Inheritance of Dynamic Behavior in UML

W.M.P. van der Aalst

Eindhoven University of Technology, Faculty of Technology and Management, Department of Information and Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tm.tue.nl

Abstract. One of the key issues of object-oriented modeling and design is inheritance. It allows for the definition of subclasses that inherit features of some superclass. Inheritance is well defined for static properties of classes such as attributes and operations. However, there is no general agreement on the meaning of inheritance when considering the dynamic behavior of objects, captured by their life cycles. This paper studies inheritance of behavior in the context of UML. This work is based on a theoretical framework which has been applied and tested in both a process-algebraic setting (ACP) and a Petri-net setting (WF-nets). In this framework, four inheritance rules are defined that can be used to construct subclasses from (super-)classes. These rules and corresponding techniques and tools are applied to UML activity diagrams, UML statechart diagrams, and UML sequence diagrams. It turns out that the combination of blocking and hiding actions captures a number of important patterns for constructing behavioral subclasses, namely choice, sequential composition, parallel composition, and iteration. Both practical insights and a firm theoretical foundation show that our framework can be used as a stepping-stone for extending UML with inheritance of behavior.

1 Introduction

The Unified Modeling Language (UML) [11, 21] has been accepted throughout the software industry as the standard object-oriented framework for specifying, constructing, visualizing, and documenting software-intensive systems. One of the main goals of object-oriented design is the *reuse* of system components. A key concept to achieve this goal is the concept of *inheritance*. The inheritance mechanism allows the designer to specify a class, the *subclass*, that inherits features of some other class, its *superclass*. Thus, it is possible to specify that the subclass has the same features as the superclass, but that in addition it may have some other features.

The concept of inheritance is usually well defined for the *static structure* of a class consisting of the set of operations (methods) and the attributes. However, as mentioned, a class should also describe the dynamic behavior of an object. We will use the term "object life cycle" to refer to this behavior. The current version of UML, Version 1.4 [11], supports nine types of diagrams: class diagrams, object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. Four of these types of diagrams, namely sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams capture (part of) the behavior of the modeled system. Sequence diagrams and collaboration diagrams typically only model examples of interactions between objects (scenarios). Activity diagrams emphasize the flow of control from activity

to activity, whereas statechart diagrams emphasize the potential states and the transitions among those states. Both statechart diagrams and activity diagrams can be used to specify the dynamics of various aspects of a system ranging from the life cycle of a single object to complex interactions between societies of objects. Activity diagrams typically address the dynamics of the whole system including interactions between objects. Statechart diagrams are typically used to model an object's life cycle. Note that UML joins and/or is inspired by the earlier work on Message Sequence Diagrams [20] (for sequence diagrams), Statecharts [13] (for statechart diagrams), and Petri nets [19] (for activity diagrams).

Looking at the informal definition of inheritance in UML, it states the following: "The mechanism by which more specific elements incorporate structure and behavior defined by more general elements." [21]. However, only the class diagrams, describing purely structural aspects of a class, are equipped with a concrete notion of inheritance. It is implicitly assumed that the behavior of the objects of a subclass is an extension of the behavior of the objects of its superclass. Clearly, this is not sufficient to realize the full potential of inheritance [8, 14, 22, 23]. Therefore, our ultimate quest is to extend each diagram type of UML with suitable notions of inheritance. For this purpose we use theoretical results presented in [1–3, 5, 6] as a stepping stone. These results provide four notions of behavioral inheritance, inheritance preserving transformation rules, transfer rules, and advanced notions such as the Greatest Common Divisor (GCD) of a set of behavioral models.

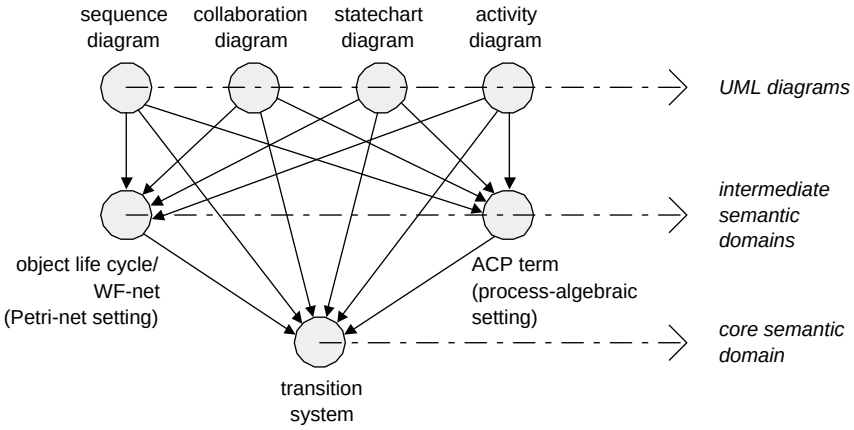

Fig. 1. Mapping UML behavior diagrams onto three semantic domains.

