Verification of Uncertainty Reducing Model Transformations

Rick Salay, Marsha Chechik, Michalis Famelis, Jan Gorzny

Department of Computer Science, University of Toronto, Toronto, Canada
{rsalay,chechik,famelis,jgorzny}@cs.toronto.edu

Abstract. Models are typically used for expressing information that is known at a particular stage in the software development process. Yet, it is also important to express what information a modeler is still uncertain about. Furthermore, when a transformation is applied to a model containing uncertainty, it is natural to consider the effect that the transformation has on the level of uncertainty, e.g., whether it always reduces it. In previous work, we have presented a general approach for precisely expressing uncertainty within models and defined formal conditions for uncertainty reduction between models. In this paper, we use these foundations to develop an automated method for verifying that a partial model transformation is uncertainty reducing.

1 Introduction

Software modelers often face the challenging task of working in the presence of uncertainty. Uncertainty in models comes from many sources, including but not limited to incomplete requirements [8], presence of alternative design decisions [31], disagreements among stakeholders [24], etc. Yet existing modeling methodologies, languages and tools rarely provide adequate support for explicating uncertainty. 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 [27]. We call the resulting models partial.

In [10], we argued that ubiquity of uncertainty should make partial models first-class artifacts in MDE, allowing these models to be manipulated “as usual”: transformed, checked for correctness, etc. In [12], we started looking at how to lift classical transformations to apply to partial models, and in [11], we conducted experiments using partial models for property checking.

In this paper, we are concerned with transformations that take the form of graph rewriting rules [9]. Such a rule is applied by finding a match of its left hand side (LHS) in the source model and replacing it with the fragment on its right hand side (RHS). A transformation corresponds to repeated applications of the rule until it can no longer be applied.

Once points of uncertainty are explicated, uncertainty resolution is reflected in the construction of a partial model refinement of the model. Yet, natural language expression of uncertainty makes it impossible to accurately (or automatically) verify the correctness of refinement.
Our formalization [27] allows us to check whether a particular rule-based transformation is a partial model refinement, i.e., that it reduces the level of uncertainty. We distinguish between two types of questions: given a rule, (1) determine whether a particular instance of applying it to a given model is correct, i.e., that it reduces uncertainty – we call this verifying correctness of refinement application, and (2) determine whether the rule, when applied to all models, is correct – we call this verifying correctness of a refining transformation.

**Motivating example.** The partial modeling approach can be applied to any modeling language. In this example we use a simplified version of UML class diagrams. Assume a scenario, depicted in Figure 1, where a modeler is facing uncertainty regarding a fragment of the class diagram in a hypothetical HVAC (Heating, Ventilation, A/C) controller for a building. The HeatingController has operations to regulate the building’s temperature and ConservationManager has operations to monitor consumption and conserving energy. A separate class SafetyController interfaces with the Security subsystem of the building, and so has operations to detect HVAC-specific malicious intrusions. The modeler also knows that there should exist operations for disabling the gas supply for the building (e.g., in case of a fire or a leak, etc) but is not sure where they belong or whether they should be in a separate class, etc.

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.

Model P1 in Figure 1 shows the use of partiality annotations, introduced in [27], 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 $\text{ecoControlOps}$ (in class ConservationManager) means that it represents a (as yet unknown) set of operations. This captures the same information as in the note attached to ConservationManager in Figure 1 – i.e., that it contains operations for energy conservation but it is still unknown what they are. The $v$ annotation on the GasDisabler 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 gasDisablerOps operations. Furthermore, the $m$-annotated composition associations say that if GasDisabler is assigned to a new class then it may have
a composition relationship either with the HeatingController class or with the SafetyController class. Yet it cannot have this relationship with both classes simultaneously since the well-formedness rules for class diagrams prohibit this.

As more information becomes available, the modeler can resolve some of these uncertainties by constructing a partial model refinement of the original model. For example, Figure 1 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: (1) the s-annotated operation ecoControlOps() is refined to a set of particular operations \{ecoOff(), setEcoLevel()\}, (2) a decision is made to put the functionality to disable the gas supply into the HeatingController class by assigning the v-annotated class GasDisabler to it, and (3) the m-composition relations are eliminated.

Figure 1 shows a refinement application, i.e., a refinement applied to a particular model. In contrast, Figure 3 gives a refining partial model transformation that can be used to generate a refinement application when applied to an arbitrary model. Syntactically, ReduceAbs removes all occurrences of s annotations on elements. Semantically, it means that these now represent particular elements rather than an arbitrary set of elements. Intuitively, this transformation reduces uncertainty about these elements and thus is an uncertainty-reducing transformation. But can we prove it?

