Towards a Methodology for Verifying Partial Model Refinements

Rick Salay, Marsha Chechik and Jan Gorzny
University of Toronto
Toronto, Canada
{rsalay,chechik,jgorzny}@cs.toronto.edu

***Abstract*—Models are good at expressing information that is known but do not typically have support for representing what information a modeler does not know or does not care about at a particular stage in the software development process. Partial models address this by being able to precisely represent uncertainty about model content. In previous work, we have defined a general approach for defining partial model semantics using a first order logic encoding. In this paper, we use this FO encoding to formally define the conditions for partial model refinement in the manner of the refinement of algebraic specifications. We use this approach to verify both manual refinements and automated transformation-based refinements. We illustrate our approach using example models and transformations.**

I. Introduction

Uncertainty in models comes from many sources: incomplete requirements, presence of alternative design decisions, disagreements among stakeholders. Model-based software development is a process in which uncertainty about the final product is incrementally reduced through a series of refinement steps. Yet modeling languages do not typically provide adequate support for explicating uncertainty.

To illustrate, consider the class diagram in Figure 1 that depicts a portion of a hypothetical automotive design project. The textual notes in the diagram represent the modeler's uncertainty by stating specific information that is known and unknown about the model. All of this information must be specified ad-hoc, using natural language, since there is no notational mechanism in the class diagram language for it. Refinement is supposed to reduce the uncertainty; yet, natural language expression of uncertainty makes it impossible to accurately (or automatically) verify correctness of refinement.

To help address this gap in expressiveness, we have proposed several types of *partiality* annotations with formal semantics that could be used to augment any modeling language with the means to accurately express uncertainty [7]. We call the resulting model *partial*. Model `P1` in Figure 2 shows the use of partiality annotations to express the uncertainty in Figure 1. In each case, the annotation is given in brackets as a prefix to the element's name. For example, the `S` annotation on the operation `cruiseControlOps` (in class `BodyController`) means that it represents a (as yet unknown) set of operations. This captures the same information as in the note attached to `BodyController` in Figure 1 – i.e., that it contains operations for cruise control but it is still unknown what they are. The `V` annotation on the `EngineDisabler` class means that it is a "variable" class and that it is still unknown whether it is assigned to a new class or to one of the existing classes; however, regardless of how it gets assigned, it must contain a set of `engineDisablerOps` operations. Furthermore, the `M`-annotated composition associations say that if `engineDisabler` is assigned to a new class then it *may* have a composition relationship either with the `EngineController` class or with the `SecurityController` class. Yet it cannot have this relationship with both classes since the well-formedness rules for class diagrams prohibit this.

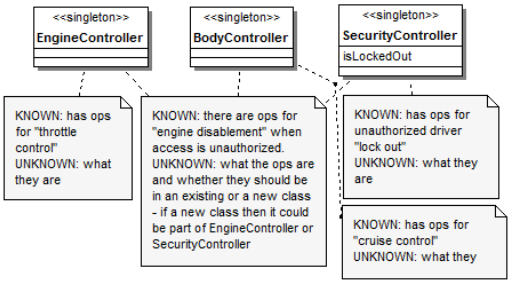


Fig. 1. Class diagram for automotive example showing ad-hoc expressions of uncertainty.

Resolving uncertainty for some of these points of partiality is reflected by constructing a *partial model refinement* of the model. For example, Figure 2 shows a partial model refinement of the partial class diagram `P1`. The refinement represents the way in which the elements in the two models are mapped to each other and captures the uncertainty resolution decisions made. To avoid visual clutter, we show only the non-obvious parts of the mapping: the `S`-annotated operation `cruiseControlOps()` is refined to a set of particular operations {`cruiseOff()`, `setCruiseSpeed`}, a decision is made to put the engine disablement functionality into the `EngineController` class by assigning the `V`-annotated class `EngineDisabler` to it, and the `M`-composition relations are eliminated.

Figure 2 shows a *manual* refinement, i.e., a refinement applied to a particular model. In contrast, Figure 3 gives a partial model transformation that can be used to generate *automated* refinements – refinements applied to arbitrary models. A rule is applied by finding an instance of its left hand side in the source model and replacing it with the fragment on the right hand side, and the transformation corresponds to the repeated application of the rule until it can no longer

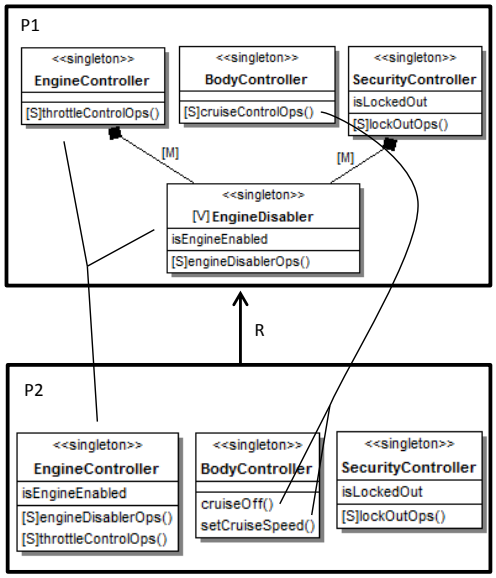


Fig. 2. Example refinement of the partial model P1.

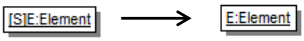


Fig. 3. A graph transformation rule defining transformation *ReduceAbs*.

be applied. Syntactically, *ReduceAbs* removes all occurrences of S annotations on elements. Semantically, it means that these elements now represent particular elements rather than an arbitrary set of elements. From a pragmatic perspective, this transformation reduces uncertainty about these elements and thus is a partiality refinement transformation.

Whether we are defining manual refinements or automated transformation-based refinements, the verification of their correctness is an important methodological consideration. This paper uses the formalization of partial models introduced in [7] as a foundation for a methodology for verifying correctness of manual refinements. It then attempts to extend this methodology for verification of refinement transformations.

