Feature-based Development of State Transition Diagrams with Property Preservation

Christian Prehofer

fortiss GmbH, Munich, Germany, `Prehofer@fortiss.de`

Abstract. In this paper, we consider incremental development of state transition diagrams by adding features, which add new states and transitions. The goal is to capture when properties of a state transition diagram are preserved when adding a feature. We classify several typical cases of such state transition diagram extensions and show when commonly used properties are preserved. In some cases, we add restrictions to the input events. In others, we need to transform properties to account for new failure cases. Properties are specified on the externally visible input and output events. To formalize the properties and to reason about internal state transition diagram extensions we use a computation tree logic with states and events.

1 Introduction

The idea of *incremental development* is to start with a base model and then to add small features in succession, which add previously unspecified behavior. Here, we extend state transition diagrams (SDs) by features, which means to add new states and transitions. With extension of an SD we refer to such a syntactic extension. The core question addressed here is whether properties are preserved when incrementally extending an SD by a new feature which adds new states and transitions.

As an example consider Figure 1, where a new, alternative path is added to an SD. By convention, added features are shown in red, with bold lines and bold labels. In this example, consider the main property that the output *"Issue ticket"* occurs eventually. As can be seen easily, the extension by this new path preserves this property. As a second example, consider the example in Figure 2, which adds a new loop. In this case, the above property (i.e., that a ticket is

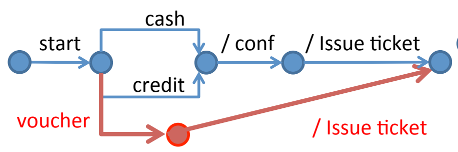

Fig. 1. Adding an alternative Path

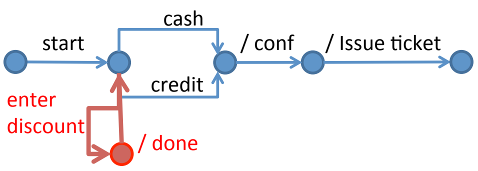

Fig. 2. Adding a local Loop

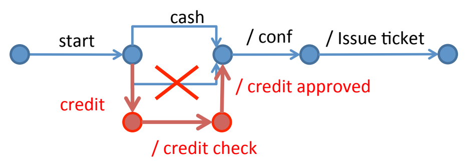

Fig. 3. Refining a Transition

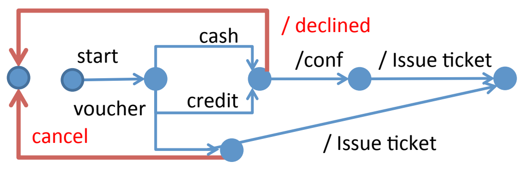

Fig. 4. Adding Failure Paths

issued eventually) is not preserved, as the loop may be traversed infinitely many times. Next, consider Figure 4, which adds two new failure paths. In this case, the above property that a ticket is issued is not preserved. We have two options here. As the first option, we may assume that such a failure does not occur. Alternatively, we may also transform the property to account for this new case.

More precisely, we consider the case where an SD is extended by a new feature, which adds new states and transitions. We classify several typical cases of such SD extensions and show when commonly used properties are preserved. In some cases, we also transform properties to account for new failure cases.

Our approach is, on a conceptual level, similar to adding aspects on a programming language level. For this, [3] has analyzed typical patterns of aspects w.r.t properties expressed in temporal logic. However, to our knowledge, such an approach has not yet been pursued for state transition diagrams. The main advantage here is that we can formalize properties on the control flow to determine when a property is preserved. Also, we adapt properties for expressiveness, unlike [3].

We formalize properties in a specific computation tree logic (CTL) which considers both states and events, following the logic in [9, 6]. Properties are specified on the externally visible traces of input and output events. To analyze when properties are preserved, we also need to capture the possible traversals of the state transition diagram, i.e., the internal view of states. This is why conventional CTL logics with consider either states or events are insufficient (see e.g. [7] for a comparison).

There exists ample work on refinement for SDs, e.g. [8], which aims to preserve all properties of an existing system, which is however not applicable in many cases. Hence we present custom solutions for specific features of SDs and specific classes of properties.

