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. To illustrate the relevance of these results, the application to workflow management is demonstrated.

Keywords: Object orientation, Petri nets, Inheritance, Workflow management, 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 [16] and OOD [6], 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 [15]) 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 [11,12]. 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 of an object of the class *person*. The creation of an object is modeled by a transition with a $\bigtriangledown$ label. Firing this transition corresponds to the birth of a

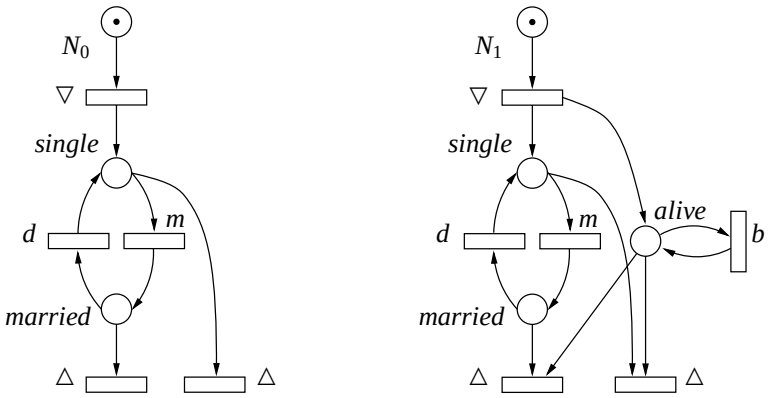

Fig. 1. Two object life cycles.

person. Firing one of the two transitions with a $\triangle$ 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 distribution 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, $\triangledown$ (object creation), $\triangle$ (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 comprises 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. Existing object-oriented methodologies such as OMT [16] and OOD [6] 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 [5], we investigate this problem using a simple ACP-like process algebra (Algebra of Communicating Processes, [3]). In that paper, *encapsulation* and *abstraction* turn out to be important concepts for the characterization of life-cycle inheritance. Based on

these concepts, four inheritance relations are defined. The process-algebraic characterization of life-cycle inheritance in [5] is rather straightforward because encapsulation and abstraction are well investigated in process algebra and because, in contrast to state-transition diagrams and Petri nets, states are not represented explicitly. It is quite easy to show that the inheritance relations have a number of desirable properties. In practice, however, state-transition diagrams are often used 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. One basic form of inheritance, called *protocol inheritance*, is based on blocking or encapsulating methods. The second form, called *projection inheritance*, is based on hiding methods by means of abstraction. The other two inheritance relations are combinations of the two basic ones. The intuition behind all four inheritance relations is that the behavior of a subclass must, in some sense, extend the behavior of its superclass, while preserving the important properties of the superclass. Depending on how strict we interpret this requirement, we get different inheritance relations. A very strict interpretation is the substitutability principle of Wegner and Zdonik [17] which says that an object of some subclass can always be used in a context where an object of its superclass is expected. As we will see, protocol inheritance adheres to this principle. However, in [14], Lakos informally discusses inheritance of dynamic behavior in a practical context. He investigates four case studies and concludes that substitutability is in practice often too strong a requirement. In his opinion, allowing to create a subclass by extending the life cycle of an object by inserting behavior in between two parts of the life cycle seems necessary. Our notion of projection inheritance allows such extensions. Identifying forms of inheritance with different characteristics, allows users to choose the appropriate form of inheritance, depending on the requirements of a design problem. To assist system designers, 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. To demonstrate the usefulness of the results presented in this paper, we focus on the application of our inheritance concepts in a workflow management setting.

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.

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. Note that any set of elements from A also denotes a unique bag over 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 X to some domain $D \subseteq A$, denoted $X \restriction D$, is defined as $[a^{X(a)} \mid a \in D]$. 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 a 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 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 is a suitable equivalence [9,10].

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 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 $\mathcal{R}$ such that $n\mathcal{R}m$. Intuitively, two nets are branching bisimilar if the environment cannot detect any differences by just observing the transitions with a label not equal to τ. Therefore, the *observable behaviors* of two nets are equivalent if and only if the nets are branching bisimilar.