Specifically, we make the following contributions:

- We develop and illustrate a methodology for verifying manual partial model refinements for a partiality mechanism that is both expressive and language-independent.
- We apply the methodology to the verification of two specific partiality refinement transformations and then discuss how our experience might be generalized to a methodology for verifying partiality refinement transformations in general.
- We describe prototype tool support based on Alloy to help automate the verification methodology.

The rest of the paper is organized as follows. In Section II, we review the concept of model partiality as introduced in [7]. In Section III, we define the verification methodology for manual partial model refinements and in Section IV, we illustrate its application. In Section V, we apply this methodology to example transformations and make observations about adapting it to the verification of partiality refinement transformations in general. In Section VI, we describe a prototype implementation of this verification method using Alloy. In

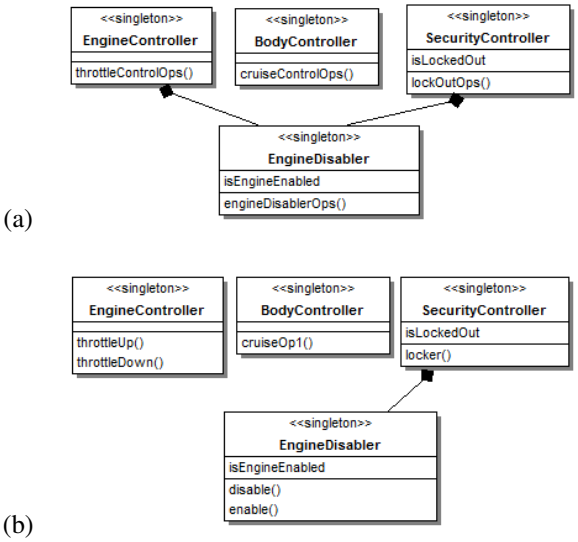


Fig. 4. The base model (a) and a concretization (b) of the partial class diagram P1 in Figure 2.

Section VII, we discuss related work. In Section VIII, we discuss conclusions and future work.

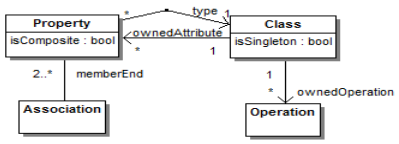
II. Background

In this section we briefly review the concepts of language-independent partial modeling introduced in [7]. When a model contains partiality information, we call it a *partial* model. Semantically, a partial model represents the set of different possible concrete (i.e., non-partial) models that would resolve the uncertainty represented by the partiality. More formally:

Definition 1 (Partial model): A partial model P consists of a *base model*, denoted $bs(P)$, and a set of annotations. Let T be the metamodel of $bs(P)$. Then, $[P]$ denotes the set of T models called the *concretizations* of P. P is called *consistent* iff $[P] \neq \emptyset$.

For example, Figure 4(a) shows the base model (i.e., what remains when the annotations are stripped away) of the partial class diagram P1 in Figure 2. Note that the base model need not necessarily be well-formed and the figure illustrates this because it violates the well-formedness rule that a singleton class cannot be composed in two different classes. This shows that non-well-formed base models are necessary to express some cases of uncertainty. Figure 4(b) shows one of the concretizations of P1. P1 has an infinite number of concretizations since each of the S-annotated operations can be replaced by any set of particular operations. Thus, although $bs(\texttt{P1})$ is not well-formed, P1 is still consistent since it has concretizations.

We use four types of partiality annotations, each adding support for a different type of uncertainty in a model: Annotating an element with M indicates that we are unsure about whether it should exist in the model; annotating an element with S indicates that we are unsure about whether it should actually be a collection of elements; annotating an element with V indicates that we are unsure about whether it should actually be merged with other elements; finally, annotating the entire model with INC indicates that we are unsure about whether it is complete.



Additional constraints:

(1) A singleton class cannot be composed in more than one class.

$\forall c\colon Class \cdot \exists c_1, c_2\colon Class \cdot isSingleton(c) \wedge IN(c, c_1) \wedge IN(c, c_2) \Rightarrow c_1 = c_2$

$IN(c, c') \Leftrightarrow \exists a\colon Association, p, p'\colon Property \cdot memberEnd(a, p) \wedge memberEnd(a, p') \wedge type(p) = c \wedge type(p') = c' \wedge isComposite(p')$

Fig. 5. A simplified metamodel of the UML class diagram language.

When these four types of partiality annotations are used together, we refer to it as *MAVO* partiality. We state the following proposition, without proof, that will be used in later sections.

Proposition 1: Given *MAVO* model P over concrete models with metamodel T, if $bs(P)$ is well-formed w.r.t T then it must also be a concretization of P and so P is consistent.

To formalize *MAVO* partiality, we begin by noting that a metamodel represents a set of models and can be expressed as a First Order Logic (FOL) theory.

Definition 2 (Metamodel): A *metamodel* is an FOL theory $T = \langle \Sigma, \Phi \rangle$, where Σ is the *signature* with sorts and predicates representing the element types, and Φ is a set of sentences representing the well-formedness constraints. The models that conform to T are the finite FO Σ-structures that satisfy Φ according to the usual FO satisfaction relation. We denote the set of models with metamodel T by $Mod(T)$.

The simple class diagram metamodel in Figure 5 fits this definition if we interpret boxes as sorts and edges as predicates comprising $\Sigma_{\texttt{CD}}$ and take the multiplicity constraints (translated to FOL) and the additional constraint (1) as comprising $\Phi_{\texttt{CD}}$.

Like a metamodel, a partial model also represents a set of models and thus can also be expressed as an FOL theory. Specifically, for a partial model P, we construct a theory $FO(P)$ s.t. $Mod(FO(P)) = [P]$. Furthermore, since P represents a subset of T models, we require that $Mod(FO(P)) \subseteq Mod(T)$. We guarantee this by defining $FO(P)$ to be an extension of T that adds constraints.