Contributions of this paper. In this paper, we look at the problem of checking correctness of uncertainty-reducing transformations of partial models, such as ReduceAbs. In previous work [25], we have presented a general approach for precisely expressing uncertainty within models and defined formal conditions for uncertainty reduction between models. We then showed how to prove that a particular application of a rule is uncertainty-reducing, i.e., that applying a given rule to a given model results in a refinement of this model.

In this paper, we use these foundations to develop an automated method for showing that a partial model transformation is uncertainty reducing, i.e., given a set of rules, applying it to any model yields an uncertainty-reducing refinement. Specifically, we make the following contributions: (1) we develop and illustrate an automated method for verifying partial model refinement transformations;
Fig. 3. A rule defining transformation Reduce Abs.

(2) we describe prototype tool support to help automate the verification method and to use the generated counterexamples in order to repair faulty rules; and
(3) we apply the method to the verification of three specific transformations. Our tool support, all models and proof of Proposition 1 are available at http://www.cs.toronto.edu/~jgorzny.

The rest of the paper is organized as follows. In Section 2, we review the concept of model partiality as introduced in [27,25]. In Section 3, we define an automated method for verifying uncertainty-reducing transformations, which consists of checking two properties. Sections 4 and 5 describe the theoretical foundations and the tool support for automating checking each of these properties. In Section 7, we apply the method on several example transformations. In Section 8, we discuss related work. In Section 9, we summarize the paper and discuss potential future research directions.

2 Background

In this section, we briefly review the concepts of language-independent partial modeling introduced in [27].

MAVO partial models. 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 $P_1$ in Figure 1. Note that the base model does not necessarily need to be well-formed. In fact, the example in the figure violates the well-formedness rule that a singleton class cannot be composed into two different classes. This shows that expressing some cases of uncertainty requires non-well-formed base models. Figure 4(b) shows one of the concretizations of $P_1$. $P_1$ has an infinite number of concretizations since each of the $s$-annotated operations can be replaced by any set of particular operations. Thus, although $bs(P_1)$ is not well-formed, $P_1$ 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
∀𝑐:𝐶𝑙𝑎𝑠𝑠· ∃𝑐1,𝑐2:𝐶𝑙𝑎𝑠𝑠· 𝑖𝑠𝑆𝑖𝑛𝑔𝑙𝑒𝑡𝑜𝑛 𝑐 ∧ 𝑐𝑜𝑚𝑝𝑜𝑠𝑒𝑑𝐼𝑛 𝑐,𝑐1 ∧ 𝑐𝑜𝑚𝑝𝑜𝑠𝑒𝑑𝐼𝑛 𝑐,𝑐2 ⇒ 𝑐1 = 𝑐2

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

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

that we are unsure about whether it should exist in the model while ∈ (default) means that it does exist; annotating an element with $s$ indicates that we are unsure about whether it should actually be a collection of elements while $p$ (default) means that it is just one element; annotating an element with $v$ indicates that we are unsure about whether it should actually be merged with other elements while $c$ (default) means that it is distinct; finally, annotating the entire model with $inc$ indicates that we are unsure about whether it is complete while $comp$ (default) means that it is complete. When these four types of partiality annotations are used together, we refer to it as MAVO partiality.

**Formalizing MAVO partiality.** 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 $\Sigma$ is the signature with sorts and predicates representing the element types, and $\Phi$ is a set of sentences representing the well-formedness constraints. The models that conform to $T$ are the finite FO $\Sigma$-structures that satisfy $\Phi$ according to the usual FO satisfaction relation. We denote the set of models with metamodel $T$ by $\text{Mod}(T)$.

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

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 $\text{FO}(P)$ s.t. $\text{Mod}(\text{FO}(P)) = \{P\}$. We proceed as follows.

(1) Let $M = \text{bs}(P)$ be the base model of a partial model $P$. We define a new
\( \Sigma_{M} \) has unary predicates \( CM(Class), ECOps(Operation)) \), ..., and binary predicates \( CMownsECOps(Class, Operation) \), ... \( \Phi_{M} \) contains the following sentences:

\[
(\text{Complete}) \ (\forall x : Class \cdot CM(x) \lor HC(x) \lor SC(x) \lor GD(x)) \land \\
(\forall x : Class, y : Operation \cdot ownedOperation(x, y) \implies (CMownsECOps(x, y) \lor ...)) \land ...
\]

\( CM \)

\[
(\exists x : Class \cdot CM(x)) \\
(\text{Unique}_{CM}) \ (\forall x, x' : Class \cdot CM(x) \land CM(x') \implies x = x') \\
(\text{Distinct}_{CM, SC}) \ (\forall x : Class \cdot CM(x) \implies -HC(x)) \\
(\text{Distinct}_{CM, GD}) \ (\forall x : Class \cdot CM(x) \implies -GD(x))
\]

similarly for all other element and relation predicates

Fig. 6. The FO encoding of \( P_{M} \).

partial model \( P_{M} \) which has \( M \) as its base model and its sole concretization, i.e., \( bs(P_{M}) = M \) and \( |P_{M}| = \{ M \} \). We call \( P_{M} \) the ground model of \( P \).

