A Methodology for Verifying Refinements of Partial Models

Rick Salay\textsuperscript{a} Marsha Chechik\textsuperscript{a} Michalis Famelis\textsuperscript{a} Jan Gorzny\textsuperscript{b}

\textsuperscript{a} Department of Computer Science, University of Toronto, Toronto, Canada
\textsuperscript{b} Department of Mathematics and Statistics, University of Victoria, Victoria, Canada

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 and to ensure that model refinements actually reduce this uncertainty. Furthermore, when a refining 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 our previous work, we have presented a general approach for precisely expressing uncertainty within models. In this paper, we use these foundations and define formal conditions for uncertainty reducing refinement between individual models and within model transformations. We describe tooling for automating the verification of these conditions within transformations and describe its application to example transformations.

Keywords Partial Models, Uncertainty, Refinement, Model Transformation.

1 Introduction

Transformations used in MDE can either be horizontal or vertical. Examples of the former include refactorings, translation and normalization, and are applied to models to change the form of their content without changing the level of abstraction. In contrast, the latter preserve relevant properties of a model while changing the level of abstraction. Vertical transformations that add detail are called refining transformations.

Such vertical transformations can be seen as resolving uncertainty within a model. For example, a vertical transformation that generates Java code from a UML model must preserve the behavioural properties of the design while resolving the uncertainty about which container classes to use to implement associations with upper multiplicity \(\ast\). In this case, the uncertainty is implicit to the modeling scenario since the choice of
Figure 1 – Class diagram for HVAC example showing ad hoc expressions of uncertainty.

container class can be left underspecified in the design but is required to be decided in the implementation. In other cases, explicit uncertainty can be stated within a model. For example, a modeler’s uncertainty about behaviour can be expressed in a state machine by allowing non-deterministic transitions between states, and can later be resolved (manually or by a transformation) by refining the model to a deterministic state machine.

Explicit uncertainty in a model can come from many sources, including but not limited to incomplete requirements [EDM05], presence of alternative design decisions [vL09], disagreements among stakeholders [SNCE10], etc. Yet existing modeling methodologies, languages and tools rarely provide adequate support for it, and uncertainty is typically expressed in an ad hoc or informal manner.

To help address this gap, 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 explicit uncertainty [SFC12]. We call the resulting models partial. Partial models can be analyzed and manipulated just like conventional models, and in previous work we have explored issues such as property checking [FCS12], change propagation [SGC13], and transformations [FSDSC13]. As more information becomes available, it can be incorporated into the partial model to reduce its degree of uncertainty. This is done in a systematic way that constitutes a metamodel-independent form of refinement [SFC12].

The use of verifiable refinement steps as part of the development process has a long tradition in software engineering [Wir71, Hoa69]. Several systematic approaches to facilitate the formal refinement of software models have been proposed (e.g., [dW98, BS03]). More recently, with the emergence of MDE, attention has turned to verifying the correctness of model transformations, including refining transformations (e.g., [LAD+14, CCCDL10, NK08]). In this paper, we continue this line of research and focus on the problem of verifying correctness of both manual refinements and automated refinement transformations that resolve explicit uncertainty within partial models.

Motivating example. Consider the scenario, depicted in Fig. 1, where a modeler is facing uncertainty regarding a fragment of a UML 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 whether they should be in a separate
The textual notes in the diagram represent the modeler’s uncertainty by stating specific information that is known and unknown about the model. Resolving uncertainty within the model requires making decisions such as what operations will be used for disabling gas, etc. All of this information is specified ad hoc, using natural language, since there is no notational mechanism in the class diagram language for expressing uncertainty. Furthermore, the lack of formal semantics for these notes prevents the creation of vertical transformations to automatically resolve uncertainty.

Model $P_1$ in Fig. 2 shows the use of partiality annotations, introduced in [SFC12], to express the uncertainty in Fig. 1. Note that although we use a simplified version of UML class diagrams here, the partial modeling approach can be applied to any modeling language, including behavioural modeling languages (e.g., UML state machines), goal modeling languages (e.g., i* [Yu97]), etc. Each 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 $\text{ConservationManager}$) means that it represents a (as yet unknown) set of operations. This captures the same information as in the note attached to $\text{ConservationManager}$ in Fig. 1 – i.e., that it contains operations for energy conservation but it is still unknown what they are. The $v$ annotation on the $\text{GasDisabler}$ 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 $\text{gasDisablerOps}$ operations. Furthermore, the $m$-annotated composition associations say that if $\text{GasDisabler}$ is assigned to a new class then it may have a composition relationship either to the $\text{HeatingController}$ class or the $\text{SafetyController}$ class. Yet it cannot have this relationship to 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, Fig. 2 shows a partial model refinement of the partial class diagram $P_1$. 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 $\text{ecoControlOps}$ is refined to a set of particular operations \{\text{ecoOff()}, \text{setEcoLevel()}\}, (2) a decision is made to put the functionality to disable the gas supply into the $\text{HeatingController}$ class by assigning the $v$-annotated class $\text{GasDisabler}$ to it, and (3) the $m$-annotated composition relations are eliminated.

Uncertainty can be resolved by eliminating partiality annotations from the model altogether, or by changing them such that the new model has a refinement relationship with the original. This notion of refinement is defined formally in Section 2. Because refinement can happen in various ways, it is necessary to verify refining transformations.

Fig. 2 shows a refinement application to a specific model. In contrast, Fig. 3 gives a refining partial model transformation that can be used to generate a refinement application when applied to an arbitrary model. We defer the explanation of the transformation rule syntax to Section 4 and only give the intuition behind the rule here. Syntactically, $\text{ReduceAbs}$ converts all occurrences of $s$ annotations on elements to $p$ annotations. Semantically, a $p$ annotation means that these now represent particular elements (i.e., $p$ for “particular”) 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 this formally?

**Contributions of this paper.** In this paper, we look at the problem of checking the correctness of both uncertainty-reducing refinements of particular models and partial model refinement transformations.

Specifically, we make the following contributions:

1. We develop a method for verifying partial model refinements applied to a particular model. As an illustration, we use it to show that Fig. 2 represents a valid refinement.

2. We define the formal correctness conditions for a partial model refinement transformation.

3. We develop a method for verifying partial model refinement rewrite rules and the corresponding refinement transformations constructed using such rules. I.e., given a set of rules defining a transformation such as in Fig. 3, our method can be used to show that applying them to any model yields an uncertainty-reducing refinement.

4. We describe prototype tool support to help automate the transformation verification method and to use the generated counterexamples in order to repair faulty rules.

5. We apply the method to the verification of three specific transformations.

A workshop version of this paper [SCG12] introduced the method for verifying refinements of particular models (item 1) and preliminarily explored the issues regarding the verification of refinement transformations. In this paper, we give the definitive version of these results and then use these foundations to give a fully automated approach to the verification of refinement transformations, including a proof of correctness and results from the implementation of the approach. Specifically, Section 3 is an improved and expanded version of the central result in [SCG12] and Sections 4-5 with the proofs in the Appendix are new. This paper also significantly extends the results of [SFC12] which introduced the concept of partial model refinement at a high level and illustrated tool support for refinement verification on an example. Specifically, we contribute the theoretical details of verifying partial model refinements of particular models (item 2) and the approach for verifying refining transformations (items 3-5).
Verifying Partial Model Refinements

The rest of the paper is organized as follows. In Section 2, we review the concept of model partiality as introduced in [SFC12]. In Section 3, we present a method for verifying partial model refinement when applied to particular models. In Section 4, we extend this to a method for verifying refinement transformations and describe the automation of this method. In Section 5, we apply the method to several example transformations. In Section 6, we discuss related work. Finally, in Section 7, 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 [SFC12].