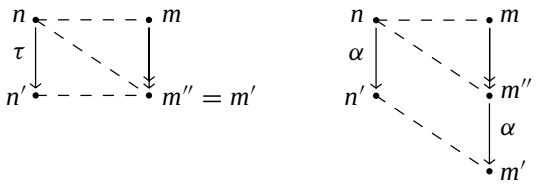

Fig. 2. Branching bisimulation.

The first two requirements in Definition 2.6 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.

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 $\triangledown$ for object creation, $\triangle$ 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., $L_s = L \cup \{\triangledown, \triangle, \tau\}$. A Petri net which specifies a life cycle has exactly one transition $t_\triangledown$ which corresponds to the creation of an object and bears a $\triangledown$ label. For convenience, we assume that $t_\triangledown$ 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 $\triangle$ label is executed. Between the creation and the termination of an object, 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 $\triangle$-labeled transition results in an empty Petri net indicating that the object has ceased to exist. 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 notion 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 $t_\triangledown$ such that ${}^\bullet i = \emptyset$, $i^\bullet = \{t_\triangledown\}$ and ${}^\bullet t_\triangledown = \{i\}$. Moreover, $t_\triangledown$ is the only transition in T such that $\ell(t_\triangledown) = \triangledown$. For any transition $t \in T$, ${}^\bullet t \neq \emptyset$.

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

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

- For any $t \in T$ such that $(N, s')[t\rangle$, $(N, s')[\ell(t)\rangle\,(N, \mathbf{0}) \Leftrightarrow \ell(t) = \triangle$. Hence, there is a one-to-one correspondence between the termination of an object and firing a $\triangle$-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. *A 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 $\triangle$. However, we can also execute σ from (N, s_2). In this case, the label of the final firing is still $\triangle$ 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? To answer this question, we have to establish an inheritance relation for object life cycles. Inspired by the process-algebraic concepts of encapsulation and abstraction, two basic forms of inheritance seem to be appropriate [5]. The first one 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 also 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. This corresponds to the encapsulation operator known from ACP. This form of inheritance is referred to as *protocol inheritance* because the subclass inherits the protocol of the superclass. By definition, it conforms to the substitutability principle. It is easy to verify that $(N_1, [i])$ in Figure 1 is a subclass of $(N_0, [i])$ under protocol inheritance. If 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 when willing to call 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 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 abstraction turning actions into internal τ actions. 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 [5], 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 on 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 [4]. It remains to show that for any two marked 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. It is also possible to combine these two basic definitions. *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 under protocol/projection inheritance if and only if it is a subclass under protocol inheritance *and* under projection inheritance. The disjunction of the two basic forms of inheritance does not yield an interesting inheritance relation. It lacks desirable properties as transitivity. However, it is possible to state that for every method new to a subclass 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, its superclass, under 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 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 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

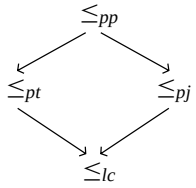

Fig. 3. An overview of life-cycle-inheritance relations.

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 [5], 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.

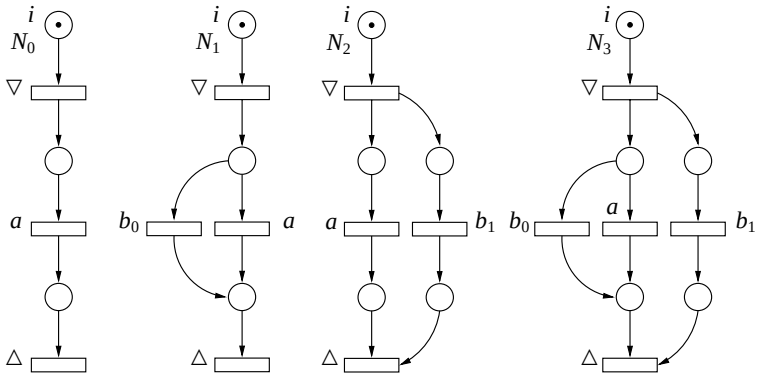

Fig. 4. $(N_1, [i]) \leq_{pt} (N_0, [i])$, $(N_2, [i]) \leq_{pj} (N_0, [i])$ and $(N_3, [i]) \leq_{lc} (N_0, [i])$.