Let $M = bs(P)$ be the base model of a partial model P and let P_M be the *ground* partial model which has M as its base model and its sole concretization – i.e., $bs(P_M) = M$ and $[P_M] = \{M\}$. We first describe the construction of $FO(P_M)$ and then define $FO(P)$ in terms of $FO(P_M)$. To construct $FO(P_M)$, we extend T to include a unary predicate for each element in M and a binary predicate for each relation instance between elements in M. Then, we add constraints to ensure that the only first order structure that satisfies the resulting theory is M itself.

We illustrate the above construction using the partial class diagram P1 in Figure 2. Let $\texttt{M1} = bs(\texttt{P1})$ be its base model and $P_{\texttt{M1}}$ be the corresponding ground partial model. We have:

$$FO(P_{\texttt{M1}}) = \langle \Sigma_{\texttt{CD}} \cup \Sigma_{\texttt{M1}}, \Phi_{\texttt{CD}} \cup \Phi_{\texttt{M1}} \rangle \quad (1)$$

(see Definition 2), where $\Sigma_{\texttt{M1}}$ and $\Phi_{\texttt{M1}}$ are model M1-specific predicates and constraints, defined in Figure 6 that extend the signature and constraints for class diagrams, as described in Figure 5. We refer to $\Sigma_{\texttt{M1}}$ and $\Phi_{\texttt{M1}}$ as the *MAVO* predicates and constraints, respectively. For conciseness, we abbreviate element names in Figure 6 - e.g., BodyController becomes BC, etc.

$\Sigma_{\texttt{M1}}$ has unary predicates $\texttt{BC}(\texttt{Class})$, $\texttt{CCOps}(\texttt{Operation})$), ..., and binary predicates $\texttt{BCownsCCOps}(\texttt{Class}, \texttt{Operation})$, ...

$\Phi_{\texttt{M1}}$ contains the following sentences:

(Complete) $(\forall x : \texttt{Class} \cdot \texttt{BC}(x) \vee \texttt{EC}(x) \vee \texttt{SC}(x) \vee \texttt{ED}(x)) \wedge$
$(\forall x : \texttt{Class}, y : \texttt{Operation} \cdot \texttt{ownedOperation}(x, y)$
$\Rightarrow (\texttt{BCownsCCOps}(x, y) \vee \ldots)) \wedge \ldots$

BC:

($Exists_{\texttt{BC}}$) $\exists x : \texttt{Class} \cdot \texttt{BC}(x)$

($Unique_{\texttt{BC}}$) $\forall x, x' : \texttt{Class} \cdot \texttt{BC}(x) \wedge \texttt{BC}(x') \Rightarrow x = x'$

($Distinct_{\texttt{BC}-\texttt{EC}}$) $\forall x : \texttt{Class} \cdot \texttt{BC}(x) \Rightarrow \neg\texttt{EC}(x)$

($Distinct_{\texttt{BC}-\texttt{SC}}$) $\forall x : \texttt{Class} \cdot \texttt{BC}(x) \Rightarrow \neg\texttt{SC}(x)$

($Distinct_{\texttt{BC}-\texttt{ED}}$) $\forall x : \texttt{Class} \cdot \texttt{BC}(x) \Rightarrow \neg\texttt{ED}(x)$

similarly for all other element and relation predicates

Fig. 6. The FO encoding of $P_{\texttt{M1}}$.

Since $FO(P_{\texttt{M1}})$ extends CD, the FO structures that satisfy $FO(P_{\texttt{M1}})$ are the class diagrams that satisfy the constraint set $\Phi_{\texttt{M1}}$ in Figure 6. Assume N is such a class diagram. The *MAVO* constraint *Complete* ensures that N contains no more elements or relation instances than M1. Now consider the class BC in M1. $Exists_{\texttt{BC}}$ says that N contains at least one class called BC, $Unique_{\texttt{BC}}$ – that it contains no more than one class called BC, and the clauses $Distinct_{\texttt{BC}-*}$ – that the class called BC is different from all the other classes. Similar *MAVO* constraints are given for all other elements and relation instances in M1. These constraints ensure that $FO(P_{\texttt{M1}})$ has exactly one concretization and thus $N = \texttt{M1}$.

Relaxing the *MAVO* constraints $\Phi_{\texttt{M1}}$ allows additional concretizations and represents a type of uncertainty indicated by a partiality annotation. For example, if we use the INC annotation to indicate that M1 is incomplete, we can express this by removing the *Complete* clause from $\Phi_{\texttt{M1}}$ and thereby allow concretizations to be class diagrams that extend M1. Similarly, expressing the effect of the M, S and V annotations for an element E correspond to relaxing $\Phi_{\texttt{M1}}$ by removing $Exists_E$, $Unique_E$ and $Distinct_{E-*}$ clauses, respectively. For example, removing the $Distinct_{\texttt{ED}-*}$ clauses is equivalent to marking the class ED with V(i.e., EngineDisabler may or may not be distinct from another class).

III. A Methodology for Verifying Manual Refinement of Partial Models

In Section II, we formally characterized the set of concretizations of a partial model using an FOL encoding. In this section, we define refinement in terms of this encoding.

Assume we have encodings $FO(P) = \langle \Sigma_P, \Phi_P \rangle$ and $FO(P') = \langle \Sigma_{P'}, \Phi_{P'} \rangle$ for partial models P and P', respectively. In the special case that they have the same base models, i.e., $\Sigma_{P'} = \Sigma_P$, P' refines P iff the following conditions hold:

$$\Phi_{P'} \text{is satisfiable} \quad (2)$$

$$\Phi_{P'} \Rightarrow \Phi_P \quad (3)$$

Given partial models P, P', the following steps verify that P' is a partial model refinement of P.

