Raising Level of Abstraction with Partial Models: A Vision

Marsha Chechik[1], Arie Gurfinkel[2], Sebastian Uchitel[3], and Shoham Ben-David[1]

[1] Department of Computer Science, University of Toronto
[2] SEI/CMU
[3] University of Buenos Aires and Imperial College London

1 Introduction and Position

We support the goals of the workshop to concentrate community efforts towards usable verification. We believe that the keys to addressing this problem are *abstraction* (i.e., raising the level of abstraction at which the software is designed) and *automation* (i.e., creating automated and scalable tools for reasoning about such software at the highest level of abstraction possible).

Specifically, we advocate designing and constructing software systems by starting with high-level "abstract" models or by synthesizing operational models from (declarative) specifications. Then, such models can be refined by adding detail, as necessary, while preserving the properties of interest. Such an approach, which builds on a recent trend in model-driven development (MDD), may allow "building quality and reliability" into the software from the very beginning of the design life-cycle.

Our position is that in order to provide *usable verification* in this domain, we need notations and techniques with the following properties:

1. **Handling incompleteness.** It is clear that "abstract" models are incomplete – i.e., some details are hidden ("abstracted") away. An ability to specify such incomplete models and reason with them is essential. The required automation involves
   (a) reasoning (using model-checking or theorem proving) about such models w.r.t. a variety of (temporal, behavioral, correctness) properties, including, properties that may depend on details that are not present at a given level of abstraction and, therefore, may warrant an "I do not know" answer.
   (b) refinement (i.e., integrating new information such as additional behavior and/or additional data objects). Refinement must preserve properties established at a higher level of abstraction (otherwise, all correctness established earlier in the software lifecycle does not carry over to later stages).
   (c) merge (combining information from multiple sources). For example, integrating (possibly conflicting) viewpoints of different stakeholders such as the end-users, maintainers, the database, etc.
   (d) operationalization (an ability to simulate, test, and validate a model). A representative example is constructing a model representing a parallel composition of several components described at different levels of abstraction and/or from viewpoints of different stakeholders.

The motivation is to do the analysis as early as possible in the design lifecycle – before the complete software is built.

2. **Handling inconsistency identification and resolution.** Models early in the design lifecycle, models with multiple stakeholders, or models coming from different sources are bound to be inconsistent. Therefore, support for tolerating (e.g., by not trivializing the entire logical inference) and resolving (e.g., through computer-aided negotiation) inconsistency must be provided.
3. **Support for operationalization of models.**
   (a) Code generation. Once a model is consistent and has sufficient level of detail, it must be converted into running code. Ideally, this step should include sufficient traceability to allow for the use of the original model for debugging and maintenance in downstream activities.
   (b) Synthesis. We believe that more analyses are possible when models of software are operationalized, e.g., expressed in the form of state machines. Such operational models can be produced directly by users or synthesized from higher-level, declarative specifications such as temporal logic formulas and scenarios.
   (c) Validation. Operational models can and should be validated using simulation and testing, and model-checked against properties of interest. Ideally, results of validation should help the developer determine how to refine the models further.

We expect that the combination of these properties allows us to start with high-level operational models and, though property-preserving refinement, ensure that the resulting software systems will have the desired properties.

Our vision is related to the classical refinement methodologies like the B method [1], to the traditional synthesis approaches like [15] and to the MDD techniques [12]. Yet, the vision is unique in the combination of the above three characteristics. For example, refinement uses partiality but does not include operationalization. Synthesis does operationalization but does not explicate partiality of the solution. MDD techniques allow inconsistencies but do not capture partiality explicitly.

While there are several modeling formalisms and reasoning frameworks that can support our vision, in this paper, we concentrate on *partial behavioral models* and some of our recent and on-going projects.

A lot of notations and formalisms for this work have been developed with the aim of capturing abstractions of complex software systems. That is, while these formalisms omitted various details, they were available in the underlying concrete models. In contrast, in our work we do not assume presence of concrete models and thus each partial model essentially represents a *set* of concrete models – those that can be obtained via refinement.

The rest of this paper is organized as follows: In Section 2, we survey related work on partial modeling formalisms and formally define and illustrate one of them – Modal Transition Systems (MTSs). In Section 3, we summarize our recent results on using MTSs as the formalism for our usable verification vision. Finally, in Section 4, we describe some limitations of the MTS-based framework and motivate some of our current/future work in this field.

