisimulation, we need to identify marked Petri nets which correspond to the successful termination of an object. A life cycle without any tokens corresponds to a terminated object, i.e., the life cycle of a terminated object is of the form $(N, \mathbf{0})$.

Definition 2.6. (Branching-bisimulation equivalence) A binary relation $\mathcal{R} : \mathcal{P}(\mathcal{N} \times \mathcal{N})$ is called a *branching bisimulation* if and only if, for any $n, n', m, m', (N, s_n), (M, s_m) \in \mathcal{N}$ and $\alpha \in \mathcal{A}$,

i) $n\mathcal{R}m \wedge n\ [\alpha\rangle\ n' \Rightarrow (\exists\, m', m'' : m', m'' \in \mathcal{N} : m\ [\ \rangle\rangle\ m''\ [(\alpha)\rangle\ m' \wedge n\mathcal{R}m'' \wedge n'\mathcal{R}m')$,

ii) $n\mathcal{R}m \wedge m\ [\alpha\rangle\ m' \Rightarrow (\exists\, n', n'' : n', n'' \in \mathcal{N} : n\ [\ \rangle\rangle\ n''\ [(\alpha)\rangle\ n' \wedge n''\mathcal{R}m \wedge n'\mathcal{R}m')$,

iii) $(N, s_n)\mathcal{R}(M, s_m) \Rightarrow (N, s_n)\ [\ \rangle\rangle\ (N, \mathbf{0}) \Leftrightarrow (M, s_m)\ [\ \rangle\rangle\ (M, \mathbf{0})$.

Two marked Petri nets n and m are called *branching bisimilar*, denoted $n \sim_b m$, if and only if there exists a branching bisimulation between n and m.

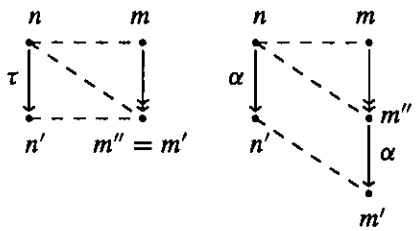

Figure 2: Branching bisimulation.

The first two requirements state that steps in the first marked Petri net are also possible in the second and vice versa. The third requirement says that if a marked Petri net can terminate via a number of τ steps, then this also holds for any other related marked Petri net. Figure 2 shows the essence of branching bisimulation. Note that for any $\alpha \in \mathcal{A}$, the relation $\_\ [\alpha\rangle\ \_$ is depicted by an α-labeled arrow, whereas the relation $\_\ [\ \rangle\rangle\ \_$ is depicted by a double-headed arrow. Also note that the definition given here differs from the original definition given in [8]. In fact, it is the definition of *semi*-branching bisimulation, which was first defined in [9], but formulated as it appears in [3]. It can be shown that the two notions are equivalent [9, 3]. The reason for using the alternative definition is that it is more concise and more intuitive than the original definition.

3 Object Life Cycles

Using Petri nets for the specification of object life cycles allows us to specify a partial ordering of methods. However, not every labeled Petri net specifies an object life cycle. As discussed in the introduction, we introduce three reserved labels, namely ∇ for object creation, Δ for object termination and τ for internal methods. Set L, not containing any of the special labels, is the set of method labels corresponding to external methods; L_s is the set of method labels including the three special labels, i.e., $L_s = L \cup \{\nabla, \Delta, \tau\}$. A Petri net which specifies a life cycle has exactly one transition t_∇ which corresponds to the creation of an object and bears a ∇ label. For convenience, we assume that t_∇ has one unique input place i and that every transition has at least one input place. A Petri net describing a life cycle refers to the life cycle of a *single object*. It suffices to consider just one object because of the fact that objects interact via the execution of methods and not directly via the life cycle. Since we focus on one object at a time, initially, place i contains one token. An object terminates the moment a method with a Δ label is executed. Between the creation of an object (i.e.,

the firing of t_∇) and the termination of an object (i.e., the firing of a Δ-labeled transition), there is always at least one token present in the life cycle. If we restrict ourselves to life cycles without inherent parallelism, it suffices to have just one token. The introduction of parallelism results in multiple tokens that are present in the life cycle. We can think of these tokens as 'stage indicators' referring to the same object. The moment an object is terminated, all tokens which correspond to the object should be removed. This means that firing a Δ-labeled transition results in an empty Petri net indicating that the object has ceased to exist and all references to the object have been removed. Finally, we assume that it is always possible to terminate by executing the appropriate sequence of methods. However, this does not mean that every object is forced to terminate. The following definition formalizes the definition of an object life cycle.

Definition 3.1. (Object life cycle) A marked L_s-labeled Petri net (N, s) where $N = (P, T, F, \ell)$ describes an object life cycle if and only if

i) P contains a special place i and T contains a special transition t_∇ such that ${}^\bullet i = \emptyset$, $i^\bullet = \{t_\nabla\}$ and ${}^\bullet t_\nabla = \{i\}$. Moreover, t_∇ is the only transition in T such that $\ell(t_\nabla) = \nabla$. For any transition $t \in T$, ${}^\bullet t \neq \emptyset$.

ii) The initial marking s contains just one token, which is a token in the initial place i: $s = [i]$.

iii) Let s' be an arbitrary state reachable from s, i.e., $(N, s)\ [*\rangle\ (N, s')$.

- For any $t \in T$ such that $(N, s')[t\rangle$, $(N, s')\ [\ell(t)\rangle\ (N, \mathbf{0}) \Leftrightarrow \ell(t) = \Delta$. Hence, there is a one-to-one correspondence between the termination of an object and the firing of a Δ-labeled transition.
- It is possible to terminate successfully from s', i.e., $(N, s')\ [*\rangle\ (N, \mathbf{0})$.

If we restrict ourselves to free-choice Petri nets, then there is a polynomial-time algorithm to verify the requirements in Definition 3.1 [1]. Moreover, for most object life cycles it is easy to see whether these requirements hold. Petri nets satisfying the requirements stated in Definition 3.1 have a number of nice properties. One of them is boundedness, i.e., the number of reachable states is finite.