(2) To construct the FOL encoding of \( P_{M} \), \( FO(P_{M}) \), 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. (3) We construct \( FO(P) \) from \( FO(P_{M}) \) by removing constraints corresponding to the annotations in \( P \). This constraint relaxation allows more concretizations and so represents increasing uncertainty. For example, if an atom \( a \) in \( P \) is annotated with \( M \) then the constraint that enforces occurrence of \( a \) in every concretization is removed. We illustrate the above construction using the partial class diagram \( P_{1} \) in Figure 1.

(1) Let \( M_{1} = bs(P_{1}) \) be its base model and \( P_{M_{1}} \) be the corresponding ground partial model.

(2) We have: \( FO(P_{M_{1}}) = \langle \Sigma_{CD} \cup \Sigma_{M_{1}}, \Phi_{CD} \cup \Phi_{M_{1}} \rangle \) (see Definition 2), where \( \Sigma_{M_{1}} \) and \( \Phi_{M_{1}} \) are model \( M_{1} \)-specific predicates and constraints, defined in Figure 6. They extend the signature and constraints for CD models described in Figure 5. For conciseness, we abbreviate element names in Figure 6, e.g., \( \text{ConservationManager} \) becomes \( CM \), etc. We refer to \( \Sigma_{M_{1}} \) and \( \Phi_{M_{1}} \) as the MAVO predicates and constraints, respectively.

Since \( FO(P_{M_{1}}) \) extends \( CD \), the FO structures that satisfy \( FO(P_{M_{1}}) \) are the class diagrams that satisfy the constraint set \( \Phi_{M_{1}} \) in Figure 6. Assume \( N \) is such a class diagram. The MAVO constraint \( \text{Complete} \) ensures that \( N \) contains no more elements or relation instances than \( M_{1} \). Now consider the class \( CM \) in \( M_{1} \). \( \text{Exists}_{CM} \) says that \( N \) contains at least one class called \( CM \), \( \text{Unique}_{CM} \) - that it contains no more than one class called \( CM \), and the clauses \( \text{Distinct}_{CM, SC} \) - that the class called \( CM \) is different from all the other classes. Similar MAVO constraints are given for all other elements and relation instances in \( M_{1} \). These constraints ensure that \( FO(P_{M_{1}}) \) has exactly one concretization and thus \( N = M_{1} \).
(3) Relaxing the MAVO constraints \( \Phi_{M1} \) allows additional concretizations and represents a type of uncertainty indicated by a partiality annotation. For example, if we use the \( \text{INC} \) annotation to indicate that \( M1 \) is incomplete, we can express this by removing the Complete clause from \( \Phi_{M1} \) and thereby allow concretizations to be class diagrams that extend \( M1 \). Similarly, expressing the effect of the \( \text{m}, \text{s} \) and \( \text{v} \) annotations for an element \( E \) correspond to relaxing \( \Phi_{M1} \) by removing Exists \( E \), Unique \( E \) and Distinct \( E \) clauses, respectively. For example, removing the Distinct \( GD \) clauses is equivalent to marking the class \( GD \) with \( \text{v} \) (i.e., GasDisabler may or may not be distinct from another class).

Partial model refinement. Intuitively, refinement of a model should not increase the set of concretizations it has (a proper refinement reduces this), while making sure at least one concretization remains. In what follows, assume that there exists a refinement mapping \( R \) which maps the atoms of the two models.

Definition 3 (MAVO Mapping) Given MAVO models \( P \) and \( P' \), based on the same metamodel, a MAVO mapping \( R(P, P') \) is a relation \( R \subseteq \text{atoms}(P) \times \text{atoms}(P') \), where \( \text{atoms}(P) \) and \( \text{atoms}(P') \) are the sets of atoms in \( P \) and \( P' \), respectively, and for all \( \langle a, a' \rangle \in R \), \( a' \) and \( a \) have the same type in the metamodel.

We now define a notion of a simple extension of a MAVO mapping:

Definition 4 (Simple extension) A MAVO mapping \( R_1(P_1, P'_1) \) is a simple extension of a mapping \( R(P, P') \) iff \( R_1(P_1, P'_1) \) is constructed by adding the same set of annotated atoms to both \( P \) and \( P' \) to form \( P_1 \) and \( P'_1 \), respectively, and adding the corresponding identity mappings to \( R \) to form \( R_1 \).

