

Consistency-Preserving Model Evolution through Transformations

Gregor Engels¹, Reiko Heckel¹, Jochen M. Küster¹, and Luuk Groenewegen²

¹ University of Paderborn, Department of Computer Science

D-33095 Paderborn, Germany

{engels,reiko,jkuester}@upb.de

phone: +49-5251-603357

² Leiden University, LIACS

P.O. Box 9512, NL-2300 RA, Leiden, The Netherlands

luuk@liacs.nl

phone: +31-71-527 7139

Abstract. With model-based development being on the verge of becoming an industrial standard, the topic of research of statically checking the consistency of a model made up of several submodels has already received increasing attention. The evolution of models within software engineering requires support for incremental consistency analysis techniques of a new version of the model after evolution, thereby avoiding a complete reiteration of all consistency tests.

In this paper, we discuss the problem of preserving consistency within model-based evolution focusing on UML-RT models. We introduce the concept of a model transformation rule that captures an evolution step. Composition of several evolution steps leads to a complex evolution of a model. For each evolution step, we study the effects on the consistency of the overall model and provide localized consistency checks for those parts of the model that have changed. For a complex evolution of a model, consistency can then be established by incrementally performing those localized consistency checks associated to the transformation rules applied within the evolution.

1 Introduction

In model-driven software engineering, the central artefact in the development are models, rather than programs coded in a programming language. Beside the generally accepted usefulness of models as a means for communication, documentation, and requirements capture, the main objective of this idea is the abstraction from programming languages and computational platforms to simplify the integration of software [11].

It is generally acknowledged that maintenance and evolution of software are still among the main challenges of software engineering. Refactoring techniques help to overcome these problems at the code-level by defining (and supporting) software transformations that restructure a software system while preserving its

behavior, which is validated by a number of test cases. If model-driven software engineering shall be successful, similar solutions are required at the model level.

Proposals for refactoring of UML models [15] suffer from the problem that no generally accepted operational interpretation of UML models is available. Therefore, validation of UML refactorings through testing is difficult. At the same time, transformations of models are difficult to describe because, unlike in programming languages where techniques based on string and tree rewriting are suitable, UML models have a graphical representation. In such situations rule-based graph transformations have been used to specify data base schema evolution [8, 7], refactoring and evolution of object-oriented programs [10], or software architecture reconfiguration [16].

One reason for the first problem, beside the poor quality of language specifications and tools, is the fact that models describe a system from different, partly overlapping perspectives in order to reduce the complexity of the problem. This separation of concerns causes the need to ensure the consistency of different submodels or views in order to ensure that there exists an implementation satisfying all requirements expressed by these different submodels.

Recently, we have proposed a methodology for defining and checking semantic consistency requirements for UML models [3]. Rather than completely formalizing UML models [9], our approach consists of a partial formalization of those aspects of a model that lead to a consistency problem and can be summarized as follows: A potential consistency problem arises as a conceptual overlap between submodels dealing with common aspects of the system to be described. To formalize and analyze this potential consistency problem, a semantic domain is chosen which supports the relevant aspects, and a mapping of (these aspects of) the models into the semantic domain is defined. Within the semantic domain, consistency conditions are formulated to enable the formal verification of the consistency of individual models, e.g., by means of a model checker. A rule-based notation has been proposed to describe flexible and extensible mappings of models to various semantic domains [2].

In this paper, we shall extend this approach to deal with the transformation of models. Thereby, we are interested in the preservation of consistency properties, like the absence of deadlocks or the conformance to some protocol, rather than of the complete behavior. For this reason we use the more general term of evolution, rather than refactoring. Transformation rules, which replace in every application one model pattern by another one, modify the model only locally. The problem we are dealing with in this paper consists in establishing for each transformation rule and each consistency property, a corresponding local condition to ensure the preservation of this property by the rule. This enables an incremental approach to the verification of consistency carrying over most of the analysis results from the previous to the current version of the model.

We shall discuss this problem for a specific class of models of concurrent object-oriented systems expressed in UML-RT [14]. First, we briefly introduce concepts and discuss consistency properties of UML-RT models. We then introduce the concepts of CSP [6] which is used as a semantic domain for check-

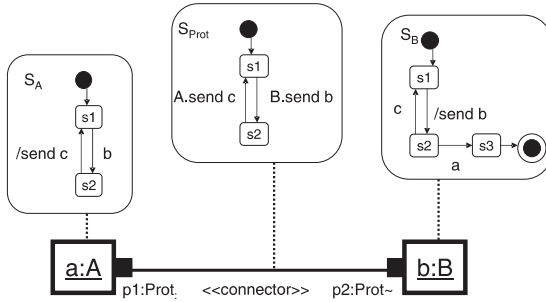


Fig. 1. The situation in UML-RT

ing UML-RT consistency, and describe the general ideas of translating UML-RT models to CSP. In section 5, we develop an approach for dealing with consistency-preserving evolution based on rule-based model transformations and algebraic properties of the semantic domain CSP.

