Partial models: A position paper

Michalis Famelis   Shoham Ben-David   Marsha Chechik   Rick Salay

University of Toronto
Toronto, Canada
{famelis,shoham,chechik,rsalay}@cs.toronto.edu

ABSTRACT

Model-based software development inevitably involves dealing with incomplete information. Yet, MDE methodologies rarely, if ever, address uncertainty in a systematic way. Drawing inspiration from the field of behavioural modeling [6, 4], we propose to use *partial models* as first-class development artifacts to abstract, reason with, visualize and manipulate sets of possible alternative models. Aiming to set up a research agenda for a systematic and robust treatment of uncertainty in MDE, we discuss how uncertainty can be captured by partial models, and how these models can be validated and correctly refined.

1. INTRODUCTION

The vision of model driven software development is to create a seamless process during which models are incrementally refined from requirements to generated code. Of course, as any other software development endeavor, this process needs to handle making decisions in the presence of incomplete and uncertain information. Uncertainty can be injected during the MDE life-cycle due to a variety of reasons such as unclear requirements allowing multiple design alternatives [12], alternative resolutions to inconsistency [8, 3], multiple stakeholder opinions [9], and many others.

Current MDE methodologies do not specifically address the handling of incomplete information in a robust way. In fact, model transformations (including those that generate correct by construction model refinements) can only be applied to concrete models, and thus incompleteness is considered to be undesirable and needs to be removed as quickly as possible. Otherwise, the development either halts or proceeds in an ad-hoc way that does not guarantee correctness.

Reasoning with incomplete information has been studied extensively in the context of behavioural modeling and verification [2, 5, 1, 6, 4]. Our goal is much larger: we aim to produce an approach to adapt existing modeling formalisms and transformations to take uncertainty into account in a systematic and sound way, aiming to encompass heterogeneous modeling languages, a variety of different kinds of uncertainty, and, most importantly, focus on MDE-style transformations.

In this paper, we propose the adoption of *partial models*, i.e., abstractions of *sets* of alternative modeling solutions, as first-class development artifacts to capture uncertainty. Partial models can be visualized to ensure human comprehension, and transformations can be lifted from concrete to partial models. Further, partial models enable efficient reasoning tasks such as property checking. Finally, partial models support refinement aimed to reduce the degree of uncertainty (we call it *uncertainty-removing* (UR), to differentiate it from the standard *detail-adding* (DA) refinement), culminating in concrete models.

Connection to workshop theme. From the perspective of modeling and verification, the novelty in this approach is the use of such concepts as *abstraction*, *refinement* and *property preservation*, developed by the verification community [2, 5, 1], to reason about MDE-type models. Additionally, the partial model representation allows us to reason about a set of models as efficiently as about each concretization. From the perspective of model transformation, our approach has the advantages of removing the need to resolve uncertainty too early, solely in order to continue development. In addition, partial modeling allows us to define UR refinement as a systematic generic refinement mechanism with well understood properties.

Paper organization. The rest of the paper is organized as follows: Section 2 illustrates the key motivating example of how partial models can be used to support software development. The remainder of the paper discusses the research agenda, including questions that need to be answered, current status of our work and some future steps. Specifically, Section 3 discusses the representation of partial models, Section 4 examines reasoning, Section 5 looks at various kinds of transformations of partial models and the associated property preservation, and Section 6 concludes the paper.

2. EXAMPLE

We motivate and explain our approach using a model (M) of a simple network controller shown in Figure 1(a). The model includes a class diagram that contains the class `Controller`, and a state machine diagram that describes the behavior of the class. The controller can be switched on and off. While it is connected, it remains in the `On` state and if it becomes disconnected, it warns the user by beeping.

In our scenario, there are two developers: Alice manages the architecture and Bob the behavioral components. There is also a consistency requirement (C1) dictating that there should not be any sink states. Bob easily identifies that the