Example 4.4. The four object life cycles shown in Figure 4 illustrate the above inheritance relations. $(N_1, [i])$ is a subclass of $(N_0, [i])$ under protocol inheritance, because removing 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 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 blocking 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 them 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 [5]). 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 [5] 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) *For any two 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. It 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

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 $t \in T_p \cap T_q$, $\ell_p(t) = \ell_q(t)$.

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.

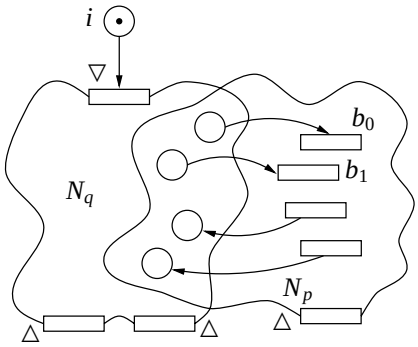

Fig. 5. Protocol-inheritance-preserving transformation rule.

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, namely*

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 $(N, [i]) \leq_{pt} (N', [i])$.

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 remove 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 after encapsulating the transitions with a new label. Let $\mathcal{R}$ be the relation $\{(\partial_H(N, u), (N', u)) \mid \partial_H(N, [i])[*\rangle \partial_H(N, u) \wedge (N', [i])[*\rangle (N', u)\}$. 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 life cycle is deactivated. In Figure 5, b_0 and b_1 operate as guards. This transformation rule shows that extending the behavior of an object with the option to choose at some point an alternative behavior yields a subclass under protocol inheritance. It follows from this rule that $(N_1, [i])$ in Figure 4 is a subclass of $(N_0, [i])$.

The transformation rule described by Theorem 5.1 is inspired by an axiom presented in [5]. To show the relation between the transformation rules in this paper and some of the algebraic rules in [5], we give an intermezzo for those familiar with process algebra.

Intermezzo 5.2. In [5], we presented an algebraic theory for studying life-cycle inheritance. In this context, an object life cycle is a closed term in the algebra. Analogous to Definition 4.3, we defined four forms of inheritance. For example, for any two object life cycles p and q, $p \leq_{pt} q$ if and only if $\partial_H(p) = q$ is derivable from the algebraic axioms.

We also presented a number of rules illustrating under what conditions inheritance is preserved. These rules form the basis for the transformation rules in this paper. Let L, L_s, and α be defined analogously to the definitions in this paper. Let p, q, q_0, q_1, and r be

closed terms and a and b actions in L_s such that $\alpha(r) \subseteq L \setminus (\alpha(q) \cup \alpha(q_0) \cup \alpha(q_1) \cup \{a\})$ 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

Axiom PT corresponds to 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 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 formulate transformation rules on Petri nets corresponding to $PJ3$, $PJ1$, and PP. Although it is possible to define a transformation rule corresponding to $PJ2$, we will not do so, because the duplication of q_0 is not very meaningful in the context of Petri nets. □**(Intermezzo)**

The second transformation rule on Petri nets corresponds to rule $PJ3$ and is illustrated in Figure 6. Extending 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 ([7]) and (4) the result is still a life cycle, yields a subclass under 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. The third requirement is useful in the proof of the following theorem. Although it is sufficient, it might be possible to relax this requirement. It is interesting to find out from practical examples whether this would be useful.

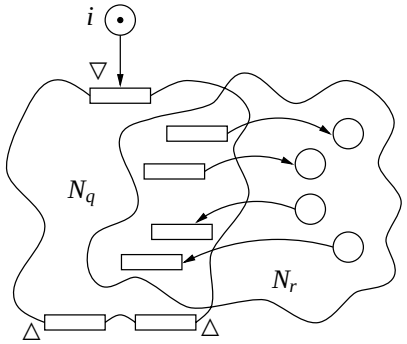

Fig. 6. Projection-inheritance-preserving transformation rule.