1) Determine first-order encodings $FO(P) = \langle \Sigma_P, \Phi_P \rangle$ and $FO(P') = \langle \Sigma_{P'}, \Phi_{P'} \rangle$.
2) Prove that $\Phi_{P'}$ is satisfiable (1st proof obligation).
3) Determine the first-order encoding of the mapping $R = \langle \Sigma_P + \Sigma_{P'}, \Phi_R \rangle$ as the intermediate theory, where Φ_R defines the elements of the signature Σ_P in terms of the signature $\Sigma_{P'}$. Note that $\Sigma_P + \Sigma_{P'}$ is the disjoint union of the signatures so that names are made distinct in case of clashes.
4) Prove that $\Phi_{P'} \Rightarrow R(\Phi_P)$ (2nd proof obligation).

Fig. 7. A method for verifying manual MAVO refinement.

Condition (2) ensures that P' is *consistent* – i.e., that $[P'] \neq \emptyset$ and thus has at least one concretization. Recall from formula (1) that $\Phi_{P'}$ consists of both the *MAVO* sentences and the well-formedndess rules for the modeling language and so these must be jointly satisfiable for this condition to hold. Condition (3) ensures that $[P'] \subseteq [P]$ so that P' has no more concretizations than P. These conditions are *proof obligations* required to be met in order to demonstrate the validity of the refinement.

When the base models are different, we cannot use this simple scheme because $\Sigma_{P'} \neq \Sigma_P$ and so the sentences are not directly comparable. For example, the base models differ for the refinement shown in Figure 2. The classic solution to this kind of problem is to first translate both $FO(P)$ and $FO(P')$ into the same intermediate theory where the signatures are appropriately related, or *mapped*, and then check whether the implication holds in this intermediate theory. For example, such an approach is taken by [6]. Specifically, we seek a mapping R that defines the signature of $FO(P)$ in terms of the signature of $FO(P')$. Such a mapping naturally defines a translation $R(\Phi_P)$ of the sentences of $FO(P)$ to equivalent sentences in terms of the signature of $FO(P')$ by replacing each occurrence of a sort or predicate of Σ_P by its definition in terms of $\Sigma_{P'}$. Since they are over the same signature, the translation $R(\Phi_P)$ can thus be compared to $\Phi_{P'}$ and the revised second proof obligation becomes:

$$\Phi_{P'} \Rightarrow R(\Phi_P) \tag{4}$$

Note that Condition (4) reduces to Condition (3) when $\Sigma_{P'} = \Sigma_P$, as expected.

A methodology for verifying a (manual) refinement based on the above discussion is given in Figure 7. In the following sections, we apply this methodology to particular examples, starting with manual refinement and then automated refinement.

IV. Applying the Methodology to Verifying Manual Refinements

In this section, we apply the refinement verification methodology in Figure 7 to show that the refinement in Figure 2 is correct. We address each of the four steps of the methodology as follows.

1) The definition of $FO(\texttt{P1})$ is given in Figure 6. Due to lack of space we omit $FO(\texttt{P2})$ but note that $\Phi_{\texttt{P2}}$ contains all *MAVO* sentences except *Unique*$_{\texttt{EDOps}}$, *Unique*$_{\texttt{TCOps}}$, and *Unique*$_{\texttt{LOOps}}$.
2) To prove the satisfiability of $\Phi_{\texttt{P2}}$, we note that the base model of P1 (i.e., the class diagram with all annotations removed) is well-formed and, by Proposition 1, a well-formed base model is always a concretization. Thus, $[\texttt{P2}] \neq \emptyset$ and so $\Phi_{\texttt{P2}}$ must be satisfiable.
3) The FOL encoding of mapping R is shown in Figure 8.
4) To prove that $\Phi_{\texttt{P2}} \Rightarrow \texttt{R}(\Phi_{\texttt{P1}})$, we must show that $\Phi \Rightarrow \texttt{R}(\phi_{\texttt{P1}})$ for each sentence $\phi_{\texttt{P1}} \in \Phi_{\texttt{P1}}$ where $\Phi \subseteq \Phi_{\texttt{P2}}$. The proof is given below.

Proof: We proceed with a proof by cases of *MAVO* constraints in $\Phi_{\texttt{P1}}$. The first four cases examine the places where P1 and P2 differ while the fifth one covers all places where they are the same.

Case 1 (*Complete*): Let $\phi_1 \in \Phi_{\texttt{P1}}$ and $\phi_2 \in \Phi_{\texttt{P2}}$ be the *Complete* constraints for P1 and P2, respectively. Now note that $\texttt{R}(\phi_1)$ is identical to ϕ_2 everywhere except for the clause for the Association elements. In that case, the clause in $\texttt{R}(\phi_1)$ is $\forall x : \texttt{Association} \cdot \text{false}(x) \vee \text{false}(x)$ whereas the clause in ϕ_2 is $\forall x : \texttt{Association} \cdot \text{false}(x)$. These are clearly semantically equivalent and so $\phi_2 \Rightarrow \texttt{R}(\phi_1)$.

Case 2 (CCOps): $\Phi_{\texttt{P1}}$ contains the *Exists* and *Distinct* constraints for operation CCOps: $\texttt{R}(\textit{Exists}_{\texttt{CCOps}}) = \exists x : \texttt{Operation} \cdot \texttt{CO}'(x) \vee \texttt{SCS}'(x)$ which clearly follows from the constraint *Exists*$_{\texttt{CO}'}$ in $\Phi_{\texttt{P2}}$.

$$\texttt{R}(\textit{Distinct}_{\texttt{CCOps}-e}) = \forall x : \texttt{Operation} \cdot (\texttt{CO}'(x) \vee \texttt{SCS}'(x)) \Rightarrow \neg e(x) \text{ and}$$
$$\texttt{R}(\textit{Distinct}_{e-\texttt{CCOps}}) = \forall x : \texttt{Operation} \cdot e(x)) \Rightarrow \neg((\texttt{CO}'(x) \vee \texttt{SCS}'(x))$$

for each operation $e \in \{\texttt{TCOps}, \texttt{LOOps}, \texttt{EDOps}\}$. Both of these follow from $\{\textit{Distinct}_{\texttt{CO}'-\texttt{e}'}, \textit{Distinct}_{\texttt{SCS}'-\texttt{e}'}\} \subseteq \Phi_{\texttt{P2}}$.