In this paper we follow the approach proposed in [8], i.e., instead of trying to give full formal semantics for UML we focus on selected parts of UML and map these parts onto a so-called *semantic domain*. A semantic domain is some formal language allowing for a precise definition of inheritance and equipped with analysis techniques to verify whether one process is a subclass of another process. Based on the theoretical

results presented in [1–3, 5, 6] we can use three semantic domains. This is illustrated in Figure 1. The *core semantic domain* is formed by transition systems using branching bisimilarity as an equivalence relation [6]. The mapping from specific models in both a Petri-net and process-algebraic setting to transition systems is given in [1–3, 5, 6]. In this paper, we explore the mapping from UML behavior diagrams to these semantic domains in order to incorporate inheritance of behavior in UML. A direct mapping from sequence, collaboration, statechart, and activity diagrams to the core semantic domain (i.e., transition systems) allows for full flexibility. An indirect mapping through one of the *intermediate semantic domains* (i.e., WF-nets and object life cycles in the Petri-net setting and ACP terms in the process-algebraic setting) allows for powerful analysis techniques, cf. the structure theory of Petri nets (e.g., invariants) and the equational theory of ACP. Moreover, the UML behavior diagrams are closely related to the two intermediate semantic domains. Consider for example the relation between activity diagrams and Petri nets.

Note that it is not our goal to provide a precise semantics for UML. This topic is relevant and has be debated many times before [7, 9], but is outside the scope of this paper.

The remainder of this paper is organized as follows. First, the theoretical results presented in [1–3, 5, 6] are introduced. Then, the four notions of inheritance are applied to sequence diagrams (Section 3), statechart diagrams (Section 4), and activity diagrams (Section 5). To conclude, we provide pointers to related work and summarize the main results.

2 Inheritance of behaviour

The goal of this section is to introduce four notions of inheritance and highlight some of the results theoretical results presented in [1–3, 5, 6]. Some of these results have been developed in a Petri-net setting (cf. [1–3, 5, 6]) others have been developed in a process-algebraic setting (cf. [5, 6]). However, the main ideas are generic and can be applied to any of the behavior diagrams in UML (i.e., sequence diagrams, activity diagrams, collaboration diagrams, and statechart diagrams). Therefore, we present the intuition behind our results and demonstrate their applicability to sequence diagrams, activity diagrams, and statechart diagrams.

2.1 An informal introduction to the four notions of inheritance

Diagrams such as sequence diagrams, activity diagrams, and statechart diagrams specify behavior of an object or system. The most elementary way of modeling behavior is the so-called *labeled transition system*. We consider this to be the core semantic domain (cf. Figure 1). A labeled transition system is a set of *states* plus a *transition relation* on states. Each transition is labeled with an *action*. Figure 2 shows a simple transition system representing an order processing system with five states (*s1*, *s2*, *s3*, *s4*, and *s5*). There are five transitions each labeled with a different action. The transition labeled with *accept_order* moves the system from the state *s1* to the state *s2*. For simplicity we assume that each transition system has one initial state (e.g., *s1* in Figure 2) and one

final state (e.g., *s5* in Figure 2). Note that any transition system with multiple initial and/or final states can be transformed in a transition system with one initial and one final state.

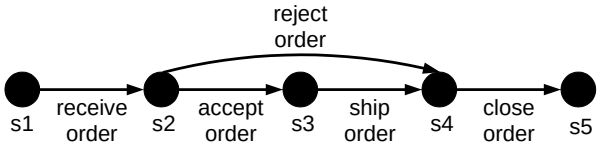

Fig. 2. A labeled transition system specifying an order processing system (TS_1).

States, transitions, and actions of a transition system should not be confused with the meaning of these notions in UML. In the context of this paper, the interpretation of these concepts depends on the UML diagrams being considered. For example, in a UML statechart diagram with a composite state consisting of multiple concurrent substates the composite state may correspond to many states in the corresponding labeled transition system (all possible combinations of states of the concurrent substates). In a UML sequence diagram, sending a message (i.e., a stimulus) corresponds to the execution of transition having the appropriate action label. In a UML activity diagram, there are action states which correspond to actions in the corresponding labeled transition system. In this section, we use the terms state, transition, and action in the context of a labeled transition system. In subsequent sections we will put these concepts in the context of UML diagrams specifying behavior.

We distinguish between *visible* actions and *invisible* actions. For comparing the behavior of two labeled transition systems only the visible actions are considered. The distinction between visible and invisible actions is fairly standard in process theory (see [6, 10] for pointers). Since it is not possible to distinguish between individual invisible actions (also referred to as silent actions), these actions are labeled τ. Two labeled transition systems are considered equivalent if their observable behaviors coincide, i.e., after abstracting from τ-actions one cannot detect any differences. From a formal point of view, branching bisimulation [6, 10] is used as an equivalence relation. However, we will avoid getting into formal definitions. Instead we refer to [1–3, 5, 6] for details.

In this paper, we focus on inheritance of dynamic behavior. Translated to labeled transition systems this translates to the following question: *When is one labeled transition system a subclass of another labeled transition system?* There seem to be many possible answers to this question. It is important to note that we have to ask this question from the viewpoint of the *environment*.

Assume that p and q are two labeled transition systems. The first answer is as follows.

> *If it is not possible to distinguish the external behavior of p and q when only actions of p that are also present in q are executed, then p is a subclass of q.*

Intuitively, this basic form of inheritance conforms to *blocking* actions new in p. In the remainder, labeled transition system p is said to inherit the *protocol* of q; the resulting

fundamental form of inheritance is referred to as *protocol inheritance*. Note that protocol inheritance specifies a *lower bound* for the behavior offered, i.e., any sequence of actions invocable on the superclass can be invoked on the subclass. Therefore, it is sometimes also referred to as "invocation consistency" [8, 22, 23].

The second answer to the above question is as follows.