2.1 Models and metamodels

A metamodel represents a set of models and can be expressed as a First Order Logic (FOL) theory. A model is taken to be a finite first order structure satisfying such a theory.

Definition 1 (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 \( T \), denoted by \( \text{Mod}(T) \), are the finite FO \( \Sigma \)-structures that satisfy \( \Phi \) according to the usual FO satisfaction relation.

The simple class diagram metamodel shown graphically in Fig. 4 fits this definition if we interpret boxes as sorts and edges as predicates comprising \( \Sigma_{\text{CD}} \) (where \( \text{CD} \) stands for “class diagram”) and take the multiplicity constraints (translated to FOL) and the additional constraint (1) as comprising \( \Phi_{\text{CD}} \). Fig. 5 shows this metamodel as an FO theory.

Sometimes it is convenient to think of a model as a typed graph where the elements are the nodes typed by sorts in the metamodel and the relation instances are edges typed by the predicates in the metamodel. We use the term atom to mean either an element or a relation instance.

2.2 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 2 (Partial model). A partial model \( P \) over a metamodel \( T = \langle \Sigma, \Phi \rangle \) consists of a base model, denoted \( \text{bs}(P) \), and a set of annotations. The metamodel of \( \text{bs}(P) \) is \( \langle \Sigma, \emptyset \rangle \). \( [P] \) denotes the set of \( T \) models called the concretizations of \( P \). \( P \) is called consistent iff \( [P] \neq \emptyset \).
Additional constraints:
(1) A singleton class cannot be composed in more than one class.

\[ \forall c, c_1, c_2 : \text{Class} \cdot \text{isSingleton}(c) \land \text{composedIn}(c, c_1) \land \text{composedIn}(c, c_2) \Rightarrow c_1 = c_2 \]

Figure 4 – An adapted and simplified metamodel of the UML class diagram language shown graphically.

<table>
<thead>
<tr>
<th>ΣCD</th>
<th>Sorts: Class, Operation, Property</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Predicates: composedIn(Class, Class), ownedOperation(Class, Operation), ownedAttribute(Class, Property), isSingleton(Class)</td>
</tr>
<tr>
<td>ΦCD</td>
<td>(1) ( \forall c, c_1, c_2 : \text{Class} \cdot \text{isSingleton}(c) \land \text{composedIn}(c, c_1) \land \text{composedIn}(c, c_2) \Rightarrow c_1 = c_2 )</td>
</tr>
<tr>
<td></td>
<td>(2) ( \forall c, c' : \text{Class}, o : \text{Operation} \cdot \text{ownedOperation}(c, o) \land \text{ownedOperation}(c', o) \Rightarrow c = c' )</td>
</tr>
<tr>
<td></td>
<td>(3) ( \forall c, c' : \text{Class}, p : \text{Property} \cdot \text{ownedAttribute}(c, p) \land \text{ownedAttribute}(c', p) \Rightarrow c = c' )</td>
</tr>
</tbody>
</table>

Figure 5 – The metamodel of class diagram in Fig. 4 as an FO theory.

The base model is the underlying model in which the annotations are stripped away. Note that the base model does not necessarily need to be a well-formed \( T \) model since it conforms to the metamodel \( (\Sigma, \emptyset) \) (i.e., \( T \) with the well-formedness constraints removed). In fact, the base model of \( \text{P1} \) in Fig. 2 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. A concretization is a well-formed model that satisfies the constraints given by the annotations. For example, one concretization of \( \text{P1} \) is obtained by removing all annotations and removing the composition relation from \text{GasDisabler} to \text{SafetyController}. \( \text{P1} \) has an infinite number of concretizations since each of the \( s \)-annotated operations can be replaced by any set of particular operations. Thus, although \( bs(\text{P1}) \) is not well-formed, \( \text{P1} \) 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:

- **May partiality** allows us to express the level of certainty we have about the presence of a particular atom in a model by annotating it with either \( m \), to indicate that it “may exist” or \( e \), to indicate that it “exists”. A May annotation is refined by changing an \( m \) to \( e \) or eliminating the atom altogether. The ground annotation \( e \) is the default if an annotation is omitted.

- **Abs partiality** allows a modeler to express uncertainty about the number of atoms in the model by letting her annotate atoms as \( p \), representing a “particular”, or \( s \), representing a “set”. A refinement of an Abs annotation elaborates the content of \( s \) atoms by replacing them with a set of \( S \) and \( p \) atoms. The ground annotation \( p \) is the default if an annotation is omitted.

- **Var partiality** allows a modeler to express uncertainty about distinctness of individual atoms in the model by annotating an atom to indicate whether it is a “constant”
Partiality Type | Target | Non-ground annotation | Ground annotation (default)
--- | --- | --- | ---
*May* | atom | M (may exist) | E (exists)
*Abs* | atom | S (set) | P (particular)
*Var* | atom | V (variable) | C (constant)
*OW* | model | INC (incomplete) | COMP (complete)

Table 1 – Summary of MAVO annotations. *May, Abs* and *Var* annotations apply to each atom while *OW* annotations apply to the entire model.

(c) or a “variable” (v). A refinement of a *Var* annotation involves reducing the set of variables by merging them with constants or other variables. The ground annotation c is the default if an annotation is omitted.

*OW partiality* allows a modeler to explicitly state whether her model is incomplete (i.e., can be extended) (INC) or complete (COMP). In contrast to the other types of partiality, here the annotation is at the level of the entire model rather than at the level of individual atoms. The ground annotation COMP is the default if an annotation is omitted.

The annotations are summarized in Table 1. When these four types of partiality annotations are used together, we refer to them as *MAVO partiality*.

**Definition 3.** The set of all *MAVO* partial models over models with metamodel T is denoted MAVO(T).

We state the following important proposition about the consistency of *MAVO* models (the proof is in the Appendix).

**Proposition 1.** Given a *MAVO* model P ∈ MAVO(T), if bs(P) is well-formed w.r.t. T, then bs(P) is a concretization of P and thus, P is consistent.

### 2.3 Formalizing *MAVO* partiality

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]. We proceed as follows.

1) Let M = bs(P) be the base model of a partial model P over metamodel T. We define a new partial model P_M which has M as its base model and its only possible concretization, i.e., bs(P_M) = M and [P_M] = {M} if M is well-formed and [P_M] = ∅ otherwise. We call P_M the *ground model* of P.

2) To construct the FOL encoding of P_M, FO(P_M), we extend T by adding 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 could satisfy the resulting theory is M itself. We refer to these additions as *MAVO* predicates and constraints, respectively.