Other work on modularity for model checking [2, 5] considers the problem of extending automata models by new states and transitions. In these works, composition of statecharts leads to proof obligations for specific properties to maintain. These are in turn to be validated by a model checker. Similar goals have been pursed in the context of aspect-modeling for state machines in [10]. These approaches require the specification and establishment of each individual property after the extension.

2 State Transition Diagrams and Event/State-based Temporal Logic

We model software systems by SDs, which we formalize as labeled transition systems.

Definition 1 (Labelled Transition System). *A* labelled transition system *(or* LTS*) is a structure $L = (S, s_0, A, \rightarrow)$ where*

- *S is a set of states.*
- *$s_0 \in S$ is the initial state.*
- *Two disjoint sets I and O of input and output events and the silent event τ is not in I nor O.*
- *Pairs of input and output events $A = (I \cup \{\tau\}) \times (O \cup \{\tau\})$, called labels.*
- *$\rightarrow \subseteq S \times A \times S$ is the* transition relation*; an element $(r, \alpha, s) \in \rightarrow$ is called a* transition*, and is usually written as $r \xrightarrow{\alpha} s$.*

To model practical examples we use explicit labels with a pair of input and output events. We write $(s, (i, o), s')$ or (s, α, s'), where $\alpha = (i, o)$, for transitions. We let $A_\tau = A \cup \{\tau\}$. We let $\alpha, \beta, \ldots$ range over A_τ. As I and O are disjoint, we also write i instead of (i, τ) and o instead of (τ, o).

Let $L = (S, s_0, A, \rightarrow)$ be an LTS. A sequence $(s_0, \alpha_0, s_1)(s_1, \alpha_1, s_2) \ldots \in \rightarrow^\infty$ is called a *path* from s_0; if a path cannot be extended anymore because it is either infinite or ends in a state without outgoing transitions, it is called a *maximal path*. We write $\text{state}_i(\pi)$ for the i'th state of the path, $i \geq 0$. We write $\text{event}_i(\pi)$ for the i'th event of the path, $i \geq 0$.

In the following, we briefly introduce a computation tree logic (CTL) based on both states and events. This is needed as we aim to separate the external view on an SD, in terms of events, and the internal view in terms of states, as discussed below.

Our logic, called ESCTL, is a special case of the UMC logic presented in [9]. In more detail, UMC permits multiple labels on a transition; here we use pairs of input and output events, which is easily embedded in UMC.

Definition 2 (ESCTL). *The syntax of the logic ESCTL(Event/State-based CTL) is defined by the following grammar where we let $\phi, \phi', \ldots$ range over* ESCTL-state formulas*: $\phi ::= T \mid s \mid \neg\phi \mid \phi \wedge \phi' \mid E\varphi \mid A\varphi$ where $s \in S$ and φ is a path formula.* ESCTL-path formulae *are formed according to the following grammar: $\varphi ::= \mathrm{X}\phi \mid \mathrm{X}_\chi\phi \mid \phi_\chi U_{\chi'} \phi' \mid \phi_\chi W_{\chi'} \phi'$ where ϕ and ϕ' are ESCTL-state formula and χ and χ' are event formulae. An* event formula *is defined by the following grammar where we let χ, χ' range over event formulas: $\chi ::= \mathit{true} \mid \alpha \mid \neg\chi \mid \chi \wedge \chi'$*

In the following we will say state and path formula instead of ESCTL-state and ESCTL-path formula.

Let $L = (S, s_0, A, \rightarrow)$ be an LTS. The satisfaction relation $\models$ between states $s \in S$ and state formulae is defined as usual. For a detailed treatment, we refer to [9]. Satisfaction of path formulae by maximal paths is defined as follows:

$$\begin{aligned}
\pi \models \mathrm{X}\phi \quad & \text{iff } \mathrm{state}_1(\pi) \models \phi;\\
\pi \models \mathrm{X}_\chi\phi \quad & \text{iff } \mathrm{event}_0(\pi) \models \chi \text{ and } \mathrm{state}_1(\pi) \models \phi;\\
\pi \models \varphi_\chi U_{\chi'}\varphi' \quad & \text{iff } \exists j \geq 0 : \mathrm{state}_j(\pi) \models \varphi \wedge \mathrm{state}_{j+1}(\pi) \models \varphi' \wedge \mathrm{event}_j(\pi) \models \chi' \wedge\\
& \forall 0 \leq k < j. \mathrm{state}_k(\pi) \models \varphi \wedge \mathrm{event}_k(\pi) \models \chi;\\
\pi \models \varphi_\chi W_{\chi'}\varphi' \quad & \text{iff } \pi \models \varphi_\chi U_{\chi'}\varphi' \text{ or } \forall k \geq 0. \mathrm{state}_k(\pi) \models \varphi \wedge \mathrm{event}_k(\pi) \models \chi;
\end{aligned}$$