We now review, following [25], how the FO encoding is used to formalize the above intuition by giving two conditions for correctness of MAVO refinement, i.e., conditions under which it reduces uncertainty.

Definition 5 (MAVO Refinement) Let \( R(P, P') \) be a MAVO mapping where we have encodings \( \text{FO}(P') = (\Sigma_{P'}, \Phi_{P'}) \) and \( \text{FO}(P) = (\Sigma_{P}, \Phi_{P}) \). \( P' \) refines \( P \) with mapping \( R \) iff the following conditions hold:

(Ref1) \( \Phi_{P'} \) is satisfiable
(Ref2) \( \Phi_{P'} \Rightarrow R(\Phi_{P}) \)

\( R \) is then called a refinement mapping.

Condition Ref1 ensures that \( P' \) has at least one concretization (see Definition 1) and condition Ref2 ensures that \( P \) has all concretizations of \( P' \). Here, \( R(\Phi_{P}) \) denotes a translation (described in [25]) of the MAVO sentences of \( P \) according to the mapping \( R \).

3 Verifying Uncertainty Reducing MAVO Transformations

Definition 5 in Section 2 defined conditions for verifying a single application of partial model refinement. In this section, we present a method for verifying that
every input/output pair of a partial model transformation is a valid refinement application (i.e., satisfies Definition 5). We call such a transformation refining.

We assume that the partial models are specified using MAVO annotations described in Section 2 and express the conditions for a refining MAVO transformation as follows.

**Definition 6 (Refining MAVO Transformation)** A MAVO transformation $F : Mod(MAVO(T)) \rightarrow Mod(MAVO(T))$ is a refining transformation iff the following conditions hold for all $P \in Mod(MAVO(T))$ where $FO(P) = \langle \Sigma_P, \Phi_P \rangle$, $FO(F(P)) = \langle \Sigma_{F(P)}, \Phi_{F(P)} \rangle$ and $R_{F(P)}$ is the mapping from $P$ to $F(P)$ induced by $F$:

(TRef1) $\Phi_P$ is satisfiable $\Rightarrow \Phi_{F(P)}$ is satisfiable

(TRef2) $\Phi_{F(P)} \Rightarrow R_{F(P)}(\Phi_P)$

These conditions mirror the conditions in Definition 5. Thus, the objective of the method we describe below is to determine whether a given MAVO transformation $F$ is a refining transformation. Definition 5 for MAVO refinement assumes that both $P$ and $P'$ are over the same metamodel and we use the same restriction for the MAVO transformations we consider.

We further assume that a candidate refining transformation $F$ is implemented as a set $\{\rho_1, ..., \rho_n\}$ of confluent and terminating refinement rewrite rules, defined as follows.

**Definition 7 (MAVO refinement rewrite rule)** A refinement rewrite rule $\rho$ on a MAVO($T$) model is a MAVO mapping $R_{\rho}(LHS, RHS)$ s.t. $LHS$ and $RHS$ are MAVO($T$) models. Rule $\rho$ is applied to a MAVO($T$) model $P$ by finding an occurrence of $LHS$ in $P$ and replacing it with $RHS$ to produce $P'$. The resulting refinement mapping between $P'$ and $P$ consists of $R_{\rho}$ at the site of the rule application and the identity mapping everywhere else.

Default annotations are not used in a rewrite rule and so even ground annotations must be specified explicitly (i.e., e, p, c and comp). In addition, we use the placeholder “?” when the annotation is not given. For example, in the rule for ReduceAbs (Figure 3), the element $E$ on the $LHS$ can match an element with any annotation as long as it includes $s$. Furthermore, if “?” is used in the same position on the $RHS$ then it must represent the same value as the instantiation on the $LHS$. For example, if the element $E$ on the $LHS$ matches an element annotated with $\langle m, s, c \rangle$, then the $RHS$ would be instantiated as $\langle m, p, c \rangle$ for element $E$.

In order to verify that a MAVO transformation is refining, we must prove that properties TRef1 and TRef2 in Definition 6 hold. To simplify this process, we note that the rules in our transformations are confluent (and thus, the order of their application does not matter) and that both TRef1 and TRef2 are transitive properties. Thus, to verify each property it is sufficient just to check it on a single

---

1 If transformations $F$ and $F'$ satisfy a transitive property then so does the composition $F \circ F'$. 
rule application for each rewrite rule of the transformation. The above process is not necessary: even if verification of a particular rule application verification fails, the combined action of multiple ones rule may still be a refinement.