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 the occurrence of a in every concretization is removed.
We illustrate the above construction using the partial class diagram $M_1$ in Fig. 2. For a description of the general case, please see [SFC12].

1) Let $M_1 = bs(P_1)$ be its base model and $P_{M_1}$ be the corresponding ground partial model.

2) We have:

$$FO(P_{M_1}) = (\Sigma_{CD} \cup \Sigma_{M_1}, \Phi_{CD} \cup \Phi_{M_1})$$

(see Def. 1), where $\Sigma_{M_1}$ and $\Phi_{M_1}$ are the MAVO predicates and constraints, defined in Fig. 6. They extend the signature and constraints for CD models described in Fig. 4. For conciseness, we abbreviate element names in Fig. 6, e.g., ConservationManager becomes CM, etc.

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 Fig. 6. Assume $N$ is such a class diagram. The MAVO constraint Complete ensures that $N$ contains no more elements or relation instances than $M_1$. Now consider the class CM in $M_1$. Exists_CM says that $N$ contains at least one class called CM, Unique_CM – that it contains no more than one class called CM, and the clauses Distinct_CM Unlike – 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 at most one concretization – in this case, it has none since $M_1$ is not well-formed.

3) Relaxing the MAVO constraints $\Phi_{M_1}$ 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 $M_1$ is incomplete, we can express this by removing the Complete clause from $\Phi_{M_1}$ and thereby allow concretizations to be class diagrams that extend $M_1$. Similarly, expressing the effect of the inc, $s$ and $v$ annotations for an element $E$ correspond to relaxing $\Phi_{M_1}$ by removing Exists_E, Unique_E and Distinct_E Unlike clauses, respectively. For example, removing the Distinct_E Unlike clauses is equivalent to marking the class GD with $v$ (i.e., GasDisabler may or may not be distinct from another class). Thus, $\Phi_{P_1}$ is constructed from $\Phi_{M_1}$ by relaxing the MAVO constraints corresponding to the annotations in Fig. 2.
The FO formalization of a MAVO model can be used for reasoning with models containing uncertainty. This includes property checking and consistency checking [SFC12], change propagation [SGC13] as well as verifying refinement, as we discuss in this paper. In addition, the FO formalization provides a way to augment a MAVO model with sentences to allow more precise expressions of uncertainty than are possible using annotations only. For example, the annotations in Fig. 2 indicate that GasDisabler can be composed in either HeatingController or SafetyController but it can also be in neither. Assume we determine that it must be in one or the other. There is no way to express this constraint using annotations alone, however, adding the sentence $\exists GD_{in}HC \lor \exists GD_{in}SC$ to $\Phi_{P_1}$ allows this to be expressed.

2.4 Mappings

Refinement requires a mapping which maps the atoms of the two models (e.g., Fig. 2). Thus, we define the notion of a mapping between MAVO models.

**Definition 4 (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 the following conditions hold:

- for all $(a, a') \in R$, $a'$ and $a$ have the same type in the metamodel, and,
- for all relation instances $r(e, e_1)$ and $r'(e', e'_1)$, $\langle r, r' \rangle \in R \Rightarrow (\langle e, e' \rangle \in R \land \langle e_1, e'_1 \rangle \in R)$

The composition $(R' \circ R)(P, P'')$ of two mappings $R(P, P')$ and $R'(P', P'')$ is the usual composition of binary relations. The set of all possible MAVO between MAVO(T) models is denoted $\text{Map}(T)$.

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

**Definition 5 (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 $R_1$.

3 Verifying Individual Refinements

Intuitively, refinement of a MAVO model should not increase the set of concretizations it has (a proper refinement reduces this), while making sure at least one concretization remains. In Section 2, we formally characterized the set of concretizations of a partial model using an FOL encoding. In this section, we formalize the intuition of refinement in terms of this encoding.

**Definition 6 (MAVO Refinement).** Let $R(P, P')$ be a MAVO mapping where we have encodings $\text{FO}(P') = \langle \Sigma_{P'}, \Phi_{P'} \rangle$ and $\text{FO}(P) = \langle \Sigma_{P}, \Phi_{P} \rangle$. $P'$ 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.

Here, $R(\Phi_P)$ denotes a translation (discussed below) of the MAVO constraints of $P$ according to the mapping $R$. Condition Ref1 ensures that $P'$ has at least one concretization (see Def. 2). Recall from formula (1) that $\Phi_{P'}$ consists of both the MAVO sentences and the well-formedness rules for the modeling language, and so these must be jointly satisfiable for this condition to hold. Condition Ref2 captures our intuition about refinement by ensuring that $P$ has all concretizations of $P'$. We state this property formally, with the proof appearing in the Appendix.

**Proposition 2.** For each MAVO mapping $R(P, P')$ such that condition Ref2 from Defn. 6 holds,

$$\forall M \cdot M \in [P'] \Rightarrow M \in [P]$$

Ref1 and Ref2 are proof obligations required to be met in order to demonstrate the validity of the refinement.

To help illustrate the importance of these two conditions, consider the two invalid refinements shown in Fig. 7. Model $P_3$ is identical to $P_1$ except that the $m$ annotations on the composedIn instances are removed. This means that both these associations must appear in every concretization of $P_3$. However, since this violates the well-formedness condition (1) in Fig. 4 and all concretizations must be well-formed (see Def. 2), this means that $P_3$ has no concretizations. Thus, $P_3$ does not satisfy condition Ref1. Condition Ref2 is satisfied since the set of concretizations of $P_3$ (i.e., the empty set) is a subset of the set of concretizations of $P_1$.

Model $P_4$ is the same as model $P_2$ in Fig. 2 except that the ConservationManager class has been removed. This satisfies Ref1 since $P_4$ has concretizations, e.g., its base model is a concretization. However, it does not satisfy Ref2. This can be seen by observing that every concretization of $P_1$ must contain the class ConservationManager whereas no concretization of $P_4$ contains it. Thus, the concretizations of $P_4$ are not concretizations of $P_1$.

In the special case that $P$ and $P'$ have the same base models (i.e., $\Sigma_{P'} = \Sigma_P$) and the mapping is the identity, Ref2 reduces to the condition that $\Phi_{P'} \Rightarrow \Phi_P$ holds. Candidate refinement $P_3$ in Fig. 7 is an example where the base model does not change. When the base models are different or the mapping is not the identity, we cannot use this simple scheme because the sentences are not directly comparable. For example, the base models differ for the refinement shown in Fig. 2. The classic solution to this kind of problem is to use a theory interpretation to translate the sentences of $FO(P)$ to equivalent sentences in terms of the signature $\Sigma_{P'}$ of $FO(P')$ so that the sentences are comparable (e.g., see [Mai97]).