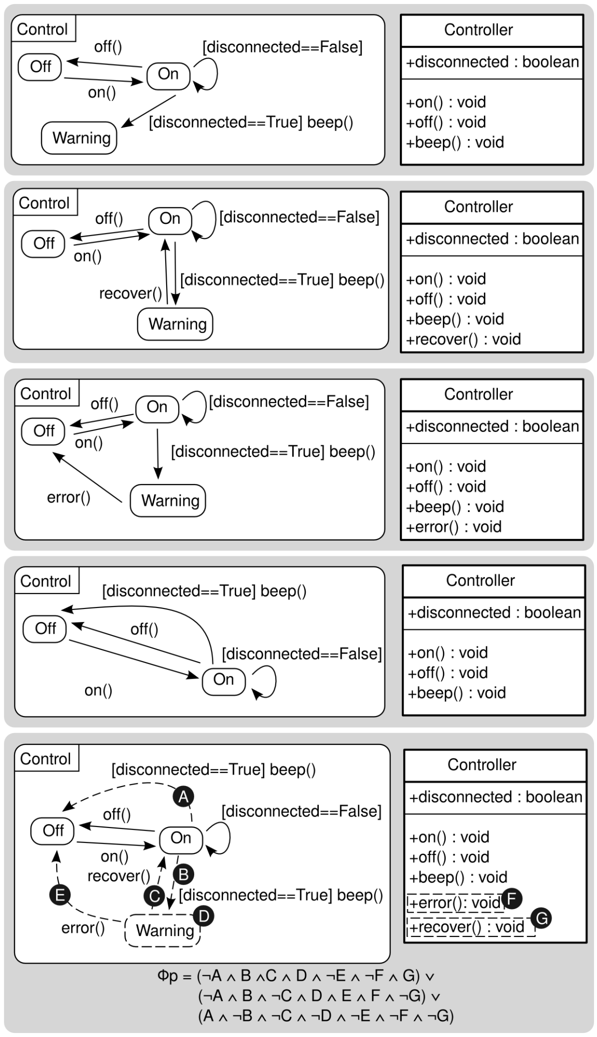


Figure 1: (a) An inconsistent model M. (b)-(d) Alternative repairs of M. (e) A partial model P capturing alternatives in (b)-(d). Dashed elements are optional.

network controller model is inconsistent with respect to C1: the state `Warning` has no outgoing transitions.

We assume that inconsistency fixing is done using repair rules [7, 8], and that two such rules for dealing with C1 exist: (R1) *"Add a transition from the sink state to some other state"* and (R2) *"Delete the sink state"*. Fixing is done semi-automatically, whereby Bob applies these rules, taking care to maintain the semantic integrity of the model.

The rules can be applied to the network controller model in three ways, resulting in three distinct alternative resolutions: 1) use R1 to add a transition from `Warning` to `On`, 2) use R1 to add a transition from `Warning` to `Off`, or 3) use R2 to delete the state `Warning`. Bob interprets each of these alternatives to ensure that the resulting state machine is meaningful and modifies the class diagram accordingly. If the state `Warning` is retained, returning to `On` signifies recovery, as shown in Figure 1(b). Turning the network controller `Off` logs an error message, as shown in Figure 1(c). If `Warning` is deleted, a disconnect causes the controller to beep and shut down, as shown in Figure 1(d).

Which of the choices should Bob make? We assume that at this stage in the development, the requirements regarding whether the network controller should have recovery capabilities are still unclear. Additionally, Bob needs to have Alice approve any modifications to the architectural model, as they could have unwanted consequences to the `Controller` class's inheritance hierarchy. Faced with these uncertainties, Bob can either halt development, waiting for the uncertainty to be lifted, or make a guess and choose one of the three alternatives, therefore risking having to undo his work. Yet another option for Bob, the one we advocate here, is to capture all possible ways of fixing the inconsistency in a single partial model, P, shown in Figure 1(e), and proceed with development using it as the primary development artifact.

Visualization. The partial model P is presented in pseudo-concrete UML syntax for brevity. It contains elements that are tagged as optional, indicated visually by dashed lines. It is also accompanied by a propositional formula, Φ_p, which describes the allowed combinations of the optional elements. Each such combination represents a *concretization* of the partial model, i.e., a concrete model that can be derived from P by removing all uncertainty. In this example, P has exactly three concretizations, corresponding to the three alternative repaired models.