The applications of a rule $\rho = \rho(LHS, RHS)$ are exactly the simple extensions of it as specified in Definition 4. Thus, for example, to check TRef1 for $\rho$ we must check that every simple extension $R(P_{LHS}, P_{RHS})$ of $\rho$ satisfies Ref1.

We summarize the verification method as follows: Given a MAVO transformation $F$ implemented as a set \{\rho_1, ..., \rho_n\} of MAVO refinement rewrite rules, for each rule $\rho_i \in \{\rho_1, ..., \rho_n\}$, we must check that it satisfies TRef1 and TRef2. If these conditions hold for all rules $\rho_i$ then $F$ is a refining transformation. In the next two sections we present the method for checking TRef1 and TRef2 on a rule $\rho_i$ by checking every simple extension $R(P_{LHS}, P_{RHS})$ of $\rho_i$.

# 4 Checking Property TRef1

To prove that TRef1 holds for a rule $\rho$ requires showing that for each simple extension $R(P_{LHS}, P_{RHS})$ of $\rho$, if $FO(P_{LHS})$ is satisfiable then $FO(P_{RHS})$ is satisfiable as well. The proof of this property is dependent both on the metamodel constraints and the MAVO constraints. Our method requires the use of tool support for this step.

Specifically, we have developed tooling that, given $\rho$, produces an Alloy module that checks TRef1 in a bounded way by exploring all simple extensions $R(P_{LHS}, P_{RHS})$ of $\rho$ up to a given scope. If Alloy finds that TRef1 does not hold for $\rho$, the counterexample it produces provides a way to “repair” the rule by adding a guard (e.g., a negative application condition [13]) that will prevent it from being applied in the bad cases. If Alloy reaches the scope without finding a counterexample, we cannot conclude that TRef1 holds. Thus this approach can only be used to provide evidence that TRef1 holds.

Our tool accepts a rule expressed in Ecore [30] as its input. The rule is then translated, using TXL [6], into an Alloy [18] encoding, which includes all of the rule’s MAVO annotations, and is combined with our encoding of the Ecore metamodel. We encoded further Alloy predicates that allow us to create arbitrary simple extensions (see Definition 4) for the LHS and the RHS of the rule, as described in Section 3. The description corresponding to the extensions of the LHS of the rule was encoded using Alloy’s facts, whereas that for the RHS - with Alloy’s predicates. This allowed us to only take into account well-formed LHS extensions, and to create instances of RHS extensions that are not well-formed, using Alloy’s assertions. Figure 7 shows an example snippet that demonstrates our Alloy encoding for rule Reduce.Abs.

Running the generated Alloy encoding enumerates all concretizations of the RHS MAVO model (i.e., the RHS of the MAVO rule and its simple extensions) that are not well-formed, up to a given scope.

This process uses concretizations rather than MAVO models but produces the same result: assume that Alloy produces a concretization that is not well formed. That means that there exists a ground MAVO simple extension (i.e.,
containing only elements annotated with \( p, c, e \) of the \( RHS \) of the \( MAVO \) rule that has at least one concretization (the one that was found) that is not well-formed. Therefore, there exists an input \( MAVO \) model for which the rule does not preserve consistency, and Alloy’s enumeration process finds it.

Moreover, the above procedure does not miss any counterexamples up to the given scope. To ensure this, we constructed the \( MAVO \) simple extensions of the rule with all elements having the “least restrictive” \( MAVO \) annotations (s, v, m). In enumerating all concretizations of this \( MAVO \) model, Alloy will end up enumerating the concretizations of all other (more restrictive) \( MAVO \) simple extensions of the rule for the given scope. In other words, the set of concretizations of any \( MAVO \) model (up to the given scope) is some subset of the set of concretizations of the “least restricted” \( MAVO \) model, and therefore no counterexamples are overlooked.

5 Checking Property \( TRef2 \)

Property \( Ref2 \) in Definition 5 has a desirable “locality” characteristic given by the following proposition.

**Proposition 1** Given a refinement rewrite rule \( \rho = R_\rho(LHS,RHS) \), if \( \rho \) satisfies \( Ref2 \) then every simple extension \( R(P_{LHS},P_{RHS}) \) of \( \rho \) satisfies \( Ref2 \).