The rules for defining the translation $R(\cdot)$ for a mapping $R(P, P')$ are given in Fig. 8. The top part of the figure shows the different cases that can occur in mapping $R$ between the base models of $P$ and $P'$, and the bottom defines the corresponding translation to be applied to the occurrences of the MAVO predicates in the sentences of $\Phi_P$. For simplicity, we only show the translation for the MAVO element predicates but the relation predicates are similar. We apply the translation to each element in $P$. Case (1) occurs when the element in $P$ is refined to at least one element in $P'$. The corresponding translation converts the MAVO predicate for the element into a disjunction of the MAVO predicates for the refined elements. This case applies when the element is s-annotated and splits into multiple elements; when it is merged with other elements due to v annotations, when it has no m annotation and when it has a m but is not removed. Case (2) occurs when a m-annotated element in $P$ is removed.
Verifying Partial Model Refinements

1. Fig. 6 shows the signature for $FO(P1)$ with all possible MAVO constraints. Based on its annotations, the set $\Phi_{P1}$ has all the MAVO constraints except $\exists GD_{in} HC$, $\exists GD_{in} SC$, $\text{UniqueGD}_{ops}$, $\text{UniqueTC}_{ops}$, $\text{UniqueGD}_{dis}$, $\text{DistinctGD}_{-HC}$, $\text{DistinctGD}_{-SC}$. $FO(P2)$ is not shown but it is encoded in $P'$; thus, the MAVO predicate is converted to the predicate $false()$ that always evaluates to $false$.

A methodology for verifying a refinement based on the above discussion is given in Fig. 9.

Finally, we state a proposition that says that mapping composition preserves refinement, the proof of which is given in the Appendix.

**Proposition 3.** Let $R(P, P')$ and $R'(P', P'')$ be two mappings that are valid refinements according to Def. 6. Then the composition $(R \circ R')(P, P'')$ is also a valid refinement.

### 3.1 Illustration

In this section, we apply the refinement verification methodology in Fig. 9 to show that the refinement in Fig. 2 is correct. Since a contribution of the current paper is to give a formal exposition of uncertainty reducing refinement, we illustrate each step of Fig. 9. Of course, we do not expect modeling practitioners to perform these steps manually. See [SFC12] for a discussion of tool support for verifying correctness of refinement.

We address each of the four steps of the methodology as follows.

1. Fig. 6 shows the signature for $FO(P1)$ with all possible MAVO constraints. Based on its annotations, the set $\Phi_{P1}$ has all the MAVO constraints except $\exists GD_{in} HC$, $\exists GD_{in} SC$, $\text{UniqueGD}_{ops}$, $\text{UniqueTC}_{ops}$, $\text{UniqueGD}_{dis}$, $\text{DistinctGD}_{-HC}$, $\text{DistinctGD}_{-SC}$. $FO(P2)$ is not shown but it is encoded

![Figure 7](image-url)
Given mapping $R(P, P')$ of MAVO models $P, P'$, the following steps verify that $R$ is a valid refinement mapping and $P'$ is a refinement of $P$.

1. Determine first-order encodings $\text{FO}(P) = \langle \Sigma_P, \Phi_P \rangle$ and $\text{FO}(P') = \langle \Sigma_{P'}, \Phi_{P'} \rangle$.
2. Prove that $\Phi_{P'}$ is satisfiable (proof obligation Ref1).
3. Determine the sentence translation function $R()$ based on Fig. 8.
4. Prove that $\Phi_{P'} \Rightarrow R(\Phi_P)$ (proof obligation Ref2).

Figure 9 – A method for verifying a MAVO refinement.

In a similar way. Based on its annotations, the set $\Phi_{P2}$ contains all MAVO constraints except $\text{Unique}_{\text{GDOps}}, \text{Unique}_{\text{TCDops}}, \text{and Unique}_{\text{LDOps}}$.

2. To prove the satisfiability of $\Phi_{P2}$, we note that the base model of $P2$ (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, $[P2] \neq \emptyset$ and so $\Phi_{P2}$ is satisfiable.

3. The mapping translation function $R()$ is shown in Fig. 10.

4. To prove that $\Phi_{P2} \Rightarrow R(\Phi_{P1})$, it is sufficient to show that $\Phi \Rightarrow R(\phi_{P1})$ for each sentence $\phi_{P1} \in \Phi_{P1}$ for some $\Phi \subseteq \Phi_{P2}$. The proof is given below.

Proof. We proceed with a proof by cases of MAVO constraints in $\Phi_{P1}$. 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 $\phi_1 \in \Phi_{P1}$ and $\phi_2 \in \Phi_{P2}$ be the Complete constraints for $P1$ and $P2$, respectively. Now note that $R(\phi_1)$ is identical to $\phi_2$ everywhere except for the clause for the composedIn elements. In that case, the clause in $\phi_1$ is $\forall x, x'$:
\( R() \) is defined as:
\[
\begin{align*}
\text{ECOps}(x) & \mapsto \text{EO}'(x) \lor \text{SEL}'(x) \\
\text{GD}(x) & \mapsto \text{HC}'(x) \\
\text{GDinHC}(x, x') & \mapsto \text{false}(x) \\
\text{GDinSC}(x, x') & \mapsto \text{false}(x)
\end{align*}
\]
for all remaining elements \( e \),
\[
e(x) \mapsto e'(x)
\]
\( \text{GDinHC}(x, x') \mapsto \text{false}(x) \)
\( \text{GDinSC}(x, x') \mapsto \text{false}(x) \)
for all remaining relation instances \( r \),
\[
r(x_1, x_2) \mapsto r'(x_1, x_2)
\]

<table>
<thead>
<tr>
<th>ECOps = ecoControlOps</th>
<th>EO = ecoOff</th>
<th>SEL = setEcoLevel</th>
<th>SC = SafetyController</th>
<th>HC = HeatingController</th>
<th>GD = GasDisabler</th>
</tr>
</thead>
</table>

Figure 10 – The definition of translation \( R() \) for mapping \( R \) in Fig. 2. The elements of \( \Sigma_{P_2} \) are “primed” to avoid name clashes.

\( \text{Class} \cdot \text{composedIn}(x, x') \Rightarrow (\text{GDinHC}(x, x') \lor \text{GDinSC}(x, x')) \) and the translation in \( R(\phi_1) \) is \( \forall x, x' : \text{Class} \cdot \text{composedIn}(x, x') \Rightarrow (\text{false}(x) \lor \text{false}(x)) \) whereas the clause in \( \phi_2 \) is \( \forall x, x' : \text{Class} \cdot \text{composedIn}(x, x') \Rightarrow (\text{false}(x)) \). These are clearly semantically equivalent and so \( \phi_2 \Rightarrow R(\phi_1) \).

Case 2 (ECOps): \( \Phi_{P_1} \) contains the \text{Exists} constraints for operation ECOps:
\[
R(\text{Exists}_{\text{ECOps}}) = \exists x : \text{Operation} \cdot \text{EO}'(x) \lor \text{SEL}'(x)
\]
which clearly follows from the constraint \( \text{Exists}_{\text{EO}} \) in \( \Phi_{P_2} \). \( \Phi_{P_1} \) also contains the \text{Distinct} constraint for ECOps:
\[
\begin{align*}
R(\text{Distinct}_{\text{ECOps} - e}) &= \forall x : \text{Operation} \cdot \text{EO}'(x) \lor \text{SEL}'(x) \Rightarrow \neg e(x) \text{, and} \\
R(\text{Distinct}_{\text{EO}' - \text{ECOps}}) &= \forall x : \text{Operation} \cdot e(x) \Rightarrow \neg((\text{EO}'(x) \lor \text{SEL}'(x))
\end{align*}
\]
for each operation \( e \in \{ \text{TCOps}, \text{LOOps}, \text{GDOps} \} \). Both of these follow from \( \{ \text{Distinct}_{\text{EO} - e'}, \text{Distinct}_{\text{SEL} - e'} \} \subseteq \Phi_{P_2} \).