Satisfaction of event formulae by events is defined as usual [9]. Informally, $\pi \models \varphi_\chi U_{\chi'}\varphi'$ holds if for some path π, the property φ holds on the states and χ holds on the events, until a transition with (s_j, α, s_{j+1}) occurs, where χ' holds for α and φ' holds for s_{j+1}.

We define the usual operators F and G, as well as F_χ, for a event χ, as follows: $F = \neg T, EF\phi = E(T_{true}U\phi), AF\phi = A(T_{true}U\phi), EF_\chi\phi = E(T_{true}U_\chi\phi), EF_\chi = E(T_{true}U_\chi T), AG\phi = \neg EF\neg\phi, AG_\chi = \neg EF_{\neg\chi}, EG\phi = AG\neg\phi, E(\varphi_\phi U\varphi') = \varphi' \vee E(\varphi_\phi U_\phi \varphi')$. Without ambiguity, we omit T in formulae. Similarly, we write eU_χ instead of $T_e U_\chi$.

Note that our combined event/state logic treats events and state separately, and many properties on states are easier to formalize than similar ones on events. For instance consider a property state s implies state s'; we denote this as $AG(s \rightarrow AFs')$, which is defined as $AG(\neg s \vee AFs')$. On the other hand, an event e implies P is denoted as $AG(e \rightarrow P)$, defined as $AG((X_e(AFP)) \vee X_{\neg e}T)$.

3 SD Composition and Property Preservation

In the following, we first motivate our specifications in state/event logic and then classify properties on SDs. This is followed by a detailed analysis of SD composition for different cases.

We use our event/state logic in order to specify externally visible behavior and internal behavior, respectively. Assume an SD S and a set of input traces I consisting only of input events.

- We express properties on *externally visible behavior* by properties on input and output events. No formulae with states are permitted, as these are considered internal.
- *(Full) behavior specifications* use properties over states, input and output events created by S, using all of the above.

One reason why we use externally visible events for behavior is that two composed SDs may share input or output events. This is not possible if only names of states are considered. On the other hand, we need states to reason about internals of a composition, as discussed below.

3.1 Specification Patterns

Our goal is to study what properties are preserved when adding a new feature to an SD. For the properties to consider, we use the specification patterns as in [4]. From the extensive study in [4], it was observed that around 80%-90% of all properties in specifications are of the three kinds. First, we have the pattern "a leads to b", called response in [4]. Formally, we have, $AG(a \rightarrow AF_b)$. The other two classes, called universality and absence, are invariants stating that a property holds globally or something never happens. Such invariants must hold for all states and/or transitions, which means that validation is compositional in this sense. We should note here that the properties in [4] also consider different temporal scopes for the validity of the formulae. This is not considered here for simplicity.

In this paper, we hence focus on leads to or response properties. Together with the invariants, this covers a significant amount of specifications. The other pattens in [4] formalize precedence of states and chained response and precedence. We conjecture that other patterns can be handled in a similar way, but detailed analysis is left to further work.

3.2 SD Composition

In the following, we define formally how to extend an SD by a new feature. We also use SDs to denote such extensions, which are then glued together at specified join states. Based on join states, similar to join points in aspect-oriented languages, we use graphical notation to denote the composition of SDs. For a more formal definition, we refer to [8].

For adding a new path, consider the schematic view in Figure 5. By convention, the SD E in red color represents the path to be added to the blue, base SD S. We assume that E and S have disjoint states, and further that E has initial state s' and a state j'. In S, we assume a start state s and a join state j for adding a path. The two SDs will be composed by merging these two states, s' and j', with the states s and j, respectively. This is denoted by the light dashed lines. In the merged state, we use the states s and j for these. Formally, the states s' and j' are renamed to s and j in E, and the two SDs are then merged to obtain $S \cup E$. Note that we do not use the same names for the states i and j in S and E to be merged in order to avoid confusion when expressing preconditions on E or S.

