Modelling Undefined Behaviour in Scenario Synthesis

Sebastian Uchitel, Jeff Kramer, Jeff Magee
Department of Computing, Imperial College London
{su2, jk, jnm}@doc.ic.ac.uk

Abstract

Current approaches to scenario synthesis do not distinguish, in the resulting state machine models, proscribed behaviour from behaviour that has not yet been defined. In this paper we propose using partial labelled transition systems (PLTS) to capture what remains undefined of the system behaviour. In the context of scenario synthesis, we show that PLTSs can be used to provide feedback to stakeholders on the parts of the behaviour specification that need further elaboration. In this way we aim to support the iterative incremental elaboration of behaviour models.

1. Introduction

Scenario-based specifications are partial descriptions of system behaviour. A scenario conveys relatively little information; it depicts an example of how system components interact. Hence, a scenario-based specification will typically have many scenarios that cover most common system behaviours and possibly some exceptional ones too. Scenario-based specifications are not particularly well suited for exhaustive description of all possible system traces and it is natural to assume that the absence of a scenario in a specification does not imply that it is an undesired system trace.

Some scenario-based notations do provide mechanisms for explicit specification of undesired system behaviour (e.g. conditions [3, 4], negative scenarios [8]); nevertheless scenario-based specifications will generally leave gaps in the specification; that is, examples of system behaviour that have not been described explicitly as positive (intended) or negative (unintended) system behaviour.

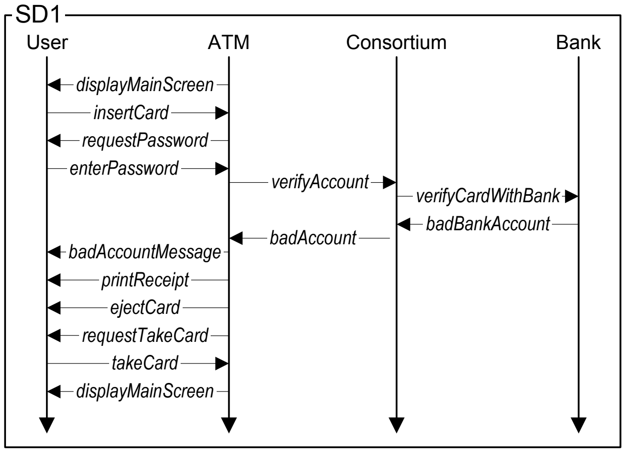
Conversely, state machine based formalisms such as labelled transition systems (LTS) are generally assumed to be complete descriptions of system behaviour up to some level of abstraction (a fixed alphabet of actions): if a labelled transition system cannot exhibit a certain sequence of actions, it is assumed that the system or component it models cannot or should not exhibit that sequence.

The difference in interpretation of scenarios and state machines is one of the causes for the former to be used in early requirements phases of the development life-cycle, where system descriptions are relatively partial and require elaboration; while the latter tend to be used more advanced stages such as design, where a more comprehensive knowledge of the system to be is available. This separation has also led to significant efforts in developing synthesis techniques that allow constructing state-machine models from scenario-based descriptions [5, 7, 9, 10].

Clearly, a question that needs to be addressed is how scenario synthesis techniques cope with moving from a partial to complete specifications. What happens with those system traces that have not been described as positive or negative behaviour? In general, they all get bundled as negative behaviour (e.g. [5, 7, 9]). Some synthesis algorithms use information from other specifications to make the synthesised state machines more accurate (we shall be looking at one of these approaches: [10]). However, even in these cases, some system behaviour may remain unspecified and will invariable fall into the positive or negative behaviour modelled in the state machine model.

Consequently, from the perspective described above, approaches to scenario synthesis lose the distinction between proscribed behaviour and behaviour that has not yet been defined. We believe that in the context of supporting the elaboration of behaviour models this is a missed opportunity. Knowing where the gaps are in a behaviour model permits the presentation of meaningful questions to stakeholders, which in turn can lead to model exploration and thus more comprehensive descriptions of the system behaviour [8]. Hence, there is a case for using, in the context of scenario synthesis, an extended notion of state machine that can explicitly capture undefined behaviour and support reasoning about aspects of system behaviour that need further elaboration.

