Towards Semantics-Aware Merge Support in Optimistic Model Versioning*

Petra Brosch[2], Uwe Egly[1], Sebastian Gabmeyer[2], Gerti Kappel[2], Martina Seidl[3], Hans Tompits[1], Magdalena Widl[1], and Manuel Wimmer[2]

[1] Institute for Information Systems, Vienna University of Technology, Austria
{uwe,tompits,widl}@kr.tuwien.ac.at
[2] Business Informatics Group, Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[3] Institute of Formal Models and Verification, Johannes Kepler University, Austria
martina.seidl@jku.at

Abstract. Current optimistic model versioning systems, which are indispensable to coordinate the collaboration within teams, are able to detect several kinds of conflicts between two concurrently modified versions of one model. These systems support the detection of syntactical problems such as contradicting changes, violations of the underlying metamodel, and violations of OCL constraints. However, violations of the semantics remain unreported. In this paper, we suggest to use redundant information inherent in models to check if the semantics is violated during the merge process. In particular, we exploit the information encoded in state machine diagrams to validate evolving sequence diagrams by means of the model checker SPIN.

1 Introduction

In model-driven engineering, *version control systems* (VCS) are an essential tool to manage the evolution of software models [4]. In this respect, *optimistic version control systems* [1] are of particular importance. They provide reliable recovery mechanisms in case changes have to be undone and support the collaboration of multiple developers.

An optimistic VCS stores the artifacts under development in a central repository, which may be accessed by all team members at any time. A typical interaction with the repository starts when a developer checks out the most recent version of the model under development. The developer then performs the desired changes on a local copy. Upon completion, the developer checks the modified local version back into the repository. If the performed changes do not interfere with the concurrently introduced modifications of another developer, the merge is straightforward and may be computed automatically. Otherwise, a *merge conflict* [4] is at hand and the divergent versions need to be merged

* This work was partially funded by the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency under grant FIT-IT-819584, by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018, by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research, and by the Austrian Science Fund (FWF) under grants P21698 and S11409-N23.

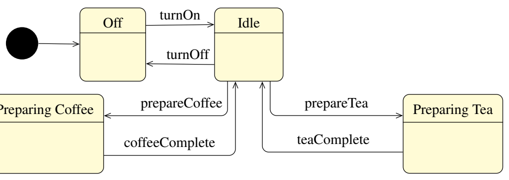

Fig. 1: State machine diagram for the class CoffeeMachine (`CM`).

manually. Without adequate tool support, the merged version may result in a syntactically and/or semantically inconsistent version, even though both versions were consistent prior to the merge. Obviously, it is of paramount importance to detect and resolve conflicts as soon as possible to prevent their propagation through multiple development cycles.

Among the many possible merge conflicts [1], the most common are *contradicting changes*. Given two developers working on the same model, this conflict may emerge if both developers commit their changes and either (a) their changes may not be applied in combination (i.e., *delete/update*), or (b) their changes are not commutable (i.e., *update/update*). In the latter case, the different ordering of the changes results in different models. In such a situation, often user interaction is required to resolve the conflict. Alternatively, a predefined heuristic-based merge strategy may be applied to automatically generate consolidated, syntactically correct versions. However, it cannot be asserted that the model is *semantically* consistent.

Consider the following example, which describes a semantically inconsistent model caused by an automatic merge of changes. Figure 1 depicts a UML state machine diagram modeling a coffee machine and the upmost diagram S in Fig. 2 a possible behavior of the same machine in terms of a sequence diagram. Two software engineers change the sequence diagram at the same time: one includes the message $\mathit{turnOff}()$, resulting in S', the other adds the message $\mathit{prepareTea}()$, resulting in S''. Each change on its own results in a sequence diagram consistent with the state machine. The next step is to merge the changes into a new sequence diagram $\hat{S}$ using an automatic versioning tool, e.g., as the one proposed by Brosch et al. [2]. As the messages of a lifeline are represented as ordered list, an update/update conflict occurs, because both newly added messages are stored at the same index of this list. A conceivable merging strategy is to consider all possible combinations of the two diagrams. This may result in several syntactically correct diagrams. Figure 2 shows two possibilities, $\hat{S}_1$ and $\hat{S}_2$: $\mathit{turnOff}()$ can be placed before or after $\mathit{prepareTea}()$. However, making tea after turning off the machine does not make much sense and a modeler would avoid such a solution in a manual merge process.