2 Consistency of Concurrent Object-Oriented Models

Concurrent object-oriented systems consist of multiple objects executed concurrently and exchanging messages. Currently, UML-RT [14] is one approach for modeling concurrent systems in an object-oriented way. UML-RT is an extension of the UML introducing the notions of *capsules*, *ports*, *connectors*, *protocols* and *protocol roles*. Originally targeted at enabling the modeling of complex real-time systems, UML-RT has also been seen as a candidate for modeling software architectures [13] and for modeling concurrent systems in general. In the following, concepts of UML-RT are briefly explained.

A *capsule* is a stereotyped active class and is used for modeling a self contained component of a system. For communication with other capsules a capsule may have one or more ports through which it is interconnected to other capsules via *connectors*. A connector is an association between capsules. It represents a hardware connection via which capsules communicate by sending and receiving signals. These signals enter or leave a capsule at a port. A *port* realizes a *protocol role* which specifies the signals sent and received via the port. One or more protocol roles form a *protocol*.

For modeling behavior, a statechart may be associated to a capsule. A capsule statechart describes how the capsule reacts to signals received via its ports and when signals are sent via its ports. State transitions of capsule statecharts may also include calls of capsule operations. For protocols, there exist also the possibility of modeling all valid sequences of signal exchanges in a protocol statechart. The protocol statechart therefore expresses requirements rather than specifying the implementation of a protocol.

In Figure 1, concepts of UML-RT are illustrated using a simple example consisting of two capsule instances *a* and *b* connected by a connector via the two

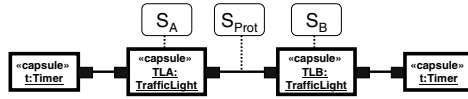


Fig. 2. The traffic lights in UML-RT

ports $p1$ and $p2$. The ports are bound to the protocol roles $Prot$ and $Prot \sim$, respectively. The protocol statechart S_{Prot} specifies the allowed sequence of signal exchanges via the connector. In this case, it is specified that first capsule B sends a b -signal, then A sends a c -signal and so on. Note that in case of binary protocols the protocol role can be directly derived from the protocol. Each capsule is associated to a capsule statechart describing the dynamic behavior of the capsule (denoted S_A and S_B , respectively).

For such a model, the following consistency properties will be required.

1. The overall system shall be deadlock-free, that is not all capsule statecharts should be blocked at the same time.
2. Communication between any two connected capsule statecharts shall conform to the corresponding protocol statechart, if any.

The importance of these consistency problems can best be illustrated using a simple example. In Figure 2, a traffic light system is presented consisting of two traffic lights, each connected to a timer capsule. The behavior of the traffic lights is defined in a statechart for each traffic light and consists of performing a red-green cycle. For synchronization with the other light, the traffic lights communicate via the lights port. Consistency of the system ensures deadlock freedom and protocol consistency. Deadlock freedom is of importance because in case of a deadlock the traffic light system would not correctly switch the lights anymore, leading possibly to car crashes. Protocol consistency ensures that the traffic lights equally perform a red-green cycle one after the other.

3 CSP as a Semantic Domain

Communicating Sequential Processes (CSP) [6] provide a mathematical model for concurrency based on a simple programming notation and supported by tools. In fact, the existence of *language* and *tool support* are most important to our aim of *specifying* and *verifying* consistency constraints, despite the existence of more expressive mathematical models. Next, we briefly review the syntax and semantics of the CSP processes we are using.

Given a set \mathcal{A} of actions and a set of process names \mathcal{N} , the syntax of CSP is given by

$$P ::= \text{STOP} \mid \text{SKIP} \mid a \rightarrow P \mid P[A \mid B]P \mid P \parallel P \mid P \sqcap P \mid P \sqcup P \mid P \setminus a \mid pn$$

where $a \in \mathcal{A}$, $A, B \subseteq \mathcal{A}$, and $pn \in \mathcal{N}$. Process names are used for defining recursive processes using equations $pn = P$.

The interpretation of the operations is as follows. The processes STOP and SKIP represent, respectively, deadlock and successful termination. The prefix process $a \rightarrow P$ performs action a and continues like P . The parallel composition $P[A \mid B]Q$ results in an interleaving of P and Q except for the actions in $A \cap B$, which have to be performed synchronously. Interleaving of processes $P \parallel\parallel Q$ results in a process that interleaves the actions of P and Q .