In this position paper we use partial labelled transition systems (PLTSs) as the target model for scenario synthesis. We follow Whittle and Shumman's approach [10] and show how additional and relevant feedback can be obtained when using PLTSs. More specifically, we start from a scenario-based specification of an ATM machine and an Object Constraint Language (OCL) based specification of message pre- and post-conditions. We use the synthesis approach of [10] to build a LTS, and then extend it to a PLTS that models which message preconditions do not hold on each state. We show that the resulting model can be used to identify behavioural aspects of the ATM system that were under-specified in the original specification. Furthermore, these undefined scenarios are not differentiated from proscribed behaviours in the synthesised model produced in [10]; hence missing an opportunity for model elicitation and elaboration. We also show how composition of PLTS can be used to combine different partial behaviour models and (potentially) reduce the number of undefined system scenarios. This supports detection and validation of gaps in the system behaviour, rather than on details of specific components, which may be irrelevant to the overall system behaviour. We conclude this position paper with some comments on future work.



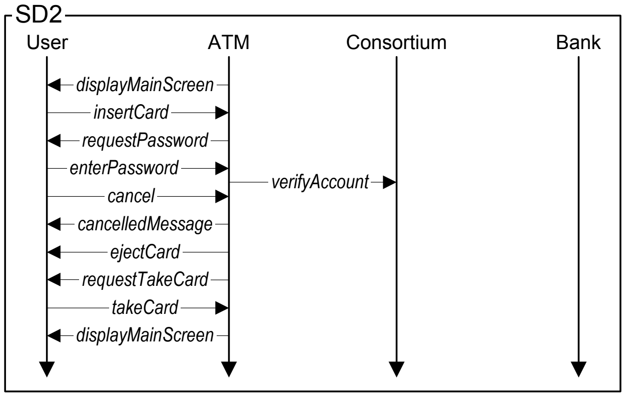
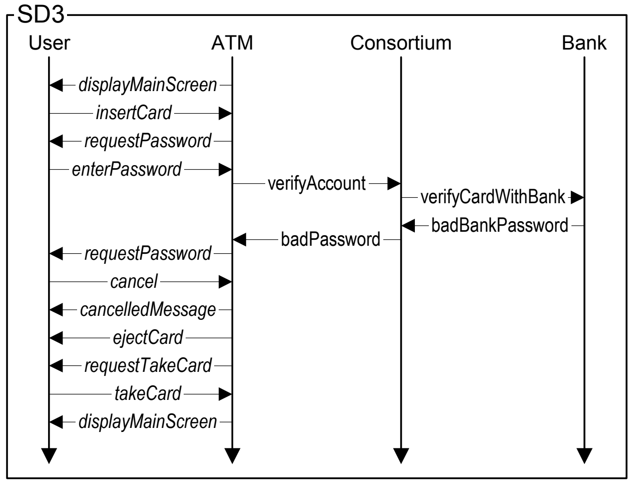


Figure 1 - Scenarios SD1 and SD2

2. LTS Synthesis

The example we use to illustrate our approach is based on a version of the ATM case study presented by Whittle and Schumann in [10]. A number of sequence diagrams (depicted in Figure 1 and Figure 2) describe how a user operates a bank account by interacting with an ATM. The ATM is connected to a network run by a consortium, which in turn interacts with the bank. In addition to the scenarios, pre- and post-conditions for some scenario messages are given in OCL (Figure 4). The post-conditions specify how messages modify the values of a set of ATM state variables (*cardIn*, *cardHalfway*, *passwdGiven*, *card*, and *passwd*). The pre-conditions specify the values these variables are expected to have before a messages occurs.