Case 3 (GDinHC, GDinSC): \( \Phi_{P_1} \) contains the Unique and Distinct constraints for composition associations GDinHC and GDinSC.
\[
\begin{align*}
R(\text{Unique}_{\text{GDinHC}}) &= \forall x, x', y, y' : \text{Class} \cdot (\text{false}(x) \land \text{false}(x)) \Rightarrow (x = x' \land y = y') \text{, and} \\
R(\text{Unique}_{\text{GDinSC}}) &= \forall x, x', y, y' : \text{Class} \cdot (\text{false}(x) \land \text{false}(x)) \Rightarrow (x = x' \land y = y') \text{, and} \\
R(\text{Distinct}_{\text{GDinHC} - \text{GDinSC}}) &= R(\text{Distinct}_{\text{GDinHC} - \text{GDinSC}}) = \forall x, x' : \text{Class} \cdot \text{false}(x) \Rightarrow \neg \text{false}(x)
\end{align*}
\]
These are always true.

Case 4 (GD): \( \Phi_{P_1} \) contains the \text{Exists} and \text{Unique} constraints for class GD.
\[
\begin{align*}
R(\text{Exists}_{\text{GD}}) &= \text{Exists}_{\text{HC}} \text{, and} \\
R(\text{Unique}_{\text{GD}}) &= \text{Unique}_{\text{HC}}
\end{align*}
\]
Both of these HC’ constraints occur in \( \Phi_{P_2} \).

Case 5: Every other element or relationship instance \( a \) in \( P_1 \) is mapped to its equivalent \( a' \) in \( P_2 \). Thus, if the MAVO constraint \( \phi_a \in \Phi_{P_1} \) holds, then the corresponding constraint \( \phi_{a'} \in \Phi_{P_2} \) holds as well. Furthermore, \( R(\phi_a) = \phi_{a'} \) and so \( \phi_{a'} \Rightarrow R(\phi_a) \).

\[\square\]
### 3.2 Annotation-only case for Ref2

The verification method given in Fig. 9 is general enough that it can be used even with MAVO models that are augmented with arbitrary FO constraints for expressing detailed cases of uncertainty, as discussed in Section 2.3. 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.

Fig. 11 summarizes these constraints, first introduced in [SCH12], 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 refer to the full set of MAVO annotations (\(m/e; s/p; v/c; \text{inc/comp})\), including those assumed by default when the annotation for a partiality type is omitted. Furthermore, we use the annotations as predicates in these sentences. For example, \(e(a)\) is true iff atom \(a\) is annotated with \(e\), and \(\text{inc}(P)\) is true iff model \(P\) is annotated with \(\text{inc}\).

![Figure 11](image)

**Figure 11** — Summary of the constraints on annotations of model atoms across a MAVO refinement mapping.

#### 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 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 into 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 at most one of the \(a_i\)s could be a constant. Finally, if a new atom, not mapped to anything in \(P\), appears in \(P'\) (case (4)), then \(P\) must be incomplete. For example, using this method it is clear that the refinement in Fig. 2 satisfies \(\text{Ref2}\).
4 Verifying Refining Transformations

Def. 6 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 Def. 6). 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 7** (Refining MAVO Transformation). A refining MAVO transformation is a transformation \( F : MAVO(T) \rightarrow MAVO(T) \times Map(T) \) such that for all \( P \in MAVO(T) \) where \( F(P) = \langle P', R_{PP'} \rangle \), \( FO(P) = \langle \Sigma_P, \Phi_P \rangle \), \( FO(P') = \langle \Sigma_{P'}, \Phi_{P'} \rangle \), \( R_{PP'} \) is the refinement mapping from \( P \) to \( P' \) and \( R_{PP'}() \) is the corresponding translation function defined in Fig. 8, the following conditions hold:

\[
\begin{align*}
& (TRef1) \quad \Phi_P \text{ is satisfiable } \Rightarrow \Phi_{P'} \text{ is satisfiable} \\
& (TRef2) \quad \Phi_{P'} \Rightarrow R_{PP'}(\Phi_P)
\end{align*}
\]

These conditions mirror those in Def. 6. Thus, the objective of the method we describe below is to determine whether a given MAVO transformation \( F \) is a refining transformation. Def. 6 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.

4.1 Transformations using rewrite rules

In this section, we consider the case that the candidate refining transformation \( F \) is implemented as the set \( \{\rho_1, \ldots, \rho_n\} \) of confluent and terminating refinement rewrite rules. A refinement rewrite rule is a variant of a graph rewrite rule [EEPT06] defined as follows:

**Definition 8** (MAVO refinement rewrite rule). A refinement rewrite rule \( \rho \) on a MAVO(T) model consists of a MAVO mapping \( R_\rho(LHS, RHS) \) s.t. LHS and RHS are MAVO(T) models and the pair \( \langle LHS, RHS \rangle \) is the underlying graph rewrite rule. Rule \( \rho \) is applied to a MAVO(T) model \( P \) by applying the underlying graph rewrite rule at a matching site of the LHS to produce \( P' \). The resulting refinement mapping between \( P' \) and \( P \) produced by this rule application consists of \( R_\rho \) at the site of the rule application and the identity mapping everywhere else.

In a MAVO rewrite rule, all default annotations must be specified explicitly (i.e, defaulting is not used) and the wildcard placeholder “?” is used when either annotation for a given annotation type can match. Thus, the LHS and RHS each have an annotation from the set \( \{INC, COMP, \text{?}\} \) and each atom of the LHS and RHS has an annotation of form \( \langle m_{may}, m_{abs}, m_{var} \rangle \) where \( m_{may} \in \{m, e, ?\} \), \( m_{abs} \in \{s, p, ?\} \) and \( m_{var} \in \{v, c, ?\} \). 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, in the rule for ReduceAbs (Fig. 3), the element \( E \) on the LHS can match an element with any annotation as long as it includes \( s \) and then the RHS converts this to \( p \) leaving the other annotations unchanged. Thus, 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 that element.
The underlying rule is applied by finding a matching site for the LHS of the rule and then applying the changes according to the RHS: annotations can be changed; atoms on the RHS that are not on the LHS are added, and atoms matched on the LHS that are not on RHS are deleted. Matching of the LHS is subject to the following constraint: any element that is deleted by the RHS can only have edges incident to it that are also matched by the LHS. Thus, MAVO rewrite rule applications never change edges that are incident to, but not included in, the matching site. For example, in rule R2 of CompReduce in Fig. 12, the LHS class C2 is deleted, and thus it cannot match classes that have other relationships beyond the two composedIn relationships indicated.

Applying a transformation $F$ to a model consists of a sequence of rule applications until no more rules can be applied. Since we assume that the set of rules is terminating, this sequence is finite, and because it is confluent, the same result is obtained regardless of the order of rule applications. The refinement mapping produced by $F$ is obtained by composing the refinement mappings produced by each individual rule application.