Case 3 (EDinEC, EDinSC): $\Phi_{\texttt{P1}}$ contains the *Unique* and *Distinct* constraints for associations EDinEC and EDinSC.

$$\texttt{R}(\textit{Unique}_{\texttt{EDinEC}}) = \texttt{R}(\textit{Unique}_{\texttt{EDinEC}}) = \forall x, y : \texttt{Association} \cdot \text{false}(x) \wedge \text{false}(y) \Rightarrow x = y \text{ and}$$
$$\texttt{R}(\textit{Distinct}_{\texttt{EDinEC}-\texttt{EDinSC}}) = \texttt{R}(\textit{Distinct}_{\texttt{EDinSC}-\texttt{EDinEC}}) = \forall x : \texttt{Operation} \cdot \text{false}(x) \Rightarrow \neg\text{false}(x)$$

Both of these are always true.

Case 4 (ED): $\Phi_{\texttt{P1}}$ contains the *Exists* and *Unique* constraints for class ED. $\texttt{R}(\textit{Exists}_{\texttt{ED}}) = \textit{Exists}_{\texttt{EC}'}$ and $\texttt{R}(\textit{Unique}_{\texttt{ED}}) = \textit{Unique}_{\texttt{EC}'}$, and both of these EC′ constraints occur in $\Phi_{\texttt{P2}}$.

Case 5: Every other element or relationship instance a in P1 is mapped to its equivalent a' in P2. Thus, if the *MAVO* constraint $\phi_a \in \Phi_{\texttt{P1}}$ holds, then the corresponding constraint $\phi_{a'} \in \Phi_{\texttt{P2}}$ holds as well. Furthermore, $\texttt{R}(\phi_a) = \phi_{a'}$ and so $\phi_{a'} \Rightarrow \texttt{R}(\phi_a)$. ■

V. Towards Applying the Methodology to Verifying Refining Transformations

A. Two Example Transformations

We consider two transformations of *MAVO* partial models defined by graphical rules. The first is the language-

$\Sigma_{\mathtt{R}} = \Sigma_{\mathtt{P1}} + \Sigma_{\mathtt{P2}}$ where the elements of $\Sigma_{\mathtt{P2}}$ are "primed" to avoid name clashes

$\Phi_{\mathtt{R}}$ contains sentences:

$\forall x : \mathtt{Operation} \cdot \mathtt{CCOps}(x) \iff \mathtt{CO}'(x) \vee \mathtt{SCS}'(x)$

$\forall x : \mathtt{Association} \cdot \mathtt{EDinEC}(x) \iff \text{false}(x)$

$\forall x : \mathtt{Association} \cdot \mathtt{EDinSC}(x) \iff \text{false}(x)$

$\forall x : \mathtt{Class} \cdot \mathtt{ED}(x) \iff \mathtt{EC}'(x)$

for all remaining elements e of type T,

$\forall x : T \cdot e(x) \iff e'(x)$

for all remaining relation instances $r(T_1, T_2)$,

$\forall x_1 : T_1, x_2 : T_2 \cdot r(x_1, x_2) \iff r'(x_1, x_2)$

Fig. 8. The FOL encoding of mapping R in Figure 2.

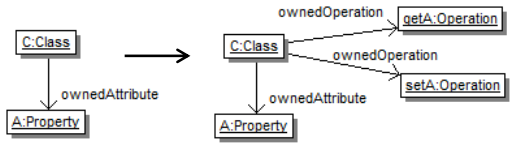


Fig. 9. The rule defining transformation *GetSet*.

independent transformation *ReduceAbs* discussed in the Section I and shown in Figure 3. The second is *GetSet* shown in Figure 9. *GetSet* is a simple detail-adding refinement transformation for class diagrams that we "lift" so that it can be applied to *MAVO* class diagrams. Our objective here is to examine the common situation where partiality-reducing refinements are interleaved with detail-adding ones. In both cases, the transformation is obtained by applying the corresponding rule repeatedly until it can no longer be applied. This process terminates for each rule, although we do not provide proofs of termination here.

In both cases, our objective is to show that the transformation F is a *refining* transformation, i.e., that *for all* models P, $F(P)$ is a valid partiality reducing refinement of P according to the methodology in Figure 7. Since our methodology requires a mapping, F must produce one between $F(P)$ and P. To define this mapping, we assume that all parts of a model external to a given rule application remain unchanged by the rule, i.e., that there is an identity mapping between these external elements. The mapping between the parts internal to a rule application is described for each rule separately.

***ReduceAbs*.** Figure 11(a) illustrates the effect of *ReduceAbs* by applying it to the model P1 in Figure 2. To simplify the verification problem, we exploit the fact that partial model refinement is transitive. Thus, if we can show that *any single* application of the rule is always a refinement, then multiple applications of the rule must be so as well.

Let P' be the result of one application of the rule to some element E of an input model P. We now apply each step of the methodology.

1) $FO(P)$ and $FO(P')$ have the same signature (with the *MAVO* predicates in the latter renamed with primes) and differ only in that the latter has an additional *MAVO* constraint $Unique_{E'}$.

2) By Proposition 1, if $bs(P')$ is well-formed then it has a concretization and so $\Phi_{P'}$ is satisfiable. Furthermore, $bs(P')$ is well-formed iff $bs(P)$ is well-formed since *ReduceAbs* does affect the base model. Thus, in the restricted case that the base model of the input model is well-formed, we can guarantee that *ReduceAbs* produces a consistent result. However, our example in Figure 11(a) is not an instance of this case – as discussed in Section II, $bs(P)$ is not well-formed – and yet both P and P' are still consistent. Furthermore, there are cases where P is consistent and applying *ReduceAbs* produces an inconsistent model P', as illustrated in Figure 10. Here, although its base model is not well-formed, the input model has concretizations (e.g., the S-annotated RemoteStart can be split over two classes) but the resulting model is not consistent. Thus, in general, *ReduceAbs* cannot be guaranteed to produce a consistent result for consistent inputs.