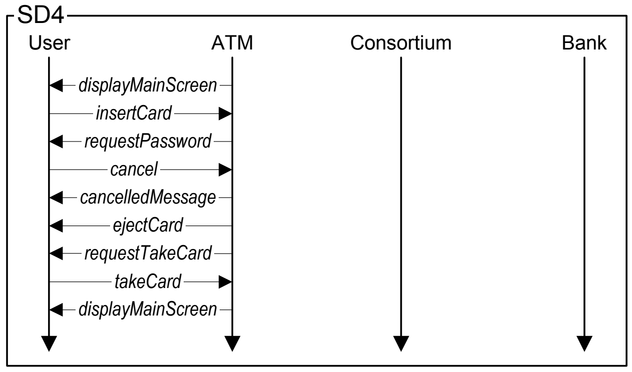


Figure 2 - Scenarios SD3 and SD4

In [10] a synthesis procedure is presented for automatically generating LTSs from the combination of the scenarios and the pre- and post-conditions. Using the pre- and post conditions, the procedure first infers the value of state variables at specific points of the scenarios. For example, for *displayMainScreen*, the first message in SD1, the OCL specification states a pre-condition that allows inferring that the value of *cardIn* and *cardHalfway* should be false at the beginning of SD1.

By considering all message pre- and post-conditions and using the unification and frame action techniques defined in [10] it is possible to infer further information on the value of state variables throughout the available scenarios. Consequently, it is possible to assign a (possibly partial) valuation of state variables to every scenario state (the gap on a scenario instance between two consecutive events). The valuations are then used to infer which scenario states should be modelled with one state in the LTS to be synthesised. We do not go in to more details on the synthesis procedure; the interested reader can refer to [10] for more information.

3. Undefined Versus Proscribed Behaviour

Figure 3 depicts the LTS for the ATM component synthesised from the scenario and OCL specification. As expected, the model captures the sequences of interactions the ATM component performs in the scenarios. For instance the LTS models an ATM that is capable of performing the sequence of actions <*displayMainScreen*, *insertCard*, *requestPassword, enterPassword, verifyAccount…*> of scenario SD1. Additionally, by omission, the LTS also models the sequences of actions that the ATM cannot perform. Thus, the ATM cannot perform the sequence <*displayMainScreen*, *insertCard*, *insertCard*> because after performing *displayMainScreen* and *insertCard* the LTS is in state 2 which does not have any outgoing transitions labelled *insertCard*. For exactly the same reasons, the ATM LTS cannot perform the following sequence: <*displayMainScreen*, *insertCard*, *ejectCard*>.

However, a closer inspection of the LTS and the OCL specification reveals that the LTS is over-specifying the behaviours that the ATM should not be capable of. Table 1 shows the value of the OCL variables in each state of the ATM LTS. Considering that the pre-condition of message *insertCard* requires variable *cardIn* to be false, we can infer that in state 2 *insertCard* should not occur. This is consistent with the fact that the LTS for the ATM component does not allow <*displayMainScreen*, *insertCard*, *insertCard*>. Contrarily, message *ejectCard* requires *cardIn* to be true, hence its precondition is satisfied in state 2. Consequently, there is no reason to dismiss the possibility of the ATM performing the sequence <*displayMainScreen*, *insertCard*, *ejectCard*>. However, the LTS for the ATM component does not allow this sequence.

```
cardIn, cardHalfway, passwdGiven : Boolean
card : Card
passwd : Sequence
insertCard(c : Card)
pre : cardIn = false
post: cardIn = true and card = c
enterPassword(p : Sequence)
pre : passwdGiven = false
post: passwdGiven = true and passwd = p
takeCard()
pre : cardHalfway = true
post: cardHalfway = false and cardIn = false
displayMainScreen()
pre: cardIn = false and cardHalfWay = false
post:
requestPassword()
pre : passwdGiven = false
post:
ejectCard()
pre : cardIn = true
post: cardIn = false and cardHalfway = false
and card = null and passwd = null and
passwdGiven = false
requestTakeCard()
pre : cardHalfway = true
post:
canceledMessage()
pre : cardIn = true
post:
```