> *If it is not possible to distinguish the external behavior of p and q when arbitrary actions of p are executed, but when only the effects of actions that are also present in q are considered, then p is a subclass of q.*

This second basic form of inheritance of behavior conforms to *hiding* the effect of actions new in p. Transition system p inherits the *projection* of transition system p onto the actions of q; the resulting form of inheritance is called *projection inheritance*. Note that projection inheritance can be considered as an *upper bound* for the behavior offered, i.e., any sequence of actions observable from the subclass should correspond to an observable sequence of the superclass (after abstraction). Therefore, it is sometimes also referred to as "observation consistency" [8, 22, 23].

To illustrate these two basic notions of inheritance we consider the labeled transition system specifying an order processing system shown in Figure 2. Suppose that this is the superclass named TS_1. Figure 3 shows another labeled transition system named TS_2. TS_2 is a subclass with respect to protocol inheritance because if we block the new actions, the observable behavior of TS_2 coincides with the observable behavior TS_1. If action *subcontract_order* is never executed, the original behavior is preserved. Note that in Figure 3 and subsequent figures new actions (i.e., action not appearing in Figure 2) are highlighted. Also note that TS_2 is *not* a subclass of TS_1 with respect to projection inheritance. If we hide the two new actions, there is an occurrence sequence where *receive_order* is directly followed by *close_order* without executing *reject_order* or *accept_order* and *ship_order* in-between. Clearly this behavior is not possible in TS_1.

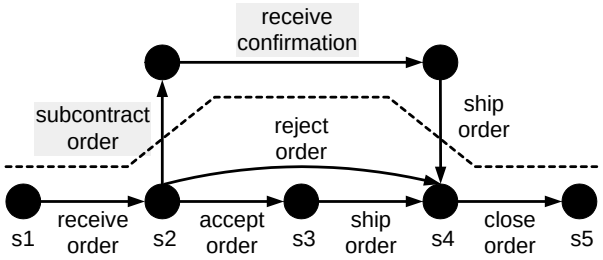

Fig. 3. A subclass with respect to protocol inheritance (TS_2).

Labeled transition system TS_3 shown in Figure 4 is a subclass with respect to projection inheritance. The new action *send_invoice* is executed in parallel with *ship_order*. (We are assuming interleaving semantics here.) If *send_invoice* is renamed to τ, then action *accept_order* is always followed by *ship_order* which in turn is followed by

close_order. Therefore, the observable behavior of TS_3 coincides with the observable behavior of TS_1 after abstracting from *send_invoice*, i.e., TS_3 is a subclass of TS_1 with respect to projection inheritance. Note that TS_3 is not a subclass of TS_1 with respect to protocol inheritance. If *send_invoice* is blocked, the process gets stuck after executing *ship_order*.

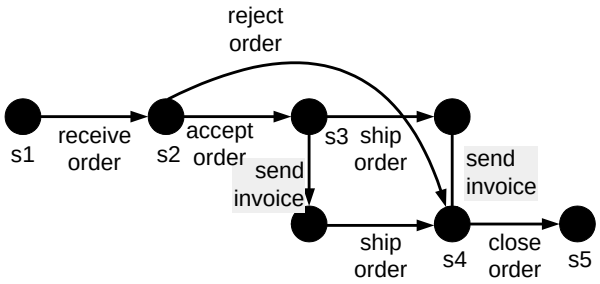

Fig. 4. A subclass with respect to projection inheritance (TS_3).

There are essentially two ways to combine the two basic notions of inheritance into stronger or weaker notions of inheritance. Protocol/projection inheritance is the most restrictive form of inheritance which combines both basic notions at the same time. If p is a subclass of q with respect to protocol/projection inheritance, then p is a subclass of q with respect to protocol inheritance *and* projection inheritance. Life-cycle inheritance is the most liberal form of inheritance: The set of new actions is partitioned into hidden and blocked such that the observable behavior of the subclass equals the behavior of the superclass. Note that protocol and/or projection inheritance implies life-cycle inheritance.

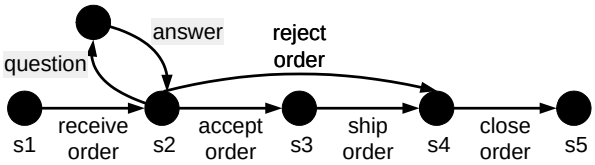

Fig. 5. A subclass with respect to protocol/projection inheritance (TS_4).

TS_4 shown in Figure 5 is a subclass of TS_1 with respect to protocol/projection inheritance because either blocking or hiding the two new actions results the observable behavior of TS_1. If action *question* is blocked, the new behavior is never activated. If actions *question* and *answer* are renamed to τ, their presence cannot be observed.

Labeled transition system TS_5 shown in Figure 6 is a subclass of TS_1 with respect to life-cycle inheritance. By blocking the new action *subcontract_order* and hiding action *send_invoice* the resulting observable behavior coincides with TS_1. Note that TS_5

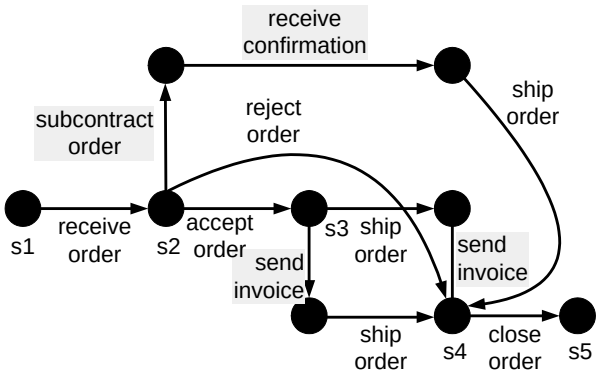

Fig. 6. A subclass with respect to life-cycle inheritance (TS_5).

is not a subclass of TS_1 with respect to any of the other three notions of inheritance. All the other notions either block and/or hide all the new actions while for life-cycle inheritance each individual new action is either blocked or hidden. Note that TS_5 is also a subclass of TS_2 with respect to projection inheritance. Moreover, TS_5 is a subclass of TS_3 with respect to protocol inheritance.