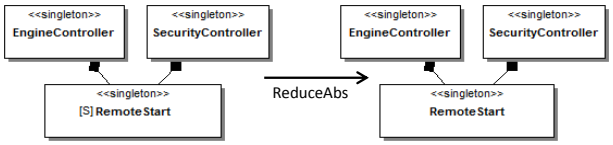


Fig. 10. An example of *ReduceAbs* producing an inconsistent model.

3) The rule maps element E to class E' as equivalence. Thus, Φ_R consists of equivalences for all elements of P and so $R(\Phi_P)$ just changes *MAVO* predicate names to their primed versions.

4) $\Phi_{P'} \Rightarrow R(\Phi_P)$ holds because

$$\Phi'_P = R(\Phi_P) \cup \{Unique_{E'}\}$$

From this analysis, we conclude that *ReduceAbs* is not a partiality-refining transformation for all input models. It is refining for inputs with well-formed base models and for some, but not all, inputs with non-well-formed base models as well. The particular criterion determining which inputs with non-well-formed are acceptable is dependent on the well-formedness rules of the language. We leave the process of constructing such a criterion for future work.

***GetSet*.** As discussed in Section II, the *MAVO* partiality mechanisms are used to express uncertainty about the syntactic structure of the model. In a development process, we expect partiality reducing refinements to be interleaved with semantically oriented detail adding refinement transformations. In this example, we observe the effect of such detail adding refinements on the partiality within a model. *GetSet* is a class diagram transformation that adds getter and setter methods to each class for each attribute. This type of transformation is common in model-driven engineering since it reduces modeler effort by automatically filling in 'boiler plate' information and enforces standards that reflect best practices.

We lift the *GetSet* class diagram transformation to partial models by interpreting the rule so that its left hand side matches only those parts of the *MAVO* class diagram that have no annotations. Figure 11(b) illustrates the application of *GetSet* to the model P1 in Figure 2. Note that our interpretation of the rule prevented its application to the attribute isEngineEnabled because of the V-annotated class EngineDisabler.

Since the purpose of *GetSet* is to add detail rather than reduce uncertainty, we want to prove that this naive transformation preserves partiality, i.e., that the input model and result model are both partiality refinements of each other. However, since *GetSet* adds information to a model, we can immediately see that it does not yield a consistent model if the input is marked as "complete" and so it is not a valid refinement in general. Thus, we do our verification analysis on models marked as "incomplete", i.e., annotated with INC . As with *ReduceAbs*, we simplify the verification problem by exploiting the transitivity of refinement and analyze a single rule application.

In applying the refinement methodology, let P' be the result of one application of the rule to some attribute A of some class C in an incomplete input model P.

1) $FO(P')$ has the signature of $FO(P')$ renamed with primes and extended with the following *MAVO* predicates: two unary predicates for the new operations `getA` and `setA`, and two binary predicates for the relation instances $\texttt{ownedOperation}(\texttt{C}', \texttt{getA})$ and $\texttt{ownedOperation}(\texttt{C}', \texttt{setA})$. $FO(P')$ has the same *MAVO* constraints as $FO(P)$ extended with *Exists*, *Unique* and *Distinct* constraints for these new predicates. Let Φ' be this set of additional constraints. Note that since we are assuming incomplete models, there is no *Complete* constraint in either P or P'.

2) If Φ_P is satisfiable then $\Phi_{P'}$ is satisfiable. We show this by exploiting the fact that *GetSet* is a lifted class diagram transformation and use it to construct a concretization of P' from a concretization of P. Since Φ_P is satisfiable, let M be a class diagram that is a concretization of P and that does not contain operations `getA` and `setA`. Now, since A and C are not annotated (by assumption), they must occur in M and so we can compute a class diagram M' that is the result of applying the rule at A and C in M. M' differs from M only in that it contains the additional operations `getA` and `setA` and relation instances $\texttt{ownedOperation}(\texttt{C}, \texttt{getA})$ and $\texttt{ownedOperation}(\texttt{C}, \texttt{setA})$. Furthermore, M' clearly satisfies the four new *MAVO* predicates and the *MAVO* constraints. Thus, we must just show that M' does not violate any of the well-formedness rules in $\Phi_{\texttt{CD}}$ for class diagrams. This follows from the fact that the only well-formedness rules that involve an `Operation` or `ownedOperation` relationship instances are the multiplicity constraints on `ownedOperation`, and these hold in M'. Therefore, we conclude that M' is a concretization of P' and so $\Phi_{P'}$ is satisfiable.

3) The rule maps A and C to A' and C', respectively, as equivalences. Thus, Φ_R consists of equivalences for all atoms of P and so $R(\Phi_P)$ just changes *MAVO* predicate names to their primed versions.

4) $\Phi_{P'} \Rightarrow R(\Phi_P)$ holds because $\Phi_{P'} = R(\Phi_P) \cup \Phi'$.

Thus, we have shown that *GetSet* is a valid partiality reducing refinement for input partial models that are incomplete. To complete the proof that it preserves partiality and P is also a refinement of P', we need to show that every concretization of P is also one of P'. Yet it is not the case. Consider the model M constructed in step (2) above. It is a concretization of P. To be a concretization of P', it needs to satisfy *MAVO* constraints $Exists_{\texttt{getA}}$ but, by assumption, it is missing the operation `getA`. Thus, M is a concretization of P but not of P'.