At first glance, it might seem necessary to provide additional knowledge, e.g., a specifically tailored ontology, to support an automatic merge process aware of the model's semantics. However, in modeling languages like UML, the required knowledge is distributed over different types of diagrams. Each diagram type provides a view on a specific aspect of the described system. Yet, these views overlap in parts, effectively duplicating certain aspects of the system across different diagrams. For our example, we may ascertain, that the first merge option, i.e., $\mathit{turnOff}()$ before $\mathit{prepareTea}()$, turns

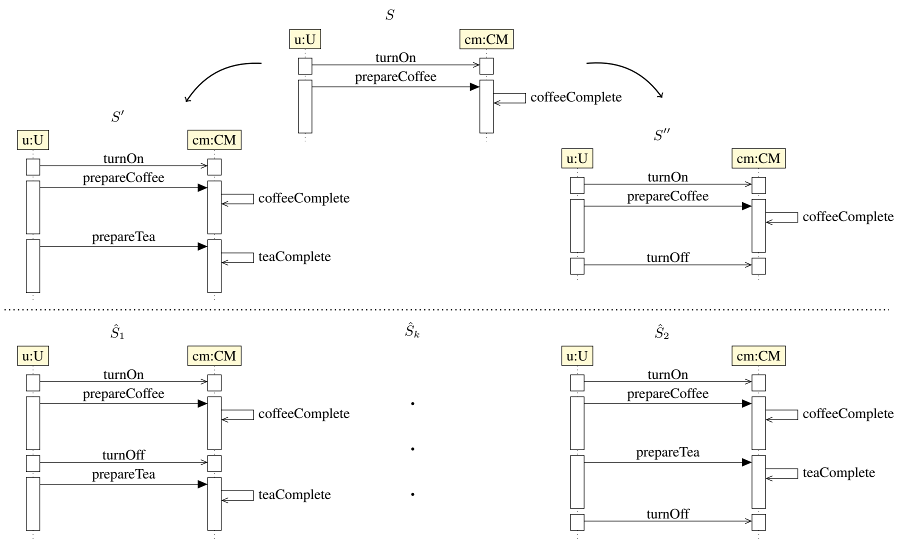

Fig. 2: Versioning scenario for a sequence diagram.

out to be inconsistent with respect to the state machine diagram, as preparing tea after turning off the machine is not possible.

In this paper, we thus propose to exploit this redundant information and use the overlapping parts of the diagrams as gluing points to construct a coherent picture of the system. In this way, we are able to assert that the modifications performed on a sequence diagram are consistent with the specification stated in a state machine diagram. For this purpose, we employ model checking techniques and integrate this approach into the merging component of the model versioning system AMOR [2].

Starting with a review of related work in Section 2, we proceed to present our semantics-aware merging approach in Section 3. We then showcase how the above presented example is solved with our approach in Section 4. Section 5 concludes the paper with a discussion of future work directions.

2 Related Work

The fields of model versioning and model verification are both related to our work, which we describe in what follows.

Model Versioning. In the last decade more than a dozen model versioning systems have been proposed (see [1] for an overview). Many existing model versioning systems take advantage of the graph-based structure of software models. As a consequence, conflicts resulting from contradicting changes are more precisely detected, sometimes even automatically resolved. Since changes are rarely introduced independently of each other, think of refactorings for example, some approaches analyze the set of composite changes to recognize the user's intention, and try to derive suitable resolution strategies when conflicting versions are checked into the repository [2, 6]. However, the semantic aspects of models are mostly neglected by current model versioning systems. To the best of our knowledge, only two approaches consider semantics in the context of model versioning. The first approach suggests the usage of *semantic views*, which are constructed by a manually defined normalization process that removes all duplicate representations of one and the same concept from the original metamodel [13]. When two divergent versions of the same base model are checked into the repository, the two versions are normalized and compared to determine possible conflicts. Although the normalization procedure integrates a semantic layer into the model versioning process, the actual comparison of the normalized models is still performed on a syntactic level.