Property 3.2. *Any marked Petri net $(N, [i])$ representing an object life cycle is bounded.*

Proof. If an object life cycle $(N, [i])$ is not bounded, then there are two reachable states s_1 and s_2 such that $(N, [i])\ [*\rangle\ (N, s_1)\ [*\rangle\ (N, s_2)$ and $s_2 > s_1$. Since $(N, [i])$ is a life cycle, there is a sequence of firings σ leading from (N, s_1) to $(N, \mathbf{0})$. The label of the last transition that fired is Δ. However, we can also execute σ from (N, s_2). In this case, the label of the final firing is still Δ but this firing does not result in $(N, \mathbf{0})$ which contradicts the fact that $(N, [i])$ is an object life cycle. Hence, an object life cycle $(N, [i])$ is bounded. □

4 Life-Cycle Inheritance

We have given a formal definition of an object life cycle in terms of a Petri net. Now, it is time to answer the following question: When is an object life cycle a subclass of some other object life cycle? For example, is $(N_1, [i])$ in Figure 1 a subclass of $(N_0, [i])$? In other words, does $(N_1, [i])$ inherit certain features of $(N_0, [i])$? To answer this question, we have to establish an inheritance relation for object life cycles. Inspired by process-algebraic concepts like encapsulation and abstraction, two basic forms of inheritance seem to be appropriate [4].

The first basic form of inheritance corresponds to encapsulation. Let $(N_0, [i])$ and $(N_1, [i])$ be two object life cycles.

> *If the environment only calls the methods of* $(N_1, [i])$ *which are not present in* $(N_0, [i])$ *and it cannot distinguish the observable behavior of* $(N_0, [i])$ *and* $(N_1, [i])$, *then* $(N_1, [i])$ *is a subclass of* $(N_0, [i])$.

This means that if the new methods added to the subclass are blocked or the environment is not willing to use the new methods, then the superclass and the subclass behave equivalently. As shown in [4] this corresponds to the encapsulation operator known from ACP (∂_H) which translates actions in H to deadlock δ. This form of inheritance is referred to as *protocol inheritance* because the subclass inherits the protocol of the superclass. It is not difficult to verify that $(N_1, [i])$ in Figure 1 is a subclass of $(N_0, [i])$ with respect to protocol inheritance. If the method *birthday* is blocked, then the observable behaviors are identical.

The second basic form of inheritance corresponds to abstraction.

> *If the environment is willing to call the methods of* $(N_1, [i])$ *which are not present in* $(N_0, [i])$ *and it cannot distinguish the observable behavior of* $(N_0, [i])$ *and* $(N_1, [i])$ *with respect to the methods of* $(N_0, [i])$, *then* $(N_1, [i])$ *is a subclass of* $(N_0, [i])$.

This means that even by calling the new methods (i.e. the methods in N_1 but not in N_0), the behavior of the subclass coincides with the behavior of the superclass with respect to the old methods. However, if the environment is reluctant to call some of the new methods, it may discover differences with respect to the old methods. If we consider the new methods to be internal methods which cannot disable or enable old methods, then the superclass and the subclass behave equivalently. For those familiar with process algebra, it is easy to see that this corresponds to the abstraction operator (τ_I) which translates actions in I to silent steps τ. This form of inheritance is referred to as *projection inheritance*. It is also easy to see that $(N_1, [i])$ in Figure 1 is a subclass of $(N_0, [i])$ with respect to projection inheritance. If we hide the method *birthday*, then the observable behaviors are identical.

Analogously to [4], we define two other forms of inheritance by combining the two basic forms just presented. But first, we define the encapsulation operator ∂_H and the abstraction operator τ_I for Petri nets.

Definition 4.1. (Encapsulation and abstraction) Let (N, s) be a marked L_s-labeled Petri net with $N = (P, T, F, \ell)$.

i) For any $H \subseteq L$, the encapsulation operator ∂_H removes all transitions with a label in H from a given Petri net. Formally, $\partial_H(N, s) = (N', s)$ such that $N' = (P, T', F', \ell')$, $T' = \{t \in T \mid \ell(t) \notin H\}$, $F' = F \cap ((P \times T') \cup (T' \times P))$ and $\ell' = \ell \cap (T' \times L_s)$.

ii) For any $I \subseteq L$, the abstraction operator τ_I renames all transition labels in I to τ. That is, $\tau_I(N, s) = (N', s)$ such that $N' = (P, T, F, \ell')$ and for any $t \in T$, $\ell(t) \in I$ implies $\ell'(t) = \tau$ and $\ell(t) \notin I$ implies $\ell'(t) = \ell(t)$.

Note that the encapsulation of methods corresponds to the removal of transitions, i.e., the blocking of a method is achieved by removing the corresponding transitions.

Property 4.2. *Branching bisimulation, $\sim_b$, is a congruence for encapsulation and abstraction.*

Proof. It is straightforward to verify that branching bisimulation, $\sim_b$, is an equivalence relation [3]. It remains to show that for any two marked Petri nets (N_0, s_0) and (N_1, s_1) and any $H, I \subseteq L$, $(N_0, s_0) \sim_b (N_1, s_1)$ implies that $\partial_H(N_0, s_0) \sim_b \partial_H(N_1, s_1)$ and $\tau_I(N_0, s_0) \sim_b \tau_I(N_1, s_1)$. Let $\mathcal{R}$ be a branching bisimulation between (N_0, s_0) and (N_1, s_1). Based on $\mathcal{R}$, we define the binary relation $Q = \{(\partial_H(N_0, u), \partial_H(N_1, v)) \mid (N_0, s_0)[*\rangle(N_0, u) \wedge (N_1, s_1)[*\rangle(N_1, v) \wedge (N_0, u)\mathcal{R}(N_1, v)\}$. It is not difficult to verify that Q is a branching bisimulation between $\partial_H(N_0, s_0)$ and $\partial_H(N_1, s_1)$. Hence, $\sim_b$ is a congruence for the encapsulation operator ∂_H. Similarly, we can prove that $\sim_b$ is a congruence for the abstraction operator τ_I. □

Using encapsulation and abstraction, we define protocol inheritance and projection inheritance respectively. However, it is possible to combine these two definitions. Therefore, we also define two additional forms of inheritance. *Protocol/projection inheritance* is the conjunction of the two basic forms of inheritance. An object life cycle is a subclass of another object life cycle with respect to protocol/projection inheritance if and only if it is a subclass with respect to protocol inheritance *and* projection inheritance. The disjunction of the two basic forms of inheritance does not yield an inheritance relation with desirable properties such as transitivity. However, it is possible to state that for every new method one of the two basic forms of inheritance should hold. This form of inheritance is called *life-cycle inheritance*. An object life cycle is a subclass of another object life cycle (i.e. the superclass) with respect to life-cycle inheritance if and only if the abstraction of some methods and the encapsulation of some other methods of the subclass results in an object life cycle equivalent to the superclass.