Figure 4 - OCL pre- and post-conditions

Clearly, our knowledge of sequences <*displayMainScreen*, *insertCard*, *insertCard*> and <*displayMainScreen*, *insertCard*, *ejectCard*> is different. We know the first one should not occur because it would violate the *insertCard* pre-condition. Whilst for the second sequence we know it does not violate any pre-conditions, thus it may be a valid ATM behaviour. This

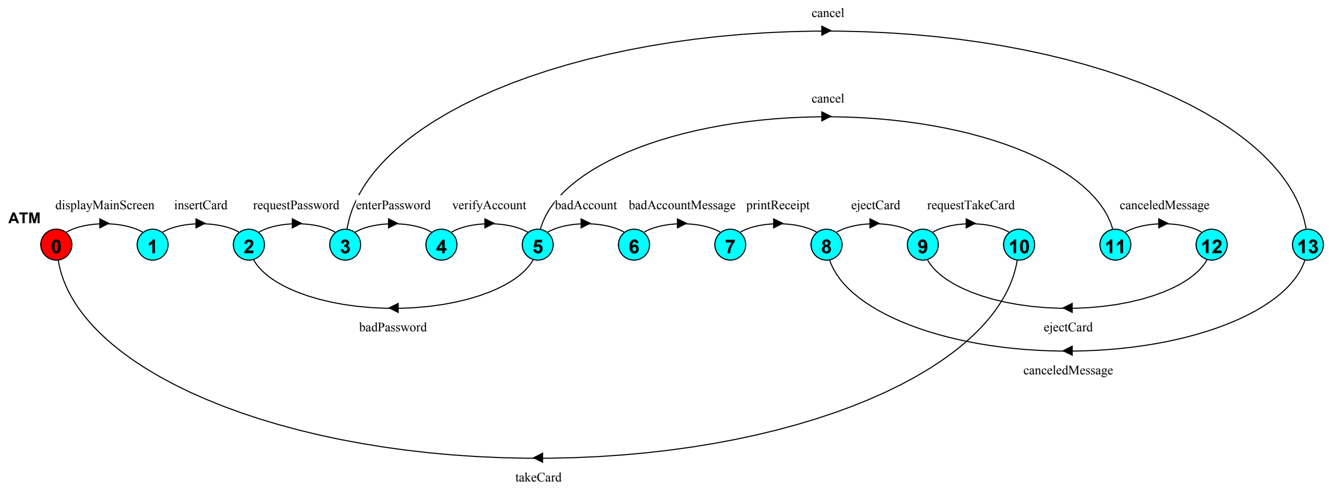


Figure 3 - Synthesized LTS for the ATM component

means that it may well be a situation that has not been explicitly specified or that has not even been considered by stakeholders. Hence it is an opportunity for providing feedback that may trigger new scenarios or strengthened pre-conditions. Either way, it is an opportunity for further elaborating the system's behaviour model.

Table 1 - Valuation of variables on states

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
cardIn	f	f	t	t	t	t	t	t	t	f	f	t	t	t
cardHalfway	f	f	f	f	f	f	f	f	f	t	t	f	f	f
passwdGiven	t	t	f	f	t	t	t	t	t	f	f	t	t	f
card	-	-	c	c	c	c	c	c	c	-	-	c	c	c
passwd	-	-	-	-	p	p	p	p	p	-	-	p	p	-

4. Partial Labelled Transition Systems

There is benefit to be gained from differentiating in LTS models behaviour that is known to be undesired from behaviour that is not yet known to be positive or negative. To capture this information we extend the notion of LTS as follows. Each LTS state is associated with a set of labels. These labels model actions that are explicitly proscribed at that state. Clearly, if an action is proscribed at a state then there can be no outgoing transitions from that state with the same label. Thus, we have that on each state every label of the PLTS alphabet is enabled, proscribed or undefined. That is, the following equation holds for each state *s*:

$$\alpha(\boldsymbol{P}) = \boldsymbol{enabled(s)} \cup \boldsymbol{proscribed(s)} \cup \boldsymbol{undefined(s)}$$

where *enabled*(*s*) is the set of labels for which there is an outgoing transition form *s* and *undefined*(*s*) is the set of labels that are not enabled nor proscribed in state *s*. Note that if *undefined*(*s*) = ∅ for all states, the PLTS can be considered a LTS.