Another elegant technique, which employs *diff* operators to compare models, is presented by Maoz et al. [10]. A *diff* operator $\mathit{diff}(m_1, m_2)$ expects two models, m_1 and m_2, as input and outputs a set of so-called diff witnesses, i.e., instances of m_1 which are not instances of m_2. For example, two *syntactically* different models m_1 and m_2 are *semantically* equivalent if each instance of m_1 is an instance of m_2 and vice versa. While [10] focuses solely on the semantic differencing aspect of model versioning, we aim to advance to a semantics-aware model merging process that is supported by an inter-diagram based consistency verification technique.

Model verification. Decoupled from the above sketched research field of model versioning systems, various works propose the verification of the syntactical consistency of models, many of which focus on the verification of UML diagrams (e.g. [7, 11]). The verification process may be enhanced by the addition of semantic information. For example, Cabot et al. [5] verify the behavioral aspects of UML class diagrams annotated with so-called operation contracts, which are declarative descriptions of operations specified as OCL pre- and postconditions. The class diagram and the operation contracts are thereby transformed into a constraint satisfaction problem, which is solved with respect to a set of consistency properties expressing, e.g., the applicability or the executability of an operation. A formal verification technique for UML 2.0 sequence diagrams employing linear temporal logic (LTL) formulas and the SPIN model checker [8] to reason about the occurrences of events is introduced by Lima et al. [9]. In contrast to these single-diagram verification techniques, multi-view approaches assert the consistency across a set of diagrams. Proponents in this area are, among others, the tools HUGO [14] and CHARMY [12]. HUGO verifies whether the interactions of a UML collaboration diagram are in accordance with the corresponding set of state machine diagrams. The tool automatically translates the state machine diagrams to PROMELA, the input language of SPIN, and generates so-called "never claims" from the collaboration diagrams. The generated artifacts form the input for SPIN, which performs the verification. While HUGO operates on UML diagrams, CHARMY provides a modeling, simulation, and verification environment for software architectures (SA), which share many commonalities with UML. SAs describe the static and behavioral structures of systems with component, state transition, and sequence diagrams. Again, CHARMY translates the modeled artifacts to PROMELA and calls upon SPIN to either locate deadlocks and unreachable states in the state machines, or to verify temporal properties of the system. In contrast to the standalone, snapshot-based verification procedure implemented by CHARMY and HUGO, our approach integrates the consistency verification procedure into the model versioning process to enable the semantics-aware merging of models.

3 Semantics-Aware Model Versioning

To detect semantic merge problems as described above, we propose to use a model checker like SPIN [8] within the merge process. The idea is to generate possible merge results and to check for each if it is consistent with the behavior defined by the corresponding state machine. We first give a short definition of the modeling language concepts needed, and then introduce our approach in detail. In particular, we consider a simplified subset of the UML state machine and sequence diagrams.

3.1 Definitions

For our purposes, a *software model* $\mathcal{U}$ consists of a set $\mathcal{M}$ of *state machines* and a *sequence diagram* $\mathcal{S}$, defined as follows: A state machine $M = (Q, T, \tau, v_0, A)$ is a deterministic finite automaton, where

- Q is a set of *states*,
- T is a set of *transition labels* (or possible *input symbols*),
- $\tau : Q \times T \rightarrow Q$ is the *transition function*,
- $q_0 \in Q$ is a designated *initial state*, and
- $A \subseteq Q$ is a set of *accepting states*.

A sequence diagram $\mathcal{S}$ is a tuple $(N, \overline{\mathcal{L}})$, where N is a set of *messages* and $\overline{\mathcal{L}}$ is a set $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ of *lifelines*. A lifeline, $\mathcal{L}$, in turn, is a tuple (M, L, tr), where

- $M \in \mathcal{M}$ is a state machine,
- L is a finite sequence $(n_1, \ldots, n_m)$ of elements of N and
- $tr : N \rightarrow T$ is a bijective function, mapping each message to a transition of the corresponding state machine.

A model $\mathcal{U}$ is *consistent* iff for each lifeline $\mathcal{L} = (M, L, tr)$ of $\mathcal{S}$, there exists a path $(tr(n_1), \ldots, tr(n_m))$ in the state machine M, where $L = (n_1, \ldots, n_m)$.

