Partial Models: Towards Modeling and Reasoning with Uncertainty

Michalis Famelis, Rick Salay and Marsha Chechik
University of Toronto, Canada
{famelis, rsalay, chechik}@cs.toronto.edu

***Abstract*—Models are good at expressing information about software but not as good at expressing modelers' uncertainty about it. The highly incremental and iterative nature of software development nonetheless requires the ability to express uncertainty and reason with models containing it. In this paper, we build on our earlier work on *expressing* uncertainty using partial models, by elaborating an approach to *reasoning* with such models. We evaluate our approach by experimentally comparing it to traditional strategies for dealing with uncertainty as well as by conducting a case study using open source software. We conclude that we are able to reap the benefits of well-managed uncertainty while incurring minimal additional cost.**

I. Introduction

Software engineering is a highly incremental and iterative endeavor where uncertainty can exist at multiple stages of the development process. Consequently, systematic approaches to handling uncertainty are essential throughout the software life-cycle.

Models are used pervasively in software engineering, and their ability to express information about different aspects of software has been studied by many researchers [25]. However, models seldom provide the means for expressing the *uncertainty that the modeler has* about this information. In this paper, by "uncertainty" we mean "multiple possibilities". This notion of uncertainty is often used in behavioral modeling [10], but we expand it to arbitrary modeling languages. For example, a modeler of a class diagram may be uncertain about which of two attributes to include in a particular class because they represent different design strategies, and it is too early to know which is correct.

In general, uncertainty can be introduced into the modeling process in many ways: alternative ways to fix model inconsistencies [14], [5], [24], different design alternatives (e.g., the above example) [26], problem-domain uncertainties [27], multiple stakeholder opinions [18], etc. In each case, the presence of uncertainty means that, rather than having a single model, we actually have a *set of possible* models and we are not sure which is the correct one. Living with uncertainty requires us to keep track of this set and use it within modeling activities wherever we would use an individual model; however, this can be challenging since modeling activities are typically intended for individual models, not sets of them. Furthermore, managing a set of models explicitly is impractical since its size might be quite large. For example, in Sec. VI we give a case study in which two inconsistencies lead to several hundred possible models. On the other hand, if uncertainty is ignored and one particular possible model is chosen prematurely, we risk having incorrect information in the model.

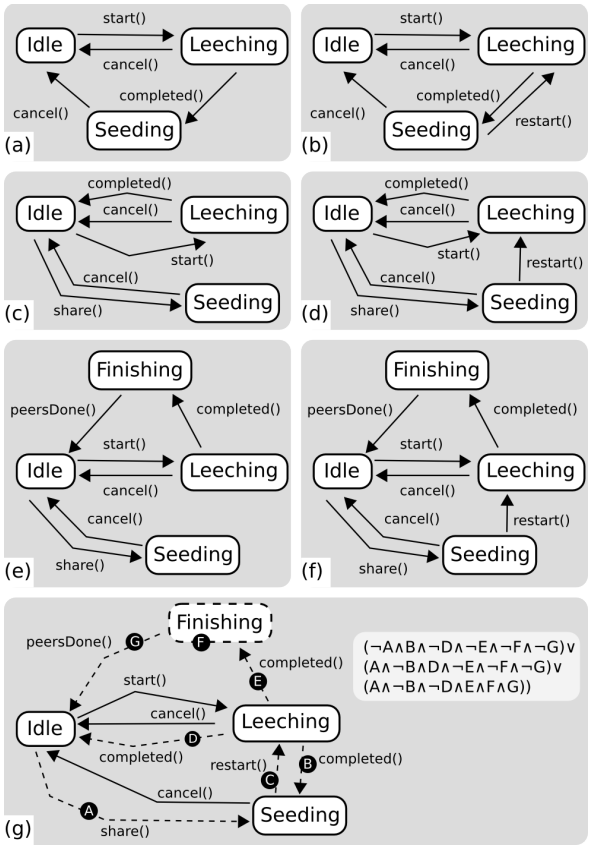

Figure 1. (a-f) Six alternative designs for a peer-to-peer file sharing system; (g) a partial model M_e for the six alternatives.

Our approach to handling uncertainty is to use annotations with well-defined semantics that change a model into a *partial* model, i.e., one that compactly *yet precisely* encodes the entire set of possible models. This representation allows us to work with a set of models as if it were a single model and do reasoning efficiently with all the possible models simultaneously.

Motivating Example. To help motivate and explain our approach, we use the example of a team engaged in the development of a simple peer-to-peer file sharing application. The team uses UML State Machine diagrams to model the behavior of this application. Its states are `Idle`, `Leeching` (downloading a file) and `Seeding` (sharing a complete local copy). Downloading always starts from the `Idle` state, and `Seeding` and `Leeching` can always be canceled. We assume that at this stage of development, the team has not finalized the exact behavior of the program, due to vague requirements

given to them by their client. The team has drafted three alternative behavioral designs:

1) "Benevolent": Once the file is downloaded, the program automatically starts `Seeding`, as shown in Fig. 1(a).
2) "Selfish": Once the file is downloaded, the program becomes `Idle`, and the user can choose whether to start `Seeding` or not – see Fig. 1(c).
3) "Compromise": Once the file is downloaded, the program stops accepting new peers. It doesn't disconnect from peers that were already connected during the `Leeching` stage, but rather waits while they are `Finishing` before it becomes `Idle` – see Fig. 1(e).

The team is also unsure whether the program should allow the user to `restart` a finished download (i.e., download the file again). The three alternatives with this feature are shown in Fig. 1(b, d, e), respectively.

Until the client clarifies the requirements, the team is faced with uncertainty over which design decision to choose. At this point, it is probably useful to be able to *reason about the available choices*, both to ensure that the models conform to the desired constraints and to explore their properties. For example, assume that the team is using a code generator that, in order to ensure determinism, requires two hard constraints:

HC1: No two transitions have the same source and target.
HC2: No state is a sink.

Additionally, the team is interested in two "nice-to-have" properties, i.e., soft constraints that are not strictly required but are desirable:

SC1: Users can share files they already have, (i.e., `Seeding` is directly reachable from `Idle`).
SC2: Users can always cancel any operation (i.e., every non-idle state has a transition to `Idle` on `cancel()`).

In order to reason effectively about any of these properties over the entire set of alternatives, the team may want to ask the following questions:

Does the property hold for all, some or none of the alternatives? This can help determine how critical some property is in selecting alternatives when uncertainty is lifted. For example, HC2 holds for all alternatives, and therefore is not going to be a main reason in selecting one, once uncertainty is resolved. Moreover, if some property does not hold for any alternative, it may be an indication that the team needs to revisit the designs, sooner rather than later. For example, knowing early on that HC1 does not hold for the alternatives in Fig. 1(c, d) may be an indication that the team needs to reconsider the design of the "selfish" scenario.

If the property does not hold for all alternatives, why is it so? This form of diagnosis can help guide development decisions even before uncertainty is lifted. Developers may be interested in finding one counter-example of an alternative where the property gets violated (or if they expected that the property would be violated – an example where it holds) to help them debug the set of alternatives. For example, locating the alternative in Fig. 1(e) might be sufficient for the team to understand why SC2 does not hold for all alternatives. In other cases, we may prefer to calculate the entire subset of alternatives that violate the property, to explore whether there is a common underlying cause, as with the alternatives in Fig. 1(c, d) that violate the hard constraint HC1.