In order to verify that a MAVO transformation is refining, we must prove that properties $TRef1$ and $TRef2$ in Def. 7 hold. To simplify this process, we note that each rule application is actually a transformation and the sequence of rule applications computing $F$ is a composition of these rule application transformations. Further, note that transformation composition preserves refinement: if $F''$ and $F'$ are refining transformations, then $F'' \circ F'$ must be as well since refinement mapping composition preserves refinement by Prop. 3. Thus, to verify each property it is sufficient just to check that it holds on a single arbitrary rule application for each rewrite rule of the transformation. Note that while sufficient, this is not a necessary condition: even if verification of a particular rule application verification fails, the combined action of multiple rule applications may still be a valid refinement.

According to Def. 8, each application of a rule $\rho$ is a simple extension of $R_{\rho}$ as specified in Def. 5. Thus, to check that $TRef1$ and $TRef2$ hold for an arbitrary application of $\rho$ we must show that $Ref1$ and $Ref2$ hold for every simple extension of $R_{\rho}$. The challenge in this “reduction” of the problem is that there are an infinite number of simple extensions of $R_{\rho}$ and we tackle this challenge separately for $TRef1$ and $TRef2$ below.

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 $\rho_i$.

4.2 Checking Property $TRef1$

To prove that $TRef1$ holds for a rule $\rho$ requires showing that $Ref1$ holds for each simple extension $R(P_{LHS}, P_{RHS})$ of $R_\rho$ - i.e., 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 [Jac06] that checks $TRef1$ in a bounded way by checking $Ref1$ on all simple extensions $R(P_{LHS}, P_{LHS})$ of $R_\rho$ up to a given scope. Here, a scope $n$ means that
$R_\rho$ is extended by up to $n$ atoms for each atom type defined by the metamodel. In addition, we exploit the following optimization: we can limit our search to those simple extensions in which $bs(P_{RHS})$ is not well-formed since, due to Prop. 1, if it is well-formed then $P_{RHS}$ is consistent and so $Ref1$ necessarily holds.

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 [HHT96]) that will prevent it from being applied in the bad cases. Note that, even if Alloy reaches the scope without finding a counterexample, we have only shown that $TRef1$ holds up to the scope and it still may not hold for larger scopes. Thus, this approach can only be used to provide evidence that $TRef1$ may hold. Note that any method for proving that $TRef1$ holds is inherently limited because of the undecidability of first order logic. The bounded approach has been shown to be effective in practice for finding errors and providing some assurance about correctness (e.g., see [MRR11]).

Our tool accepts a rule expressed in Ecore [SBPM07] as its input. The rule is then translated, using TXL [Cor06], into an Alloy encoding, which includes all of the rule’s MAVO annotations, and is combined with our encoding of the Ecore metamodel. To this, we add Alloy predicates that allow us to create arbitrary simple extensions (see Definition 5) for the LHS and the RHS of the rule, as described in Section 4. 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. Running the generated Alloy encoding enumerates all RHS MAVO models (i.e., the RHS of the MAVO rule and its simple extensions) with base models that are not well-formed, up to a given scope.

4.3 Checking Property $TRef2$

In Sec. 3.2, we discussed syntactic sufficient conditions for showing that a candidate refinement mapping satisfies property $Ref2$. This method is applicable only when the MAVO models on either side of the mapping use only annotations and no additional FO constraints. Fortunately, for a refinement rewrite rule $\rho = R_\rho(LHS, RHS)$, this is the case. The syntactic refinement conditions also have the desirable “locality” characteristic given by the following proposition (the proof is in the Appendix).

**Proposition 4.** Given a refinement rewrite rule $\rho = R_\rho(LHS, RHS)$, if $\rho$ satisfies the syntactic refinement conditions in Fig. 11, then every simple extension $R(P_{LHS}, P_{RHS})$ of $R_\rho$ satisfies these conditions.

Thus, we can reduce the problem of checking condition $TRef2$ to simply checking the syntactic refinement conditions on the LHS and RHS of the rule – i.e., we have reduced the problem of checking the proof obligation $TRef2$ to the much simpler problem of checking the syntactic refinement conditions given in Fig. 11. Note however, that since this is only a sufficient condition, $\rho$ may fail this test and still satisfy $TRef2$.

For a given refinement rewrite rule, these conditions can be easily checked with existing tools. In particular, checking $TRef2$ involves (a) expressing the constraints shown in Fig. 11 as OCL constraints over the Ecore representation of the rule, and (b) using an off-the-shelf OCL constraint checker, such as DresdenOCL [HDF00].
5 Applying the Transformation Verification Method

We now illustrate the transformation verification method on three confluent and terminating 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. 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 Fig. 3. The second is GetSet with the rule shown in Fig. 12(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 [SCG12], whereas here we show how our method can automate the verification.

The third example is CompReduce, shown in Fig. 12(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 Fig. 4 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 the LHS must have at least two classes corresponding to \( C_2 \). 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 2 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 result of checking $TRef_1$ for scopes 3 and 7 (using the Alloy-based tool described in Sec. 4.2) and the result for $TRef_2$. Recall that checking $TRef_1$ on a rule for a scope $n$ means that $Ref_1$ is checked for all simple extensions of the rule containing up to $n$ additional atoms for each atom type.

$TRef_2$ holds for ReduceAbs but $TRef_1$ fails (at scope 3) to hold and a counterexample is shown in Fig. 13. The LHS is a model with an $s$-annotated singleton class $C2$ that has two $\text{composedIn}$ associations to different classes. The RHS changes $C2$ to being $p$-annotated by applying ReduceAbs. The RHS has concretizations but the RHS does not because of well-formedness constraint (1) in Fig. 4 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) [HHT96] that guards the rule application from such situations. The NAC is created by encoding the relevant slice of the discovered counterexample. In the case of ReduceAbs, the repair involves constructing a NAC from the LHS of Fig. 13 in order to prevent the rule from being applied to singleton classes that are composed in more than one class. The resulting fixed rule satisfies both $TRef_1$ (at least up to scope 7) and $TRef_2$, as shown in the second row of Table 2.

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.

$TRef_1$ holds for scope 7 for the transformation GetSet, but $TRef_2$ fails. The counterexample here occurs when the RHS model is $\text{comp}$-annotated since the elements $\text{getY : Operation}$, $\text{setY : Operation}$ and the corresponding $\text{ownedOperation}$ relations are added to the LHS but case (4) in Fig. 11 says that such additions can only occur in a refinement if the model is $\text{inc}$-annotated (i.e., incomplete). Thus, we repair this rule by refining the $\text{OW}$ annotation from $?$ to $\text{inc}$. The resulting transformation satisfies both properties. For the four rules of the transformation CompReduce, both $TRef_2$ and $TRef_1$ are satisfied and thus we have evidence that this transformation is valid refinement.

As this is prototype tooling, we did not focus on optimizing its performance. Even with an unoptimized tool, the results indicate that all scope 3 checks finish quickly ($< 1s$). Scope 7 checks take on the order of minutes, with the longest taking just over 10 (CompReduce (R4)). Such runtimes can still be reasonable since transformation verification needs to be done only once when the transformation implementation is changed. We leave the problem of determine the reasonable scope for a given rule for
## Table 2 – Results of applying the verification method to the six rules and their counter-example-based repairs.