Definition 4.3. (Inheritance relations) For any two marked Petri nets (N, s), $(N', s') \in \mathcal{N}$,

i) *protocol inheritance:*
(N, s) is a subclass of (N', s') under *protocol inheritance*, denoted $(N, s) \leq_{pt} (N', s')$, if and only if there is an $H \subseteq L$ such that $\partial_H(N, s) \sim_b (N', s')$,

ii) *projection inheritance:*
(N, s) is a subclass of (N', s') under *projection inheritance*, denoted $(N, s) \leq_{pj} (N', s')$, if and only if there is an $I \subseteq L$ such that $\tau_I(N, s) \sim_b (N', s')$,

iii) *protocol/projection inheritance:*
(N, s) is a subclass of (N', s') under *protocol/projection inheritance*, denoted $(N, s) \leq_{pp} (N', s')$, if and only if there is an $H \subseteq L$ such that $\partial_H(N, s) \sim_b (N', s')$ *and* an $I \subseteq L$ such that $\tau_I(N, s) \sim_b (N', s')$,

iv) *life-cycle inheritance:*
(N, s) is a subclass of (N', s') under *life-cycle inheritance*, denoted $(N, s) \leq_{lc} (N', s')$, if and only if there is an $I \subseteq L$ and an $H \subseteq L$ such that $I \cap H = \emptyset$ and $\tau_I \circ \partial_H(N, s) \sim_b (N', s')$.

Note that life-cycle inheritance is defined in terms of a function composition ($\tau_I \circ \partial_H$). Since we demand that I and H are disjoint, we may change the order of encapsulation and abstraction without changing the definition of life-cycle inheritance.

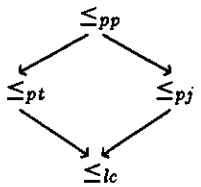

Figure 3: An overview of life-cycle-inheritance relations.

Figure 3 shows an overview of the four inheritance relations. Protocol/projection inheritance is the strongest form of inheritance. If an object life cycle is a subclass with respect to protocol/projection inheritance, then it is also a subclass with respect to the other three forms of inheritance. In Figure 1, $(N_1, [i])$ is a subclass of $(N_0, [i])$ with respect to protocol/projection inheritance. Therefore, $(N_1, [i])$ is also a subclass of $(N_0, [i])$ with respect to the other three forms of inheritance. Life-cycle inheritance is the weakest form of inheritance. If an object life cycle is a subclass with respect to any of the four forms of inheritance, then it is also a subclass with respect to life-cycle inheritance. In [4] it is shown that the inclusion relations in Figure 3 are strict and that there are no inclusion relations between protocol inheritance and projection inheritance.

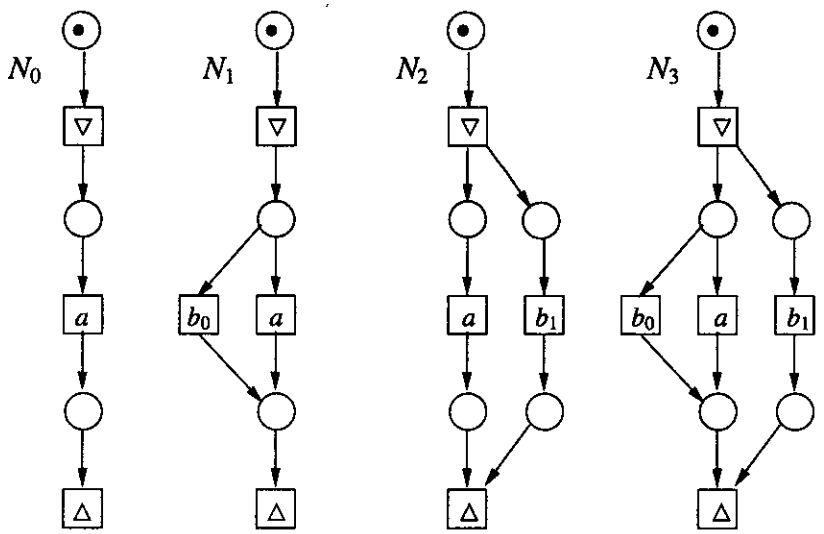

Figure 4: $(N_1, [i]) \leq_{pt} (N_0, [i])$, $(N_2, [i]) \leq_{pj} (N_0, [i])$ and $(N_3, [i]) \leq_{lc} (N_0, [i])$.

Example 4.4. The four object life cycles shown in Figure 4 illustrate these inheritance relations. $(N_1, [i])$ is the subclass of $(N_0, [i])$ under protocol inheritance, because the omission of the transition labeled b_0 in N_1 yields a net structurally equivalent and, hence, branching bisimilar to $(N_0, [i])$. $(N_2, [i])$ is a subclass of $(N_0, [i])$ under projection inheritance, because hiding the transition labeled b_1 in N_2 yields a marked Petri net which is branching bisimilar to $(N_0, [i])$. $(N_2, [i])$ is not a subclass of $(N_0, [i])$ under protocol inheritance, because the blocking of b_1 yields a net which cannot

terminate successfully. $(N_3, [i])$ is not a subclass of $(N_0, [i])$ under protocol inheritance, nor is it a subclass under projection inheritance. However, $(N_3, [i])$ is a subclass of $(N_0, [i])$ under life-cycle inheritance.

The object life cycles shown in Figures 1 and 4 illustrate that the four inheritance relations are complementary. Moreover, our belief that the four inheritance relations are valuable is strengthened by the fact that each of the four relations is reflexive and transitive.