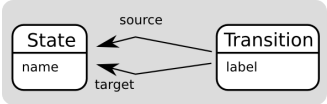

Figure 2. Simplified metamodel used for defining State Machines.

If the property is a necessary constraint, how to filter out the alternatives for which it gets violated? Developers may be interested in this sort of property-driven refinement of the set of alternatives. For example, if the team decides that SC2 is a necessary feature, they should be able to restrict their working set of alternatives to those that satisfy it, namely, those in Fig. 1(a-d).

Contributions. In this paper, we elaborate and evaluate a key component of our broad research agenda for managing uncertainty within models [7]: reasoning with models containing uncertainty. Specifically, we define partial models, show how to construct them and then describe the three reasoning operators aimed to answer the questions posed by the motivating example. We then extensively evaluate our approach by experimentally comparing it to conventional strategies for dealing with uncertainty as well as by conducting a case study of an open source software project.

Organization of the paper. The rest of this paper is organized as follows: In Sec. II, we provide the necessary background. In Sec. III, we formally define partial models. Sec. IV develops the core methods of reasoning with partial models. These are experimentally evaluated in Sec. V and then applied to a case study in Sec. VI. We discuss related work in Sec. VII and conclude the paper with a summary and suggestions for further research in Sec. VIII.

II. Background

In this section, we establish the notation and introduce concepts used in the remainder of the paper. Specifically, we ground our approach to graph-based modeling languages and propositional logic.

Modeling Formalisms. In this paper, a model is a *typed graph* that conforms to some *metamodel* represented by a distinguished *type graph*. Our approach is domain-independent, in the sense that it can handle arbitrary graph-based modeling languages. The definitions that follow are based on [6].

Definition 1: A *graph* is a tuple $G = \langle V, E, s, t \rangle$, where V is a set of nodes, E is a set of edges, and $s, t : E \to V$ are the source and target functions, respectively, that assign each edge a source and target node.

Definition 2: A *typed graph* (model) of type T is a triple $\langle G, type, T \rangle$ consisting of a graph G, a metamodel T and a typing function $type : G \to T$ that assigns types to the elements of G.

For example, the models shown in Fig. 1 are typed with the metamodel shown in Fig. 2.

Definition 3: The *scope* (or *vocabulary*) of a model $\langle G = \langle V, E, s, t \rangle, type, T \rangle$ is the set $S = V \cup E$ of its typed nodes and edges.

For example, the scope of the model in Fig. 1(a) consists of the states `Idle`, `Leeching` and `Seeding` and the edges `start()`, `completed()`, etc.

In the following, we often refer to nodes and edges that are in the scope of a model as *elements* or *atoms* of the model.

From models to formulas and back. To encode a model in propositional logic, we first map elements in its scope into propositional variables and then conjoin them. To ensure that this operation is reversible, we define specific naming conventions for the propositional variables:

- A node element N of type T is mapped to a propositional variable "`N_T`".
- An edge element E of type T with source node N_1 and target node N_2 is mapped to a propositional variable "`E_`N_1`_`N_2`_T`".

For example, the propositional encoding of the model in Fig. 1(b) is:

```
Idle_State ∧ Leeching_State ∧ Seeding_State ∧
start_Idle_Leeching_Transition ∧
cancel_Leeching_Idle_Transition ∧
completed_Leeching_Seeding_Transition ∧
cancel_Seeding_Idle_Transition ∧
restart_Seeding_Leeching_Transition
```

Given a propositional encoding $P(m)$, of a model m, we can uniquely reconstruct the model m using the naming conventions. First, for every propositional whose name fits the pattern `N_T`, we create a node of type T, named N. Then, for every propositional variable whose name follows the pattern `E_`N_1`_`N_2`_T`, we create an edge of type T between the nodes N_1 and N_2, with the label E.

This propositional encoding also allows us to embed models into larger scopes, by negating all the variables not in the original scope. For example, the model in Fig. 1(a) can be expressed in the scope of the model in Fig. 1(b) as:

```
Idle_State ∧ Leeching_State ∧ Seeding_State ∧
start_Idle_Leeching_Transition ∧
cancel_Leeching_Idle_Transition ∧
completed_Leeching_Seeding_Transition ∧
cancel_Seeding_Idle_Transition ∧
¬ restart_Seeding_Leeching_Transition
```

Using the propositional representation, we also define a simple form of *model union*. Assuming two elements with the same name are considered identical, the union of two models is a model that corresponds to a formula that is a conjunction of all the variables in the union of their scopes. For example, the union of the models in Fig. 1(b, d) is:

```
Idle_State ∧ Leeching_State ∧ Seeding_State ∧
start_Idle_Leeching_Transition ∧
cancel_Leeching_Idle_Transition ∧
completed_Leeching_Seeding_Transition ∧
cancel_Seeding_Idle_Transition ∧
restart_Seeding_Leeching_Transition ∧
completed_Leeching_Idle_Transition ∧
share_Idle_Seeding_Transition
```

A useful extended scope is the embedding of a sparse graph into the scope of its corresponding complete graph. In the union of models with extended scopes, variables only appear negated if they are negated in both input models.

Properties. We consider properties expressed in first order logic (FOL) or in a similar language such as the Object Constraint Language (OCL) [15]. For example, the property HC1 is expressed in FOL as:

$$\forall t_1, t_2 : \text{Transition} \quad \cdot \quad (\text{Source}(t_1) = \text{Source}(t_2) \wedge \text{Target}(t_1) = \text{Target}(t_2)) \Leftrightarrow (t_1 = t_2)$$

An FOL formula can be *grounded* over the vocabulary of a particular model that is encoded in a propositional formula. For example, grounding HC1 over the vocabulary of the model in Fig. 1(a), given that it contains 4 transition elements and that HC1 is a universal property, results in Φ_{HC1} – a conjunction of 10 unique terms of the form $(S_i = S_j \wedge T_i = T_j) \Leftrightarrow E_i = E_j$, where $E_{i,j}$ are propositional variables representing transitions, $S_{i,j}, T_{i,j}$ are variables representing their respective source and target states, and "=" signifies identity.

III. Partial Model Preliminaries

In this section, we formally define partial models and their associated operations. Semantically, a partial model represents a set of *classical* (i.e., non-partial) models.

Partial Models. The particular type of partiality we consider in this paper is the one that allows a modeler to express uncertainty as to whether particular model atoms should be present the model. The model is accompanied by a propositional formula, called *may formula*, which explicates allowable combinations of such atoms.

Definition 4: A *Partial Model* is a tuple $\langle G, vm, em, \phi \rangle$, where $G = \langle \langle V, E, s, t \rangle, type \rangle$ is a *complete* typed graph, $vm : V \rightarrow \mathcal{B}$ and $em : E \rightarrow \mathcal{B}$, where $\mathcal{B}$ is the set $\{\mathsf{True}, \mathsf{False}, \mathsf{Maybe}\}$, are functions for annotating atoms in G, and ϕ is a propositional *may* formula over the scope $S = V \cup E$, built as described in Sec. II.

In the above definition, an annotation True (False) means that the atom must (must not) be present in the model, whereas Maybe indicates uncertainty about whether the atom should be present in the model. In other words, a partial model consists of a complete typed graph whose elements are annotated with True, False or Maybe, and a may formula that describes the allowed configurations of its elements. The annotation functions are often omitted for brevity.