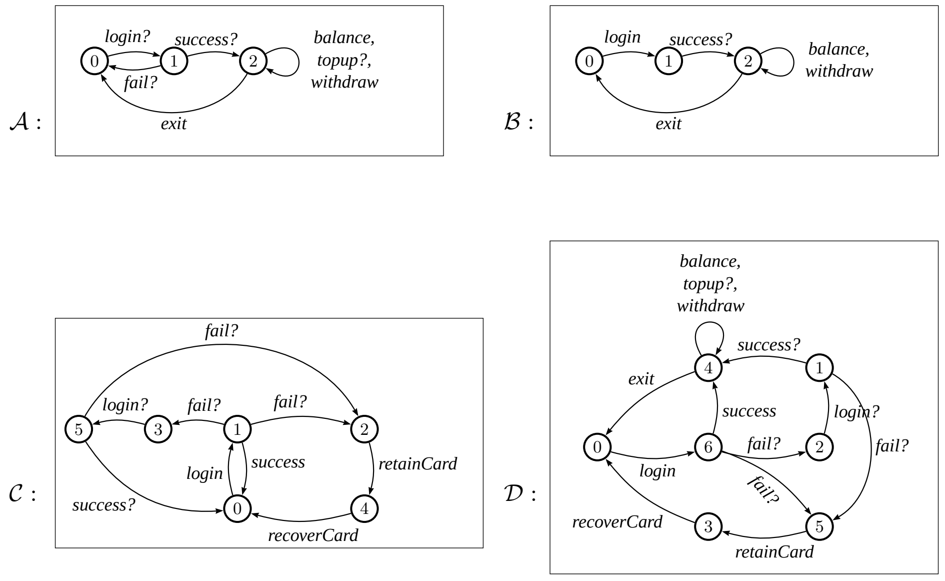

Fig. 1. Some MTSs: $\mathcal{A}$ and $\mathcal{C}$ are ATM models; $\mathcal{B}$ is a refinement of $\mathcal{A}$; $\mathcal{D}$ is a merge of $\mathcal{A}$ and $\mathcal{C}$.

2 Partial Models

We begin with a well-known concept of Labeled Transition Systems.

Definition 1. (Labeled Transition System) *A* Labeled Transition System *(LTS) is a tuple $L = (S, A, \Delta, s_0)$, where S is a finite set of states, A is a set of actions, $\Delta \subseteq (S \times A \times S)$ is a transition relation between states, and $s_0 \in S$ is the initial state.*

Modal Transition Systems (MTSs) [10, 14] allow explicit modeling of what is *not* known about the behavior of a system. They extend LTSs with an additional set of transitions that model the interactions with the environment that the system cannot guarantee to provide, but, equally, cannot guarantee to prohibit.

Definition 2. (Modal Transition System) *A* Modal Transition System *(MTS) M is a structure $(S, A, \Delta^r, \Delta^p, s_0)$, where $\Delta^r \subseteq \Delta^p$, (S, A, Δ^r, s_0) is an LTS representing* required *transitions of the system and (S, A, Δ^p, s_0) is an LTS representing* possible *(but not necessarily required) transitions of the system.*

MTSs specify *partial behavioral models* which distinguish between required, prohibited, or unknown behaviors. When depicting MTSs, we enumerate states for reference and assume that state 0 is the initial state. *Required* transitions are denoted by a solid labeled arrow. Transitions that are *possible but not required* (a.k.a. *maybe* transitions) are denoted by a question mark following the label. Transitions on sets are shorthand for a single transition on every element of the set.

For example, consider a specification of software controlling a bank Automated Teller Machine (ATM). The specification may consist of a number of use cases exemplifying how the ATM is to be used, and some properties it is expected to satisfy. An example use case is “when a user has successfully logged in, i.e., inserted a valid card and keyed in a valid password, the user must be offered the following choices: withdraw cash, balance slip or log out”. In addition, some ATMs may provide an optional feature of topping up a pay-as-you-go mobile phone. A possible safety property of an ATM is to prohibit withdrawals, balances and top-ups if the user is not logged in. An operational model, in the form of an MTS that captures the above use-case and property, is depicted in model $\mathcal{A}$ in Figure 1. Here, the initial state of the model is labeled 0, the transition from state 0 on *login* is allowed (but not required); all other transitions from state 0 are prohibited. If the system has provisions for logging in the user and the login is successful, the user (in state 2) must be given a choice to withdraw cash, obtain a balance or exit. The *top-up* feature is optional, i.e., allowed but not required.

$\mathcal{C}$ in Figure 1 is another ATM model. Unlike $\mathcal{A}$, which gives an overview of the entire ATM system, $\mathcal{C}$ concentrates just on the possible protocols for a failed login attempt. It allows zero, one and two failures, and requires that in both cases there are *retainCard* and *recoverCard* transitions. This model specifies nothing about the operations allowed once a successful login has been achieved.