The processes $P \sqcap Q$ and $P \square Q$ represent internal and external choice between P and Q , respectively. That means, while $P \sqcap Q$ performs an internal (τ -)action when evolving into P or into Q , for $P \square Q$ this requires an observable action of either P or Q . For example, $(a \rightarrow P) \sqcap (b \rightarrow Q)$ performs τ in order to become either $a \rightarrow P$ or $b \rightarrow Q$. Instead, $(a \rightarrow P) \square (b \rightarrow Q)$ must perform a or b and evolves into P or Q , respectively. Finally, the process $P \setminus a$ behaves like P except that all occurrences of action a are hidden.

The semantics of CSP is usually defined in terms of *traces*, *failures*, and *divergences* [6]. A trace is just a finite sequence $s \in \mathcal{A}^*$ of actions which may be observed when a process is executing. A failure (s, A) provides, in addition, the set A of actions (also denoted *refusal set*) that can be refused by the process after executing s . Divergences are traces that are followed by infinite internal computations (without any communication). In the following, for a process P , we denote with $traces(P)$ the set of all traces of P and with $failures(P)$ the set of all failures of P .

Together with these semantic models come several notions of process refinement. We write $P \sqsubseteq_{\mathcal{T}} Q$ if $traces(Q) \subseteq traces(P)$, i.e., every trace of Q is also a trace of P . Analogously, $P \sqsubseteq_{\mathcal{F}} Q$ if $traces(Q) \subseteq traces(P)$ and $failures(Q) \subseteq failures(P)$. In general, the idea is that Q is a refinement of P if Q is more deterministic (more completely specified) than P . These refinement relations shall be used to express consistency constraints.

In particular, deadlock freedom of a process P defined over \mathcal{A} can be checked by determining whether $DF \sqsubseteq_{\mathcal{F}} P$. Here, DF denotes the most nondeterministic process on the alphabet \mathcal{A} and is defined as $DF = \prod\{a \rightarrow DF \mid a \in \mathcal{A}\}$. For a detailed introduction to CSP the reader is referred to Roscoe et al. [12].

4 Translating UML-RT Models into CSP

In order to analyze UML-RT models for consistency, our approach is to map models to CSP processes [3]. In the following, we briefly summarize the structure of such a translation, restricting ourselves to non-hierarchical UML-RT models (no capsules contained in other capsules) and considering only binary protocols. Furthermore, we assume that the signal sets of all protocols are disjoint.

A statechart S translates into a CSP process $CSP(S)$ specifying the behavior of the statechart and its associated event queue (cf. [5]). In case of a capsule statechart, the alphabet on which this process will synchronize with the port processes includes events $send_m$ for all messages m sent and $queue_n$ for all messages n received by the capsule.

<pre> A(State) = act_A -> if (state == s1) then b?x -> if (x == 1) then ackn_A -> A(s2) else ackn_A -> A(s1) else if (state == s2) then send_c -> ackn_A -> A(s1) else STOP </pre>	<pre> A_Port_in = A_p_in ? x -> if (x == B_b) then queue_b -> A_Port_in else A_Port_in A_Port_out = send_c -> A_p_out ! A_c -> A_Port_out </pre>
--	--

Fig. 3. The statechart process and the port processes

For each port p_i , a process $CSP(p_i) = Port_i^{in} \parallel Port_i^{out}$ is composed by interleaving two unidirectional port processes for in- and outgoing messages which synchronize over events $p_i^{in}?m$ and $p_i^{out}!n$ with the attached connector process and over events $send_n$ and $queue_m$ with the statechart process. In Figure 3, the statechart process derived from the statechart S_A in Figure 1 and the two port processes, one for incoming and one for outgoing messages, is shown (the process associated to the event queue is not shown in the figure).

Since explicit specification of connector behavior is not supported by UML-RT, we use *connector stereotypes* for selecting specific, pre-defined connector behavior. For example, one can thus select blocking or non-blocking behavior, or the restriction to a certain buffer capacity, etc. For each connector of stereotype c between two ports p_i and p_j , we construct a connector process $CSP(c) = CSP_{p_i,p_j}(c) \parallel CSP_{p_j,p_i}(c)$ by interleaving two unidirectional connector processes which formalize the intended behavior associated with the stereotype. They synchronize with the port processes over events $p_i^{in}?m, p_i^{out}!n$ and $p_j^{in}?n, p_j^{out}!m$.