To summarize, we have identified four notions of inheritance based on two fundamental mechanisms: hiding and blocking. The most restrictive notion of inheritance is protocol/projection inheritance. Of the four transition systems TS_2, TS_3, TS_4, and TS_5 only TS_4 is a subclass of TS_1 with respect to protocol/projection inheritance. Life-cycle inheritance is the most liberal form of inheritance and each of the four transition systems TS_2, TS_3, TS_4, and TS_5 is a subclass of TS_1 with respect to life-cycle inheritance.

2.2 Inheritance preserving transformation rules and other results

Based on the four notions of inheritance, we have developed a set of tools (e.g., Woflan [24]) and obtained powerful theoretical results. These theoretical results have been presented in [1–3, 5, 6] and in this section we only highlight some of them.

In both a Petri-net and a process-algebraic setting we have developed a comprehensive set of *inheritance preserving transformation rules* (cf. Table 1). A detailed description of these rules is beyond the scope of this paper. Therefore, we give an informal description of four inheritance preserving transformation rules: PP, PT, PJ, and PJ3. Transformation rule PP is used to add loops such that each loop eventually returns to the state where it was initiated. If these loops contain only new or invisible actions, protocol/projection inheritance, and therefore also the other three notions of inheritance, are preserved. TS_4 can be constructed from TS_1 using this rule, and therefore, it automatically follows that TS_4 is a subclass of TS_1 under protocol/projection inheritance. PT preserves protocol inheritance and adds alternative behavior. TS_2 can be constructed from TS_1 using this rule. PJ and PJ3 both preserve projection inheritance. PJ can be used to insert new actions in-between existing actions. PJ3 can be used to add parallel

behavior. TS_3 can be constructed from TS_1 using rule PJ3. The four rules (PP, PT, PJ, and PJ3) correspond to design constructs that are often used in practice, namely iteration, choice, sequential composition, and parallel composition. If the designer sticks to these rules, inheritance is guaranteed. It should be noted that the precise formulation of these rules depends of the modeling language being used and is not as straightforward as it may seem (e.g., the added parts should not introduce deadlocks and terminate properly). The four inheritance preserving transformation rules have been formulated in terms of object life cycles and WF-nets (Petri-net-based modeling languages, cf. [1–3, 5, 6]) but also in terms of ACP (a process-algebraic language, cf. [5, 6]). Moreover, in the process-algebraic setting additional rules (PJ2, LC1, LC2, and LC3) have been formulated [5, 6].

name	adds	preserves
PP	loops containing new behavior	all notions of inheritance
PT	new alternatives starting with a new action	only protocol and life-cycle inheritance
PJ	new actions inserted in-between existing ones	only projection and life-cycle inheritance
PJ3	new actions in parallel with existing ones	only projection and life-cycle inheritance

Table 1. Overview of inheritance preserving transformation rules.

Based on the inheritance preserving transformation rules we have also developed a comprehensive set of *transfer rules*. These transfer rules can be used to migrate instances from a subclass to a superclass and vice versa. Suppose that p is a subclass of q constructed using the rules PP, PT, PJ, and PJ3. For any state in p it is possible to transfer an instance (e.g., an object of a class whose life-cycle is specified by transition system p) to q such that the transfer is instantaneous (i.e., no postponements needed) and does not introduce syntactic errors (e.g., deadlocks, livelocks, and improper termination) nor semantic errors (e.g., the double execution of actions or unnecessary skipping of actions). Moreover, it is also possible to transfer instances from subclass p to superclass q without any problems. Note that the transfer rules are derived from the transformation rules introduced earlier. The transfer rules to move a case to a subclass are: r_{PT}, r_{PP}, r_{PJ}, $r_{PJ3,C}$ and $r_{PJ3,P}$. The transfer rules to move a case to a superclass are: $r^{-1}_{PT,C}$, $r^{-1}_{PT,P}$, r^{-1}_{PP}, r^{-1}_{PJ}, and r^{-1}_{PJ3}. See [3] for the specification of these transfer rules in a Petri-net setting.

Each of the four inheritance relations provides an ordering on labeled transition systems and can be used to define concepts such as the GCD (Greatest Common Divisor) of two processes. The concept of GCD was introduced in [3] and a detailed analysis of this concept is given in [2]. Since none of the inheritance relations is a lattice, there is a trade-off between "uniqueness" and "existence". By using a weak notion of GCD, existence is guaranteed but there may be multiple GCD's. By using a stronger notion, existence is no longer guaranteed but if the GCD exists, it is unique. Given this tradeoff, we define the notion of Maximal Common Divisor (MCD). An MCD is a "smallest" superclass of both p and q under life-cycle inheritance. If a set of labeled transition systems is related under inheritance via subclass-superclass relationships, it is generally

quite easy to find the GCD. If this is not the case, the computation of a GCD is more involved and there are typically multiple candidates (i.e., MCD's). Similarly results hold for LCM's (Least Common Multiple) and MCM's (Minimal Common Multiple) [2, 3].