Model M_e in Fig. 1(g) is an example of a partial model. The elements annotated with True, such as the state `Idle` and the transition `start()`, appear with solid lines, and its Maybe elements, such as the state `Finishing`, with dashed lines. The edges that are not shown (such as any edge between the states `Finishing` and `Leeching`) are annotated with False. M_e is accompanied by the may formula ϕ_e, shown next to it in the figure. We have used capital letters as shortcuts for the full names of the propositional variables that correspond to the Maybe elements. For example, `F` stands for the variable `Finishing_State`.

Given a partial model M, let $\mathcal{C}(M)$ be the set of classical (or *concrete*) models that it represents, called *concretizations*. For example, $\mathcal{C}(M_e)$ consists of the models shown in Fig. 1(a)-(f). A partial model with an empty set of concretizations is called *inconsistent*. In what follows, we only assume consistent partial models.

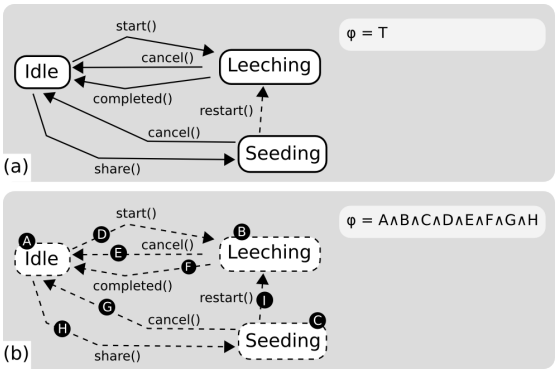

Figure 3. Normal forms of the model $M_e^{\neg HC1}$, shown in Fig. 4(a): (a) Graphical Normal Form (GNF) (b) Propositional Normal Form (PNF).

The size of the set of concretizations reflects the modeler's degree of uncertainty. Uncertainty can be reduced by reducing the set of concretizations via *refinement*. A partial model is refined by changing the annotations of its elements to increase the level of certainty: Maybe elements can be assigned to True, False or Maybe; True and False annotations must remain unchanged since information about them is already certain. Changes to Maybe elements must not violate the formula of the original partial model, and thus produce a (nonempty) subset of concretizations allowed by it.

Definition 5: Given two partial models M_1 and M_2, where $M_i = \langle G_i, vm_i, em_i, \phi_i \rangle$, with $G_1 = G_2$, we say that M_2 *refines* M_1 (or that M_1 is *more abstract* than M_2), denoted $M_2 \preceq M_1$ iff $\mathcal{C}(M_2) \subseteq \mathcal{C}(M_1)$ over the same scope S.

For example, the model M_e^{HC1} in Fig. 4(b) is more refined than the model M_e in Fig. 1(g). In particular, $\mathcal{C}(M_e)$ consists of the models in Fig. 1(a)-(f), whereas $\mathcal{C}(M_e^{HC1})$ consists of the models in Fig. 1(a,b,e,f). Thus, the model M_e^{HC1} has less uncertainty.

A partial model without Maybe elements has exactly one concretization. The naming conventions in Sec. II allow us to define a unique conversion between a classical model and a corresponding partial model with a unique concretization.

Normal Forms. Given a set of classical models, there is no unique way to represent them as a partial model. For example, $M_e^{\neg HC1}$ in Fig. 4(a) represents the models in Fig. 1(c, d). However, the same set of concretizations could be expressed by: (a) removing from the scope of $M_e^{\neg HC1}$ extraneous False elements, such as the state `Finishing`, and (b) rewriting its propositional formula only in terms of its Maybe elements. In the case of $M_e^{\neg HC1}$, the partial model has only one Maybe element (the transition on `restart()`), which can be either True or False, and therefore, the attached propositional formula $\phi_e^{\neg HC1}$ is a tautology.

Definition 6: Two partial models M_1, M_2 are *equivalent*, denoted $M_1 \sim M_2$, iff $\mathcal{C}(M_1) = \mathcal{C}(M_2)$. Obviously, $M_1 \sim M_2$ iff $M_1 \preceq M_2$ and $M_2 \preceq M_1$.

To help represent models, we define two normal forms: *Graphical Normal Form* (GNF) and *Propositional Normal Form* (PNF). Intuitively, a model in GNF represents most information in the graph, whereas in PNF it represents all the information in the formula. For example, the GNF and PNF for $M_e^{\neg HC1}$ are shown in Fig. 3(a-b), respectively. In the latter, we did not represent False edges which would otherwise be represented by negated variables.

As partial models are complete graphs, the normal form of M should be restricted to its largest complete subgraph that only contains True and Maybe nodes. We call the scope of this subgraph *minimal*.

In the following, the symbol $\vDash$ signifies logical entailment.

Definition 7: Given a partial model $M = \langle G, \phi \rangle$, its GNF is a partial model $M^{\text{GNF}} = \langle G^{\text{GNF}}, \phi^{\text{GNF}} \rangle$, constructed as follows:

- $G^{\text{GNF}} \subseteq G$ and the scope S of M^{GNF} is minimal.
- For every atom a in G, if $\phi \vDash a$, then a is annotated with True in G^{GNF}.
- For every atom a in G, if $\phi \vDash \neg a$, then a is annotated with False in G^{GNF}.
- ϕ^{GNF} is specified only in terms of elements annotated with Maybe in G^{GNF}.
- $\phi \vDash \phi^{\text{GNF}}$.

Proposition 1: Let M be a partial model and M^{GNF} be a result of applying Definition 7. Then, $M \sim M^{\text{GNF}}$.

Definition 8: Given a partial model $M = \langle G, \phi \rangle$, its PNF is a partial model $M^{\text{PNF}} = \langle G^{\text{PNF}}, \phi^{\text{PNF}} \rangle$ constructed as follows:

- $G^{\text{PNF}} \subseteq G$ and the scope S of M^{PNF} is minimal.
- All elements in G^{PNF} are annotated with Maybe.
- $\phi^{\text{PNF}} \vDash \phi$.

Proposition 2: Let M be a partial model and M^{PNF} be a result of applying Definition 8. Then, $M \sim M^{\text{PNF}}$.

Properties of Partial Models. The result of checking a property on a partial model can be True, False or Maybe. True means that the property holds for all concretizations, False that it does not hold for any of them, and Maybe that it holds for some, but not all concretizations. This is called *thorough* checking [2]. Moreover, by Definition 5, refinement preserves True and False properties. That is, as uncertainty gets reduced, values of properties about which we were certain remain unaffected.

IV. Reasoning with Partial Models

In this section, we describe how to facilitate decision deferral in the presence of uncertainty by using partial models to reason with sets of alternatives. In particular, we define four reasoning operations:

OP1: **Construction**: how to create a partial model to (precisely) represent a set of alternatives.
OP2: **Verification**: how to check whether a partial model satisfies a property.
OP3: **Diagnosis**: how to find out which alternatives violate the property.
OP4: **Refinement**: how to filter out the alternatives that violate the property.

OP1: Construction. Construction of partial models is achieved by merging the alternatives and annotating the elements that vary between them by Maybe. Additionally, *may*

Algorithm 1 Construction of partial models.