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 object life cycles. If the following additional properties are satisfied, namely*

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 $(N, [i]) \leq_{pj} (N', [i])$.

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

i) Let u' be such that $\tau_I(N, u)\, [\alpha\rangle\, \tau_I(N, u')$. We must check 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 firing a transition in $T_r \setminus T_q$, then $\alpha = \tau$ and $v' = v$. Otherwise, there is a transition in T_q labeled α such that $(N', v)\, [\alpha\rangle\, (N', u' \restriction P_q)$.

ii) Let v' be such that $(N', v)\, [\alpha\rangle\, (N', v')$. It must be shown that there are states u', 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 it is easy to verify that u', u'' as above exist. Assume t is not enabled in $\tau_I(N, u)$. We have to prove that t can be enabled without firing transitions in N_q. It follows from the assumptions that ${}^\bullet t \cap P_r \neq \emptyset$ 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 in P_q contains a token (t is enabled in (N', v) and $u \restriction P_q = v$) and because of the third requirement above all other transitions in T_q consuming from this place will never be able to fire. However, this means that $\tau_I(N, u)$ is not a life cycle because it cannot terminate properly. Hence, it must be possible to enable t in $\tau_I(N, u)$ without firing transitions in N_q. It is now straightforward to verify that there must be states u', 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 $\tau_I(N, u') \mathcal{R} (N', v')$.

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 $\triangle$-labeled transition. 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 $u \restriction P_q = \mathbf{0}$ and $u \neq \mathbf{0}$. Hence, $u = \mathbf{0} \Leftrightarrow v = \mathbf{0}$.

□

The transformation rule of Theorem 5.3 can be used to show that $(N_2, [i])$ in Figure 4 is a subclass of $(N_0, [i])$.

There are many other transformation rules which preserve some kind of inheritance. Figure 7 shows a transformation rule inspired by the algebraic axiom $PJ1$. It shows that new behavior may be inserted between parts of a 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 rule shown in Figure 7 boils down to the replacement of an arc by an entire net. The third requirement in the following theorem guarantees that every token that transition t_* would normally have put into place p_* eventually indeed appears in p_* by firing only transitions of N_r.

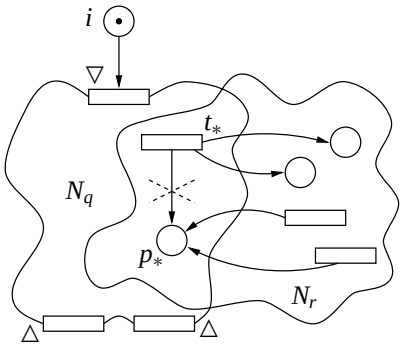

Fig. 7. Another projection-inheritance-preserving transformation rule.

Theorem 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, then* $(N, [i]) \leq_{pj} (N', [i])$.

Proof. (sketch) We have to prove that $\tau_I(N, [i]) \sim_b (N', [i])$ with $I = \alpha(N_r) \setminus \alpha(N_q)$. Let $\mathcal{R}$ be the relation $\{(\tau_I(N, u), (N', v)) \mid \tau_I(N, [i])\ [*\rangle\ \tau_I(N, u) \wedge (N', [i])\ [*\rangle\ (N', v) \wedge u \lceil (P_q \setminus \{p_*\}) = v \lceil (P_q \setminus \{p_*\}) \wedge v \lceil \{p_*\}$ is a home-marking of $(N_r', u \lceil P_r)\}$. Using the theory of free-choice nets (Home-marking theorem, liveness and boundedness results, see [7]), in combination with the life-cycle properties of Definition 3.1, it is possible to show that $\mathcal{R}$ satisfies the three requirements stated in Definition 2.6. □