These results illustrate the strong theoretical foundation for inheritance of dynamic behavior. Unfortunately, its application has been limited to the workflow domain [3]. In this paper, we want to demonstrate that these results can also be used as a stepping stone for extending the behavior diagrams in UML (i.e., sequence diagrams, activity diagrams, collaboration diagrams, and statechart diagrams) with inheritance.

2.3 Checking inheritance using Wolfan

To illustrate the applicability of the four inheritance concepts we refer to our workflow analysis tool Woflan [24]. Woflan is based on Petri-nets and aims at the verification of workflow processes. Besides checking for deadlocks and other design errors, Woflan also supports the four notions of inheritance. Given two workflow models, Woflan is able to check whether one model is a subclass of the other model. The current version assumes that these models are expressed in terms of Petri nets. However, we have developed translations from concrete systems such as COSA (COSA Solutions/Thiel AG), Staffware (Staffware PLC), and Protos (Pallas Athena) and alternative modeling techniques such as workflow graphs and event-driven process chains [3]. Moreover, the inheritance-checker is relatively independent of the modeling language used and the results are not restricted to workflow processes but apply to any behavioral model. These practical results demonstrate the practical potential of the results presented in [1–3, 5, 6] in the context of UML.

3 Sequence diagrams

The first diagram type we consider is the UML sequence diagram. A sequence diagram has two dimensions: (1) the vertical dimension represents time and (2) the horizontal dimension represents different instances. Sequence diagrams are typically used to describe specific scenarios of interaction among objects. Although UML allows for variations such as iteration, conditional, and timed behavior, we assume that the sequence diagram is restricted to lifelines, messages (i.e., communications of type procedure call, asynchronous, and return), activation, and concurrent branching. Under these assumptions it is fairly straightforward to map a sequence diagram onto an labeled transition system (core semantic domain) or a Petri net (intermediate semantic domain) [16, 20]. In fact, under these assumptions any sequence diagram corresponds to a so-called marked graph [19], i.e., a Petri net where places cannot have multiple inputs/outputs. This observation indicates that only projection inheritance is relevant for this diagram type. (If there are no choices, it makes no sense to block behavior since this will only cause deadlocks.) As a result, only the projection-inheritance preserving transformation rules are relevant.

Consider the two sequence diagrams shown Figure 7. The right-hand side diagram is a subclass of the left-hand side diagram under projection inheritance. There are two ways to verify this. First of all, we can map both sequence diagrams onto any of the

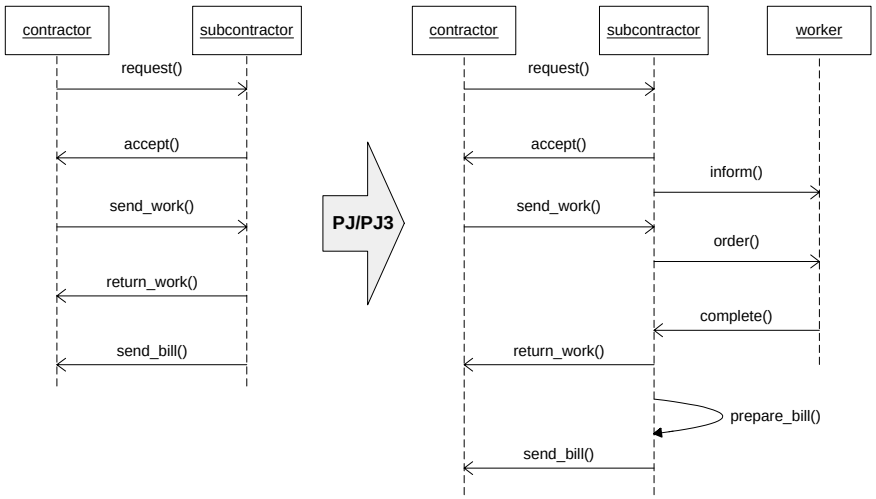

Fig. 7. A subclass sequence diagram constructed using rules PJ and PJ3.

semantics domains shown in Figure 1 and then check inheritance in the corresponding domain. Second, one can apply the projection-inheritance preserving transformation rules PJ and PJ3 described in Section 2.2. PJ can be used to add action *prepare_bill* and PJ3 can be used to add the lifeline *worker* and the actions *inform*, *order*, and *complete*. Note that in Figure 7 we assumed a one-to-one correspondence between messages and actions. Moreover, we did not use communications of type procedure call, activation, and concurrent branching. However, translation of PJ and PJ3 to sequence diagrams can easily deal with these concepts.

Collaboration diagrams are closely related to sequence diagrams. In essence they provide a different view on the identical structures [21]. Therefore, the results obtained for inheritance of sequence diagrams can easily be transferred to collaboration diagrams.

4 Statechart diagrams

In contrast to sequence diagrams, statechart diagrams are typically not used to specify scenarios. Instead they are used to model the life cycle of object. Since the 1987 paper by David Harel [13] there has been an ongoing discussion on the semantics of statecharts. Clearly, any of these semantics can be mapped onto our core semantic domain (transition systems) and thus implicitly use the four notions of inheritance described in Section 2.1. Moreover, in [8] a partial mapping onto CSP is given (we can use a similar semantic mapping to ACP) and in [22, 23] a mapping onto object behavior diagrams is given. Instead of providing yet another mapping of statechart diagrams onto some semantic domain we provide two examples to demonstrate our inheritance notions and the related transformation rules.