Property 4.5. *Protocol inheritance, projection inheritance, protocol/projection inheritance, and life-cycle inheritance are preorders.*

Proof. For any labeled marked Petri net (N, s), $\partial_\emptyset(N, s)$ is equal to (N, s) and $\tau_\emptyset(N, s)$ is equal to (N, s). Hence, $\leq_{pt}$, $\leq_{pj}$, $\leq_{pp}$, and $\leq_{lc}$ are reflexive. It is fairly straightforward to show that $\leq_{pt}$ is transitive. Let (N_0, s_0), (N_1, s_1) and (N_2, s_2) be three marked Petri nets such that $(N_0, s_0) \leq_{pt} (N_1, s_1)$ and $(N_1, s_1) \leq_{pt} (N_2, s_2)$. It is possible to find two sets of labels $H, H' \subseteq L$ such that $\partial_H(N_0, s_0) \sim_b (N_1, s_1)$ and $\partial_{H'}(N_1, s_1) \sim_b (N_2, s_2)$. Since $\sim_b$ is a congruence for ∂_H (see Property 4.2), it is easy to verify that $\partial_{H' \cup H}(N_0, s_0) = \partial_{H'} \circ \partial_H(N_0, s_0) \sim_b \partial_{H'}(N_1, s_1) \sim_b (N_2, s_2)$. Hence, $(N_0, s_0) \leq_{pt} (N_2, s_2)$. Analogously, we can prove that $\leq_{pt}$ is transitive. Since $\leq_{pp} = \leq_{pt} \cap \leq_{pj}$, it follows immediately that $\leq_{pp}$ is transitive. Showing that life-cycle inheritance is transitive is more involved. Assume $(N_0, s_0) \leq_{lc} (N_1, s_1) \leq_{lc} (N_2, s_2)$. From the definition of life-cycle inheritance it follows that there are subsets H, H', I, and I' of L such that $\tau_I \circ \partial_H(N_0, s_0) \sim_b (N_1, s_1)$ and $\tau_{I'} \circ \partial_{H'}(N_1, s_1) \sim_b (N_2, s_2)$, $H \cap I = \emptyset$ and $H' \cap I' = \emptyset$. Moreover, it is possible to choose H, H', I, and I' such that $(H \cup I) \cap (H' \cup I') = \emptyset$ (see [4]). Since $\sim_b$ is a congruence for abstraction and encapsulation, it follows that $\tau_{I' \cup I} \circ \partial_{H' \cup H}(N_0, s_0) = \tau_{I'} \circ \tau_I \circ \partial_{H'} \circ \partial_H(N_0, s_0) = \tau_{I'} \circ \partial_{H'} \circ \tau_I \circ \partial_H(N_0, s_0) \sim_b \tau_{I'} \circ \partial_{H'}(N_1, s_1) \sim_b (N_2, s_2)$. Hence, $\leq_{lc}$ is also transitive. □

Analogously to the result in [4] we can also show that *subclass equivalence* coincides with branching-bisimulation equivalence, i.e., given two object life cycles and one of the four inheritance relations, if the first life cycle is a subclass of the second life cycle and vice versa, then the two life cycles are branching bisimilar. This is another result showing that the definitions are sound.

Theorem 4.6. (Decidability of inheritance) *For any two object life cycles $(N_0, [i])$ and $(N_1, [i])$ it is decidable whether $(N_1, [i])$ is a subclass of $(N_0, [i])$ with respect to $\leq_{pt}$, $\leq_{pj}$, $\leq_{pp}$, or $\leq_{lc}$.*

Proof. If follows from Property 3.2 that the two object life cycles are bounded. Each of the modified object life cycles used in Definition 4.3 (i.e., $\partial_H(N_1, [i])$, $\tau_I(N_1, [i])$ and $\tau_I \circ \partial_H(N_1, [i])$ with $H, I \subseteq L$) is also bounded (Although they may not satisfy the requirements in Definition 3.1). Therefore, checking whether such a modified life cycle and $(N_0, [i])$ are branching bisimilar is decidable. □

5 Inheritance-Preserving-Transformation Rules on Petri Nets

As long as life cycles are not too complex, it is easy to check whether a specific inheritance relation holds. Unfortunately, object life cycles tend to become very complex. Although it is possible to check the inheritance relations automatically, such a check may require a lot of computing power. Therefore, we propose a number of transformation rules which preserve inheritance. Moreover, these transformation rules reveal the essence of the inheritance relations described in Definition 4.3.

For convenience, we introduce the alphabet operator α on Petri nets. For any L_s-labeled Petri net $N = (P, T, F, \ell)$, $\alpha(N) = \{\ell(t) \mid t \in T \wedge \ell(t) \in L_s \setminus \{\tau\}\}$. The union of two Petri nets is defined as the union of the components, i.e., $N_p \cup N_q = (P_p \cup P_q, T_p \cup T_q, F_p \cup F_q, \ell_p \cup \ell_q)$ under the assumption that for any $t \in T_p \cap T_q$, $\ell_p(t) = \ell_q(t)$.

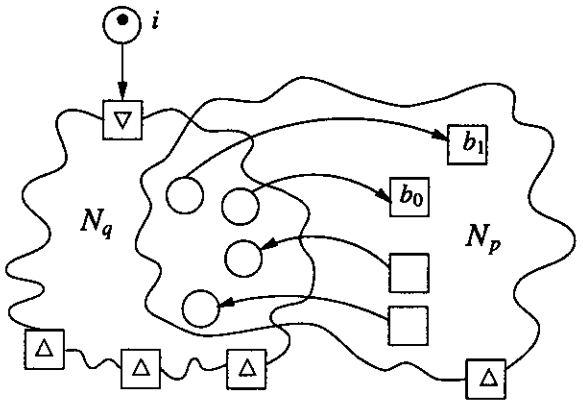

Figure 5: Protocol-inheritance-preserving transformation rule.

The first transformation rule preserves protocol inheritance and is illustrated in Figure 5. If we extend a life cycle $(N_q, [i])$ with a Petri net N_p such that (1) no transitions are shared among both nets, (2) all new transitions consuming from places in N_q have a label not in $\alpha(N_q)$ and (3) the result is still a life cycle, then the extended life cycle is a subclass of the original life cycle with respect to protocol inheritance.