Composing all statechart, port and connector processes P_1, \dots, P_n in parallel, we obtain a process $CSP(M) = \parallel_{i=1..n} (P_i, \alpha P_i)$ which represents the complete behavior of the model M . Here, αP_i denotes the synchronization alphabet of process P_i as described above. Sometimes we are specifically interested in the communication of two capsules over a connector as shown in Fig. 1. In this case, we construct the capsule-connector-capsule process

$$CCC(A, p_1, c, p_2, B) = \parallel_{i=1..5} (P_i, \alpha P_i)$$

where $P_1 = V^{p_1}(S_A)$ and $P_2 = V^{p_2}(S_B)$ represent the views of port p_1 and p_2 on S_A and S_B , respectively, obtained by hiding from the statechart processes all events not relevant to the ports. Moreover, $P_3 = CSP(p_1)$ and $P_4 = CSP(p_2)$ are the port processes while $P_5 = CSP(c)$ models the connector.

The two consistency issues identified above can be formalized as follows.

1. *Deadlock freedom.* The model M is deadlock free if the process $CSP(M)$ is deadlock free, that is, $DF \sqsubseteq_{\mathcal{F}} CSP(M)$.
2. *Protocol consistency.* A capsule-connector-capsule combination as shown in Fig. 1 is consistent with a protocol statechart S_{Prot} if $CSP(S_{Prot}) \sqsubseteq_{\mathcal{T}} CCC(A, p_1, c, p_2, B)$. The model M is *protocol consistent* if all capsules-connector-capsule combinations are consistent with their protocols.

Protocol consistency requires that all traces of the process $CCC(A, p1, c, p2, B)$ are contained in the traces of the process $CSP(S_{Prot})$. Referring to the example in Figure 1, protocol consistency holds because all traces generated by the execution of the statecharts conform to the specified protocol.

By translating UML-RT models to CSP we have therefore gained a precise formal definition for protocol consistency and deadlock freedom. Automating the translation process [2] enables automated checking of consistency using tools such as the FDR model checker [4].

Within model evolution, the UML-RT model changes and therefore also its CSP counterpart. The UML-RT model remains consistent if its CSP translation remains consistent. As a consequence, it is necessary to understand the effects that CSP refinement relations induce on the UML-RT level. In other words, we need to discuss what it means for a statechart to refine another one, more precisely, how can we change a statechart such that it remains a refinement in the terms of the CSP translation?

Concerning trace refinement, a statechart $S_{B'}$ trace refines another statechart S_B if $CSP(S_B) \sqsubseteq_T CSP(S_{B'})$. That means that all traces generated by $S_{B'}$ must be contained in the set of traces of S_B . Intuitively, for a statechart that means that every sequence of events and actions possible for $S_{B'}$ must also be possible for S_B . As a consequence, in the process of refinement, we are allowed to delete transitions in the statechart S_B to get $S_{B'}$. However, we cannot introduce new transitions with new events. As an example, consider the protocol statechart in Figure 1. If we construct a new protocol statechart by deleting an arbitrary transition (e. g. the $B.send\ b$ transition), then the new protocol statechart is a trace refinement of the old protocol statechart.

Concerning failures refinement, first of all the condition of trace refinement must hold (note that failures refinement is stronger than trace refinement). Additionally, the set of failures of $CSP(S_{B'})$ must be included in the set of failures of $CSP(S_B)$. Recalling failures calculations for CSP processes, we can also annotate a statechart with a refusal set. We note that in general we are not allowed to delete transitions in the process of failures refinement because this introduces additional failures. Referring to the properties of CSP, we are only allowed to make the statechart more deterministic.

5 Consistency-Preserving Model Evolution through Transformations

One of the main advantages of the component-based approach to software development over the classical paradigm of object-oriented programming is the possibility to keep the effect of changes local. This is achieved by strict encapsulation using interfaces (ports) which do not only specify the messages that a component is able to receive, but also the messages it is able to send.

However, despite these advantages, we still face the problem of ensuring consistency after the model has evolved. Consider as an example the evolution of the traffic lights described in Figure 4. A new car sensor capsule is added to the

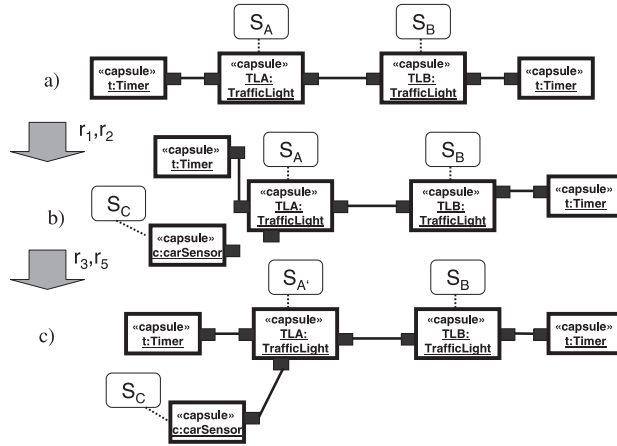


Fig. 4. Evolution of the traffic lights model in UML-RT

system and connected to one of the traffic lights. In order to ensure the correct functioning of the system, deadlock freedom and protocol consistency must be ensured. One way would be to reiterate all consistency checks which is obviously very time-consuming.