Refinement of MTSs maintains required behaviors, does not introduce behaviors that are prohibited, but can change unknown behaviors into required or prohibited. For example, model $\mathcal{B}$ of Figure 1 shows one possible refinement of model $\mathcal{A}$: the unknown transition on *login* became a required transition, and the optional transition on *fail* and the self-loop on *top-up* in state 2 were omitted.

MTSs are members of a large family of partial modeling formalisms including Partial Labeled Transition Systems (PLTSs) [18], multi-valued state machines [8], Mixed Transition Systems [7] and multi-valued Kripke structures [4, 6, 11] among others.

3 Modeling and Analysis with Modal Transition Systems

In our work [5, 10, 16, 17], we have studied MTSs as the underlying formalism for enabling usable verification. Specifically, our goal was to interpret an MTS as a concise representation of a set of LTSs and to define refinement as resolving partiality. Moreover, we found it important to consider models with different vocabularies, describing different aspects from the perspectives of different stakeholders (and, hence, of diverse scopes, as correspond to the elaboration of behavioral models in practice). For example, models $\mathcal{A}$ and $\mathcal{C}$ in Figure 1 describe different aspects of the ATM model and thus their vocabularies overlap but are not the same.

In our study of MTSs, we concentrated on defining operations on MTSs: refinement, merge (defined as the least common refinement of two models), consistency checking (defined as absence of the merge of two models), synthesis, parallel composition and model-checking of MTSs [10, 16]. These operations were implemented in MTSA – a tool for specifying, animating and reasoning about MTS models [9]. A screenshot of MTSA is shown in Figure 2. We have not explored code generation.

We discuss some of MTS operations below. Merge is perhaps the more unusual one. Its goal is to combine information coming from various sources. Defined as the least common refinement of two models, it preserves all required behaviors of the models. It also preserves all prohibited behaviors (i.e., if neither model was allowed to have a

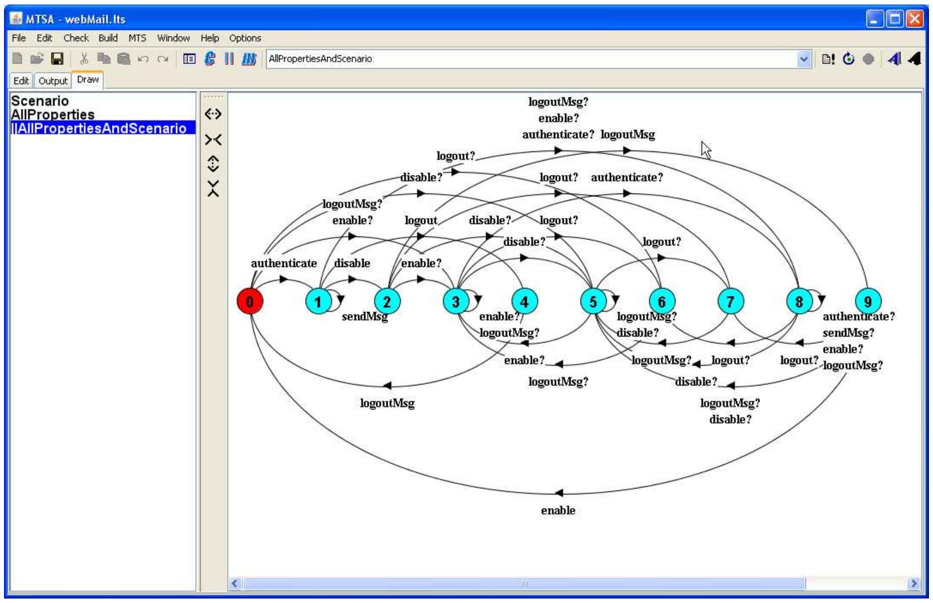

Fig. 2. MTSA screenshot – Draw View.

particular behavior then neither would their merge). However, if a particular behavior is possible but not required in one model and is required (prohibited) in the other, then it is required (prohibited) in the merge. For example, model $\mathcal{D}$ in Figure 1 is the merge of $\mathcal{A}$ and $\mathcal{C}$. Note that the *login* transition from state 0 is possible but not required in $\mathcal{A}$ but is required in $\mathcal{C}$; thus, it is required in the merge.

We showed that our refinement operator preserves properties expressed in Fluent LTL [13], whether they are true or false in the less refined model. Of course, properties which are "unknown" in the less refined model might become true or false in the more refined one, or remain unknown. For example, consider the safety property for the ATM which states that if the user is not logged in, withdrawals and balance checks are prohibited. It is expressed in Fluent LTL as $G(\mathit{LoggedOut} \Rightarrow (\neg \mathit{balance} \wedge \neg \mathit{withdraw}))$, where *LoggedOut* is a *fluent* (i.e., a state description) indicating that the user either has not executed a *login* or has *exit*ed from the system. This property holds for the model $\mathcal{A}$ shown in Figure 1 and thus is preserved in its refinement, $\mathcal{B}$.