Theorem 5.1. (Protocol-inheritance-preserving transformation rule) *Let $N_q = (P_q, T_q, F_q, \ell_q)$ and $N_p = (P_p, T_p, F_p, \ell_p)$ be two Petri nets. Let $(N, [i]) = (N_q \cup N_p, [i])$ and $(N', [i]) = (N_q, [i])$ be two object life cycles satisfying the requirements stated in Definition 3.1. If the following additional properties are satisfied,*

i) $T_q \cap T_p = \emptyset$,

ii) $(\forall\, p, t : t \in T_p \wedge p \in P_q \cap {}^\bullet t : \ell(t) \in L \setminus \alpha(N_q))$,

then $(N, [i])$ is a subclass of $(N', [i])$ with respect to protocol inheritance, i.e. $(N, [i]) \leq_{pi} (N', [i])$.

Proof. We show that $\partial_H(N, [i]) \sim_b (N', [i])$ with $H = \alpha(N_p) \setminus \alpha(N_q)$. Consider the marked Petri net $(N, [i])$. Initially, the places in $P_p \setminus P_q$ are empty. The only way to add tokens to one of these places is by firing a transition consuming tokens from $P_q \cap P_p$. So, if we encapsulate these transitions $(\partial_H(N, [i]))$, then the places in $P_p \setminus P_q$ will remain empty and none of the remaining transitions

in T_p will ever be able to fire. Hence, the subnet added to $(N', [i])$ in $\partial_H(N, [i])$ is dead if we encapsulate the transitions having a new label. Let $\mathcal{R} = \{(\partial_H(N, u), (N', u)) \mid \partial_H(N, [i])\ [*\rangle\ \partial_H(N, u) \wedge (N', [i])\ [*\rangle\ (N', u)\}$ be a binary relation. Since the subnet added to $(N', [i])$ in $\partial_H(N, [i])$ is dead, it is straightforward to verify that $\mathcal{R}$ is a branching bisimulation between $\partial_H(N, [i])$ and $(N', [i])$. □

The transitions in the set $\{t \in T_p \mid P_q \cap {}^\bullet t \neq \emptyset\}$ operate as 'guards'. By blocking these guards, the new part of the object life cycle is deactivated. In Figure 5, b_0 and b_1 operate as guards. By applying this transformation rule, we are able to show that $(N_1, [i])$ in Figure 4 is a subclass of $(N_0, [i])$. We can also apply this rule to show that $(N_1, [i])$ in Figure 1 is a subclass of $(N_0, [i])$.

The transformation rule described by Theorem 5.1 is inspired by an axiom presented in [4]. To show the relation between the inheritance-preserving transformation rules presented in this paper and some of the algebraic rules in [4], we give an intermezzo for those familiar with process algebra.

Intermezzo 5.2. In [4] we presented an algebraic theory $\mathrm{PA}^\tau_{\delta\rho}$ for studying life-cycle inheritance. $\mathrm{PA}^\tau_{\delta\rho}$ is an ACP-like process algebra with deadlock, internal actions and renaming. In this context, an object life cycle is defined to be a closed PA^τ term starting with the object-creation action $\triangledown$. Based on this theory, we have defined four forms of inheritance analogous to Definition 4.3. For example, for any two object life cycles p and q, $p \leq_{pt} q$ if and only if $\mathrm{PA}^\tau_{\delta\rho} \vdash \partial_H(p) = q$.

In [4], we also presented a number of rules which illustrate under what conditions inheritance is preserved. In the remainder of this intermezzo, we concentrate on these rules. L, L_s and α are defined analogous to the definitions in this paper. Let p, q, q_0, q_1 and r be closed PA^τ terms and a and b actions in L_s such that and $\alpha(r) \subseteq L \setminus (\alpha(q) \cup \alpha(q_0) \cup \alpha(q_1) \cup \{a\})$ and $b \in L \setminus (\alpha(q) \cup \{a\})$. Under these conditions the following axioms apply.

$q + b \cdot p \leq_{pt} q$	PT
$q \cdot r \leq_{pj} q$	$PJ1$
$a \cdot (r \cdot (q_0 + q_1) + q_0) \leq_{pj} a \cdot (q_0 + q_1)$	$PJ2$
$a \cdot (q \parallel r) \leq_{pj} a \cdot q$	$PJ3$
$a \cdot ((b \cdot r) \cdot q + q) \leq_{pp} a \cdot q$	PP

The first rule (axiom PT) corresponds to the rule in Theorem 5.1. Method b functions as a guard. By blocking the guard, the environment is forced to follow the original life cycle q. Rules $PJ1$ and $PJ2$ state that inserting new behavior in an object life cycle that does not disable any behavior of the original life cycle, yields a subclass under projection inheritance. Rule $PJ3$ shows that putting alternative behavior in parallel with the original life cycle also yields a subclass under projection inheritance. Rule PP shows that under protocol/projection inheritance it is allowed to postpone behavior. In the remainder of this section, we show that we can formulate transformation rules on Petri nets which correspond to $PJ3$, $PJ1$ and PP. Although it is possible to define a transformation rule which corresponds to $PJ2$, we will not do so, because the duplication of q_0 is not very meaningful in the context of Petri nets.

The second transformation rule corresponds to rule $PJ3$ and is illustrated in Figure 6. If we extend a life cycle $(N_q, [i])$ with a Petri net N_r such that (1) no places are shared among both nets, (2) all new transitions have a label not in $\alpha(N_q)$, (3) the transitions in N_q consuming tokens from N_r obey

the free-choice property ([6]) and (4) the result is still a life cycle, then the extended life cycle is a subclass of the original life cycle with respect to projection inheritance. Hence, we can add parts to the life cycle which are executed in parallel with the original life cycle while preserving projection inheritance.

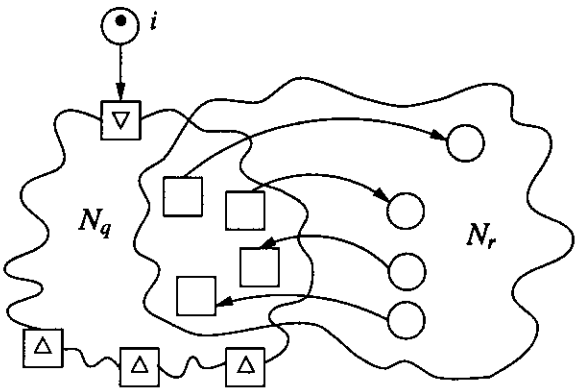

Figure 6: Projection-inheritance-preserving transformation rule.