<table>
<thead>
<tr>
<th>Rule</th>
<th>TRef1</th>
<th>Time(ms) Scope 3</th>
<th>Time(ms) Scope 7</th>
<th>TRef2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReduceAbs</td>
<td>fail</td>
<td>562</td>
<td>N/A</td>
<td>pass</td>
</tr>
<tr>
<td>ReduceAbs (repaired)</td>
<td>pass</td>
<td>733</td>
<td>313166</td>
<td>pass</td>
</tr>
<tr>
<td>GetSet</td>
<td>pass</td>
<td>636</td>
<td>4770</td>
<td>fail</td>
</tr>
<tr>
<td>GetSet (repaired)</td>
<td>pass</td>
<td>668</td>
<td>226128</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R1)</td>
<td>pass</td>
<td>645</td>
<td>350476</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R2)</td>
<td>pass</td>
<td>224</td>
<td>127505</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R3)</td>
<td>pass</td>
<td>256</td>
<td>114916</td>
<td>pass</td>
</tr>
<tr>
<td>CompReduce (R4)</td>
<td>pass</td>
<td>689</td>
<td>623241</td>
<td>pass</td>
</tr>
</tbody>
</table>

The concept of refinement is central to formal software engineering methodologies and is supported by Z [WD96], B [AA05], Abstract State Machines [BS03], OBJ [GWM⁺00], algebraic specifications [Mai97], etc. In this paper we showed that refinement supported by MAVO is correct w.r.t. its language semantics. While it is the closest to the approach used in algebraic specifications [Mai97], it differs from this and all of the abovementioned languages in several important ways. First, MAVO is not designed for software specification, but rather for expressing the uncertainty of a modeler. This means that it provides mechanisms for explicitly specifying what is unknown rather than expressing this only implicitly via omitted information (i.e.,
under-specification). Second, MAVO is a meta-language that can be used to augment any modeling language defined using a metamodel in order to allow uncertainty specification. Thus, MAVO can be used with existing formal software modeling languages, non-formal software modeling languages (e.g., i* [Yu97]) and even non-software modeling languages (e.g., chemical structure models [ACH+97]), etc. MAVO achieves this language independence because it is based only on the syntax of the modeling language while ignoring its semantics – see [SFC12] for a more detailed discussion – and thus MAVO refinement described in this paper is independent of the language semantics as well.

**Set-reducing operations.** Uncertainty-reducing transformations are a special case of a wider class of operations that, given an artifact that abstracts a set of models, produce a new artifact abstracting a subset of the original. Such operations can be found in domains such as product line engineering, megamodeling and metamodeling.

In product line engineering, staged configuration [CHE04] is a method for incrementally making choices about which features to include in a product. This is achieved by stepwise reducing the set of possible configurations of the input feature model. To guarantee this correctness conditions, six allowable specialization steps are provided, each one representing a possible way to remove a configuration choice. These steps are formalized using context-free grammars [CHU05] and are implemented in the FeaturePlugin tool for the Eclipse platform [AC04].

Metamodels can also be understood as abstractions of sets of (instance) models, since “a modeling language can be seen as delimiting all possible specifications which can be constructed using that language” [Gui07]. A method typically used to reduce the set of admissible instances of a metamodel is the addition of constraints [KNLS00], often in a language such as the Object Constraint Language (OCL) [Obj06]. A different approach is to enable the creation of metamodel sub-types. There are various approaches for creating model subtypes [GCD+12], mainly focusing on achieving model substitutability, especially in the context of model transformations. Metamodel pruning is a dual to subtyping, aiming to create metamodel super-types [SMBJ09].

Meganmodels are used to model the macroscopic view of software development. The elements of a megamodel are themselves models, interconnected with various macro-relations [FN05, SME09]. In that sense, it is an abstraction of a set of models. Reducing this set (in order to, for example, create task-specific views) can be accomplished with model slicing [BCE+06]. Since megamodels are themselves models, it is possible to use submodel extraction techniques [CVC13].

**Verifying model transformations.** More broadly, our work is related to a number of approaches for verifying properties of model transformations. Some of them employ theorem proving [GGL+06, Sch10], whereas others do some form of model checking [He98, BHM09]. Like our approach to proving TRef1, many use Alloy. For example, Baresi et al. [BS06] represent subsequent applications of rules to an input model as a state-space, similarly to the standard method for representing traces with Alloy [JSS01]. This allows property checking for graph transformation systems, similar to bounded model checking. Anastasakis et al. [ABK07] take a similar approach, using Alloy to verify ATL-like transformations [JABK08]. 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. [SMTC12] 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.

7 Conclusion and Future Work

In this paper, we described an approach for verifying uncertainty reducing refinements of partial models. In particular, we defined a method for verifying refinements applied to particular models and then extended this to verify refining transformations of partial models. In both cases, the verification depends on satisfying two proof obligations and can be automated. For transformation verification, we then showed that the first condition (TRef1) can be checked by a special-purpose tool built on top of Alloy, and the second condition (TRef2) — by 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 gathering evidence of their correctness.

Our approach has a number of limitations which we intend to address in follow-on 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 computing a problem diameter [BCCZ99]. We also plan to study the more general problem of verifying uncertainty-reducing refining transformations that also involve metamodel translations.

References


A Proofs

A.1 Proof of Prop. 1

By Def. 2, $P$ is consistent iff it contains at least one concretization. Let $P'$ be the refinement of $P$ obtained by making all annotations default (i.e., removing any explicit annotations). Clearly $P'$ has at most one concretization: if $bs(P')$ is well-formed, then this is the sole concretization; otherwise, it has no concretizations. But, $bs(P) = bs(P')$ since we are just changing the annotations. Therefore, since all concretizations of $P'$ are also concretizations of $P$, this means that when $bs(P)$ is well-formed, it is a concretization of $P$ and $P$ is consistent.

A.2 Proof of Prop. 2

We wish to show that if Ref2 holds then every concretization of $P'$ is also a concretization of $P$. When $P$ and $P'$ have the same base model, $R()$ is the identity, and the proposition clearly holds. When $P'$ and $P$ have different base models, then concretizations are not directly comparable since $\text{Mod} (FO(P'))$ and $\text{Mod}(FO(P))$ are based on different signatures. To address this, we work with a representation of a concretization that is not dependent on the signature. Specifically, we note that every concretization of a MAVO model can also be viewed as its refinement – a ground MAVO model containing no annotations. Furthermore, the corresponding MAVO
mapping for this refinement is easy to obtain from the FOL structure representation of the concretization.

**Definition 9.** Given a MAVO model $P$ with $\text{FO}(P) = (\Sigma_P, \Phi_P)$, let $M \in [P]$ be a concretization and $\text{FO}(M)$ be its representation as a FOL structure satisfying $\Phi_P$. Let $M'$ be the MAVO model with base model $M$ and with no annotations. We define the MAVO mapping $R_M$ between $M'$ and $P$ using the following condition: $\forall a \in \text{atoms}(M'), a' \in \text{atoms}(P) \cdot (a, a') \in R_M$ iff in $\text{FO}(M)$, the MAVO predicate for $a'$ holds for $a$.