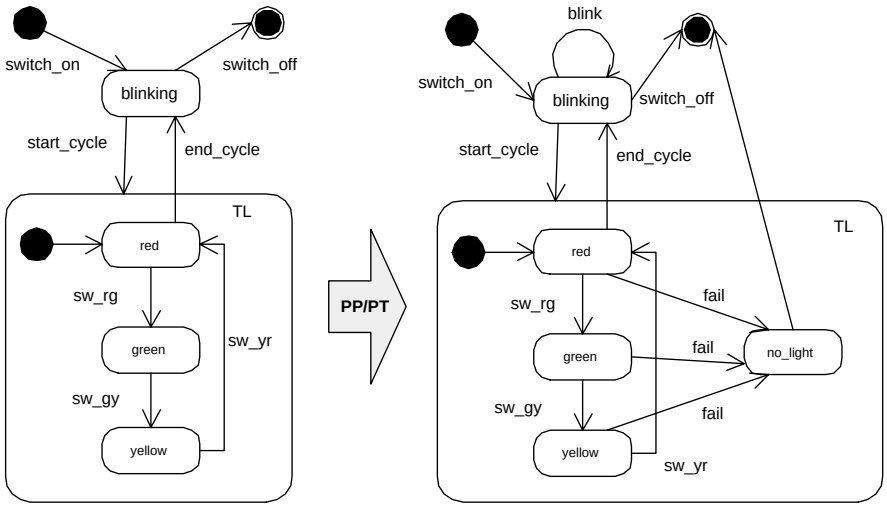

Fig. 8. A subclass statechart diagram constructed using rules PP and PT.

Figure 8 shows two statechart diagrams. The right-hand side diagram is a subclass of the left-hand side diagram under protocol inheritance. Any sequence of actions possible in the left-hand side diagram is also possible in the right-hand side diagram. The superclass statechart diagram models a Dutch traffic light which is either in state *blinking* or in composite state *TL*. The composite state is decomposed in three substates: *red*, *green*, and *yellow*. The subclass statechart diagram extends the superclass in two ways. First of all, the self transition *blink* is added. Second, the composite state is extended to allow for a traffic light which fails. State *no_light* corresponds to a malfunctioning traffic light which is shut down. The extension involving transition *blink* can be realized by applying PP, this is the protocol/projection inheritance preserving transformation rule which introduces loops which can be blocked or hidden. The extension involving state *no_light* can be realized by applying PT, this is the protocol inheritance preserving transformation rule which introduces alternatives which can be blocked.

Another example illustrating the application of our framework to statechart diagrams is given in Figure 9. The right-hand side diagram is a subclass of the left-hand side diagram under projection inheritance. The subclass statechart diagram (right) extends the superclass (left) in two ways. First of all, the traffic light has four phases instead of three including a state *red+yellow*. Second, the composite state *TL* has now two concurrent regions: one corresponding to the original traffic light with one additional phase and one corresponding to a mechanism to count the number of cars. If we abstract from this new mechanism and the additional phase, we obtain the original traffic light. Therefore, it is easy to verify that the right-hand side diagram is a subclass of the left-hand side diagram under projection inheritance. However, it is also possible to demonstrate this by applying the two projection inheritance preserving transformation

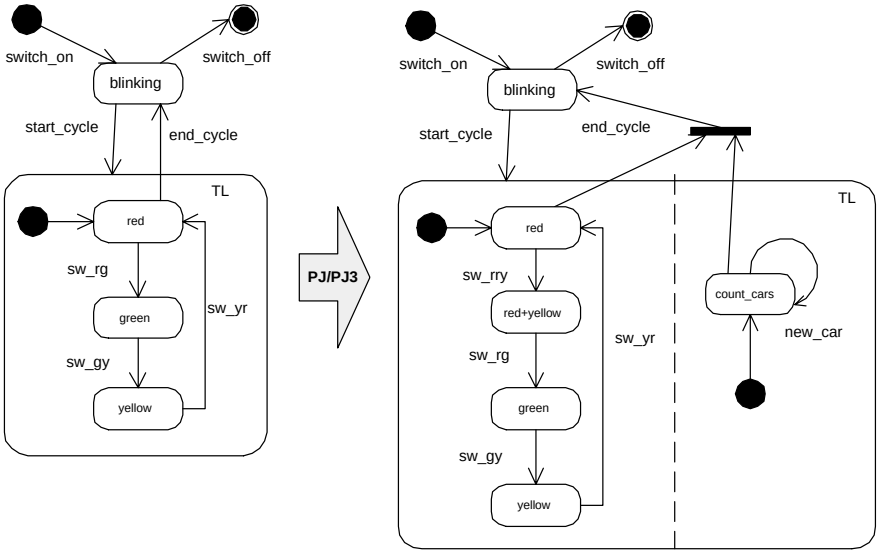

Fig. 9. A subclass statechart diagram constructed using rules PJ and PJ3.

rules PJ and PJ3 mentioned in Section 2.2. PJ can be used to insert the additional state *red+yellow* and PJ3 can be used to add the concurrent region for counting cars.