Therefore, we shall try to exploit the locality principle. That means to formalize evolution by means of local transformations, like the creation or deletion of a capsule or connector, the update of a capsule or protocol statechart, etc., and to study the effects of these transformations on various consistency properties. The aim is to establish the consistency of the new model in an incremental way rather than to verify again the complete model. In fact, we believe that any approach to consistency that cannot deal with evolution locally must fail in realistic development scenarios with frequent re-iterations of modeling, implementation, and testing.

Our approach consists of the following steps. First, model transformations are specified by means of transformation rules representing elementary evolution steps that may be combined in various ways to achieve more complex changes. Second, for each transformation rule local application conditions are specified which, when satisfied, ensure the preservation of a certain more global consistency property. The specification of rules and conditions happens once and for all at the language level, although the experience from concrete examples is required to validate their practicability.

In order to manipulate a given model we have to determine the kind of change we want to achieve as well as its location in the model. This corresponds to the selection of a rule and its application area. Then, the rule may be applied provided that all those application conditions are satisfied that are required for the consistency properties to be preserved. If there is no consistency-preserving transformation for the change we intend to do, we have to resort to ad-hoc evolution with a complete consistency check of the new model.

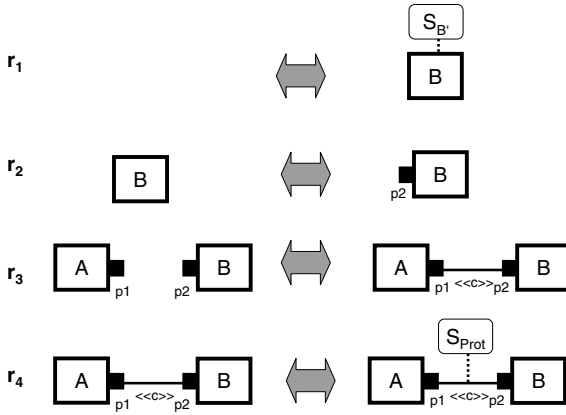


Fig. 5. Transformation rules for creation and deletion

Notice that we do not deal with dynamic (i.e., run-time) evolution. This would require, in addition, to take into account the states of capsules and protocols in order not to disturb ongoing interactions.

5.1 Rule-Based Model Transformations

We provide *creation/deletion rules* for model elements like capsules, connectors, and protocol statecharts (cf. Fig. 5) as well as *update rules* for capsule and protocol statecharts and connectors (cf. Fig. 6). Each rule consists of two patterns representing the situation before and after the change. For example, the rule r_3 assumes two capsules A and B with an unconnected port each and creates a connector between them. When read from right to left, the inverse rule r_3^{-1} removes an existing connector from its ports. The possibility to use a rule in both ways is represented by a double arrow.

Rules r_1, r_2, r_4 and their inverses work analogously. In the case of r_1 the empty pattern on the left means that the rule can be applied anywhere. For the inverse r_1^{-1} it is important to note that the deletion of a capsule is only allowed if there is no port attached to it. This is no real limitation since we may use r_3^{-1} first to remove all connectors, then removing all ports with r_2^{-1} . Removing connectors, in turn, we have to take care to delete any attached protocol statecharts first using rule r_4^{-1} , etc.

Formally, our rules can be interpreted as graph transformation rules on instances of the meta model defining the syntax of the language UML-RT. The restriction of applicability described above are typical of the so called *algebraic approach* [1] to graph transformation which takes a conservative approach, disallowing the deletion of vertices if this would cause “dangling links”.

The rules in Fig. 6 specify the update of model elements, by exchanging a capsule statechart (rule r_5), replacing a protocol statechart (rule r_6) and modifying the type of a connector (rule r_7), e.g., to increase its buffer size. Often,

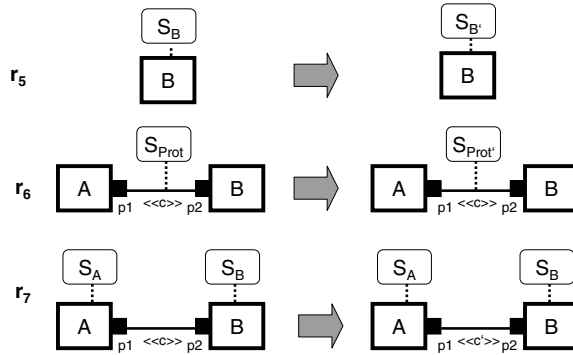


Fig. 6. Transformation rules for update

a similar effect can be achieved by first deleting a model element and then creating a new one using the rules of Fig. 5. However, to combine these two steps in one rule allows us to relate the semantics of the original and its replacement, e.g. by requiring that the second is a refinement or extension of the first.