Input: Set A of n concrete models $m_i, i \in [0..n-1]$.
Output: A partial model $M = \langle G_M, \Phi_M \rangle$

```
Construct G_M as the union of all m_i ∈ A.
Annotate non-common elements in G_M by Maybe.
Create Φ_M :=False
for all m_i ∈ A, e_x ∈ G_M e annotated with Maybe do
    Create φ_i = e_0 ∧ ¬e_1 ∧ ... ∧ e_k,
        where if e_x ∉ m_i, it appears negated.
    Φ_M := Φ_M ∨ φ_i
end for
return M = ⟨G_M, Φ_M⟩
```

formula is constructed to capture the allowable configurations of the Maybe elements.

Algorithm 1 shows how to create a partial model M from a set A of alternatives. By construction, $\mathcal{C}(M) = A$, which establishes the algorithm's correctness.

In our motivating example, the six alternative behavioral designs can be represented using the partial model shown in Fig. 1(g). In the figure, elements annotated as Maybe appear dashed. For example, the state `Finishing` exists in only two alternatives, and the transition on `restart()` – in three; therefore, both are represented as Maybe. The rest of the elements are present in all of the alternatives, and thus are represented as True and appear solid. The corresponding *may* formula is shown in Fig. 1(g).

OP2: Verification. The purpose of the verification task is to answer the question "*Does the desired property hold?*".

In order to facilitate reasoning, we put the partial model in PNF and appropriately combine its PNF may formula with the formula representing the property we want to check. A SAT solver is then used to check whether the encoding of the model entails that of the property.

Specifically, the verification engine receives a partial model M that is represented in PNF by the propositional formula Φ_M and a property expressed as a propositional formula Φ_p. We then check satisfiability of the expressions $\Phi_M \wedge \neg\Phi_p$ and $\Phi_M \wedge \Phi_p$, using two queries to a SAT solver, combining the results to determine the outcome of the property on the partial model as described in Table I. For example, if both the property and its negation are satisfiable, then there is at least one concretization of the partial model where the property holds and another – where it does not. Thus, in the partial model the property has value Maybe.

Returning to our running example, in order to check whether the property HC1 holds for the partial model M_e in Fig. 1(g), we first put M_e in PNF to get the propositional formula Φ_e. Then we express HC1 as a propositional formula Φ_{HC1}, by grounding it over the vocabulary of M_e, as described in Sec. II. Checking the property means checking satisfiability $\Phi_e \wedge \neg\Phi_{HC1}$ and $\Phi_e \wedge \Phi_{HC1}$. The SAT solver returns one of the two models from Fig. 1(c, d) as the satisfying assignment for $\Phi \wedge \neg\Phi_{HC1}$, and one of those in Fig. 1(a, b, e, f) for $\Phi \wedge \Phi_{HC1}$. Thus, the value of HC1 is uncertain (Maybe) on the model.

Table I
CHECKING PROPERTY p ON THE PARTIAL MODEL M.

$\Phi_M \wedge \Phi_p$	$\Phi_M \wedge \neg\Phi_p$	Property p
SAT	SAT	Maybe
SAT	UNSAT	True
UNSAT	SAT	False
UNSAT	UNSAT	(Inconsistent M)

OP3: Diagnosis. If the result of the verification task is False or Maybe, the next step is to do diagnosis, i.e., to answer the question "*Why does the property of interest not hold?*". Or, conversely, if the outcome was Maybe where it was expected to be False, to answer the question "*Why is the property not violated?*". There are three forms of feedback that can be given to the developer:

*1) Return one counter-example – a particular concretization for which the property does not hold **(OP3a)**:* Such a counter-example is provided "for free" as a by-product of SAT-based verification. In particular, if the property is False, the SAT solver produces a satisfying assignment for $\Phi_M \wedge \neg\Phi_p$.

This assignment is a valuation for all propositional variables that correspond to elements in the scope of M and can thus be visualized as a classical model for presentation to the user. To create the visualization, we conjoin all variables, negating those that had value False in the satisfying assignment. Provided the naming conventions in Sec. II are followed, this conjunction uniquely corresponds to a classical model, which is then presented as the feedback.

In our running example, verifying SC2 on the model M_e involves checking the satisfiability of $\Phi_e \wedge \neg\Phi_{SC2}$. This formula is satisfiable, and the SAT solver returns one of the concretizations in Fig. 1(e, f) as a satisfying assignment.

*2) Return a concretization where the property does hold **(OP3b)**:* This is also a by-product of the verification stage: if the result of checking the property is Maybe, the SAT solver produces a satisfying assignment for the formula $\Phi_M \wedge \Phi_p$. This valuation is expressed as a model (as discussed above) and provided to the user.

In the case of verifying SC2, the SAT solver returns a valuation that corresponds to one of the concretizations in Fig. 1(a,b,c,d) as a satisfying assignment to the formula $\Phi_e \wedge \Phi_{SC2}$.

3) Return a partial model representing the set of all *concretizations for which the property does not hold **(OP3c)**:* These concretizations are characterized by the formula $\Phi_M \wedge \neg\Phi_p$. In our example, the concretizations of M_e that violate HC1 are those that satisfy the formula $\Phi_e \wedge \neg\Phi_{HC1}$, i.e., those in Fig. 1(c,d).

In order to create useful feedback to the user, we consider a new partial model $M^{\neg p}$ with the same vocabulary as M, that is represented in PNF by the formula $\Phi_M \wedge \neg\Phi_p$. We visualize $M^{\neg p}$ by putting it into GNF. In our example, the partial model $M_e^{\neg HC1}$ that represents the set of concretizations of M_e that violate HC1 is shown in Fig. 4(a). $M_e^{\neg HC1}$ is expressed in terms of the larger scope of M_e and therefore certain elements are tagged as False and omitted from the diagram. The overall process is described in Algorithm 2. As the resulting

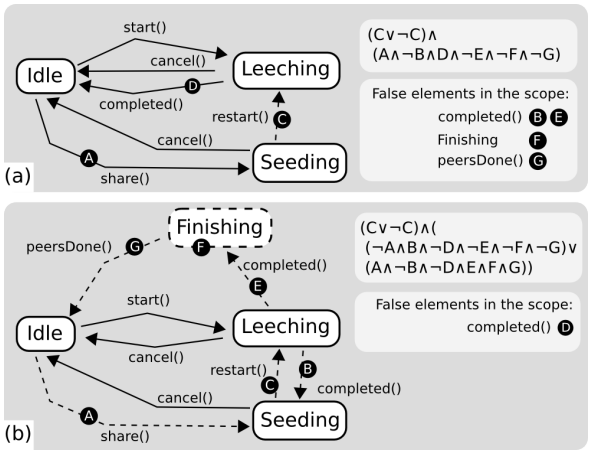

Figure 4. (a) Partial model $M_e^{\neg HC1}$ representing all concretizations of M_e that violate HC1. (b) Partial model M_e^{HC1} representing all concretizations of M_e that satisfy HC1.

model is constructed by the formula $\Phi_M \wedge \neg\Phi_p$, its set of concretizations is exactly the subset of concretizations of the original partial model for which the property was violated. In other words, $M^{\neg p} \preceq M$.