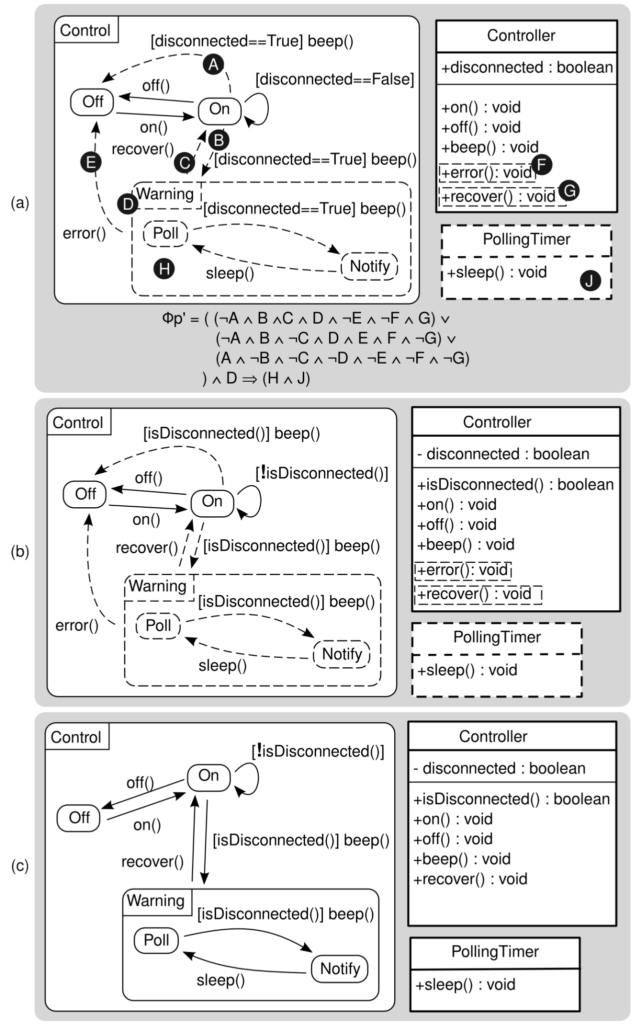
Analysis. Bob can use P to check properties. For example, the result of checking property C1 is `True`, as none of the concretizations of P have a sink state. Thus, the outcome of checking C1 is interpreted as "property holds regardless of how the partial model will be concretized". Suppose an additional requirement (C2) dictates that multiple transitions with exactly the same source and target states should not exist. Checking C2 on P results in `Maybe`, as it holds for some concretizations of P (those in Figures 1(b) and 1(c)) but not for others (the one in Figure 1(d)). Thus, the outcome of checking C2 is interpreted as "it depends on how this model will get concretized".

Detail-adding refinements. Bob can also use P to continue development. For example, a useful detail-adding (DA) refinement is to elaborate the `Warning` state to periodically check if the controller is still disconnected. Bob can do it directly on P to get the new model P', shown in Figure 2(a). In P', two sub-states, `Poll` and `Notify`, have been added to `Warning`. To implement the waiting functionality, Bob also needs Alice to add a `PollingTimer` class to the architectural diagram. A required property of all detail-adding refinements is the transformation property T1: they should result in models with "more information" than the input model[1] (Section 5 discusses the use of T1 for reasoning). The transformation from P to P' satisfies this property, as the set of concretizations of P' only contains transformed versions of all of the concretizations of P.

Refactoring. *Refactoring* is a transformation characterized by the transformation property T2: it should not add or remove details. Alice can do it directly on the partial model P'. A classic refactoring is making the public instance variable `disconnected` private and only accessible via the getter method `isDisconnected()`. The resulting model P'' is shown in Figure 2(b). The transformation between P and P'' satisfies T2 for the same reasons as T1.

Uncertainty-reducing refinements. Finally, once more information becomes available, Bob and Alice can apply uncertainty-reducing (UR) refinements to derive a less par-

[1]For behavioral models, this is called *simulation*.



(a)
Control
[disconnected==True] beep()
off()
[disconnected==False]
Off
On
on()
recover()
[disconnected==True] beep()
Warning
[disconnected==True] beep()
error()
Poll
Notify
sleep()
Controller
+disconnected : boolean
+on() : void
+off() : void
+beep() : void
+error(): void
+recover() : void
PollingTimer
+sleep() : void
Φp' = ((¬A ∧ B ∧C ∧ D ∧ ¬E ∧ ¬F ∧ G) ∨
(¬A ∧ B ∧ ¬C ∧ D ∧ E ∧ F ∧ ¬G) ∨
(A ∧ ¬B ∧ ¬C ∧ ¬D ∧ ¬E ∧ ¬F ∧ ¬G)
) ∧ D ⇒ (H ∧ J)
(b)
[isDisconnected()] beep()
[!isDisconnected()]
[isDisconnected()] beep()
[isDisconnected()] beep()
- disconnected : boolean
+isDisconnected() : boolean
(c)
[!isDisconnected()]
[isDisconnected()] beep()
[isDisconnected()] beep()
+recover() : void

Figure 2: (a) The model P' after detail-adding refinement of P. (b) Refactored model P'' ($\Phi_{p''} = \Phi_{p'}$). (c) Concretization M' of the partial model after uncertainty-removing refinement.

tial and ultimately concrete model from P''. In our scenario, after negotiation with the client and between each other, they decided to keep the `Warning` state and implement the polling system designed by Bob. The result of the UR refinement is the model M' in Figure 2(c). The transformation property T3 for UR refinements is that optional elements can be kept optional, removed or made mandatory (see Section 5 about how T3 is used for reasoning). T3 holds for the transformation between P'' and M' as all optional elements are either made mandatory or removed. A UR refinement such as this, that does not retain any optional elements is called a *concretization refinement*. Additionally, checking M' for properties C1 and C2 yields `True` for the former, as expected, but also `True` for the latter. Subsequent development can continue using M', whereas P'' can be archived as a record of the design alternatives considered so that they can be revisited if needed.