If we contrast the preconditions of Figure 4 with the valuation of state variables for each state of the ATM LTS (Table 1) we can determine which message should not occur on each state. For instance, the precondition of *insertCard* determines that it cannot occur on states 2 to 8, 11 to 13. Consequently, we can extend the LTS of Figure 4 with the sets modelling the messages proscribed at each state. Cells marked "p" in Table 2 represent the pairs of proscribed action labels at sets for the ATM PLTS.

We can then add to Table 2, the information on enabled messages for each state (cells marked "e"). For example, as a transition labelled *insertCard* has been defined from state 1 to state 2, *insertCard* is marked as enabled in state 1 in Table 2.

Pairs of messages and labels that are not marked with either "e" or "p" are highlighted with "**?**" and model the states where messages could occur (according to their preconditions), but the consequences of such occurrences are not yet known. Thus, we have states 0, 9 and 10 modelling that *insertCard* could occur, but the state to which its occurrence would lead to is not known.

Table 2 can now be used to prompt stakeholders on hypothetical situations. For instance, the fact that in state 0, insert card is undefined may prompt the following question: Can a card be inserted into the ATM before a message is displayed?

Note that PLTS differ from multi-valued state-machines (e.g. [1]), in that in the latter transitions are assigned truth values (e.g. true, false, unknown) rather than transition labels being undefined at states. Thus, multi-valued state-machines require much finer grained knowledge about what is unknown. For instance, in the ATM example we would have to speculate on all the possible destinations of *insertCard* from state 0: transitions labelled *insertCard* with value unknown would be needed from state 0 to all other states. In the setting we propose, this is not useful as the true transition from state 0 for insert card –supposing that it should exist, but has not appeared in the given scenarios– could lead to a new PLTS state altogether.

Table 2 - Classification of ATM states

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
insertCard	**?**	e	p	p	p	p	p	p	p	**?**	**?**	p	p	p
enterPassword	**?**	**?**	**?**	e	p	p	p	p	p	p	p	p	p	**?**
takeCard	p	p	p	p	p	p	p	p	p	**?**	e	p	p	p
displayMainScreen	e	**?**	p	p	p	p	p	p	p	p	p	p	p	p
requestPassword	**?**	**?**	e	**?**	p	p	p	p	p	p	p	p	p	**?**
ejectCard	p	p	**?**	**?**	**?**	**?**	**?**	**?**	e	p	p	**?**	e	**?**
cancelledMessage	p	p	**?**	**?**	**?**	**?**	**?**	**?**	**?**	p	p	e	**?**	e
requestTakeCard	p	p	p	p	p	p	p	p	p	e	**?**	p	p	p