OP4: Property-driven refinement. If the result of verification of an important property is Maybe, the developer may want to refine the partial model to a constrained version such that all of its concretizations satisfy the property. This subset of concretizations exactly characterized by the formula $\Phi_M \wedge \Phi_p$. We use it to construct the partial model M^p in the same manner as we did for constructing $M^{\neg p}$.

In our example, the set of concretizations of M_e that satisfy HC1 consists of those in Fig. 1(a, b, e, f). Constructing the partial model M_e^{HC1} that represents these is done using the same method (shown in Algorithm 2) as for constructing its complement, $M_e^{\neg HC1}$. Namely, the formula $\Phi_e \wedge \Phi_{HC1}$ is constructed and then put into GNF. The result is shown in Fig. 4(b).

As M^p is constructed using the formula $\Phi_M \wedge \Phi_p$, its set of concretizations is exactly the subset of concretizations of the original M for which the property holds; thus, $M^p \preceq M$.

V. Experiments

We conducted a preliminary empirical study to assess the feasibility and scalability of our approach to reasoning using partial models. More specifically, we attempted to answer the following research questions:

RQ1: How **feasible** is reasoning with sets of models with the partial model representation in comparison to the classical approach?

RQ2: How **sensitive** are the partial modeling representation and reasoning techniques to the varying degree of uncertainty?

To get answers to RQ1 and RQ2, we set up experiments with parameterized random inputs to simulate various categories of realistic reasoning settings.

Experimental setup. The reasoning tasks described in Sec. IV are operationalized using two fundamental tasks:

Algorithm 2 Get all concretizations that violate (satisfy) a property.

Input: A partial model M_{in} and a property C
Output: A partial model M_{out} abstracting exactly the concretizations of M_{in} that violate (satisfy) C.

1: Put M_{in} in PNF, to get Φ_{in}.
2: Ground C, to get Φ_c
3: Construct $\Phi_{out} := \Phi_{in} \wedge \neg\Phi_c$ ($\Phi_{out} := \Phi_{in} \wedge \Phi_c$)
4: Create M_{out} with the same vocabulary as M_{in} and PNF formula Φ_{out}
5: Put M_{out} in GNF and **return** it

T1: *Check the satisfiability of the formulas* $\Phi_M \wedge \Phi_P$ *and* $\Phi_M \wedge \neg\Phi_P$ (for OP2, OP3a and OP3b).

T2: *Construct a new partial model in GNF that has a PNF formula* $\Phi_M \wedge \Phi_P$ (for OP3c with $\neg\Phi_P$ and OP4).

We focus our experimental evaluation on T1 and T2 because they require the use of SAT-solving technology, as opposed to Construction (OP1) which is linear to the number of input classical models and their elements (see Algorithm 1). Specifically, to answer RQ1, we conducted two experiments:

E1 Compare the relative performance of doing reasoning by running the task T1 to the performance of classical reasoning by considering the set of concretizations represented by M.

E2 Compare the relative performance of running T2 to get a partial model representing the subset of concretizations that satisfy a property, to the performance of incrementally collecting all the classical models as satisfying assignments of the formula $\Phi_M \wedge \Phi_P$.

To answer RQ2, we executed the experiments E1 and E2 with randomly generated experimental inputs that were parameterized to allow for different sizes, both with respect to model size and the size of the set of concretizations.

Experimental inputs. The metamodel of typed models corresponds to additional constraints in their propositional encoding. This makes the problem easier for the SAT solver, as it constrains the search space. We chose to use untyped models for inputs to our experiments, as the least constrained and thus the most difficult for the SAT solver.

We considered the following experimental parameters: 1) size of the partial model, 2) size of its set of concretizations, 3) quantification (e.g., existential, universal, mixed) of the property, and 4) result of property checking (True, False, Maybe). To manage the multitude of possible combinations of these, we discretized the domain of each parameter into categories.

We defined four size categories, based on the total number of elements (nodes and edges) in the partial model: Small (S), Medium (M), Large (L) and Extra-Large (XL). Based on pilot experiments, we defined ranges of reasonable values for each size category and selected a representative exemplar. The ranges of the categories and the selected exemplars for each category are shown in Table II.

Table II
CATEGORIES OF THE SIZE OF MODELS.

Size of Model	S	M	L	XL
Nodes	(0,10]	(10, 20]	(20, 40]	(40, 80]
Elements	(0,110]	(110, 420]	(420, 1640]	(1640, 6480]
Exemplar	30	240	930	2550

Table III
CATEGORIES OF THE SIZE OF THE CONCRETIZATION SET.

Size of Set	S	M	L	XL
Concretizations	(0,10]	(10, 75]	(75, 150]	(150, 300]
Exemplar	5	50	100	200

In a similar manner, we defined four categories (S, M, L, XL) for the size of the set of concretizations of the generated model. The size of this set reflects the degree of uncertainty encoded in the partial model, so that the category S corresponds to little uncertainty over which alternative to chose, and the category XL corresponds to extreme uncertainty. Based on pilot experiments, we defined reasonable ranges and selected a representative exemplar for each category, as shown in Table III.

We also defined four property types (based on the quantification of FOL formulas): "fully existential" (E), "fully universal" (A) and two "mixed" categories: "exists-forall" (EA) and "forall-exists" (AE). Additionally, we considered the three possible results that can be yielded by property checking – True, Maybe and False.

Implementation. We implemented tooling support to randomly generate inputs based on the experimental properties outlined in Sec. V. Specifically, we generate propositional formulas expressed in the input format of the MathSAT 4 SMT Solver [3]. Each such propositional formula Φ_r is a conjunction of the form $\Phi_r = \Phi_a \wedge \Phi_c \wedge \Phi_p$, where Φ_a represents the annotations of the elements of the partial model, Φ_c – its set of concretizations and Φ_p – the property being checked. We describe these below.

For each random partial model, we considered a complete graph whose elements are in the model's finite vocabulary of N_1 nodes and N_1^2 edges. Each element is randomly annotated as True or False, and N_2 elements are annotated as Maybe. Each element in the model is represented by a boolean variable. The formula Φ_a captures the set of variables that make up the model as well as their annotations. In particular, Φ_a is a conjunction of $N_1(N_1+1)$ terms, one for each element. If an element α is annotated as True, its corresponding term is the non-negated variable v_α. If it is annotated as False, its term is $\neg v_\alpha$, and if it is annotated as Maybe– $(v_\alpha \vee \neg v_\alpha)$. This tautological disjunction is necessary for v_α to be considered by the SAT solver even if it doesn't appear elsewhere in Φ_r.

Each model is accompanied by the formula Φ_c that captures its set of concretizations. Φ_c is a disjunction of N_3 unique sub-formulas representing individual concretizations. Each one is a conjunction of the N_2 Maybe variables, a random number of which is negated. This way, each sub-formula defines an allowable configuration of Maybe elements.

Defining specific values for N_1 and N_3, we were able to generate models for each of the combinations of the parameters in Tables II and III.

```
VAR v0, v1, v2, v0_v0, v0_v1, v0_v2, v1_v0,
v1_v1, v1_v2, v2_v0, v2_v1, v2_v2 : BOOLEAN
FORMULA
# Start of Φa
v0 and (v1 or (not v1)) and (v2 or (not v2)) and
(not v0_v0) and (not v0_v1) and (v0_v2 or
(not v0_v2)) and (not v1_v0) and (not v1_v1) and
(not v1_v2) and (not v2_v0) and (not v2_v1) and
(not v2_v2) and (
# Start of Φc
 ((v1 and v2 and (not v0_v2)) or
 (v1 and (not v2) and (not v0_v2)) or
 ((not v1) and v2 and v0_v2)))
# Start of Φp
and not (
 (not v0 implies not v1_v0) and (not v1_v0 implies not v2)
 and (not v1 implies not v0_v2))
```

Figure 5. A randomly generated input in MathSAT's encoding language.