Thus, we can dramatically simplify the effort of proving condition \( TRef2 \) by simply checking \( Ref2 \) on the \( LHS \) and \( RHS \) of the rule itself. Furthermore, although \( Ref2 \) is general enough that it can be used with \( MAVO \) models that are augmented with arbitrary FO constraints for expressing detailed cases of uncertainty, when we limit ourselves to just using \( MAVO \) annotations, we can simplify checking the refinement condition \( Ref2 \) by defining syntactic constraints (i.e., sufficient conditions) on the annotations.
Figure 8 summarizes these constraints, first introduced in [26], which we refer to as MAVO syntactic refinement conditions. Each of the five columns indicates a different case (case number is on the top) in the refinement mapping, and the sentences in the lower part of each case give the constraints on the MAVO annotations for the atoms of that case. A valid refinement must satisfy all of these constraints. The sentences make reference to the full set of MAVO annotations (M/E; S/P; V/C; INC/COMP), including those assumed by default when the annotation for a partiality type is omitted specified by default by omitting the annotation for a partiality type), each non-ground/ground pair.

Case (0) says that if $P$ is complete then $P'$ must be as well. In case (1), when an atom $a$ of model $P$ is refined to a set of atoms $a_1, \ldots, a_n$ of $P'$, the first sentence says that if $a$ is annotated with $E$ (i.e., it is not $M$), then at least one of the atoms $a_i$ must also be annotated with $E$. Thus, if $a$ exists and it is refined to the set of $a_i$s then at least one of these should exist. The second sentence says that if $a$ is a particular (i.e., not a set) then there can only be one $a_i$ and it too must be a particular. The third sentence says that if $a$ is a constant and thus it can't merge with any other atom then neither can any of the $a_i$s it refines to and so they too must be constants. Case (2) says that if $a$ is not propagated in to the new model, then it must have been annotated with $M$. Case (3) states that if multiple $a_i$s in $P$ are mapped into a single $a'$ in $P'$, then if any of the $a_i$s had definite information, or were particular, or were a constant, then so is $a'$. The last sentence in case (3) says that at most one of the $a_i$s could be a variable. Finally, if a new atom, not mapped to anything in $P$, appears in $P'$ (case (4)), then $P'$ could not be complete. For example, using this method it is clear that the refinement in Figure 1 is correct.

For a given refinement rewrite rule, the condition TRef2 can be easily checked with existing tools. In particular, checking TRef2 involves (a) expressing the constraints shown in Figure 8 as OCL constraints over the Ecore representation of
the rule, and (b) using an off-the-shelf OCL constraint checker, such as DresdenOCL [17].

5.1 Proof of Proposition 1

We prove this by using the syntactic refinement conditions as the definition for Ref2.

Proof. The proof is by induction on the number of atoms in the simple extension $R(P_{\text{LHS}}, P_{\text{RHS}})$ of $\rho$. The base case is $\rho$ itself and it satisfies the syntactic refinement conditions by assumption. For the inductive step, we show that if the simple extension $R(P_{\text{LHS}}, P_{\text{RHS}})$ of $\rho$ satisfies the syntactic refinement conditions then so does the simple extension $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ that is minimally larger. We construct $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ by choosing an atom $\alpha$, $\alpha \notin P_{\text{LHS}}$, $\alpha \notin P_{\text{RHS}}$ and define $P_{\text{LHS}}^\# = P_{\text{LHS}} \cup \{\alpha\}$, $P_{\text{RHS}}^\# = P_{\text{RHS}} \cup \{\alpha\}$ and $R^\# = R \cup \{(\alpha, \alpha)\}$. $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ is the unique (up to isomorphism) simple extension of $R(P_{\text{LHS}}, P_{\text{RHS}})$ with the least additional atoms. Although the atom $\alpha$ can have any annotation, we will initially consider the case that it is annotated with $\text{epc}$ in both $P_{\text{LHS}}^\#$ and $P_{\text{RHS}}^\#$.

To check whether $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ satisfies the syntactic refinement conditions, first note that since case (0) is not dependent on atoms, it must be satisfied by $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ since, by assumption, $R(P_{\text{LHS}}, P_{\text{RHS}})$ satisfies it. Next we must check the constraints in cases (1) and (2) for each atom $a$ in $P_{\text{LHS}}^\#$. First consider the atom $a = \alpha$ in $P_{\text{LHS}}^\#$. It is mapped to a single atom $\alpha$ in $P_{\text{RHS}}^\#$ and so only case (1) applies and all the constraints are clearly met. Every other atom $a \neq \alpha$ in $P_{\text{LHS}}^\#$ is also in $P_{\text{LHS}}$ and so, by the inductive assumption and the fact that $\alpha$ is not mapped to any of these, we can conclude that cases (1) and (2) are satisfied for these. We can argue similarly, for the atoms of $P_{\text{RHS}}^\#$ and show that all the constraints for cases (3) and (4) are met.