This idea can be generalized by constructing more powerful, but less generic rules dedicated to handle evolution scenarios which occur frequently in a certain class of applications. For example, the change to the traffic light model in Fig. 4 could be either obtained by a sequence of elementary transformation steps or by a single complex step.

5.2 Preservation of Consistency

After identifying possible evolution steps by means of model transformation rules, we now consider the question how the application of these rules has to be constrained if a certain consistency property shall be preserved. That means, we shall assume that the original model has a certain property, like deadlock freedom or protocol consistency, and we shall formulate conditions for the application of transformation rules which ensure that the transformed model enjoys a similar property.

Formally, we will exploit the mapping of UML-RT models to CSP processes to prove that the stated conditions are indeed sufficient to ensure the preservation of the property. Here, the choice of CSP as a semantic domain is most important: Beside the fact that properties can be formulated and automatically verified, this semantic domain provides useful theoretical results about refinement and composition of processes that we shall use in the proofs.

First, we state some simple preservation properties of creation/deletion rules.

Theorem 1 (simple preservation properties).

1. *The rules r_1 , r_2 and r_3 , as well as their inverses, preserve protocol consistency.*

2. Rule $r_1, r_2, r_4, r_1^{-1}, r_2^{-1}, r_4^{-1}$ preserve deadlock freedom.
3. Rule r_4^{-1} preserves protocol consistency, while rule r_4 does so under the condition that the communication of A and B over the connector refines S_{Prot} .

Proof. We only give a formal proof for r_2 : Recall that $CSP(M) = \parallel_{i=1..n} (P_i, \alpha P_i)$ and $CSP(M') = \parallel_{i=1..n+1} (P_i, \alpha P_i)$ where P_{n+1} is the new port process. As the capsule statechart is not changed by this rule, it does not reference any signals associated to the protocol of the new port. Therefore, αP_{n+1} is disjoint from all other alphabets and thus the port process does not synchronize on any event with any other process. By prerequisite, $CSP(M)$ is deadlock free and therefore also $CSP(M')$ because the behavior of all other processes remains the same.

The properties of r_1/r_1^{-1} are based on the fact that there is no connection whatsoever between the the newly created or deleted capsule and the rest of the model.

For rules r_2/r_2^{-1} we have to observe that the capsule behavior does not change when introducing a port because the capsule statechart remains the same. In the case of creation, the capsule statechart before the transformation could not refer to events related to the port because the port did not yet exist. In the case of deletion, the application of the rule is prohibited as long as there is any reference to the port. This is, again, a consequence of the conservative approach which does not allow for dangling references thus maintaining syntactic well-formedness of the model. Therefore, rule r_2/r_2^{-1} is only meaningful in combination with r_4 which changes the behavior of the capsules to exploit/ignore the port.

More interesting is the rule r_4 introducing a new protocol statechart because it requires to check the consistency of the capsule-connector-capsule construct with the new protocol. Rule r_4^{-1} , instead, is just removing this requirement.

For r_3/r_3^{-1} , it is crucial to observe that, although there is no change to the capsule statecharts, the overall behavior may change dramatically if the capsule statecharts specify send or receive operations via the newly connected/disconnected ports, which become activated/blocked by the change in the communication structure. We first note that that creation and deletion of connectors preserves protocol consistency because protocol consistency considers only local behavior of the capsules involved in the communication over the connector the protocol statechart is associated with. However, in general deadlock freedom is not preserved when creating or deleting a connector.

As simple example, consider Figure 7. The system before application of r_3 is deadlock free because A and B continuously exchange messages. After plugging in the connector, C sends an a to B causing B to end its activities. After that, A is blocked because it does not receive any messages from B anymore, B has terminated and so has C . As a consequence, the system deadlocks.

Concerning the evolution of the capsule statechart itself, we can establish the following theorem.

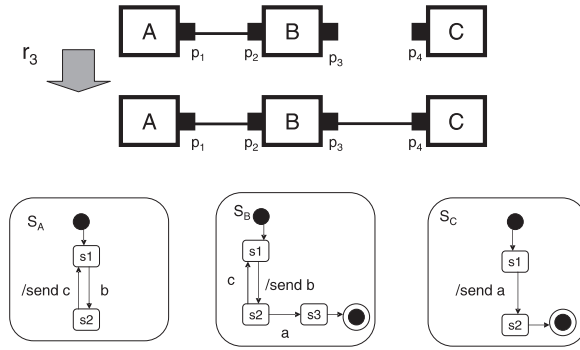


Fig. 7. Rule r_3 does not preserve deadlock freedom

Theorem 2 (capsule statechart evolution).