To generate formulas Φ_p that simulate grounded FOL properties, we used property "templates". For example, to capture the (trivial) FOL formula $\phi_{ex} = \exists x, y : x \Rightarrow y$, we created the template *"X implies Y"*. Given a partial model with elements represented by the set of four variables $\{v_1, v_2, v_3, v_4\}$, the propositional formula that corresponds to grounding ϕ_{ex} over the vocabulary of the model is created as a randomly instantiated disjunction of copies of the template, e.g., *"(v3 implies v2) or (v1 implies v4)"*. To run experiments, our goal was creating templates for realistic properties such as the ownership relationship. For example, the template *"(not X) implies (not Y)"* indicates that Y cannot exist without its "owner" element, X.

Each template was repeated N_6 times, with N_6 large enough so that Φ_p contains N_4 variables, out of which N_5 correspond to Maybe elements. Preliminary results by pilot experiments indicated that these parameters did not significantly affect the observed times and therefore in the generated inputs we fixed them to $N_4 = 0.1 \times N_1$ and $N_5 = min(N_2, 0.05 \times N_1)$.

To create properties in the FE ("fully existential") category, the template is repeated as a series of N_6 disjunctions and for FA properties – as a series of N_6 conjunctions. EA properties were generated as N_7 disjunctions of conjunctions of N_8 instantiations of the template, where N_7 and N_8 were random numbers s.t. $N_7 \times N_8 = N_6$. Similarly, AE properties were comprised of N_7 conjunctions of disjunctions of N_8 instantiations of the template.

Fig. 5 shows an example of an input formula generated randomly using a FA "ownership property" .

Methodology. We conducted a series of experiments generating inputs along the dimensions specified by three parameters: model size, size of set of concretizations and type of property. For each combination of the parameters, we produced inputs using the selected exemplar values shown in Tables II and III.We did multiple runs for each combination and then picked at least 3 runs for each data-point that produced results in each of the three possible return values (True, Maybe, False).

For each run, we used the generated input to execute the two experiments, E1 and E2. For each, we recorded the speedup $S_p = \frac{T_c}{T_{pm}}$, where T_c and T_{pm} were the times to do a task with sets of classical models and with partial models.

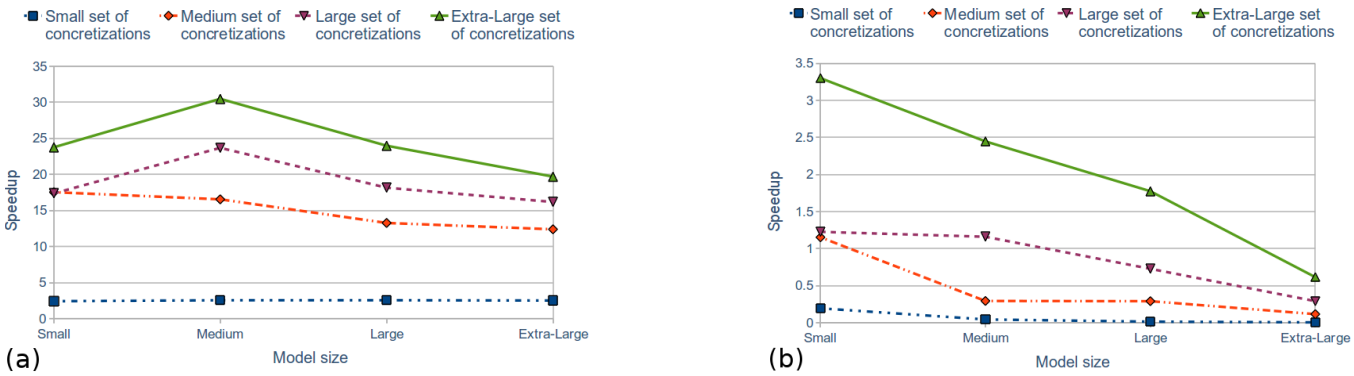

Figure 6. Speedup versus model size for different degrees of uncertainty: (a) experiment E1, (b) experiment E2.

Results. The experiments[1] did not show dramatic differences in speedup between the different property and return types. The biggest difference in speedup for E1 was recorded in the AE category between properties that return Maybe (21.65) and those that return False (29.13), for M-sized models with Large sets of concretizations. For E2, the biggest difference in speedup was recored for S-sized models with XL sets of concretizations, for properties that return Maybe, between the EA (0.36) and AE categories (5.62). This indicates that property and return types are not the prime determinants for the performance of our approach.

On the other hand, the size of the partial model and the size of the set of concretizations had a much larger effect on the recorded variance of speedup. The ranges of recorded speedups for E1 and E2 are shown in Fig. 6(a, b), respectively. The plotted values are averages for type of property and return value for each combination of size of model and size of set of concretizations. This is an indication that these parameters are the most important factors for studying the effectiveness of reasoning with partial models.

Fig. 6(a) shows that for verification and simple diagnostic tasks, such as producing a counter-example, there is a significant speedup from using partial models. The smallest speedups were observed in the inputs with S sets of concretizations (between 2.45 for S-sized models and 2.59 for L-sized models). The increase from these values was dramatic for M, L and XL sets of concretizations. For these categories, the smallest speedup was 19.72 for XL-sized models with M sets of concretizations and the biggest speedup was 30.49 for M-sized models with XL sets of concretizations.

For more complex tasks, such as property-driven refinement, the effect of the size of concretizations, as shown in Fig. 6(b), seems to be the determinant parameter, as the technique offers a speedup greater than 1 for larger sets of concretizations. Our approach was significantly slow for S sets of concretizations (the largest speedup being 0.05 for M-sized models). Moderate speedups were only recorded for XL sets of concretizations and smaller models (3.30 for S, 2.25 for M and 1.78 for L). This points to the conclusion that for more complex tasks, speedup is best for smaller models with larger sets of concretizations.

[1]All results available at http://www.cs.toronto.edu/~famelis/icse12.html

These observations, lead us to the conclusion that, regarding RQ1 (feasibility), there is a significant net gain from using our approach for tasks like verification and counter-example guided diagnosis, whereas for tasks like property-driven refinement there are certain cases where it is preferable to use the classical approach.

Regarding RQ2 (sensitivity to degree of uncertainty), the observations point to the conclusion that the speedup offered by our approach is positively correlated to the degree of uncertainty. In fact, the greatest speedups were observed for inputs that had bigger sizes of sets of concretizations. For smaller levels of uncertainty, explicitly handling the set is more efficient.

These results, albeit preliminary, are encouraging and motivate further research, as they indicate that partial models are a useful representation that can offer significant gains compared to handling the entire set of concretizations.

Threats to Validity. The most important threat to validity stems from the use of experimental inputs that were randomly generated. The formulas that we created for properties were randomly grounded and were generated from a few arbitrarily defined templates.