Example 5.5. Figure 8 illustrates the three transformation rules presented thus far. The first transformation rule can be used to prove that $(N_1, [i]) \leq_{pt} (N_0, [i])$. The second rule proves that $(N_2, [i]) \leq_{pj} (N_1, [i])$. Applying the third transformation rule shows that $(N_3, [i]) \leq_{pj} (N_2, [i])$. 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 transformation rule which preserves protocol *and* projection inheritance. This transformation rule corresponds to the algebraic rule *PP* and is illustrated in Figure 9. It shows that under protocol/projection inheritance, it is allowed to *postpone* behavior. When a token appears in place p_*, it is possible to iterate the behavior of N_r an arbitrary number of times before continuing with the original behavior. The third requirement in the theorem below plays a similar role as the equivalent requirement of the previous transformation rule. It guarantees that eventually all tokens consumed from place p_* by transitions of N_r are returned.

Theorem 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:*

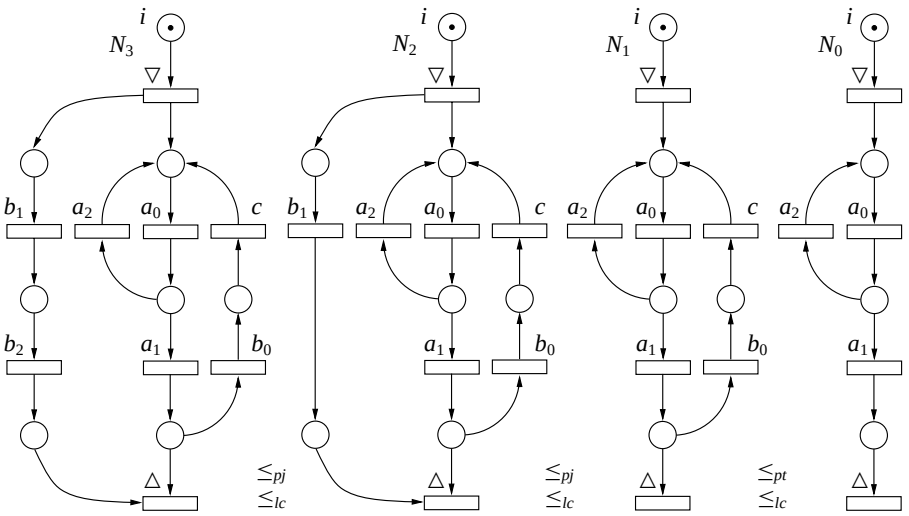

Fig. 8. The application of the three transformation rules leads from $(N_0, [i])$ to $(N_3, [i])$ while preserving life-cycle inheritance.

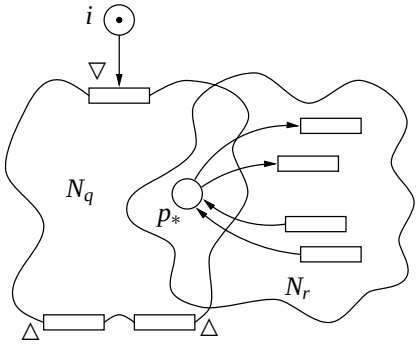

Fig. 9. A protocol/projection-inheritance-preserving transformation rule.

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 of Definition 3.1, then* $(N, [i]) \leq_{pp} (N', [i])$.

Proof. (Sketch) Since the conditions of Theorem 5.1 are satisfied, protocol inheritance is preserved. It remains to prove that $\tau_I(N, [i]) \sim_b (N', [i])$ with $I = \alpha(N_r)$. Let $\mathcal{R}$ be the relation $\{(\tau_I(N, u), (N', v)) \mid \tau_I(N, [i])\ [*\rangle\ \tau_I(N, u) \wedge (N', [i])\ [*\rangle\ (N', v) \wedge u \upharpoonright (P_q \setminus \{p_*\}) = v \upharpoonright (P_q \setminus \{p_*\}) \wedge v \upharpoonright \{p_*\}$ is a home-marking of $(N_r, u \upharpoonright P_r)\}$. As in Theorem 5.4, the theory of free-choice nets (Home-marking theorem, liveness and boundedness results, see [7]), in combination with the life-cycle properties of Definition 3.1, can be used to show that $\mathcal{R}$ satisfies the three requirements stated in Definition 2.6. □

The four transformation rules presented in this section give a good characterization of the various forms of inheritance. In contrast to [5], 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 [5] correspond to elegant transformation rules in a Petri-net context is encouraging. It appears that the inheritance concepts presented in this paper are quite universal and transcend the two formalisms.