Therefore, when $\alpha$ is annotated with $\text{epc}$, $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ satisfies all the syntactic refinement conditions. Now, if any of these annotations are weakened the result is that fewer syntactic refinement conditions are applicable but this does not change the fact that they are all satisfied. For example, if $\alpha$ in $P_{\text{LHS}}^\#$ is annotated with $\text{mpc}$, then it must have the same annotation in $P_{\text{RHS}}^\#$ (by definition of a simple extension), and the first constraint in cases (1) and (3) no longer applies.

Therefore, for any annotation on $\alpha$, $R^\#(P_{\text{LHS}}^\#, P_{\text{RHS}}^\#)$ satisfies all the syntactic refinement constraints. \hfill \Box

6 Applying the Verification Method

We now illustrate the verification method on three transformations of MAVO partial models defined by rewrite rules. In all cases, the transformation is obtained by applying the corresponding rule(s) repeatedly until it can no longer
be applied. The result of this process is unique regardless of the order of rule applications due to the assumption of confluence and is guaranteed to terminate due to the assumption of termination. We show how results of the analysis can either give evidence of correctness of each transformation or help repair it.

**Example Transformation Rules.** The first example is the language-independent transformation *ReduceAbs* discussed in Section 1 with the rule shown in Figure 3. The second is *GetSet* with the rule shown in Figure 10(a). *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. *ReduceAbs* and *GetSet* are toy transformations and are considered here because they have been analyzed manually in [25], whereas here we show how our method can automate it.

The third example is *CompReduce*, shown in Figure 10(b). It consists of four rules. This transformation has pragmatic utility: there are cases in MAVO models when a refinement can be *implied* by the interaction between annotations and well-formedness rules, and *CompReduce* constructs these implied refinements for instances of the *composedIn* association. Rule (R1) encodes the fact that if an instance of the *composedIn* relation exists (i.e., is \(e\)-annotated) between two classes then the classes must exist as well, since an association cannot exist without its endpoints. Rule (R2) is due to the the well-formedness constraint in Figure 5 that forbids a singleton class from being composed in two classes simultaneously. The rule says that the \(s\)-annotated class \(C_2\) on the LHS can be split since it has two \(ec\)-annotated *composedIn* associations and thus any concretization of \(C_2\) must have at least two classes. Rule (R3) says that the *composedIn* association between two \(p\)-annotated classes can only be particular (i.e., there cannot be a set of them) and so it should be \(p\)-annotated as well. Finally, rule (R4) is similar to (R3) but for \(c\)-annotated classes.

**Verifying the transformations.** Table 1 shows the results of applying the method to the six rules of the three example transformations. The experiments were run on a laptop with an Intel Core i7 processor and 8 GB of RAM using Alloy 4.2. For each rule, we report the results of checking TRef1 and TRef2. For rules that fail TRef1, we record the minimum scope for a failure. For those that pass it, we record the maximum scope that could be checked before the solver ran out of memory. We also report the overall time required for Alloy to create the first counterexample, if any.

![Fig. 9. An example of *ReduceAbs* producing an inconsistent model.](image-url)
The rules defining transformations `GetSet` and `CompReduce`.

TRef2 holds for `ReduceAbs` but TRef1 fails to hold and a counterexample is shown in Figure 9. The LHS is a model with an s-annotated singleton class C2 that has two composedIn associations to different classes. The RHS changes C2 to being p-annotated by applying `ReduceAbs`. The LHS has concretizations but the RHS does not because of the well-formedness constraint in Figure 5 that forbids a singleton class from being composed into two classes simultaneously.

One way to repair this rule is to restrict it by adding a negative application condition (NAC) [15] that guards the rule application from situations such as this. The NAC is created by encoding the relevant slice of the discovered counterexample. The resulting fixed rule satisfies both TRef1 (up to scope 20) and TRef2, as shown in Table 1.

Another interesting way to repair `ReduceAbs` is to restrict it to apply only after the `CompReduce` transformation, to “normalize” the input model. In this case, rule (R2) of `CompReduce` would split the problematic case into two s-annotated singleton classes and then `ReduceAbs` could be applied. Note that these two possible repairs yield different results.

TRef1 holds for the scope 20 for the transformation `GetSet`, but TRef2 fails. The counterexample here occurs when the RHS model is comp-annotated since the elements `getY: Operation`, `setY: Operation` and the corresponding `ownedOperation` relations are added to the LHS but case (4) in Figure 8 says that such additions can only occur in a refinement if the model is inc-annotated (i.e., incomplete). Thus, we repair this rule by refining the OW annotation from ? to inc. The resulting transformation satisfies both properties.