5 Activity diagrams

An activity diagram is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. Activity diagrams are typically used for modeling behavior which transcends the life cycle a single object. Therefore, it supports notations such swimlanes and is often used for workflow modeling. Compared to classical statecharts, activity diagrams allow for actions states, subactivity states, decisions and merges (both denoted by a diamond shape), object flows, and concurrent transitions (to model synchronization and forks). Note that swimlanes do not influence the behavior of an activity diagram. The semantics of activity diagrams is still under discussion. However, clearly many ideas have been adopted from Petri nets and in the proposal for UML 2.0 token passing is used as the main mechanism to specify the semantics of activity diagrams [12]. Therefore, we use WF-nets [3] as the semantic domain to map activity diagrams on. Concurrent transitions are mapped Petri-net τ transitions, decisions and merges are mapped onto places, actions states are mapped onto Petri-net transitions, object flows are mapped onto places, transitions are mapped onto τ transitions and places, etc.

Again we use an example to illustrate the application of our inheritance notions. Consider the two activity diagrams shown in Figure 10. The right-hand side diagram is a subclass of the left-hand side diagram under life-cycle inheritance. Note that the

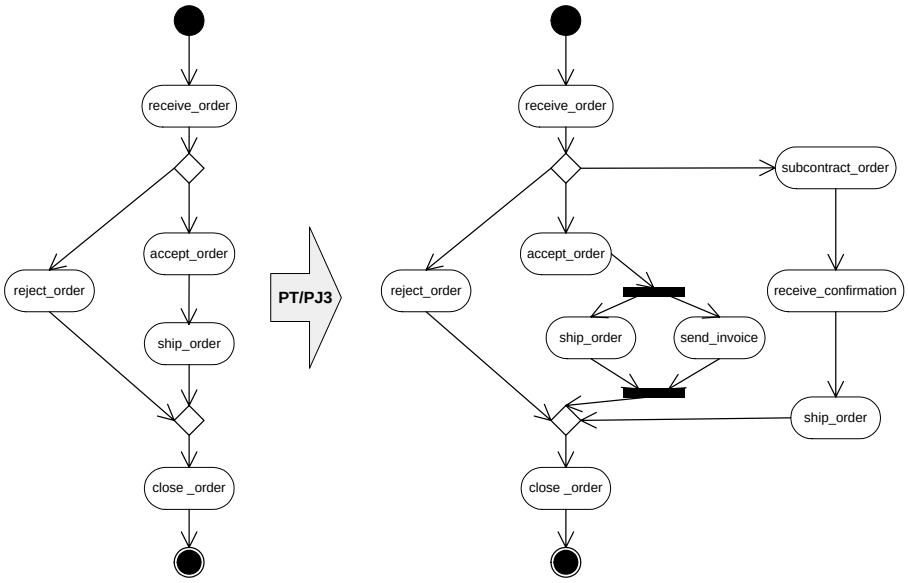

Fig. 10. A subclass activity diagram constructed using rules PT and PJ3.

right-hand side diagram is not a subclass under any of the other three notions of inheritance: Action *send_invoice* needs to be hidden while action *subcontract_order* needs to be blocked, therefore none of the other notions applies. Also note that left-hand side diagram corresponds to the labeled transition system shown in Figure 2. The right-hand side diagram corresponds to the labeled transition system shown in Figure 6. The subclass activity diagram (right) extends the superclass (left) in two ways: (1) *send_invoice* can be added using projection inheritance preserving transformation rule PJ3, and (2) the alternative sequence starting with *subcontract_order* can be added using protocol inheritance preserving transformation rule PT.

In Section 2.3 our verification tool Woflan [24] was already mentioned. Using a straightforward mapping of activity diagrams onto WF-nets, we can use Woflan to check whether the right-hand side activity diagram in Figure 10 is a subclass of the left-hand side diagram under life-cycle inheritance. As Figure 11 shows, this is indeed the case. The interested reader can download Woflan from http://www.tm.tue.nl/it/woflan.

6 Related work

The literature on object-oriented design and its theoretical foundations contains several studies related to the research described in this paper. In [26], abstraction in a process-algebraic setting is suggested as an inheritance relation for behavior. Other research on inheritance of behavior or related concepts such as behavioral subtyping are presented in [4, 15, 17, 18, 25]. The variety of inheritance relations reported in the literature is not surprising if one considers, for example, the large number of semantics that exist for concurrent systems (see, for example, [10]). For an elaborate overview of other

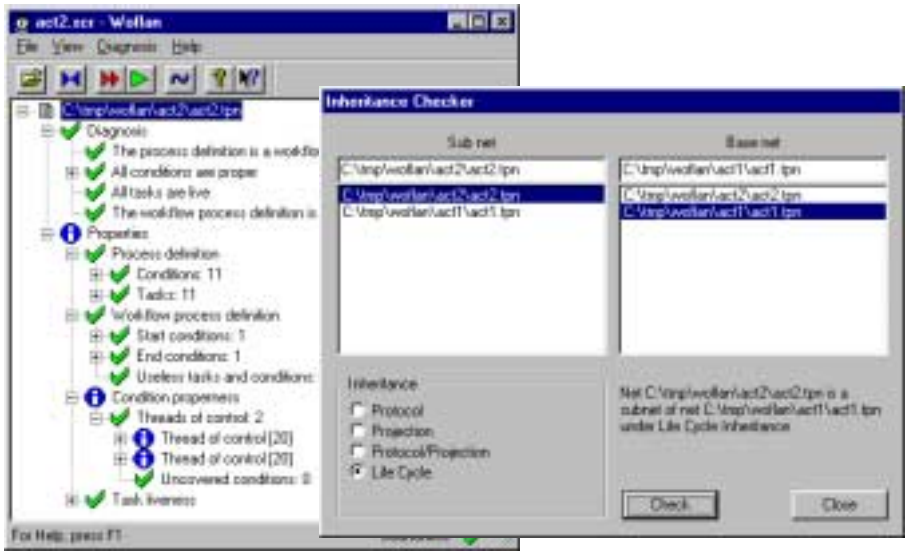

Fig. 11. Woflan shows that the right-hand side activity diagram in Figure 10 is indeed a subclass of the left-hand side diagram under life-cycle inheritance.