Theorem 5.3. (Projection-inheritance-preserving transformation rule) *Let $N_q = (P_q, T_q, F_q, \ell_q)$ and $N_r = (P_r, T_r, F_r, \ell_r)$ be two Petri nets with for any $t \in T_q \cap T_r$, $\ell_q(t) = \ell_r(t)$. Let $(N, [i]) = (N_q \cup N_r, [i])$ and $(N', [i]) = (N_q, [i])$ be two object life cycles satisfying the requirements stated in Definition 3.1. If the following additional properties are satisfied,*

i) $P_q \cap P_r = \emptyset$,

ii) $(\forall\, t : t \in T_r \setminus T_q : \ell(t) \notin \alpha(N_q))$,

iii) $(\forall\, p, t : p \in P_r \wedge t \in T_q \cap p^\bullet : (\forall t' : t' \in T_q : {}^\bullet t \cap {}^\bullet t' \neq \emptyset \Rightarrow {}^\bullet t = {}^\bullet t'))$,

then $(N, [i])$ is a subclass of $(N', [i])$ with respect to projection inheritance, i.e. $(N, [i]) \leq_{pj} (N', [i])$.

Proof. We have to prove that $\tau_I(N, [i]) \sim_b (N', [i])$ with $I = \alpha(N_r) \setminus \alpha(N_q)$.
Let $\mathcal{R} = \{(\tau_I(N, u), (N', v)) \mid u \restriction P_q = v \wedge \tau_I(N, [i])\ [*\rangle\ \tau_I(N, u) \wedge (N', [i])\ [*\rangle\ (N', v)\}$ be a binary relation. To prove that $\mathcal{R}$ is a branching bisimulation between $\tau_I(N, [i])$ and $(N', [i])$, we show that the three requirements stated in Definition 2.6 hold. Let u be a state of N and let v be a state of N' such that $\tau_I(N, u)\mathcal{R}(N', v)$.

i) If u' is a state of N such that $\tau_I(N, u)\ [\alpha\rangle\ \tau_I(N, u')$, then it is easy to verify that there is a state v' of N' such that $\tau_I(N, u')\mathcal{R}(N', v')$ and $(N', v)\ [(\alpha)\rangle\ (N', v')$. If $\tau_I(N, u)\ [\alpha\rangle\ \tau_I(N, u')$ corresponds to the firing of a transition in T_r, then $v' = v$. Otherwise, there is a transition t in T_q such that $\ell(t) = \alpha$ and $(N', v)\ [\alpha\rangle\ (N', v')$.

ii) If v' is a state of N' such that $(N', v)\ [\alpha\rangle\ (N', v')$, then we have to prove that there is a state u' and a state u'' such that $\tau_I(N, u)\ [\ \rangle\rangle\ \tau_I(N, u'')\ [(\alpha)\rangle\ \tau_I(N, u')$, $\tau_I(N, u'')\mathcal{R}(N', v)$ and $\tau_I(N, u')\mathcal{R}(N', v')$. Let t be a transition in T_q such that $(N', v)\ [\ell(t)\rangle\ (N', v')$. If t is enabled in $\tau_I(N, u)$, then $\tau_I(N, u)\ [\ell(t)\rangle\ \tau_I(N, u')$ and $\tau_I(N, u')\mathcal{R}(N', v')$. If t is not enabled in $\tau_I(N, u)$, then we have to prove that t can be enabled without firing transitions in N_q. If t

is not enabled in $\tau_I(N, u)$, then ${}^\bullet t \cap P_r \neq \emptyset$ and one of the places in ${}^\bullet t \cap P_r$ is empty. Assume that t will never be able to fire in $\tau_I(N, u)$. One of the input places of t contains a token (t is enabled in (N', v) and $u \restriction P_q = v$) and because of the third property all other transitions also consuming from this input place will never be able to to fire. However, this means that $\tau_I(N, u)$ is not a life cycle because successful termination is not possible. Hence, we conclude that it is possible to enable t in $\tau_I(N, u)$ without firing transitions in N_q and firing t results in a state u' such that $\tau_I(N, u')\mathcal{R}(N', v')$.

iii) Remains to prove that $\tau_I(N, u)\ [\ \rangle\rangle\ \tau_I(N, \mathbf{0}) \Leftrightarrow (N', v)\ [\ \rangle\rangle\ (N', \mathbf{0})$. The only way to reach a state with no tokens in a marked Petri net satisfying Definition 3.1 is by firing a Δ-labeled transition just before reaching state $\mathbf{0}$. Hence $\tau_I(N, u)\ [\ \rangle\rangle\ \tau_I(N, \mathbf{0}) \Leftrightarrow u = \mathbf{0}$ and $(N', v)\ [\ \rangle\rangle\ (N', \mathbf{0}) \Leftrightarrow v = \mathbf{0}$. If $u = \mathbf{0}$, then $v = u \restriction P_q = \mathbf{0}$. If $v = \mathbf{0}$, then $u \restriction P_q = \mathbf{0}$. Since $(N, [i])$ satisfies Definition 3.1, it is not possible that $u \neq \mathbf{0}$ and $u \restriction P_q = \mathbf{0}$. Hence, $u = \mathbf{0} \Leftrightarrow v = \mathbf{0}$.

□

Theorem 5.3 specifies sufficient requirements such that the extension of the life cycle with a part that is executed in parallel yields a life cycle under projection inheritance. We can use this transformation rule, to show that $(N_2, [i])$ in Figure 4 is a subclass of $(N_0, [i])$.

It is not difficult to find other transformation rules which preserve some kind of inheritance. Figure 7 shows a transformation rule inspired by rule $PJ1$. This transformation rule shows that we can insert new behavior between two parts of the original life cycle that are executed sequentially while preserving projection inheritance. In contrast to the previous two transformation rules, the Petri net which corresponds to the superclass is modified. The transformation rule shown in Figure 7 boils down to the replacement of an arc by an entire Petri net. This transformation rule preserves projection inheritance if the requirements stated in the following conjecture are met.