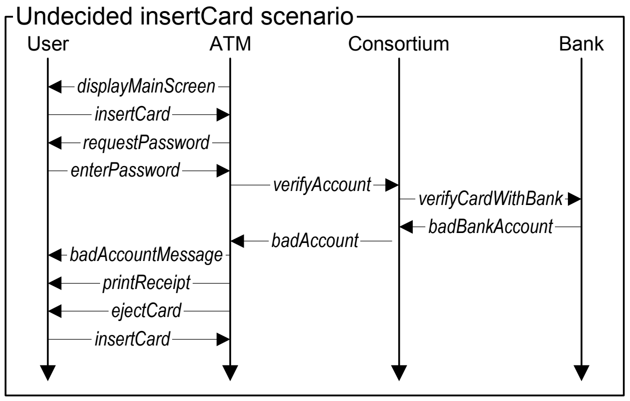


Figure 5 - Undefined insertCard scenario

5. Composition of Partial Behaviour Models

Although benefits may be obtained from inspecting a PLTS, a more appealing approach is to generate feedback in the form of scenarios. Thus, we want to compose partially specified models of the system components appearing in scenarios and to reason about how they interact and if they reach states for which certain message labels are undefined. For instance, if a PLTS for the user, consortium and bank where built, the scenario of Figure 5 could automatically be generated through some kind of reachability analysis. The scenario depicts the case where after a session, the ATM ejects the card, which is left halfway in the machine, and the user instead of taking the card pushes it back in.

Thus, we need to extend the notion of parallel composition of LTSs to PLTSs. Intuitively, parallel composition of LTSs models a system in which components execute asynchronously and synchronize on shared observable message labels. Given a shared label *a,* one LTSs can take an *a*-labelled transition if and only if the LTS it is being composed with can do so too. Consequently, a LTS in a state where *a* is not enabled, will prevent the other LTS from taking a transition labelled *a*.

In the parallel composition of PLTSs this changes because *a* not being enabled does not imply that *a* is proscribed. For instance, suppose we are composing PLTSs *P* and *Q,* which are in states *p* and *q* respectively. In addition suppose there is a shared label *a* that is enabled in *p*. If *a* is undefined on state *q* then *a* should also be undefined in the composite process because we do not know if *Q* can synchronize on *a* when in *q*. Clearly, if *a* is proscribed in *q,* then *a* should be also proscribed in the composite process (as with standard LTSs).

On the other hand, if *a* is undefined in state *p*. If *a* has been explicitly proscribed on state *q* then *a* should also be proscribed in the composite process. This means that the fact that *a* is undefined in *p* is irrelevant with respect to the composite behavior. In other words, although we have a gap in the specification of component *P*, providing feedback on it is not necessary in the context of *Q*.

More generally this allows us to combine different partial behaviour models and (potentially) reduce the number of unclassified system scenarios. Allowing detection and validation of gaps in the behaviour specification to focus on the emerging behaviour of system components working together, rather than on details of components that may be irrelevant to the overall system behaviour.

Table 3 provides a summary intuition as to how enabled, proscribed and undefined message labels work together in parallel composition of PLTSs.

Table 3 – Proscribed, enabled and undefined messages in PLTS parallel composition

	Enabled	*Proscribed*	*Undefined*
Enabled	Enabled	Proscribed	Undefined
Proscribed	Proscribed	Proscribed	Proscribed
Undefined	Undefined	Proscribed	Undefined

Consider the ATM example, and suppose we have additional information on the how the card is managed between the ATM and the user, which we call for short the *card protocol.* We describe the behaviour with a PLTS depicted in Figure 6 and Table 4. If we compose the behaviour models for the card protocol and the ATM, the resulting PLTS will no longer have the following pairs of undefined behaviour (compare with Table 2): {(*insertCard*, 9), (*insertCard*, 10), (*ejectCard*, 3), (*ejectCard*, 4), (*ejectCard*, 5), (*ejectCard*, 6), (*ejectCard*, 7), (*requestTakeCard*, 10), (*takeCard, 9*)}.

As a consequence, we know have a composite behaviour model that has fewer gaps that need stakeholder validation.

Although the preceding example reduces the number of undefined pairs of states and labels (compared to the ATM component on its own), this is not always the case. Clearly, composition of PLTSs can introduce new cases of undefinedness in the composite behaviour. Thus, parallel composition does not always reduce the number of gaps in the overall specification that need stakeholder validation.