We also consider refinement of transitions as follows. We write $S - E$, where E denotes a set of transitions in S which are removed. Based on this, we can express transition refinement by $S - E \cup E'$ as shown in Figure 3. If E removes a single transition for which a corresponding path with same start and end states and the same trigger event in the first transition in this path, then we denote this as a transition refinement by $S \cup^r E'$ and leave E implicit (without ambiguity).

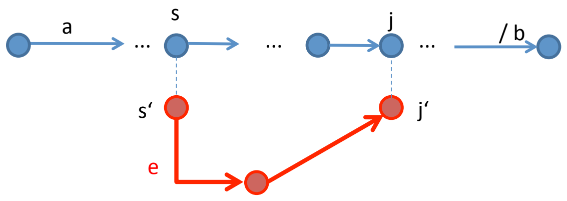

Fig. 5. Adding an Alternative Path (Schematic view)

3.3 Property Preservation for Response/Leadsto Properties

The main goal of this section is to define rules for the extension of an SD. As defined above, assume an SD S which is extended to $S \cup E$ by an extension E. The goal is now to define criteria when a property for S, e.g. $S \models P$, also holds for $S \cup E$, i.e., $S \cup E \models P$.

This achieves modularity on properties w.r.t. extension, and also simplifies formal proofs as $S \cup E$ is larger than S and E. Furthermore, we will see that many properties can be obtained easily by local analysis of the state diagram. We should observe also that some properties are not preserved by extensions. For this, we will transform the properties into extended properties with extra conditions.

Adding Alternative Paths We consider the schematic view of adding an alternative path in Figure 5. We use the states s, s' and j, j' as shown in this figure. However, the schematic case as in Figure 5 is simplified for illustration and does not show all cases covered by this rule. More precisely, this case is specified by the following assumptions: First, there is a path from s to j in S, i.e., $S \models AG(s \rightarrow EF_j)$. This means that j is reachable from s on at least one path; thus we add an alternative path to this. Second, $E \models AG(s' \rightarrow AF_{j'})$. This means that j' is reached eventually when entering E. Furthermore, we assume $s \neq j$ to avoid trivial loops which are covered in a separate case below. Third, we assume that j' is a final state in E. Hence we have no newly added infinite loops and termination only in state j for E.

We consider first a property of the form $AG(a \rightarrow AF_b)$. The main idea here is to identify cases when an alternative path does not obstruct this property. In other words, the event b is observed in any path after a, even with the new alternative path.

Assuming S, E with states s, s' and j, j' as specified above for Figure 5. In case $S \models AG(a \rightarrow AF_b)$ holds for S, we obtain $S \cup E \models AG(a \rightarrow AF_b)$ if one of the following holds:

- $S \models AG(a \rightarrow (\neg s U_b))$
- or $S \models AG(s \rightarrow (\neg j W_b))$
- or $E \models AG(s' \rightarrow AF_b)$.

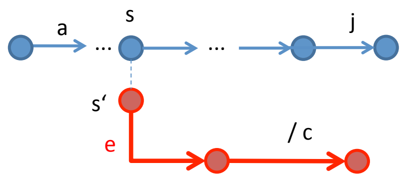

Fig. 6. Adding a Failure Path

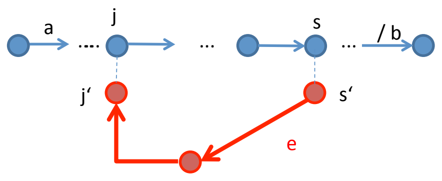

Fig. 7. Adding a Fallback Path

Adding a Failure Path We consider the schematic view of adding a failure path in Figure 6. The assumptions for this case are as follows: there is a failure event c which occurs in E, E is entered by event e and furthermore that there is no paths back to the SD S. Formally, we have no j' in this case and $E \models AG(e \rightarrow AF_c)$.