approaches we refer to [6]. Although many authors mention the need for inheritance of behavior in the context of UML, in most cases the application to concrete UML diagrams is missing. Consider for example the work presented in [14]. The authors provide a rigorous framework for behavioral inheritance, but do not "lift" the framework to the level of statechart or activity diagrams. Other authors focus specifically on inheritance of statecharts [8, 22, 23]. In [8] inheritance of statechart diagrams is investigated using CSP as a semantic domain. In [22, 23] Object/Behavior Diagrams are used as a semantic domain. It is encouraging to see that in [8, 22, 23] similar notions of inheritance are used: "invocation consistency" corresponds to our protocol inheritance and "observation consistency" corresponds to our projection inheritance. Note that we allow for two additional notions of inheritance (protocol/projection inheritance and life-cycle inheritance), provide inheritance preserving transformation and transfer rules, and also extend our work to sequence and activity diagrams.

7 Conclusion

In this paper, we investigated the applicability of the theoretical results on behavioral inheritance presented in [1–3, 5, 6] to the UML diagrams dealing with behavior. Although these theoretical results have been developed in the context of specific models for concurrency (i.e., Petri-nets and process algebra) there is a common core which has been illustrated using labeled transition systems (our core semantic domain). We have demonstrated that this core can be lifted to the level of concrete UML diagrams. In particular, we have applied the four inheritance notions and corresponding transformation

rules to sequence diagrams, activity diagrams, and statechart diagrams. In this paper, we used a rather pragmatic approach not aiming at the full expressive power of UML. The full UML standard simply contains too many features and is not defined (yet) sufficiently to allow for our ultimate quest: Defining inheritance for sequence diagrams, activity diagrams, and statechart diagrams as it is defined for object diagrams. However, the examples presented in this paper show that the theoretical results can be lifted to the level of concrete UML diagrams.

References

1. W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst and T. Basten. Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In J.M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52. Springer-Verlag, Berlin, 2001.
3. W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
4. P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundation of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, Berlin, 1991.
5. T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1998.
6. T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.
7. R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In M. Aksit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–366. Springer-Verlag, Berlin, 1997.
8. G. Engels, R. Heckel, and J.M. Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In M. Gogella and C. Kobryn, editors, *4th International Conference on The Unified Modeling Language (UML 2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 272–286. Springer-Verlag, Berlin, 2001.
9. R. Eshuis and R. Wieringa. Comparing Petri Nets and Activity Diagram Variants for Workflow Modelling- A Quest for Reactive Petri Nets. In H. Ehrig, W. Reisig, and G. Rozenberg, editors, *Petri Net Technologies for Communication Based Systems*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2002.
10. R.J. van Glabbeek. The Linear Time - Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves. In E. Best, editor, *Proceedings of CONCUR 1993*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, Berlin, 1993.
11. Object Management Group. *OMG Unified Modeling Language, Version 1.4*. OMG, http://www.omg.com/uml/, 2001.
12. Object Management Group. *OMG Unified Modeling Language 2.0 Proposal, Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02, Version 0.671*. OMG, http://www.omg.com/uml/, 2002.

13. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
14. D. Harel and O. Kupferman. On the Behavioral Inheritance of State-Based Objects. Technical Report MCS99-12, The Weizmann Institute of Science, Israel, 1999.
15. G. Kappel and M. Schrefl. Inheritance of Object Behavior - Consistent Extension of Object Life Cycles. In J. Eder and L.A. Kalinichenko, editors, *Proceedings of the Second International East/West Database Workshop (EWDW 1994)* , pages 289–300. Springer-Verlag, Berlin, 1995.
16. E. Kindler, A. Martens, and W. Reisig. Inter-Operability of Workflow Applications: Local Criteria for Global Soundness. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 235–253. Springer-Verlag, Berlin, 2000.
17. B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
18. O. Nierstrasz. Regular Types for Active Objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993. Special issue containing the proceedings of the 8th. annual conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'93, Washington DC, 1993.
19. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
20. E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, MA, USA, 1998.
22. M. Schrefl and M. Stumptner. On the Design of Behavior Consistent Specialization of Object Life Cycles in OBD and UML. In M. Papazoglou, S. Spaccapietra, and Z. Tari, editors, *Advances in Object-Oriented Data Modelling*, pages 65–104. MIT Press, 2000.
23. M. Stumptner and M. Schrefl. Behavior Consistent Inheritance in UML. In Alberto H. F. Laender et al. editor, *Proceedings of the 19th International Conference on Conceptual Modeling (ER 2000)*, volume 1920 of *Lecture Notes in Computer Science*, pages 527–542. Springer-Verlag, Berlin, 2000.
24. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
25. H. Wehrheim. Subtyping Patterns for Active Objects. In H. Giese and S. Philippi, editors, *Proceedings 8ter Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung)*, volume 24/00, Münster, Germany, 2000. University of Münster.
26. R.J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models*. PhD thesis, Free University, Amsterdam, The Netherlands, 1990.