6 An Application to Workflow Management

Workflow management. To date, over 250 workflow management tools are commercially available. These tools will have a tremendous impact on the way information systems are being developed. By using a *Workflow Management System* (WFMS), one is able to develop information systems which can easily be adapted to a changing environment ([13]). Traditionally, an information system is a collection of applications centered around a database system. This means that the business processes supported by the information system are hard-coded in the applications. As a result, the applications need to be modified the moment the underlying business process changes. Moreover, management information is missing and the processes are hard to control. By using a WFMS, one is able to push the business processes out of the applications. The WFMS can take care of controlling, monitoring and executing the business processes. Applications are started by the WFMS and are unaware of the process they contribute to. By reconfiguring the WFMS, one can support new and/or modified processes. Processes supported by a WFMS are *case based*, i.e., the primary objective is to handle cases (e.g. purchase orders, insurance claims, proposal forms, loan requests or tax papers) efficiently. Each case is handled by the WFMS by executing a sequence of tasks. The *routing* of cases (i.e., the partial ordering of tasks which need to be executed) is specified by a *process definition*. Iteration, sequential, parallel and selective routing are the basic ingredients of a process definition. There are two types of tasks; (1) tasks which require a trigger and (2) task which do not require a trigger. Tasks which do not require a trigger correspond to fully automatic tasks which are to be executed as soon as possible. The other tasks require a trigger before execution. Examples of triggers are: documents, electronic messages, time triggers and user triggers. Interactive tasks require a user trigger; the WFMS cannot force a user to execute a task.

As we will see, a workflow process definition corresponds to an object life cycle. The process definition specifies the life cycle of a case. Cases correspond to objects, tasks correspond to methods and triggers correspond to method invocation. Tasks which do not require a trigger correspond to internal methods. If a task requires a trigger, it corresponds to an external method. Given the similarities between workflow process definitions and object life cycles, it is interesting to see whether the results presented in this paper can be applied to workflow management. Moreover, the techniques used in most of the available WFMS's are close to the Petri-net-based technique used in this paper ([2,8]). Some of the WFMS's are actually based on Petri nets (e.g. COSA by Software-Ley). Most of the other tools provide a subset of the Petri-net functionality (e.g. state transition diagrams) and/or use an ad-hoc technique which can be mapped onto Petri nets. A WFMS supports changes to the underlying business process. However, in many

situations it is not possible to replace a process by an arbitrary new process. Often the modified process is an extension of the old process. Clearly, this corresponds to some form of inheritance. There are several reasons for the application of inheritance concepts in a workflow management context.

- If a process definition is replaced by a new one, it is important that the existing cases can be handled the 'old way'. If the new process definition is a subclass of the old one with respect to some form of inheritance, this is possible by blocking and/or executing new tasks.
- Consider the situation where a number of departments has to share some common process definition. However, each department may add some extra tasks that are executed locally. If each of the local process definitions is a subclass of some common superclass process definition, then the superclass may provide handles to manage, share and exchange cases.
- The user of the WFMS is familiar with the existing processes. If a process is replaced by a new one, it is important to identify the differences. By establishing an inheritance relation, it becomes clear what the differences are and how the new process can be used to handle things the old way.

Note that a process can be modified for a single case (ad-hoc workflow) or an entire class of cases (production workflow).