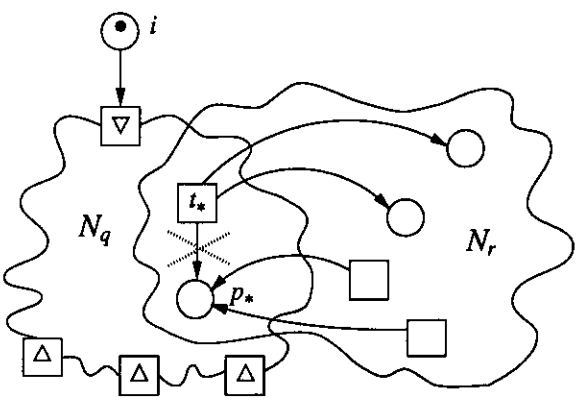

Figure 7: Another projection-inheritance-preserving transformation rule.

Conjecture 5.4. (Another projection-inheritance-preserving transformation rule) *Let* $N_q = (P_q, T_q, F_q, \ell_q)$ *and* $N_r = (P_r, T_r, F_r, \ell_r)$ *be two Petri nets such that:*

i) $P_q \cap P_r = \{p_*\}$, $T_q \cap T_r = \{t_*\}$, $(t_*, p_*) \in F_q$ *and* $\ell_q(t_*) = \ell_r(t_*)$ *for some place* p_* *and transition* t_*,

ii) $(\forall t : t \in T_r : t \neq t_* \Rightarrow \ell(t) \notin \alpha(N_q))$,

iii) $N'_r = (P_r, T_r, F'_r, \ell_r)$ *with* $F'_r = F_r \cup \{(p_*, t_*)\}$ *is a free-choice Petri net and* $(N'_r, [p_*])$ *is live and bounded.*

Let $N'_q = (P_q, T_q, F'_q, \ell_q)$ *with* $F'_q = F_q \setminus \{(t_*, p_*)\}$. *If* $(N, [i]) = (N'_q \cup N_r, [i])$ *and* $(N', [i]) = (N_q, [i])$ *are object life cycles satisfying the requirements stated in Definition 3.1, then* $(N, [i])$ *is a subclass of* $(N', [i])$ *with respect to projection inheritance, i.e.,* $(N, [i]) \leq_{pj} (N', [i])$.

For people not familiar with free-choice Petri nets, requirement *iii)* may be hard to swallow. Using the rich theory of free-choice Petri nets ([6]) it is easy to prove that $[p_*]$ is a so-called home marking of $(N'_r, [p_*])$, see [1]. This implies that eventually every token consumed from place p_* by N_r is returned. If we abstract from the methods added by N_r, then the replacement of the arc between t_* and p_* by N_r does not change the external behavior. Therefore, it can be shown that the replacement of this arc by N_r preserves projection inheritance.

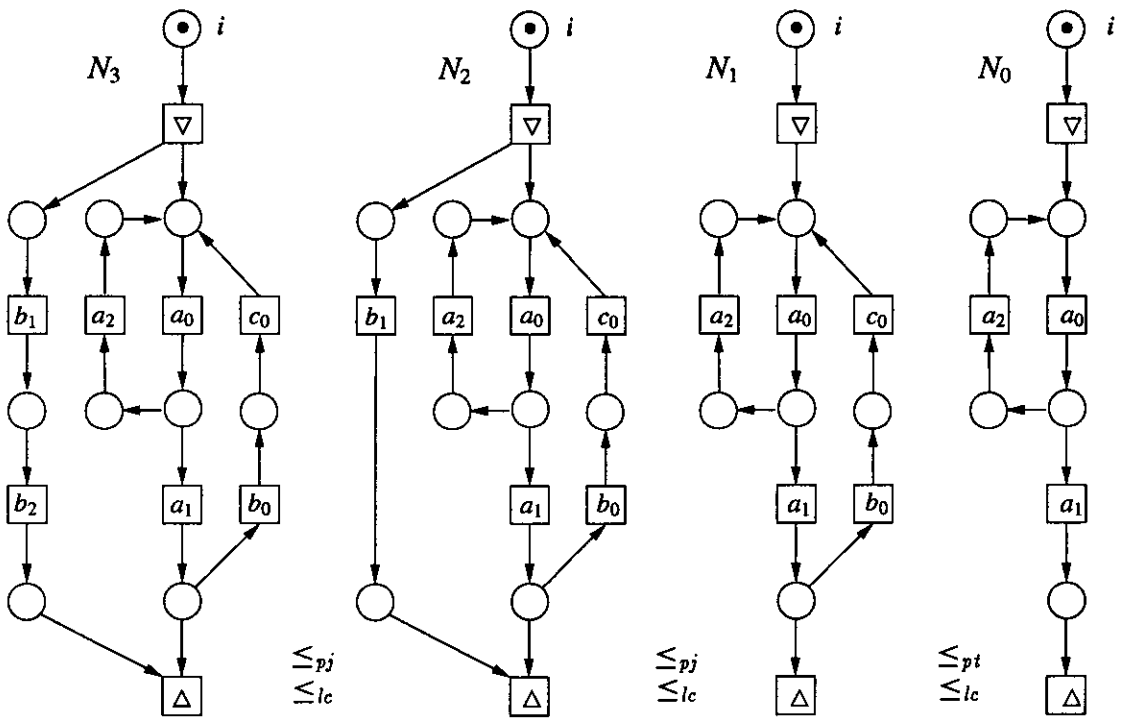

Figure 8: The application of the three transformation rules leads from $(N_0, [i])$ to $(N_3, [i])$ while preserving life-cycle inheritance.

Example 5.5. Figure 8 illustrates the three transformation rules presented thus far. The first transformation rule can be used to prove that $(N_1, [i]) \leq_{pt} (N_0, [i])$ (b_0 is removed from N_1). The second rule can be used to prove that $(N_2, [i]) \leq_{pj} (N_1, [i])$ (b_1 is relabeled to τ). Application of the third transformation rule shows that $(N_3, [i]) \leq_{pj} (N_2, [i])$ (b_2 is relabeled to τ). The three transformation rules also preserve life-cycle inheritance. Since $\leq_{lc}$ is transitive, we deduce that $(N_3, [i]) \leq_{lc} (N_0, [i])$.

Finally, we present a rule which preserves protocol *and* projection inheritance. This transformation rule corresponds to rule *PP* and is illustrated in Figure 9.