Since merge is based on refinement, a property which is true (false) in two models is also true (false) in their merge.

To operationalize partial descriptions of behavior of different aspects of software, we have developed synthesis algorithms. These compute MTSs from safety properties described in temporal logic (they give the "upper bound" of the MTS behavior, specifying prohibited behavior), and from scenarios expressed as Message Sequence Charts [16] (they give the "lower bound" of the MTS behavior, specifying required behavior).

4 Challenges

In this section, we discuss some challenges in using MTSs as the partial behavioral models to support our Usable Verification position.

1. While MTSs can be used to detect inconsistency (and even potentially help with negotiation), they do not allow analysis in the presence of inconsistency. We have looked at other partial formalisms that allow such reasoning – most notably, multi-valued Kripke structures [6] and built a model-checking engine that can determine whether a (CTL) property evaluates to "inconsistent", in addition to true, false and unknown. However, it is unclear how to combine multi-valued Kripke structures with labeled transition-like approaches such as MTSs, to take advantage of other operations defined over them.
2. The main drawback of MTSs and similar operational partial modeling formalisms is that the complexity of many key operations associated with them is EXPTIME-complete [2, 3]. This restricts the use of MTSs to only modestly sized programs.
   Yet, our experience using MTSs for specification and modeling of (relatively complex) software systems indicates that most of their use is limited either to capturing particular traces or, at most, linear behavior with some branching parts.
   We are now working on defining a new formalism for which manipulations of partial models can be achieved relatively inexpensively without compromising too much on expressive power. A first attempt is to describe *required* and *possible* behavioral traces as regular expressions (REs). These can be represented as automata, and manipulated in linear time. However, regular expressions are less expressive than other modeling formalisms: for example, they cannot express triggered scenarios: if a prefix of a behavior trace appears, some suffix behavior must be possible (e.g., absence of deadlocks). Instead, we are looking to enrich linear temporal logic with trigger abilities, to identify a "sweet spot": a language with sufficient expressive power and yet with low-complexity of analysis for the software-engineering tasks outlined in Section 1.
3. Finally, our proposal is best categorized as "green field" research. While having many desirable characteristics, it is unclear how to combine our approach with more traditional methods for software development.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. A. Antonik, M. Huth, K. Guldstrand L., U. Nyman, and A. Wasowski. "EXPTIME-complete Decision Problems for Modal and Mixed Specifications". *Electronic Notes in Theoretical Computer Science*, 242(1):19–33, 2009.
3. N. Benes, J. Kretnsky, K. Larsen, and J. Srba. "Checking Thorough Refinement on Modal Transition Systems is EXPTIME-complete". In *Proceedings of ICTAC'09*, pages 112–126, 2009.
4. G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of Proceedings of 11th International Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 274–287. Springer, 1999.

5. M. Chechik, G. Brunet, D. Fischbein, and S. Uchitel. "Partial Behavioural Models for Requirements and Early Design". In *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, pages 1–10, 2007.
6. M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM Transactions on Software Engineering and Methodology*, 12(4):1–38, October 2003.
7. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, The Netherlands, July 1996.
8. R. Diaz-Redondo, J. Pazos-Arias, and A. Fernandez-Vilas. "Reusing Verification Information of Incomplete Specifications". In *Proceedings of the 5th Workshop on Component-Based Software Engineering*, 2002.
9. N. D'Ippolito, D. Fishbein, M. Chechik, and S. Uchitel. "MTSA: The Modal Transition System Analyzer". In *Proceedings of International Conference on Automated Software Engineering (ASE'08)*, pages 475–476, September 2008.
10. D. Fischbein, G. Brunet, N. D'Ippolito, M. Chechik, and S. Uchitel. "Weak Alphabet Merging of Partial Behaviour Models". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 1–49, 2010.
11. M. Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3-4):335–350, 1991.
12. D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.
13. D. Giannakopoulou and J. Magee. "Fluent Model Checking for Event-Based Systems". In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 257–266. ACM Press, September 2003.
14. K. Larsen and B. Thomsen. "A Modal Process Logic". In *Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society Press, 1988.
15. A. Pnueli and R. Rosner. "On the Synthesis of a Reactive Module". In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, January 1989.
16. S. Uchitel, G. Brunet, and M. Chechik. "Synthesis of Partial Behaviour Models from Properties and Scenarios". *IEEE Transactions on Software Engineering*, 3(35):384–406, 2009.
17. S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *Proceedings of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 43–52, November 2004.
18. S. Uchitel, J. Kramer, and J. Magee. "Behaviour Model Elaboration using Partial Labelled Transition Systems". In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 19–27, 2003.