In case of leadsto properties, we have several cases. The base case happens when the property is not affected by the failure, i.e., if the added path is not taken before b, formally $a \rightarrow (\neg s U_b)$. Then, we have the case that b always occurs in E, similar to above.

If these simple cases do not hold, we have the option to weaken the property. Assuming $S \models AG(a \rightarrow AF_b)$ and $AG(E \models e \rightarrow AF_c)$, we obtain $S \cup E \models AG(a \rightarrow AF_{c \vee b})$.

Adding Fallback Paths The idea of a fallback path is that a new path is added, similar to the alternative path case. Here, the new path leads to a state already traversed earlier. Thus we fall back to an earlier state and create a possible loop, which is different from the above cases.

We consider the schematic view of adding an alternative path in Figure 7. As above, the SD E in red color represents the path to be added to the blue, base SD S, with join states s and j. We assume that E has initial state s' and a final state j'.

The fallback case, as illustrated schematically in Figure 7, is specified by the following assumptions: First, there is a path from j to s in S, i.e., $S \models AG(j \rightarrow EF_s)$. Secondly, $E \models AG(s' \rightarrow AF_{j'})$. Third, j' is a final state in E. The main problem in this case is that the fallback path adds a loop which may be traversed infinitely often.

Consider a response property of the form a leads to b. The main idea here is to identify cases when an alternative path does not obstruct this property. In other words, the event b is observed in any path after a, even with the new alternative path.

Assuming S, E with states s, s' and j, j' as specified above for Figure 7. Assuming $S \models AG(a \rightarrow AF_b)$, we obtain $S \cup E \models AG(a \rightarrow AF_b)$ if

– $S \models AG(a \rightarrow (\neg s W_b))$
– or $E \models AG(s' \rightarrow AF_b)$.

For adding a fallback path, we have another important case. If we can avoid non-termination by the added loop of the fallback path, we can also establish

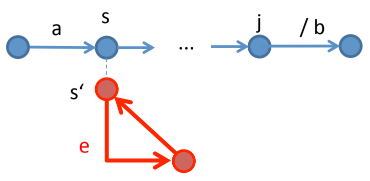

Fig. 8. Adding a local Loop

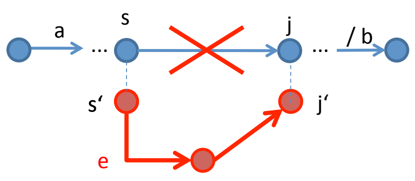

Fig. 9. Refining a Transition

the desired property. It is sufficient to express that in state s, the input e does not occur infinitely often.

The problem is that we cannot express such fairness properties in a CTL-logic. We can formalize this as $S \cup E \models \neg AG(s \wedge X_e)$, but this is a CTL* formula and not a CTL formla. Fortunately, for our state/event logic, there also exists an extension, including model checking, for the μ-calculus, which can embed CTL* [6]. Another, suitable solution is to add fairness conditions on the permitted input events, as done for CTL-logic in [1]. We expect that this can also be transferred to action/state CTL logics, but this goes beyond the scope of the paper.

Refining Transitions In this case, a transition is refined into several new states and new transitions. This is typical when moving from a higher level model to a more detailed model, which shows more details by refining a transition into several transitions. Note that removing transitions is a special case of this refinement. An example is shown in Figure 9.

The schematic case is illustrated in Figure 9. Formally, it is specified by the following assumptions: First, there is a transition from s to j in S. Second, $E \models AG(s' \rightarrow A_{j'})$. This means that j is reached eventually when entering E. Hence we have no infinite loops and termination only in state j. Refining a transition could be modeled by removing a transition and adding a new path; for refinement it is however more effective to consider this in conjunction.

We consider first a property of the form $a \rightarrow AF_b$. The main idea here is to identify cases when the added path does not obstruct this property. In other words, the event b is observed in any path after a, even with the new alternative path.

Assuming $S \models AG(a \rightarrow AF_b)$, we obtain $S \cup^r E \models AG(a \rightarrow AF_b)$ if

- $S \models AG(a \rightarrow (\neg s W_b))$
- or $S \models AG(s \rightarrow (\neg j W_b))$
- or $E \models AG(s' \rightarrow AF_b)$.