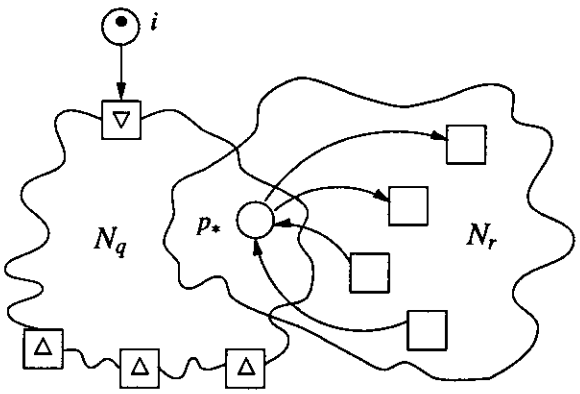

Figure 9: A protocol/projection-inheritance-preserving transformation rule.

Conjecture 5.6. (Protocol/projection-inheritance-preserving transformation rule) *Let* $N_q = (P_q, T_q, F_q, \ell_q)$ *and* $N_r = (P_r, T_r, F_r, \ell_r)$ *be two Petri nets such that:*

i) $T_q \cap T_r = \emptyset$ *and there is a place* p_* *such that* $P_q \cap P_r = \{p_*\}$,

ii) $\alpha(N_r) \cap \alpha(N_q) = \emptyset$,

iii) $N_r = (P_r, T_r, F_r, \ell_r)$ *is a free-choice Petri net and* $(N_r, [p_*])$ *is live and bounded.*

If $(N, [i]) = (N_q \cup N_r, [i])$ *and* $(N', [i]) = (N_q, [i])$ *are object life cycles satisfying the requirements stated in Definition 3.1, then* $(N, [i])$ *is a subclass of* $(N', [i])$ *with respect to protocol/projection inheritance, i.e.,* $(N, [i]) \leq_{pp} (N', [i])$.

It is easy to see that protocol inheritance is preserved, because the conditions stated in Theorem 5.1 apply. Projection inheritance is also preserved because the transitions in N_r are renamed to τ and are willing to fire such that any token consumed from $[p_*]$ is returned eventually. The latter is a direct result of the fact that $[p_*]$ is a home marking of $(N_r, [p_*])$. The transformation rule described in Conjecture 5.6 shows that under protocol/projection inheritance it is allowed to postpone part of the life cycle.

Note that, in contrast to the rules presented in [4], the four transformation rules presented in this paper are also applicable to object life cycles with recursion. In fact, it is easy to see that the application of the last transformation rule introduces recursion.

The four transformation rules give a good characterization of the various forms of inheritance. In contrast to [4], we did not provide rules for the preservation of life-cycle inheritance, because these rules are combinations of the rules for protocol and projection inheritance (See Example 5.5). The fact that the rules in [4] correspond to elegant transformation rules in a Petri-net context is encouraging. It appears that the inheritance concepts used in this paper are quite universal and transcend the two formalisms.

6 Concluding Remarks

A framework for the specification and verification of life-cycle inheritance has been presented. The framework is based on Petri nets and, therefore, close to the professional experience of people engaged in object-oriented design. The four inheritance relations presented in this paper have been inspired by the process-algebraic concepts of encapsulation and abstraction [4]. It has been shown that these inheritance relations can be checked automatically. Moreover, a number of powerful inheritance-preserving transformation rules have been presented. These transformation rules show how an object life cycle may be extended while preserving certain dynamical properties.

Acknowledgements. The authors would like to thank Jos Baeten and Marc Voorhoeve for their valuable suggestions.

References

1. W.M.P. van der Aalst. A Class of Petri Nets for Modeling and Analyzing Business Processes. Computing Science Reports 95/26, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.
2. G. Agha et al. Panel discussion at the workshop on Object-Oriented Programming and Models of Concurrency. 16th. International Conference on the Application and Theory of Petri Nets, Torino, Italy, June 1995.
3. T. Basten. Branching Bisimilarity is an Equivalence indeed! To appear in Information Processing Letters.
4. T. Basten and W.M.P. van der Aalst. A Process-Algebraic Approach to Life-Cycle Inheritance: Inheritance = encapsulation + abstraction. Computing Science Reports 96/05, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996.
5. G. Booch. *Object-Oriented Analysis and Design: With Applications.* Benjamin/Cummings, Redwood City, CA, USA, 1994.
6. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1995.
7. R.J. van Glabbeek. What is Branching Time Semantics and Why to Use It? In *Bulletin of the EATCS,* number 53, pages 191–198. European Association for Theoretical Computer Science, June 1994.
8. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89: Proceedings of the IFIP 11th. World Computer Congress*, pages 613–618, San Fransisco, CA, USA, August/September 1989. Elsevier Science Publishers B.V., North-Holland, 1989.
9. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. Report CS-R9120, Centre for Mathematics and Computer Science, CWI, Amsterdam, The Netherlands, 1991. A revised version will appear in *Journal of the ACM.*
10. K.M. van Hee. *Information System Engineering: a Formal Approach.* Cambridge University Press, Cambridge, UK, 1994.

11. K. Jensen. *Coloured Petri Nets. Basic concepts, analysis methods and practical use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.

12. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science : An EATCS Series*. Springer-Verlag, Berlin, 1985.

13. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness, p. 24
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doombos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks		On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein		Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch		Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken		Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld		Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma R. Gerth W. Janssen S. Katz M. Poel C. Rump	J. Davies S. Graf B. Jonsson G. Lowe A. Pnueli J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben		Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt		The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt		On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group		Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra		Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers		A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller		Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper		Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek		A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien		The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote		Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef		Concrete process algebra, p. 134.
95/04	J. Hidders		An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi		A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts		A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen		Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo		Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra		Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber		Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14
95/30	J. Hidders, C. Hoskens, J. Paredaens	The Formal Model of a Pattern Browsing Technique, p.24.
95/31	P. Kelb, D. Dams and R. Gerth	Practical Symbolic Model Checking of the full μ-calculus using Compositional Abstractions, p. 17.
95/32	W.M.P. van der Aalst	Handboek simulatie, p. 51.
95/33	J. Engelfriet and JJ. Vereijken	Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
95/34	J. Zwanenburg	Record concatenation with intersection types, p. 46.
95/35	T. Basten and M. Voorhoeve	An algebraic semantics for hierarchical P/T Nets, p. 32.
96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.