1. Rule r_5 preserves the protocol consistency of all capsule-connector-capsule constructs (X, p_1, c, p_2, Y) where $X = B$ implies that $V^{p_1}(S'_B)$ is a trace refinement of $V^{p_1}(S_B)$ and, analogously, $Y = B$ implies that $V^{p_2}(S'_B)$ is a trace refinement of $V^{p_2}(S_B)$
2. If the new capsule statechart S'_B is a failures refinement of S_B , then r_5 preserves deadlock freedom.

Proof. 1. Let $X=B$. We know that $CSP(S_{Prot}) \sqsubseteq_T CCC(B, p_1, c, p_2, Y)$. By prerequisite, $V^{p_1}(CSP(S_B)) \sqsubseteq_T V^{p_1}(CSP(S_{B'}))$ and thus, by closure of traces refinement under context, $CCC(B, p_1, c, p_2, Y) \sqsubseteq_T CCC(B', p_1, c, p_2, Y)$ and by transitivity the result follows.

2. If $CSP(S_B) \sqsubseteq_F CSP(S_{B'})$ then $CSP(M) \sqsubseteq_F CSP(M')$ by properties of refinement. As $DF \sqsubseteq_F CSP(M)$, deadlock freedom of $CSP(M')$ follows by transitivity.

With respect to preserving protocol consistency, we have to ensure that the view of a port on the statechart remains the same. For a statechart that means that we can change it as long as we do not affect the traces concerned with the exchange of messages via the port. As an example, recall Figure 1 and statechart S_A . In order to keep the view on the port the same, we could for example add a concurrent substate to the statechart exchanging messages via a new port. We could also expand the state s_1 or s_2 and do some additional transitions, but not any sending or receiving messages via the port: We are in general not allowed to add a transition sending or receiving a message via the old port, thereby introducing a new trace and violating the trace refinement property.

Concerning the evolution of the protocol statechart using rule r_6 we can show that, if the original model is protocol consistent and the new protocol statechart is an extension (an inverse refinement) of the old one, then the new model is also protocol consistent. That means, we may weaken the protocol statechart to allow more sequences of message as long as we keep the original sequences.

Theorem 3 (protocol statechart evolution). *Rule r_6 preserves protocol consistency if S_{Prot} is a trace refinement of $S_{Prot'}$.*

Proof. Let S_{Prot} be associated to $CCC(A, p1, c, p2, B)$. If $CSP(S_{Prot'}) \sqsubseteq_{\mathcal{T}} CSP(S_{Prot})$ then all traces contained in the old protocol are also contained in the new protocol. As $CSP(S_{Prot'}) \sqsubseteq_{\mathcal{T}} CCC(A, p1, c, p2, B)$, by transitivity of trace refinement it follows that $CSP(S_{Prot'}) \sqsubseteq_{\mathcal{T}} CCC(A, p1, c, p2, B)$.

Concerning the exchange of a connector depicted in r_7 , we can show that if the new connector (process) is a trace refinement of the old one, then r_7 preserves protocol consistency and if the new connector (process) is a failures refinement of the old one, then r_7 preserves deadlock freedom. Providing predefined connector stereotypes, each of it associated to a specific connector process, we can check connector processes for refinement properties once and for all. In principle, refinement properties of connectors depend very much on their underlying behavior (i. e. whether a connector is non-blocking or blocking and how large its capacity is).

After discussing the evolution according to minimal evolution steps, one can also establish theorems for consistency-preserving complex evolution steps combining the application of a number of the rules discussed. For example, the introduction of a new connector (r_3) often leads to a change of capsule behavior (r_5). In the following, we give an exemplary theorem for such a combined evolution. We assume here, that the left hand sides of rules are identified (i. e. that capsule B is the same capsule in both rule applications) which need not be the case in general.

Theorem 4 (combined evolution). *An evolution according to r_3 and r_5 preserves deadlock freedom under the condition that $V^{p_2}(CSP(S_{B'})) \parallel\parallel CSP(S_B)$ is failures equivalent to $CSP(S_{B'})$.*

Proof. If $CSP(S_{B'})$ is failures equivalent to $V^{p_2}(CSP(S_{B'})) \parallel\parallel CSP(S_B)$, then in $CSP(M')$ the complete deadlock free behavior of $CSP(M)$ is contained and this is not influenced by the addition of the new connector and new behavior. As a consequence, $CSP(M')$ is deadlock free.