Another threat to validity is induced by our choice to use a few exemplar values of the experimental parameters in order to manage the combinatorial explosion of options. It is evident that more experimentation is required, to generalize our results and further investigate effects of the experimental parameters that may not have been made obvious by our set of experiments.

To compensate for these threats to validity, we additionally conducted a Case Study, to triangulate our experimental results with experience from applying our technique to a real world application. The size of the models that we extracted from the Case Study fell in the XL category, with M and L sets of concretizations, whereas the properties were in the FE category and returned True and Maybe. The observed speedups (detailed in the next section))were consistent with our experimental results.

VI. Case Study

Problem Description. In this case study, we aim to illustrate the following MDE software maintenance scenario: An engineer is given the task of fixing a software defect by modifying

its UML model which will subsequently be used to construct the modified software (e.g., via a transformation). However, after creating the modifications to the model, the engineer finds that some model constraints are violated and thus the software cannot be constructed. For example, she may have modified a sequence diagram without properly synchronizing it with the structural aspects (e.g., classes) of the model. To help her resolve these constraint violations, she uses a tool that can automatically propose different model repair alternatives (e.g., [12]). Suppose the engineer is uncertain about which alternative to choose because their relative merits are unclear – and thus she would like to reason with the set of alternatives to help her make the choice and possibly even defer the decision until more information is available. In this case study, we apply the partiality techniques developed in this paper to show how they could help her in this scenario and to illustrate the feasibility of the approach.

We use an open source project UMLet [23], which is a simple Java-based UML editor, as the software on which our user is requested to perform a maintenance task. This project has also been used by Van Der Straeten et al. for finding model inconsistencies with a model finder [24]. The goal of the maintenance task is to fix the following bug, referred to as Issue 10 on the online issue log [22]: "copied items should have a higher z-order priority". That is, if the user copies and then pastes an item within the editor, it is not the topmost item if it overlaps with other existing items. Thus, any fix to the bug must satisfy the following property P1: "Each item that is pasted from the clipboard must have z-order = 0." The paste functionality is implemented in UMLet by instantiating the class *Paste* and invoking its *execute* operation. Fig. 7 shows a fragment of the sequence diagram, generated from the code using the Borland TogetherJ tool [1] for *execute* with the circled portion representing a bug fix we propose. The full sequence diagram has 12 objects, 53 messages and 8 statement blocks. Although UMLet has 214 classes in total, we restrict ourselves to a slice that covers the sequence diagram for *execute* consisting of 6 classes (plus 5 Java library classes) with 44 operations. Of the 12 objects in the sequence diagram, 5 are instances of Java library classes and 7 – of UMLet classes. In the fragment shown, the *for* loop statement block iterates through every item in the clipboard (indexed by variable *e*) and adds it to the editor window (represented by the object *pnl*:*DrawPanel*). When an entity is added to a *DrawPanel*, the z-order is not set to 0 by default, causing the bug. In our proposed fix (shown in the dashed circle), we create a transient object *positioner* and tell it to *moveToTop*(*e*), using the Swing operation *setComponentZOrder*.

Inconsistencies and the Partial Model. Our fix is conceptually correct but it violates two consistency rules required for code generation:

1) **ClasslessInstance:** Every object must have a class. Possible repairs:
   - *RC1:* Remove the object.
   - *RC2:*(*obj*) Replace the object with an existing object *obj* that has a class.

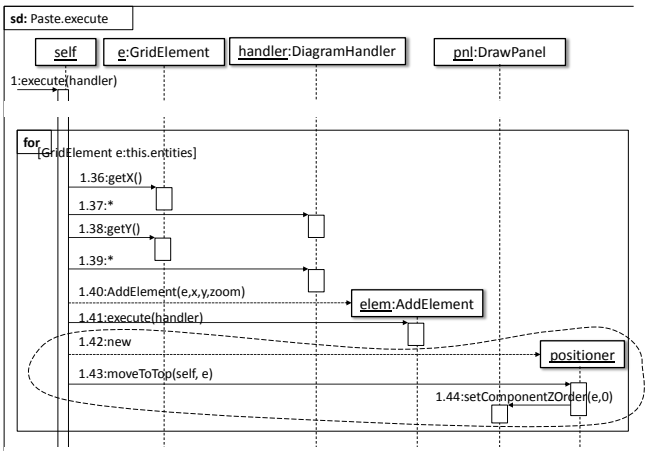

Figure 7. Our fix for the *execute* operation of the UMLet paste function.

   - *RC3:*(*cls*) Assign the object to the existing class *cls*.
   - *RC4:* Assign the object to a new class.
2) **DanglingOperation:** The operation used by a message in a sequence diagram must be an operation of the class of the receiving object. Possible repairs:
   - *RD1:* Put the operation into the receiving object's class.
   - *RD2:*(*op*) Change the operation to the operation *op* that is already in the receiving object's class.
   - *RD3:* Remove the message.

ClasslessInstance and DanglingOperation are both based on [21]. In our case, the *positioner* object violates ClasslessInstance and the message with operation *moveToTop* violates DanglingOperation because it is not in *positioner's* class (since *positioner* has no class).

If we apply all possible repairs, we get a set of alternative ways to fix the inconsistency, summarized as follows:

1) *Positioner* can be removed (*RC1*), can be replaced by one of the existing 7 objects (*RC2*), can be assigned to one of the existing 6 classes (*RC3*), or can be an instance of a new class (*RC4*).
2) The operation *moveToTop* can be added to the *positioner's* class (*RD1*), can be changed to one of the other 44 operations depending on *positioner's* class (*RD2*), or can be removed (*RD3*).

Only certain repairs are mutually compatible – for example, *RC1* cannot be used with *RD2* since the latter depends on *positioner*'s class but the former removes *positioner* entirely. There are 220 alternatives in total for all valid combinations.

If we construct a partial model to represent this set of alternatives, all the model elements in the proposed fix in Fig. 7 become Maybe since they are present in some alternatives and absent in others. Furthermore, based on the compatible combinations of repairs, the *may* formula portion of the partial model is expressed as

$$\phi_M \quad = \quad \mathit{Choose}((\phi_{RC1} \wedge \phi_{RD3}), \\ (\phi_{RC2(e)} \wedge \phi_{RD1}), (\phi_{RC2(e)} \wedge \phi_{RD2(getX)}), \ldots),$$

where *Choose*($\phi_1, \ldots, \phi_n$) is a logical function that holds when exactly one of $\{\phi_i\}_{1 \le i \le n}$ hold. Each of the formulas

for the individual repairs can be further expanded and expressed in terms of the UML 2 metamodel [16]. For example, $\phi_{RC2(e)}$ represents the condition that object *positioner* is replaced by object *e* in Fig. 7, expressed as

$$\phi_{RC2(e)} \quad = \quad \textit{covered}(\textit{receiveEvent}(\textit{Message\_1.43})) = \textit{Lifeline\_e}$$

which says that the lifeline covered by the receiving event of message 1.43 is the one for object *e*:*GridElement*.