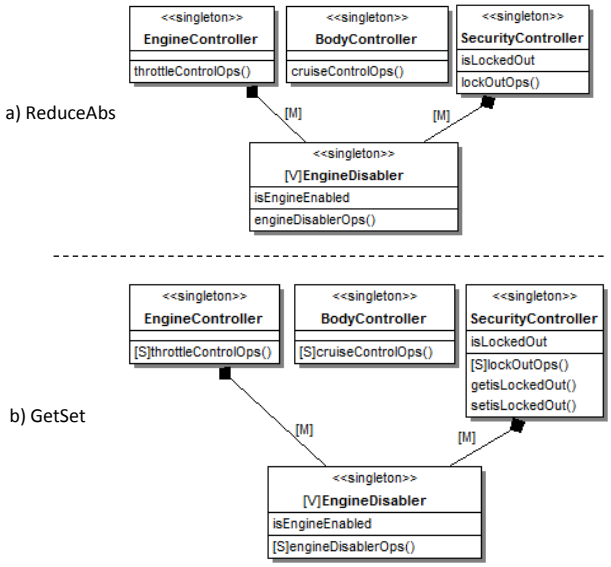


Fig. 11. The result of applying *ReduceAbs* (a) and *GetSet* (b) to the partial class diagram `P1` in Figure 2.

To summarize, we have shown that in general, *GetSet* is not a partiality reducing refinement. However, when restricted to incomplete input models, it is a partiality reducing refinement although it does not preserve partiality.

B. Discussion

The above transformation examples are simple and seem intuitively to be correct; yet our analyses showed that they were not. The analyses also revealed some of the key challenges as well as some opportunities associated with applying the manual methodology to these automatic transformations. We summarize these below.

Exploiting the transitivity of refinement. Transformations are often defined using a set of rules. In such cases, we can exploit the transitivity of refinement to decompose the verification problem by checking each rule independently. Furthermore, we can decompose the effect of a particular rule into a sequence of individual rule applications. This approach yields a sufficient but not a necessary condition for the transformation to be refining: it is possible to define transformations such that a single rule application is not a refinement but the final result, after all rule applications are made, is always a refinement due to the combined action of rules.

Exploiting locality. In the above examples, we only considered the verification of individual rule applications (at an arbitrary site in the input model). Furthermore, we tried to make our analysis as local as possible around the rule application site. The ideal situation for analysis is when the rule has the following property: if the rule itself is a correct refinement (i.e., if we can prove that its right hand side of the

rule is a valid refinement of its left hand side), then any rule application to a model produces a correct refinement of the model. Unfortunately, this is seldom the case, even for simple rules. For example, *ReduceAbs* does not have this property – the rule is a valid refinement but, as illustrated in Figure 10, not all of its applications are. However, if we restrict the locality property to just the implication proof obligation (step (4) in the methodology), the situation improves. The restricted locality property is: if the implication holds in the rule then it holds between models when the rule is applied. This is the case for both *ReduceAbs* and *GetSet* because the effect of the rule application is to increase the set of *MAVO* sentences, and hence, it must be more constraining. One investigation we leave for future work is to determine whether this restricted locality property holds for any *MAVO* refining transformation rule.

Addressing satisfiability. From the examples it appears that the satisfiability proof obligation (step (2)) seems like the more difficult one to ensure. This is because, unlike the implication proof obligation (step (4)), it depends on the well-formedness rules of the modeling language (in this case, class diagrams) and these can be arbitrarily complex. However, we have observed some special cases that simplify the problem:

- Showing that $\Phi_{P'}$ is satisfiable only requires us to find a single concretization of P' for each consistent input model P. The analysis of *GetSet* follows this strategy by showing how to construct a concretization of P' from any concretization of P for a consistent input P. This construction was possible because *GetSet* is a lifted version of a transformation of concrete models. Thus, we can use this approach to satisfiability whenever we are dealing with lifted concrete transformations.
- If we can ensure that the transformation preserves well-formedness of base models, then the result of the transformation is a consistent partial model (by Proposition 1) when the input model has a well-formed base model. This is the case for both *ReduceAbs* and *GetSet*. Thus, the satisfiability proof obligation can be met for a broad class of input models.

We intend to use the above observations in a future development of a methodology to address verification of partial model transformations.

VI. Tool Support

We developed a prototype tool designed to automate the methodology for verifying manual *MAVO* refinements as described in Figure 7. The prototype uses TXL [1] for translation of eCore models into first-order encodings, and the Alloy Analyzer [2] for generating the encodings and checking the proof obligations. The implementation of each step is accomplished as follows:

1) The first-order encodings in Alloy are generated by translating the input eCore models via a TXL program. The input eCore models consist of a partial model *P*, a candidate refinement *P'*, and a refinement mapping *R* between *P* and *P'*. Three Alloy predicates, `P`, $\texttt{P}'$ and `R`, are generated corresponding to the FO constraints Φ_P, $\Phi_{P'}$ and Φ_R, respectively.

2) The first proof obligation, satisfiability of $\Phi_{P'}$, is checked by the Alloy Analyzer by running the Alloy predicate ($\texttt{P}'$ `and` `R`). `R` must be included in this check because in the Alloy implementation both models and the mapping are encoded as part of the same theory. If Alloy finds an instance of the predicate, then the proof obligation is met. If no instance is found, all we can say with certainty is that the proof obligation fails within the given scope.

3) The Alloy code of the mapping is generated at the same time as the code for the models in step (1).

4) The second proof obligation, that $\Phi_{P'} \Rightarrow R(\Phi_P)$, is checked by running the predicate (($\texttt{P}'$ `and` `R`) `and not` `P`). If Alloy is able to find an instance for this predicate then the proof obligation fails. If no instance is found then all we can say with certainty is that the proof obligation is met within the given scope.

Note that an Alloy experiment does not constitute a proof that one model is a refinement of another – both steps (2) and (4) include an analysis outcome "only sure up to a given scope". Thus, selecting the scope becomes an important task. The default scope set by the tool is equal to the total number of distinct signatures of the translated model. While many models will not require a scope this large, some will. For example, in a model with element annotated with a `V` partiality, a refinement of the model may merge this element with every other element of the same type. Thus this refinement would require as many instances of this element as there are total elements in the model.