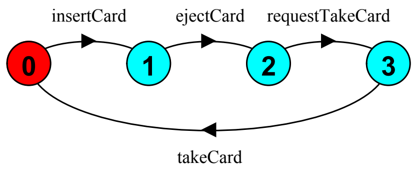


Figure 6 - Card Protocol

Table 4 - Classification of Card Protocol states

	0	1	2	3
insertCard	e	p	p	p
takeCard	p	p	?	e
ejectCard	p	e	p	p
requestTakeCard	p	p	e	p

6. Future Work and Conclusions

In this paper we have shown how explicit modelling of aspects of system behaviour that are unknown can be beneficial in the context of scenario-based specifications. Synthesis of PLTSs captures the gaps in the behaviour model that need to be elaborated. As a result undefined scenarios can be used as cues to stakeholders for providing the behaviour information that is missing from the model; hence, supporting the iterative incremental elaboration of behaviour models.

Although we have used in this paper an example based on scenarios and OCL pre- and post-conditions, the use of PLTS can be useful when used with other sources of negative behavioural information such as MSC conditions [3, 4] or negative scenarios [8].

We envisage using PLTSs to support the elaboration of behaviour models. Unknown behaviour can be modelled explicitly, and then models can be used to query users on whether a particular scenario is possible or not. Our previous work on implied scenarios [18] shares this approach to model elaboration based on scenario generation and validation. However, implied scenarios address a very specific aspect of partial scenario-based descriptions while PLTSs provide a more general framework for model elaboration. This approach to model iterative construction of scenario-based specifications is shared with work of Mäkinen and Systä [15]. However, in their work scenarios that are fed back to users are the result of over-generalisations of the synthesis procedures used. The scenarios we aim to generate are a result under-specification.

We are currently only starting to experiment with PLTSs as the target for scenario synthesis. We are working on the formal definitions of PLTS and PLTS parallel composition, implementing synthesis procedures and detection methods that allow detecting undefined scenarios. They are being developed on LTSA [6] exploiting its new fluent linear temporal logic (FLTL) model checking features. We are also looking into using undefined scenarios in the context of our simulation tool to guide model exploration and elaboration. Undefined scenarios could also provide a framework for guided play-in scenarios [2]

7. Acknowledgements

We gratefully acknowledge the support of the European Union under grant STATUS (IST-2001-32298), and EPSRC grant READS GR/S03270/01.

References

[1] M. Chechik, S. Easterbrook, and B. Devereux, *Model Checking with Multi-Valued Temporal Logics* in 31st IEEE International Symposium on Multiple Valued Logics (ISMVL'01), Warsaw, 2001.

[2] D. Harel, *From Play-In Scenarios to Code: An achievable Dream*, IEEE Software, vol. 34, pp. 53-60, 2001.

[3] D. Harel and W. Damm, *LSCs: Breathing Life into Message Sequence Charts* in 3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, New York, 1999.

[4] ITU, *Message Sequence Charts*, International Telecommunications Union. Telecommunication Standardisation Sector, Recommendation Z.120, 2000.

[5] I. Krüger, R. Grosu, P. Scholz, and M. Broy, *From MSCs to Statecharts* in *Distributed and Parallel Embedded Systems*, F. J. Rammig, Ed.: Kluwer Academic Publishers, 1999, pp. 61-71.

[6] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. New York: John Wiley & Sons Ltd., 1999.

[7] E. Mäkinen and T. Systä, *MAS – An Interactive Synthesizer to Support Behavioral Modeling in UML,* in 23rd IEEE International Conference on Software Engineering (ICSE '01), Toronto, 2001.

[8] S. Uchitel, J. Kramer, and J. Magee, *Negative Scenarios for Implied Scenario Elicitation* in 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'02), Charleston, 2002.

[9] S. Uchitel, J. Kramer, and J. Magee, *Synthesis of Behavioural Models from Scenarios*, IEEE Transactions on Software Engineering, vol. To appear., 2002.

[10] J. Whittle and J. Schumann, *Generating Statechart Designs from Scenarios* in 22nd International Conference on Software Engineering (ICSE'00), Limerick, Ireland, 2000.