3.2 Versioning Scenarios

Our versioning scenarios involve concurrent modifications on a sequence diagram. The state machine diagrams remain unchanged. A modification concerns one or more messages, each being of either of the following three types:

- *insert*: a message $n \in N$ is inserted at any index of a lifeline;
- *delete*: a message n is removed from a lifeline; and
- *update*: a message n is replaced by $n' \in N$ different from n.

Concurrent changes may result in different sequence diagrams. It is then up to the versioning tool to merge these changes into a new version of the diagram, which must be syntactically correct and consistent with the state machine diagrams.

Merging sequence diagrams is done as follows: For each lifeline, any possible sequence of messages originating from both diagrams is syntactically correct, but possibly inconsistent with the behavior defined in the corresponding state machine diagrams.

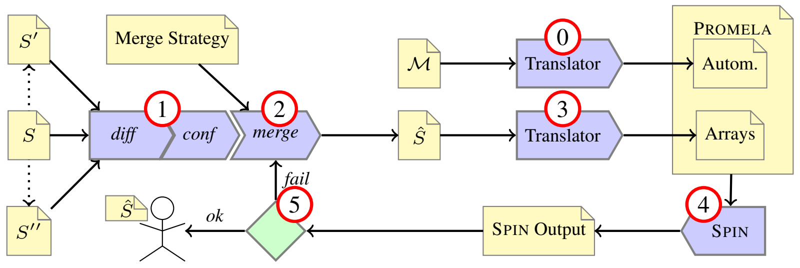

Fig. 3: Workflow of the merging process.

3.3 Semantics-Aware Model Merging

We propose to integrate the model checker SPIN [8] to support the generation of merged sequences. SPIN is a software verification tool: It takes as input a software abstraction, or model, encoded in SPIN's input language PROMELA and relevant properties of the software model in LTL. SPIN can be run in two modes: In *simulation mode*, where the PROMELA model is executed, and in *verification mode*, where the LTL formula is checked for satisfiability with respect to the PROMELA model.

For our basic definition of a software model, we propose a simple encoding that allows to check for the consistency between a sequence diagram and a set of state machines by running SPIN in simulation mode, which is much faster than verification mode and sufficient for our purpose. The state machines are encoded as deterministic finite automata and the sequence diagram as set of arrays containing transition labels of the respective automata. The verification task in this case is to check if each word (i.e., array of transition labels) is accepted by its automaton.

The workflow of the merging process, as depicted in Fig. 3, is as follows:

0. The set $\mathcal{M}$ of state machine diagrams is encoded in PROMELA automata. Other than for sequence diagrams, this encoding is done only once per application scenario.
1. The versioning operations *diff* (comparison) and *conf* (conflict detection) are executed on the original sequence diagram S and the two modifications S' and S''.
2. The versioning operation *merge* is performed based on the output of Step 1 and a merge strategy. In order to produce a syntactically correct sequence diagram, the merge strategy defines conditions on the possible orderings of the merged messages on a lifeline. A possible strategy is one that orders messages in a first-come, first-serve manner, or one that allows any possible combination. A strategy may allow more than one possible sequence diagram as result of the merge. In this case, the choice is made deterministically.
3. The output of *merge*, i.e., a syntactically correct sequence diagram $\hat{S}$, is encoded as set of PROMELA arrays, describing each lifeline as a word from the alphabet of the respective automaton encoded in Step 0.
4. The PROMELA code is fed into SPIN, which checks if each of the words generated in Step 3 is accepted by the respective automaton. It returns either a success message or the state and transition where the verification failed.

5. If the SPIN output does not contain an error message, the current merged sequence diagram $\hat{S}$ and the SPIN output are returned to the user. Otherwise, the procedure continues at Step 2 with a new merged sequence diagram $\hat{S}$ different from the previous ones.

For the encoding we make use of the following elements of PROMELA [8]:

– `active proctype`: defines a process behavior that is automatically instantiated at program start;
– `label`: identifies a unique control state (we also use the prefix `end`, which defines a termination state);
– `mtype`: a declaration of symbolic names for constant values;
– `array`: a one-dimensional array of variables (we use arrays of `mtype` elements to encode words checked by the automaton);
– `if`: a selection construct, used to define the structure of the automaton; and
– `goto`: an unconditional jump to a label, also used to define the structure of the automaton.