The model `P1` in Figure 2 has 16 elements that must be encoded in Alloy – each class, operation, and attribute is encoded as an Alloy signature, and each association requires three signatures: one for the association and one for each endpoint.

We used Alloy to show that `P2` in Figure 2 is a valid refinement of `P1`. We also used Alloy to prove that applying the transformations *ReduceAbs* and *GetSet* to `P1`, as depicted in Figure 11, also yield valid refinements. We ran our experiments on a laptop with an Intel Core i7 processor and 8 GB of RAM using Alloy 4.2. The run times of the various experiments with the various scopes are summarized in Table I. The first column indicates the proof obligation tested with the scope indicated in the second column. The third column is the total time required by Alloy to generate and solve the CNF formula for the predicate using SAT4J. The largest scope size that could be checked before the solver ran out of memory in our experiments was about 32.

Although our experiments with the prototype are preliminary, the results indicate that the proof obligations for these small models can be checked in a matter of seconds for a moderately-sized scope. As the size of the scope increases, the time increases exponentially, as expected.

Our next step is to do in-depth scalability testing varying both model sizes and the degree of partiality in the models (since this affects the number of concretizations). In addition,

Predicate Checked	Scope	Time (ms)
Figure 11(a), proof obl. 1	16	13 542
Figure 11(a), proof obl. 2	16	4 695
Figure 2, proof obl. 1	16	5 171
Figure 2, proof obl. 2	16	2 817
Figure 11(b), proof obl. 1	16	5 611
Figure 11(b), proof obl. 2	16	3 824
Figure 11(b), proof obl. 2	25	71 698
Figure 11(b), proof obl. 2	30	294 302

TABLE I
THE REFINEMENT EXPERIMENTS.

although the current prototype is focused on verifying manual refinements, we are interested in adapting it to help with the verification of refining transformations.

VII. Related Work

In this section, we briefly discuss other work related to the verification of partiality refinement.

Partial *behavioural* modeling formalisms such as Modal Transition Systems (MTSs) [4] allow introduction of uncertainty about transitions on a given event, whereas Disjunctive Modal Transition Systems (DMTSs) [5] add a constraint that the refinement should include at least one of the possible transitions. Concretizations of these models are Labelled Transition Systems (LTSs). MTSs and DMTSs can be thought of as the application of the M annotation in our framework to LTS models. The MTS and DMTS refinement mechanism allows resulting LTS models to have an arbitrary number of states. This is different from the treatment provided in this paper, where we concentrated only on syntactic rather than semantic partiality and thus state duplication was not applicable.

Another approach to partiality is the work of Herrmann [3] which studied the value of being able to express *vagueness* within design models. His modeling language SeeMe has notational mechanisms similar to the INC and M annotations; however, there is no formal foundation for these mechanisms and no well-defined notion of refinement.

VIII. Conclusion and Future Work

Partial modeling is an approach for expressing a modeler's uncertainty about a model. Partial model refinement expresses the reduction of uncertainty as model development proceeds. Such refinements can happen manually, or as is often the case with model-driven development, using refining model transformations. In previous work, we have defined various language-independent partial modeling mechanisms, together called *MAVO*, as model annotations with formal semantics [7]. In this paper, we considered the issue of formally verifying *MAVO* partial model refinements and made several advances in this area. Specifically, we defined a methodology for formally verifying manual *MAVO* refinements. We then applied that methodology to examples of refining transformations and observed some patterns. We intend to use them in the future as we tailor the manual methodology to verifying refining transformations. Finally, we implemented a prototype Alloy-based tool to help automate the methodology and reported on the results of applying it to the examples in the paper.

As discussed above, the primary follow-on work is development of a methodology for verifying refining partial model tranformations, but other directions of investigation seem fruitful as well. In this paper, we focused on studying partiality reducing transformations; however, there are many other types of transformations that could be applied to partial models. In particular, since any modeling language can be extended to its *MAVO* partial model version, it is natural to consider how to "lift" transformations of the modeling language to transformations between the *MAVO* versions of it. We illustrated this with *GetSet* example. Our method of lifting was naive because it didn't attempt to take into account the intent of the transformation and use it to make the partial version meaningful. For example, if an attribute is M-annotated, indicating that it may or may not be present, we should M-annotate the new getter and setter operations and also add a constraint ensuring that their existence depends on the existence of the attribute; thus, refinements of partiality on the attribute force corresponding refinements on the partiality of the getter and setter operations. We are currently developing a theory of how to lift such transformations in a sound and complete way.

Finally, our use of FOL as the means to formalize metamodels and partial models gives our work a strong algebraic specification flavor and allows us to adapt existing work in this area to partial models. In this paper, we have explored the most basic aspect of this: defining partial model refinement as a special kind of specification refinement [8]. In the future, we hope to adapt other aspects of algebraic specification to partial models, e.g., specification composition and structured specifications, parameterized specifications, transformations between different specification languages, etc.

References

[1] J. Cordy. The TXL Source Transformation Language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[2] D. Jackson. *Alloy Analyzer Website*, 2012.

[3] T. Herrmann. *Handbook of Research on Socio-Technical Design and Social Networking Systems*, chapter Systems Design with the Socio-Technical Walkthrough, pages 336–351. 2009.

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

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

[6] T. Maibaum. Conservative Extensions, Interpretations between Theories and All That! In *Proc. of TAPSOFT'97*, pages 40–66. Springer, 1997.

[7] R. Salay, M. Famelis, and M. Chechik. Language Independent Refinement Using Partial Modeling. In *Proc. of FASE'12*, March 2012. To appear.

[8] D. Sannella and A. Tarlecki. Essential Concepts of Algebraic Specification and Program Development. *Formal Asp. Comput.*, 9(3):229–269, 1997.