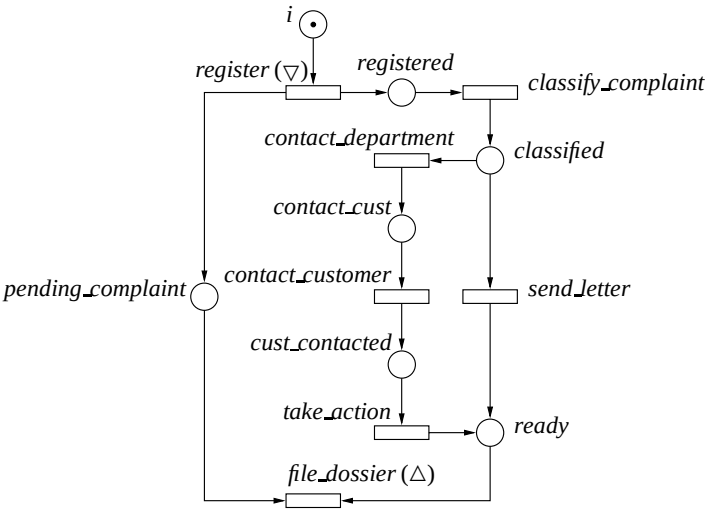

Fig. 10. The life cycle of a complaint.

An example. To illustrate the use of life-cycle inheritance in the context of workflow management, we consider the complaints desk of a fictitious Company X. The complaints desk handles complaints of customers about the products produced by Company X. Each complaint is registered before it is classified. Depending on the classification of the complaint, a letter is sent to the customer or an inquiry is started. The inquiry starts

with a consultation of the department involved, followed by a discussion with the customer. Based on this inquiry, the necessary actions are taken. Finally, the dossier is filed. Figure 10 shows the process definition which is used to configure the WFMS used by the employees of the complaints desk. It is easy to see that the Petri net represents an object life cycle.

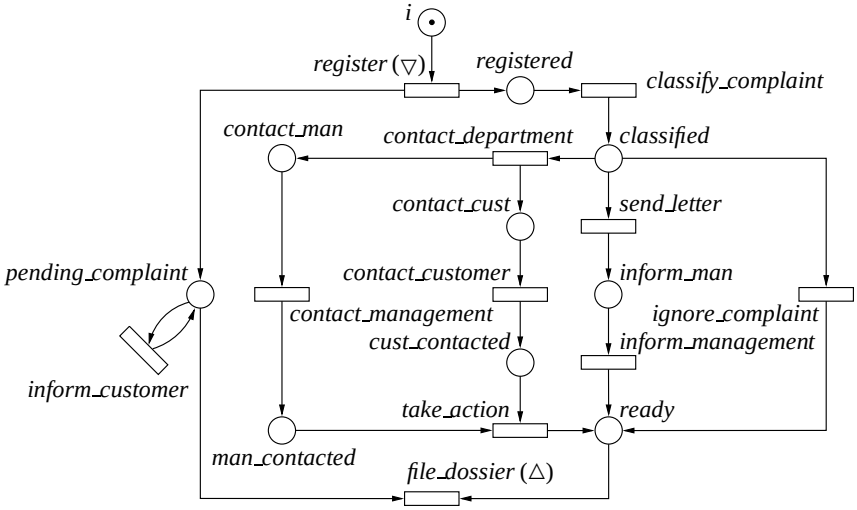

Fig. 11. The extended life cycle of a complaint.

The management of Company X is not satisfied with the processing of complaints. Therefore, the WFMS has to be reconfigured to support an extended complaint handling process. The process definition shown in Figure 11 is used to reconfigure the WFMS. Since the new process should inherit some of the properties of the old process, it is interesting to establish a life-cycle-inheritance relationship between the two processes. Transition *ignore_complaint* has been added for complaints that make no sense, for example complaints without the name and address of the complainant. The addition of this task preserves protocol inheritance, as we can show by applying Theorem 5.1. Transition *contact_management* has been added because, in the new process, the management of the department wants to be kept informed. Contacting the management and the customer can be done in parallel. We can use Theorem 5.3 to show that the addition of the *contact_management* task preserves projection inheritance. Another addition is that in the new process, after sending a letter to the complainant a new task is performed, namely *inform_management*. Theorem 5.4 can be applied to prove that the addition of *inform_management* also preserves projection inheritance. At any time during the processing of a complaint, the complainant may request information about the processing of the corresponding complaint. Therefore, the transition *inform_customer* has been added. By applying Theorem 5.6, it is not difficult to see that the addition of this task preserves protocol and projection inheritance. Since each of the extensions preserves protocol and/or projection inheritance, the process definition shown in Figure 11 is a subclass of the process definition shown in Figure 10 with respect to life-cycle inheritance.