In practice, this means that we can add a new capsule together with a capsule statechart and a connector attached to an old capsule and the complete model remains deadlock free, if we extend for example the existing statechart with a concurrent substate.

Returning to our motivating example of the traffic lights (see Figure 4), we can now study the evolution in detail. We first identify the individual transformation rules that might have been used to change the model. In our case, we may apply rules r_1 for adding capsule instance c of type `carSensor`, twice r_2 for adding two new ports for traffic light TLA and `carSensor`, r_3 adding a new connector and r_5 changing the behavior of the statechart for TLA for example for counting the cars crossing the intersection at a traffic light.

Using the theorems above, one can verify in an incremental way that the model remains protocol consistent. In fact, this preservation holds unconditionally for all but the last step, while for r_5 according to Theorem 2.1 we have to check if S'_A refines S_A with respect to the two original ports. In the present example, this might well be the case if the new interaction with the car sensor is transparent for the other components. However, even under this assumption, we cannot automatically conclude that the evolution preserves deadlock freedom because of the application of r_3 to add the new connector. Using the theorem for combined evolution, deadlock freedom can be established if the change from S_A to $S_{A'}$ consists of adding a new concurrent substate capturing the new behavior via the new port.

6 Conclusion

Model-level evolution is important because it allows to deal with a key problem of software development at high level of abstraction. However, in order to be of any use, the consistency of models must be maintained as a key prerequisite for a successful mapping to an implementation.

In this paper, we have shown how consistency of UML-RT models can be formalized by mapping important aspects of the model into a semantic domain, specifying the consistency requirements as relations between semantic objects and performing static analysis to verify these relations. We have introduced rule-based transformations of UML-RT models to describe both structural and behavioral evolution steps. For each rule, we have formally shown which conditions are required to preserve certain consistency properties, and discussed examples where consistency properties are, in general, not preserved.

We have learned that consistency is a semantic issue which should be dealt with by formal methods (in our case CSP) providing a specification language and tool support for analysis and, last but not least, an established theory for verifying models and transformation rules. In our specific example, we have largely exploited the fact that CSP refinements are transitive and closed under substitution of processes. This allowed us to show that a local relation between the replaced submodels is sufficient to preserve global consistency. Here, the choice of the semantic domain is crucial because, e.g., in a model checking approach not supporting such compositionality results, local analysis is not sufficient.

Future work includes the establishment of more powerful combined evolution theorems as well as the extension of this approach to support further aspects of UML-RT such as hierarchy of capsules. Furthermore, the approach shall be applied to other consistency problems in other semantic domains, thereby enabling consistency-preserving evolution also for other UML dialects.

References

- [1] H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973. [220](#)
- [2] G. Engels, R. Heckel, and J.M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *LNCS*, pages 272–287. Springer, 2001. [213](#), [218](#)
- [3] G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, 2001. [213](#), [216](#)
- [4] Formal Systems Europe (Ltd). *Failures-Divergence-Refinement: FDR2 User Manual*, 1997. [218](#)
- [5] J.-J. Hiemer. *Statecharts in CSP: Ein Prozessmodell in CSP zur Analyse von STATEMATE-Statecharts*. DrKovac Verlag, 1999. [216](#)
- [6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. [213](#), [215](#), [216](#)
- [7] J. Jahnke and A. Zündorf. Using graph grammars for building the VARLET database reverse engineering environment. In G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999. [213](#)
- [8] M. Löwe. Evolution patterns. Postdoctoral thesis, Technical University of Berlin. Tech. Report 98-4, Dept. of Comp. Sci, 1997. [213](#)
- [9] W. McUumber and B. Cheng. A General Framework for Formalizing UML with Formal Languages. In *Proceedings 23rd International Conference on Software Engineering*. IEEE Computer Society, May 2001. [213](#)
- [10] T. Mens, S. Demeyer, and D. Janssens. Object-oriented refactoring using graph rewriting. Technical Report vub-prog-tr-02-01, Vrije Universiteit Brussel, 2001. [213](#)
- [11] Object Management Group. Model driven architecture, 2001. <http://www.omg.org/mda>. [212](#)
- [12] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. [216](#)
- [13] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. UML + ROOM as a standard ADL? In *Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems, Las Vegas, USA*. IEEE Computer Society Press, 1999. [214](#)
- [14] B. Selic. Using UML for modeling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 250–262. Springer Verlag, 1998. [213](#), [214](#)
- [15] G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel. Refactoring UML models. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools., 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001. [213](#)

- [16] M. Wermelinger and J. Fiadero. A graph transformation approach to software architecture reconfiguration. In H. Ehrig and G. Taentzer, editors, *Joint APPLICRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000)*, Berlin, Germany, March 2000. 213