3. REPRESENTING PARTIAL MODELS

We now discuss how to represent sets of models as partial models. There are two key questions: (Q1) What kinds of sets of concrete models can be efficiently abstracted into a single partial model, i.e., what kinds of uncertainty can be captured this way? (Q2) How can we efficiently construct partial models, avoiding the enumeration of all possible alternatives and what are the circumstances under which a construction is feasible?

In our example in Section 2, we used a particular kind of partiality, called *May partiality*, formally defined in [10], which allows certain modeling elements to be tagged as optional. Constructing a May partial model from a set of alternative models entails merging the vocabularies of the different alternatives and identifying differing elements, like we did in constructing the partial model P in Figure 1(e). Some elements are mandatory for all concretizations, while others are optional. A propositional formula (Φ_p for P) is added to the partial model to capture the set of the allowable configurations of optional elements. In our example, the state `Warning` is tagged as optional, as it is not allowed in all possible configurations of the partial model.

P and other partial models we work with are "model like" (as opposed to being expressed in a generic description language such as first-order logic), in that they are structurally similar to the concrete models they abstract and hence are easy to understand. This allows partial models to remain first-class development artifacts within the framework of existing MDE approaches. Other forms of partiality that can be captured in "model-like" partial models exist as well. For example, when a model is known to be *incomplete*, the partial model represents the set of its possible completions. Another case occurs when it is not clear whether certain model elements should be *distinct*, and the partial model represents all possible ways to merge these elements. These and other partiality types and their combinations are described in [10].

We are currently working on constructing partial May models using symbolic model transformation, with a preliminary implementation using Kodkod [11]. The construction avoids building all concrete models and then merging them, and thus remains scalable and efficient. Our specific emphasis is on capturing alternatives which result from applying multiple repair rules.

4. REASONING WITH PARTIAL MODELS

Partial models are formal artifacts that enable reasoning. The key question here is: (Q3) What kinds of properties can be (efficiently!) checked on partial models?

In order to facilitate reasoning, we translate the visual rendering of partial models presented in the examples of Section 2 into a symbolic one and appropriately combine it with the formula representing the constraints on the model. A SAT solver is then used to check whether the model entails the property.

For example, to check whether the property C2 holds for the partial model P, shown in Figure 1(e), we first translate P into the propositional formula $\Phi = \Phi_p \wedge \texttt{Control} \wedge \texttt{Off} \wedge \ldots \wedge \texttt{Controller} \wedge \texttt{on()} \wedge \ldots$ that extends Φ_p with all the mandatory elements in P. Then we express C2 as a propositional formula Φ_{C2}. Checking the property is therefore reduced to checking satisfiability of the propositional expressions $\Phi \wedge \neg\Phi_{C2}$ and $\Phi \wedge \Phi_{C2}$, using a SAT solver.

The above approach allows us to reason about *all* concretizations together, using one or two queries to the SAT solver. For May models, our checking returns the answer `Maybe` iff the partial model has a concretization where the property is `True`, and another one where the property is `False` [2] This is the case for C2, as the SAT solver returns a satisfying assignment for $\Phi \wedge \neg \Phi_{C2}$ (the concretization shown in Figure 1(d)) and one for $\Phi \wedge \Phi_{C2}$ (one of the concretizations shown in Figures 1(b-c)).

There are more questions stemming from the method outlined above: (Q4) What types of properties can be effectively checked using this method? (Q5) Is SAT-solving the best alternative for reasoning about various types of partial models? (Q6) What useful feedback can we give to the user as the result of our analysis?

5. TRANSFORMING PARTIAL MODELS

Transformation is an essential part of any MDE methodology. The key questions that need to be answered here are: (Q7) Are there transformations specific to partial models? (Q8) How can existing transformations be adapted to operate on partial models? (Q9) How are the properties of partial models affected by the different transformations?