For the four rules of the transformation `CompReduce`, both TRef2 and TRef1 are satisfied and thus we have evidence that this transformation is uncertainty-reducing.
Table 1. Results of applying the verification method to the six rules and their counterexample-based repairs.

<table>
<thead>
<tr>
<th>Rule</th>
<th>TRef1</th>
<th>Scope</th>
<th>Time (ms)</th>
<th>TRef2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce Abs</td>
<td>fail</td>
<td>3</td>
<td>562</td>
<td>pass</td>
</tr>
<tr>
<td>Reduce Abs(repaired)</td>
<td>pass</td>
<td>20</td>
<td>10593296</td>
<td>pass</td>
</tr>
<tr>
<td>GetSet</td>
<td>pass</td>
<td>15</td>
<td>925032</td>
<td>fail</td>
</tr>
<tr>
<td>GetSet (repaired)</td>
<td>pass</td>
<td>20</td>
<td>10706024</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R1)</td>
<td>pass</td>
<td>9</td>
<td>2502127</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R2)</td>
<td>pass</td>
<td>15</td>
<td>3297282</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R3)</td>
<td>pass</td>
<td>10</td>
<td>1035124</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R4)</td>
<td>pass</td>
<td>10</td>
<td>3901705</td>
<td>pass</td>
</tr>
</tbody>
</table>

7 Related Work

The uncertainty-reducing transformations that we studied in this paper are closely related to refinement of partial behavioral models. Well-known examples of such formalisms include Modal Transition Systems (MTSs) [21] and Featured Transition Systems (FTSs) [3]. The concretizations of MTSs and FTSs are Labeled Transition Systems (LTSs).

In MTSs, uncertainty is captured using *maybe*-annotated transitions. Existing methods of checking MTS refinement, e.g., [22,13], verify that it holds for specific pairs of models. Our approach, on the other hand, aims to verify that a transformation is refining regardless of particular input and output models.

Featured Transition Systems (FTSs) [3] are precise representations of sets of models, used in the area of Software Product Line (SPL) engineering [23] to capture the variability in the behavior of products in a product family. An FTS encodes a set of LTSs using annotations that associate each of its transitions with specific features from a feature diagram. FTS refinement is studied in [7] for the case where new features are added to the SPL, by classifying the new features w.r.t. whether they add or remove new behavior. MAVO partiality can express more nuanced kinds of variability than the *m*-like variability in FTSs. This impacts the difficulty of verification of refinement of MAVO models. Moreover, the notion of feature-set evolution for which FTSs refinement is studied, means that at a certain level, verification also happens at the level of specific pairs of partial models. Finally, MAVO refining transformations can describe more general scenarios where uncertainty is systematically removed from a system.

More broadly, our work is related to a number of approaches for verifying properties of model transformations. Some of them employ theorem proving [14,28], whereas others do some form of model checking [16,4]. Like our approach to proving TRef1, many use Alloy. For example, Barasi et al. [2] represent subsequent applications of rules to an input model as a state-space, similarly to the standard method for representing traces with Alloy [19]. This allows property checking for graph transformation systems, similar to bounded model checking. Anastasakis et al. [1] take a similar approach, using Alloy to verify ATL-like transformations [20]. They create the Alloy encoding of the transformation and
its source and target metamodels and run the tool to produce instances of transformed models, trying to verify that, given well-formed inputs, the rule produces well-formed outputs. However, neither of the above approaches proposes a systematic method to repair the transformation in case counter-examples are produced. Sen et al. [29] use Alloy to create complete versions of partially defined models to use for testing model transformations. This process is reminiscent of the way we use Alloy to generate all extensions of the graph rewrite rule, even though the eventual goal is different.

8 Conclusion and Future Work

This paper presented a contribution to model-driven engineering in the context of uncertainty. Specifically, we described an automated approach to showing correctness of uncertainty-reducing transformations on partial models. The approach builds on our earlier work on verifying uncertainty-reducing refinement applications [25]. We extended the two proof obligations from [25] to showing that they hold for all possible input/output pairs of models. We then showed that the first condition can be checked using a special-purpose tool built on top of Alloy, and the second condition can be checked using a standard OCL checker using a set of syntactic conditions on the transformation. Applying the method on several examples showed that it is effective for debugging transformations and for gathering evidence of their correctness.

Our approach has a number of limitations which we intend to address in later work. Specifically, we are interested in investigating ways to prove the transformation condition TRef1, instead of collecting evidence for it using Alloy. In some cases, this can be done by calculating the maximum scope under which an absence of a counterexample guarantees correctness. This notion is similar to a problem diameter [3]. We also plan to study the more general problem of verifying uncertainty-reducing refining transformations that also involve metamodel translations.
References