The cases above are as in the case of adding a transition: In the first case, $(\neg s U b)$ means s never happens before b. Hence b occurs always before the alternative path is taken. In the second case, $s \rightarrow (\neg j W_b)$ means this: if s happens, then b happens after j. This is to avoid the case that b only occurs between s and j. Finally, $E \models AG(s' \rightarrow AF_b)$ means that b happens in E when the path is taken. Hence the property also holds for the combined SD.

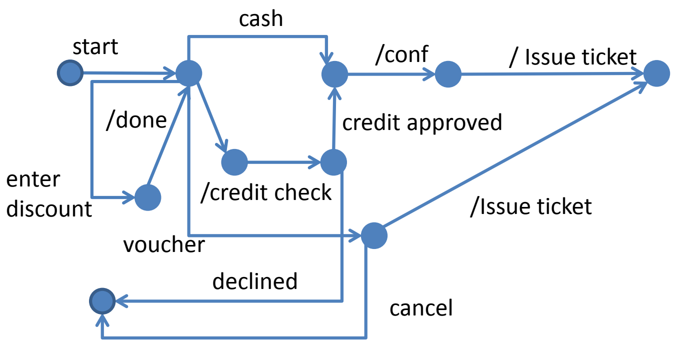

Fig. 10. Combining the Examples

3.4 Example

In the following, we apply the above results to the example in the introduction. We consider the base SD as in Figure 1 and the property $AG(start \rightarrow AF_{Issue\_ticket})$. Combining the examples from Section 1, we obtain the SD in Figure 10

1. Adding the voucher option in Figure 1 preserves this property.
2. Adding the discount option in Figure 2 does not preserve this property. Assuming a fairness precondition, this turns into $AG(start \rightarrow AF_{Issue\_ticket})$. More formally, we need to assume in the following that $enter\_discount$ does not occur infinitely often in sequence.
3. Refining the credit transition in Figure 3 preserves this last property.
4. Adding the failure cases as in Figure 4 does not preserve this property. We need to weaken the property as follows: $AG(start \rightarrow AF_{declined \vee cancel \vee Issue\_ticket})$, assuming the above fairness precondition regarding $enter\_discount$.

4 Conclusions

In this paper, we have presented a new approach for incremental development of state transition diagrams. We have developed a classification of individual features to be added to an SD, and then discussed when properties can be established. In some failure cases, we had to modify the properties to account for failure cases. In case of loops, we may need a notion of fairness for some properties. The conditions for the rules only assume properties of the SDs to be composed, which means that the rules are modular. Hence, one benefit is that we can validate the rules more efficiently. In many cases the conditions can be checked by simple analysis on the SD level.

We have formalized our rule by using a recently developed Event/State CTL logic. This permits us to express properties on externally visible behavior in terms of input and output events, while expressing composition conditions by internal states. We have validated our rules by several examples in a workflow example. We assume that our rules and examples are easy to express in a model checker for our state/event logic, e.g. in the UMC tool [6].

The main novelty is to focus on different kinds of properties individually, which permits new ways for incremental development. Regarding other works on refinement, we note that usual notions of refinement preserve all properties, which is too strong for our case. Further work will address other patterns of property specifications, also including different scopes.

Acknowledgements The author would like to thank Sebastian Bauer, Martin Wirsing, Franco Mazzanti and Rolf Hennicker for stimulating discussions.

References

1. C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.
2. C. Blundell, K. Fisler, S. Krishnamurthi, and P. Van Hentenrvck. Parameterized interfaces for open system verification of product lines. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 258 – 267, sept. 2004.
3. S. D. Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 135–145, New York, NY, USA, 2008. ACM.
4. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411 –420, 1999.
5. J. Liu, S. Basu, and R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18, 2011.
6. F. Mazzanti. UMC logics. http://fmt.isti.cnr.it/umc/V4.1/umc.html. [Online; accessed July 9th, 2013].
7. R. Nicola and F. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.
8. C. Prehofer. Assume-guarantee specifications of state transition diagrams for behavioral refinement. In *iFM 2013: 10th International Conference on integrated Formal Methods*. Springer-Verlag, June 2013.
9. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119 – 135, 2011.
10. G. Zhang and M. Hölzl. Hila: High-level aspects for uml state machines. In S. Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, 2010.