Special Transformations. Partial models are closely associated with a particular vertical transformation, that we call *uncertainty-removing* (UR) refinement. A UR refinement makes a model "less partial" by resolving some or all of its uncertainty. For May partiality, this is ensured by the property T3 which prescribes that the removal of uncertainty is achieved by making some optional elements of the model mandatory or deleting them. Similar correctness properties can be defined for other kinds of partiality. Since the removal of uncertainty is "problem-" and "domain-independent", UR refinement is a generic refinement mechanism, with well understood properties [10]. A UR refinement that completely removes uncertainty is called a *concretization*, e.g., going from the partial model P'' in Figure 2(b) to the concrete model M' in Figure 2(c).

By satisfying the transformation property T3, a UR refinement reduces the number of concretizations of a partial model. Therefore, any `True` (`False`) properties, i.e., properties that were `True` (`False`) for all its concretizations, remain `True` (`False`). For the same reason, `Maybe` properties can be changed into `True` or `False` or remain unaffected. This is illustrated by the property C2 in our example, which evaluates to `Maybe` in P, P' and P'', and to `True` in M'.

Adapting Classical Transformations. In order to adapt an existing transformation to partial modeling, we need to specify how it affects the set of concretizations of its input partial model. We formulate this as follows: if $\mathcal{F}$ is the adapted version of a classical transformation F, and if the partial model P' is the result of applying $\mathcal{F}$ to a partial model P, then for each concretization p of P there exists a concretization p' of P' s.t. p' is the result of applying F to p. In other words, $\mathcal{F}$ is total and surjective.

We call the adapted version of classical refinements *detail-adding* (DA) refinements. A DA refinement adds new information to a partial model without removing uncertainty. Recall one such refinement in Section 2: model P, shown in Figure 1(e), became model P', shown in Figure 2(a). While more detail was added to the state `Warning`, the model did not become "less partial". Additionally, in keeping with property T1, the `Warning` state is only elaborated for those concretizations of P that have it.

Certain classical transformations can be applied directly on partial models without the need for explicit adaptation. We illustrated this with the refactoring of the model P' in Figure 2(a) to the model P'' in Figure 2(b).

By satisfying the transformation property T1, a DA refinement only adds information and thus preserves `True` existential and `False` universal properties. For example, the property C3: *"there exists a state with a self-looping transition"* is an existential property which holds in P. Thus, it is guaranteed to hold in P'.

We are currently working on an implementation to apply partialized versions of DA transformations using symbolic model transformation. Immediate follow-up questions are: (Q10) How can adaptations of classical transformations be systematized? (Q11) What are the conditions under which classical transformations can be applied to partial models without adaptation? Important future questions include: (Q12) How are the properties of combinations of structural and behavioral partial models affected by transformation? (Q13) What effect does combining the different kinds of refinement (UR and DA) have to property preservation? (Q14) What properties are preserved if the transformation also involves changes to the metamodel?

6. CONCLUSION

We have outlined a research agenda for developing a systematic and robust treatment of uncertainty in MDE. Our approach consists of extending existing MDE methodologies by using partial models as first-class development artifacts. We illustrated how partial models can be represented and how they can be used for reasoning, and for defining property-preserving transformations.

7. REFERENCES

[1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In *Proc. of CAV'00*, pages 154–169, 2000.

[2] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proc. of POPL'77*, pages 238–252, 1977.

[3] A. Egyed, E. Letier, and A. Finkelstein. "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models". In *Proc. of ASE'08*, pages 99–108, 2008.

[4] D. Fischbein, G. Brunet, N. D'ippolito, M. Chechik, and S. Uchitel. "Weak Alphabet Merging of Partial Behaviour Models". *ACM TOSEM*, 2011. in press.

[5] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In *Proc. of CAV'97*, pages 72–83, 1997.

[6] K. G. Larsen and B. Thomsen. "A Modal Process Logic". In *Proc. of LICS'88*, pages 203–210, 1988.

[7] T. Mens and R. V. D. Straeten. "Incremental Resolution of Model Inconsistencies". In *Proc. of WADT'06*, 2007.

[8] C. Nentwich, W. Emmerich, and A. Finkelstein. "Consistency Management with Repair Actions". In *Proc. of ICSE'03*, pages 455–464, 2003.

[9] M. Sabetzadeh and S. Easterbrook. "View Merging in the Presence of Incompleteness and Inconsistency". *J. of Requirements Engineering*, 11(3):174–193, 2006.

[10] R. Salay and M. Chechik. "Language Independent Refinement using Partial Modeling". Technical report, Univ. of Toronto, 2011.

[11] E. Torlak and D. Jackson. "Kodkod: A Relational Model Finder". In *Proc. of TACAS'07*, pages 632–647, 2007.

[12] A. van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009.

[2] This form of analysis, which does not produce erroneous `Maybe` answers is called *thorough*.