Analysis. Having defined a partial model whose set of concretizations are the possible alternative ways of making our bug fix consistent with the required rules, we can use the techniques discussed in Sec. IV to reason about the alternatives using properties. The first question is whether any of the alternatives "break" the paste functionality. For example, consider the property P2: "Whenever an item is pasted, a new item is created in the editor window" which should hold if the paste functionality is implemented correctly. To check this against the partial model, we encode it into a propositional formula ϕ_{P2} over the UML metamodel in the same way as the repair formulas. Due to lack of space, we limit ourselves to a high-level description of the encoding: "ϕ_{P2} holds iff in the sequence diagram for *execute* there exists an iteration over the items in the clipboard (e.g., a *for* block) that creates a copy of each item using the operation *CloneFromMe*() and later adds the item to the editor window using the command *AddElement*". Only the *AddElement* portion is visible in the fragment in Fig. 7.

We used the MathSAT implementation described in Sec. V to verify this property (OP2), checking the queries $\phi_M \wedge \phi_{P2}$ and $\phi_M \wedge \neg\phi_{P2}$. The result was True,indicating that all concretizations satisfy the property. We also did the comparison with doing this task using classical models as in Sec. V and found a speedup of 30.68. Next we considered the critical property P1 that is required for any fix to our bug. In this case, OP2 yielded Maybe indicating that some but not all concretizations are acceptable bug fixes. In this case, we found a speedup of 31.23.

To investigate why some concretizations did not fix the bug, we used our diagnosis technique (OP3c) to produce the partial model representing the counterexamples to P1 by setting the *may* formula to $\phi_M \wedge \neg\phi_{P1}$ and computing the GNF. In the resulting may model, the *moveToTop* operation is absent and thus *moveToTop* is necessary for P1 to hold. This is reasonable since if this operation is not invoked, then the z-order is never set to 0. In a similar way, we used property-guided refinement (OP4) to refine our partial model to represent only the satisfying concretizations by setting the *may* formula $\phi_M \wedge \phi_{P2}$ and computing the GNF. The speedup for these two tests were 0.19 and 0.05, respectively.

VII. Related Work

A number of partial *behavioral* modeling formalisms, have been studied in the context of abstraction (for verification) or for capturing early design models [9]. For example, Modal Transition Systems (MTSs) [10] allow introduction of uncertainty about transitions on a given event, whereas Disjunctive Modal Transition Systems (DMTSs) [11] add an additional constraint that at least one of the possible transitions must be taken in the refinement. These approaches compactly encode an over-approximation of the set of possible LTSs and thus reasoning over them suffers from information loss. Moreover, the MTS and DMTS refinement mechanism allows resulting LTS models to have an arbitrary number of states which is different from the treatment provided in this paper, where we concentrated only on "structural" partiality and thus state duplication was not applicable.

Another relevant area is product line software development [17] which captures the set of potential models by identifying their commonalities and variabilities. Most approaches keep the expressions of variability in a separate feature model but some incorporate these directly into the model using notational extensions in the metamodel [13]. Featured Transition Systems (FTSs) [4] are most closely related to the notion of partial models presented in this paper. FTSs encode a set of products by annotating transitions with specific features from a feature diagram (much like our *may* formula), and differ from MTSs and DMTSs in that they support *precise* representation and reasoning with a set of models.

Our approach is distinct from related work in a number of important ways. First, it applies to *any* kind of modeling language (not just behavioral models) that can be defined using a metamodel. Second, our viewpoint is the comprehensive handling of uncertainty rather than just reasoning over variability. In this context, partial models support changes in the level of uncertainty, with tasks such as property-driven (OP4) and more generally *uncertainty-removing* refinement [20]. Third, partial models are first-class development artifacts that can be manipulated throughout the software engineering life cycle with *detail-adding* transformations that do not affect the level of uncertainty [8]. Finally, the notion of partiality studied in this paper (where model elements can be optional or mandatory) is only one of several kinds of partiality, developed in [19].

VIII. Conclusion and Future Work

This paper presented an approach for reasoning in the presence of uncertainty. We showed how to construct partial models to represent sets of alternatives and how to use them for reasoning. We evaluated the approach by running experiments using randomly generated inputs and triangulated our results with a case study dealing with alternative repairs to inconsistency for a real world software project. Our evaluation, while preliminary, showed that in the presence of high degrees of uncertainty, using partial models offers significant improvements for reasoning tasks.

Our work is part of a broader research agenda, outlined in [7]. Our next steps include studying how partial models can be used as first-class development items. In particular, we want to investigate model transformation of partial models, as well as the effects of transformation on the properties of the concretizations.

REFERENCES

[1] Borland TogetherJ. *website : http://www.borland.com/us/products/together/*, accessed 03-20-2012.

[2] G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proc. of CAV'99*, pages 274–287, 1999.

[3] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, and R. Sebastiani. "The MathSAT 4 SMT Solver". In *Proc. of CAV'08*, pages 299–303, 2008.

[4] P. Classen, A.and Heymans, P. Schobbens, A. Legay, and J. Raskin. "Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines". In *Proc. of ICSE'10*, pages 335–344, 2010.

[5] 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.

[6] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS. Springer, 2006.

[7] M. Famelis, S. Ben-David, M. Chechik, and R. Salay. "Partial Models: A Position Paper". In *Proc. of MoDeVVa'11*, pages 1–6, 2011.

[8] M. Famelis, R. Salay, and M. Chechik. "The Semantics of Partial Model Transformations". In *Proc. of MiSE'12*, 2012. To appear.

[9] D. Fischbein, G. Brunet, N. D'ippolito, M. Chechik, and S. Uchitel. "Weak Alphabet Merging of Partial Behaviour Models". *ACM TOSEM*, 21, 2012.

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

[11] P. Larsen. "The Expressive Power of Implicit Specifications". In *Proc. of ICALP'91*, volume 510 of *LNCS*, pages 204–216, 1991.

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

[13] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. M. Jézéquel. "Weaving Variability into Domain Metamodels". *J. Model Driven Engineering Languages and Systems*, pages 690–705, 2009.

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

[15] Object Management Group. *Object Constraint Language OMG Available Specification Version 2.0*, 2006.

[16] OMG. *UML Superstructure Specification Version 2.3*, 2010.

[17] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag New York Inc, 2005.

[18] M. Sabetzadeh, S. Nejati, M. Chechik, and S. Easterbrook. "Reasoning about Consistency in Model Merging". In *Proc. of LWI'10*, 2010.

[19] R. Salay and M. Chechik. "Language Independent Refinement using Partial Modeling". In *Proc. of FASE'12*, 2012. To appear.

[20] R. Salay, M. Chechik, and J. Gorzny. "Towards a Methodology for Verifying Partial Model Refinements". In *Proc. of VOLT'12*, 2012. To appear.

[21] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. "Using Description Logic to Maintain Consistency between UML Models". In *Proc. of UML'03*, pages 326–340, 2003.

[22] UMLet. *UMLet Issue List : http://code.google.com/p/umlet/issues/list*, accessed 03-20-2012.

[23] UMLet. *UMLet website : http://www.umlet.com/*, accessed 03-20-2012.

[24] R. Van Der Straeten, J. Pinna Puissant, and T. Mens. "Assessing the Kodkod Model Finder for Resolving Model Inconsistencies". *Modelling Foundations and Applications*, 6698:69–84, 2011.

[25] A. Van Deursen, P. Klint, and J. Visser. "Domain-Specific Languages: An Annotated Bibliography". *ACM Sigplan Notices*, 35(6):26–36, 2000.

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

[27] H. Ziv, D. Richardson, and R. Klösch. "The Uncertainty Principle in Software Engineering", 1996.