The PROMELA encoding of a state machine is done as follows:

– The state machine is encoded as `active proctype` that contains all the necessary elements of the state machine.
– Each transition $t \in T$ is encoded as an element of `mtype`. The additional element `acc` is added to model transitions to the `end` state.
– Each $q \in Q$ is encoded as a label marking a state of the `active proctype`. The additional state `end` is added.
– The state q_0 is placed at the beginning of the respective process in order to be executed at process initiation.
– τ is encoded as a set of `if` conditions inside each PROMELA state q: For each t such that (q, t) is defined by τ, the current symbol of an input sequence (which is, as described below, the encoding of a lifeline) is compared to t. If the condition holds, a `goto` statement jumps to state $\tau(q_0, t)$.
– Our sequence diagram semantics does not require a lifeline to end with a specific message, so all states are accepting states. We thus place a transition `goto end` if the current symbol equals our additional transition label `acc` into each state except the `end` state.

A lifeline is encoded as array `S` of `mtype`. Each field of `S` with index `i` contains the `mtype` element $tr(e_i)$ where e_i is the i-th element of the sequence L.

The PROMELA code is executed as simulation. It prints a success message if the word encoded in the array is accepted. In this case, the lifeline is consistent with the corresponding state machine. Otherwise it aborts when it hits a transition label that is undefined in the current state.

We have implemented the outlined approach based on the Eclipse Modeling Framework (EMF)[4]. In particular, the presented language excerpt of UML has been specified as an Ecore-based metamodel. The transformations of state machines into PROMELA

[4] `http://www.eclipse.org/modeling/emf`.

automata and sequence diagrams into PROMELA arrays have been implemented as model-to-text transformations using Xpand[5]. The implementation is available at `http://modelevolution.org`.

4 Application Scenario

We illustrate our approach using the example from Section 1. First, we translate the state machine of Fig. 1 by means of the encoding presented in the previous section as follows:

- The state machine is defined as `active proctype` named `Coffeemachine`.
- The transition labels of the coffee machine, along with an additional label `acc`, are contained in `mtype`.
- Each state of the coffee machine is represented by a label, such as `Off` or `Idle`, and an `end` state is added. The start and end states of the coffee machine are summarized in label `Off`.
- For each state, all defined transitions are encoded using `if` and `goto` statements.
- A counter is added to keep track of the current index of the input word.

Listing 4.1: State machine encoding in PROMELA.

```
mtype = {turnOff,turnOn,prepareCoffee,coffeeComplete,prepareTea,teaComplete,
         acc};

 active  proctype Coffeemachine() {
byte h = 0;
mtype CM[3];

CM[0] = turnOn; CM[1] = prepareCoffee; CM[2] = coffeeComplete; CM[3] = acc;

Off:
 printf("Off\t %e\n", CM[h]);
 if
 :: CM[h] == turnOn -> h++; goto Idle
 :: CM[h] == acc -> goto end
 fi;
Idle:
 printf("Idle\t %e\n", CM[h]);
 if
 :: CM[h] == prepareCoffee -> h++; goto PreparingCoffee
 :: CM[h] == prepareTea -> h++; goto PreparingTea
 :: CM[h] == turnOff -> h++; goto Off
 :: CM[h] == acc ->  goto end
 fi;
PreparingCoffee:
 printf("PreparingCoffee\t %e\n", CM[h]);
 if
 :: CM[h] == coffeeComplete -> h++; goto Idle
 :: CM[h] == acc -> goto end
 fi;
PreparingTea:
 printf("PreparingTea\t %e\n", CM[h]);
 if
 :: CM[h] == teaComplete -> h++;  goto Idle
 :: CM[h] == acc -> goto end
 fi;
end:
 printf("end!\n")
}
```

[5] `http://www.eclipse.org/modeling/m2t/?project=xpand`.

The sequence diagram contains one relevant lifeline, the instance cm of the coffee machine, which is encoded as array `CM` of `mtype`: For each message n_i received by cm, `CM[i]=` $tr(n_i)$. Recall that tr returns an element of the set of transition labels and that those are encoded as elements of `mtype`.