7 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 system developers engaged in object-oriented design.

Four inheritance relations have been presented, inspired by the process-algebraic concepts of encapsulation and abstraction [5]. Encapsulation corresponds to *blocking* method calls, whereas abstraction conforms to *hiding* method calls. By identifying four forms of inheritance with different characteristics, designers can choose approriate inheritance relations at design time. It has been shown that in principle all four inheritance relations can be checked automatically.

Furthermore, 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. They capture four basic constructs to build a subclass from a superclass, namely choice, parallel composition, sequential composition, and iteration. Since these constructs are fundamental to any (concurrent) object-oriented language, this is an important reason why we believe that encapsulation and abstraction capture the essence of inheritance of dynamic behavior. The transformation rules make it possible to create subclasses of some given class in a constructive way at design time, making static analysis of subclass relations unnecessary. In this way, the transformation rules support the *reuse* of life-cycle specifications during a design.

As an example of the applicability of the concepts presented in this paper, a simple problem from the area of workflow management has been discussed. This example shows that the notion of inheritance of dynamic behavior is useful in the adaptation of business processes to new requirements while maintaining the important properties of the old processes.

A future challenge is the validation of our approach by applying it to non-trivial object-oriented design problems. For this purpose, it is needed to incorporate the results either in a full fledged object-oriented language based on Petri nets or to translate them to an existing language, possibly based on another formalism for specifying life cycles. It is also interesting to investigate in more detail the applicability of the results in the context of workflow management.

Acknowledgements. The authors would like to thank Marc Voorhoeve and the anonymous referees 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 Report 95/26, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, 1995.
2. W.M.P. van der Aalst. Petri-Net-based Workflow Management Software. In A. Sheth, editor, *Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems*, pages 114–118, Athens, Georgia, USA, May 1996.
3. J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1990.
4. T. Basten. Branching Bisimilarity is an Equivalence indeed! *Information Processing Letters*, 58(3):141–147, May 1996.

5. T. Basten and W.M.P. van der Aalst. A Process-Algebraic Approach to Life-Cycle Inheritance: Inheritance = Encapsulation + Abstraction. Computing Science Report 96/05, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, March 1996.
6. G. Booch. *Object-Oriented Analysis and Design: With Applications*. Benjamin/Cummings, Redwood City, CA, USA, 1994.
7. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.
8. C.A. Ellis and G.J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993, 14th. International Conference, Proceedings*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16, Chicago, USA, June 1993. Springer–Verlag, Berlin, Germany, 1993.
9. 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.
10. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
11. K.M. van Hee. *Information Systems Engineering: A Formal Approach*. Cambridge University Press, Cambridge, UK, 1994.
12. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1, *Basic Concepts*. EATCS monographs on Theoretical Computer Science. Springer–Verlag, Berlin, Germany, 1992.
13. T.M. Koulopoulos. *The Workflow Imperative*. Van Nostrand Reinhold, New York, USA, 1995.
14. C.A. Lakos. Pragmatic Inheritance Issues for Object Petri Nets. In *TOOLS Pacific 1995, Proceedings*, pages 309–321, Melbourne, Australia, 1995. Prentice-Hall.
15. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS monographs on Theoretical Computer Science*. Springer–Verlag, Berlin, Germany, 1985.
16. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.
17. P. Wegner and S.B. Zdonik. Inheritance as an Incremental Modification Mechanism or What like is and isn't like. In S. Gjessing and K. Nygaard, editors, *ECOOP '88, European Conference on Object-Oriented Programming, Proceedings*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77, Oslo, Norway, August 1988. Springer–Verlag, Berlin, Germany, 1988.