The resulting encoding of the state machine with the initial version S of the sequence diagram is shown in Listing 4.1. It is easy to see that the above code eventually reaches the `end` state. Replacing the array `CM` by the two modified sequence diagrams S' and S'', encoded in the same manner, the code also reaches the `end` state. However, on the merged sequence diagram $\hat{S}_1$, given in the following, the model checker will give up when it reaches the `Off` state trying to match `CM[4]`.

```
mtype CM[7];
CM[0] = turnOn; CM[1] = prepareCoffee; CM[2] = coffeeComplete; CM[3] = turnOff;
CM[4] = prepareTea; CM[5] = teaComplete; CM[6] = acc;
```

On the other hand, the second merged sequence diagram $\hat{S}_2$, given in the following, is consistent. Hence, in our merging workflow, $\hat{S}_2$ will be returned to the user.

```
mtype CM[7];
CM[0] = turnOn; CM[1] = prepareCoffee; CM[2] = coffeeComplete;
CM[3] = prepareTea; CM[4] = teaComplete; CM[5] = turnOff; CM[6] = acc;
```

5 Conclusion and Future Work

In this paper, we proposed to use a model checker to detect semantic merge conflicts in the context of model versioning. Model checkers are powerful tools used for the verification of hardware and software. A model checker takes as input a model of a system and a formal specification of the system and verifies if the former meets the latter. We applied this technique to check the semantic consistency of an evolving UML sequence diagram with respect to state machine diagrams that remain unchanged. When contradicting changes occur, a unique automatic merge is not possible in general. However, additional information on violations of the model's semantics allows to identify invalid solutions. Hence, a more goal-oriented search for a consistent merged version is supported.

Our first experiments on this approach gave promising results, but for the full integration into the versioning process several issues have to be considered which we discuss in the following.

Extension of the Language Features. So far, we considered only a restricted, simplified subset of the UML metamodel. In this setting, the execution semantics of the considered diagrams is quite unambiguous. With the introduction of more advanced concepts, several questions concerning the execution semantics will arise, which are not covered by the UML standard and need detailed elaboration in order to translate them to the formalisms supported by the model checker. When including these language features, we expect to fully exploit the expressiveness of LTL for the needed assertions.

Integration in the Merge Component. We use the information obtained by the model checker not only to verify the consistency of two diagrams, but to support the merge process as necessary when models are versioned in an optimistic way. At the moment, only the fact that the model checker failed to verify the provided encoding is propagated back to the merge component. We plan to build an analyzer which is able to deduce constraints from the output of the model checker. These constraints can then be used to create an alternative merged version.

Visualization of the Conflicts. For reasons of usability, the representation of conflicts is of paramount importance. In particular, we conjecture that conflicts have to be reported in the concrete syntax of the modeling language [3]. Therefore, we propose a mechanism based on UML profiles to include merging information directly into the model. We plan to extend this mechanism to report semantical problems in the concrete syntax.

Benchmarking. Finally, we need more test cases to evaluate our approach. In particular, it will be interesting to learn about precision and recall in various merging scenarios as well as to study scalability with growing model size.

References

1. P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. The Past, Present, and Future of Model Versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011.
2. P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, and P. Langer. Adaptable Model Versioning in Action. In *Modellierung*, volume 161 of *LNI*, pages 221–236. GI, 2010.
3. P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 184–193. Springer, 2011.
4. P. Brosch, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *IWMCP @ TOOLS'10*, pages 42–49, 2010.
5. J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL Operation Contracts. In *7th Int. Conf. on Integrated Formal Methods*, pages 40–55. Springer, 2009.
6. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *11th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'08*, volume 5301 of *LNCS*, pages 311–325, 2008.
7. A. Egyed. UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models. In *29th Int. Conf. on Software Engineering*, pages 793–796. IEEE, 2007.
8. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
9. V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi. Formal Verification and Validation of UML 2.0 Sequence Diagrams using Source and Destination of Messages. *ENTCS*, 254:143–160, 2009.
10. S. Maoz, J. O. Ringert, and B. Rumpe. A Manifesto for Semantic Model Differencing. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 194–203. Springer, 2010.
11. T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *9th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS'06*, volume 4199 of *LNCS*, pages 200–214. Springer, 2006.
12. P. Pelliccione, P. Inverardi, and H. Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *TSE*, 35(3):325–346, 2008.
13. T. Reiter, K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis. Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In *MDEIS @ ICEIS'07*, pages 29–40, 2007.
14. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *ENTCS*, 55(3):357–369, 2001.