For simplicity, in the following proof of Prop. 2, we use $M$ to denote either a concretization as a model, its FOL representation $\text{FO}(M)$ or its corresponding ground MAVO model $M'$, and our usage should be clear from context. Furthermore, for any MAVO model $P$ with $\text{FO}(P) = (\Sigma_P, \Phi_P)$, there is a one-to-one correspondence between the atoms of $P$ and the MAVO predicates in $\Sigma_P$ and we will rely on this to switch between the different usages.

Assume that the condition $\text{Ref2}$ holds and let $M$ be a model s.t. $M \in [P']$, with $R_M$ being the mapping between $M$ and $P'$. Without loss of generality, we assume that the metamodel of $M$ consists of a single element type and relation type. Thus, we will omit the typing of variables where its meaning is clear. Since $\text{Ref2}$ holds, we know that $(\Phi_P \Rightarrow R(\Phi_P))$, and since $M \models \Phi_P$, therefore, $M \models R(\Phi_P)$. We now show that this also means that $M \models \Phi_P$ when we consider the mapping between $M$ and $P$ to be $R \circ R_M$. It is sufficient to show for each sentence $\phi \in \Phi_P$ that $M \models \phi$. We proceed by cases considering each type of MAVO constraint in $\Phi_P$. Our strategy in each case is to consider the different possible sentence translations $R()$ of $\phi$ defined by Fig. 8. For each possible translation, we show that $M \models \phi$ using the mapping $R \circ R_M$.

Case 1 (**Complete**): If $P$ has the inc annotation then it has no **Complete** constraint; thus we only need to consider when it is missing this annotation. The **Complete** constraint consists of a conjunction of clauses. We prove that $M$ satisfies the clause for model elements; the proof for model relations is similar. The clause corresponding to model elements is the disjunction $\forall x \cdot a_1(x) \lor \ldots \lor a_m(x)$ containing every MAVO element predicate $a_i(x)$ of $\Sigma_P$. Let $\phi_1$ denote this clause. Each $a_i(x)$ can fall into case (1) or (2) in the sentence translation $R()$ defined in Fig. 8. If case (1) applies, then $a_i(x)$ is replaced by the disjunction $a'_i(x) \lor \ldots \lor a''_i(x)$. If case (2) applies, it is replaced by $\text{false}(x)$ Thus, $R(\phi_1)$ is a disjunction of a subset of MAVO element predicates of $\Sigma_P$. But since $M \models R(\phi_1)$, every element of $M$ must be mapped by $R_M$ to an element of this subset. Thus, $R \circ R_M$ maps every element of $M$ to an element of $P$ which must occur as a disjunct in $\phi_1$. Therefore, $M \models \phi_1$.

Case 2 (**Exists**): For each clause $\text{Exists}_n$ in $\Phi_P$ for an atom $a \in \text{atoms}(P)$, we consider the translation cases in Fig. 8 that can apply to $R()$:

- If case (1) applies, $R(\text{Exists}_n) = \exists x \cdot a'_i(x) \lor \ldots \lor a''_i(x)$ . Since $M \models R(\text{Exists}_n)$, it must be that $(a'', a'_i) \in R_M$ for some $i$ and $a'' \in \text{atoms}(M)$. But $(a'_i, a) \in R$ for all $i \in \{1, \ldots, n\}$. Therefore, $(a'', a) \in R \circ R_M$ and thus $M \models \text{Exists}_n$.

- We show that case (2) cannot occur. If case (2) applies, $R(\text{Exists}_n) = \exists x \cdot \text{false}(x)$. This is always false and so $R(\Phi_P) = \text{false}$. But $M \models R(\Phi)$ and so $M \models \text{false}$ which is a contradiction because $\text{false}$ is unsatisfiable. Therefore, this case cannot occur.
Case 3 (Unique): For each clause $Unique_a$ in $\Phi_P$ for an atom $a \in atoms(P)$, we consider the translation cases in Fig. 8 that can apply for $R()$:

- If case (1) applies, it must be that $n = 1$ since $Unique_a$ means that there is at most one $a$ in a concretization. Thus, $R(Unique_a) = Unique_a'$. Since $M \models R(Unique_a)$, it must be that either $(a'',a'_1) \in R_M$ for exactly one $a'' \in atoms(M)$ or there is no such $a''$. In the first case, this means that $(a'',a) \in R \circ R_M$ and thus $M \models Unique_a$. In the second case, $R_M$ maps nothing in $M$ to $a'_1$ and so $R \circ R_M$ maps nothing in $M$ to $a$. Thus, $M \models Unique_a$.

- If case (2) applies, $R$ maps nothing in $P'$ to $a$ and so $R \circ R_M$ maps nothing in $M$ to $a$. Thus, $M \models Unique_a$.

Case 4 (Distinct): For each clause $Distinct_{a\rightarrow b}$ in $\Phi_P$ for atoms $a,b \in atoms(P)$, we consider the translation cases in Fig. 8 that can apply for $R()$. There are four possibilities since case (1) or (2) could apply to either $a$ or $b$.

- If case (1) applies to both $a$ and $b$, then $R(Distinct_{a\rightarrow b}) = \forall x \cdot (a'_1(x) \lor \ldots \lor a'_n(x)) \Rightarrow \neg(b'_1(x) \lor \ldots \lor b'_m(x))$. Note that $a'_i \neq b'_j$ for all $(i,j)$ since $Distinct_{a\rightarrow b}$ means that $a$ and $b$ cannot overlap in a concretization. Since $M \models R(Distinct_{a\rightarrow b})$, it must be the case that the set of atoms in $M$ that $R_M$ maps to, $\{a'_1,\ldots,a'_n\}$, is disjoint from the set that $R_M$ maps to, $\{b'_1,\ldots,b'_m\}$. Thus, the set of atoms in $M$ that $R \circ R_M$ maps to $a$ is disjoint from the set of atoms in $M$ that $R \circ R_M$ maps to $b$. Therefore, $M \models R(Distinct_{a\rightarrow b})$.

- If case (1) applies to $a$ and case (2) to $b$, then $R(Distinct_{a\rightarrow b}) = \forall x \cdot (a'_1(x) \lor \ldots \lor a'_n(x)) \Rightarrow \neg false(x)$. Thus, $R(Distinct_{a\rightarrow b})$ is always true and $M \models R(Distinct_{a\rightarrow b})$ trivially.

- If case (2) applies to $a$ and case (1) to $b$, then $R(Distinct_{a\rightarrow b}) = \forall x \cdot false(x) \Rightarrow \neg(b'_1(x) \lor \ldots \lor b'_m(x))$. Thus, $R(Distinct_{a\rightarrow b})$ is always true and $M \models R(Distinct_{a\rightarrow b})$ trivially.

- If case (3) applies to both $a$ and $b$, then $R(Distinct_{a\rightarrow b}) = \forall x \cdot false(x) \Rightarrow \neg false(x)$. Thus, $R(Distinct_{a\rightarrow b})$ is always true and $M \models R(Distinct_{a\rightarrow b})$ trivially.

Since all the cases have been considered, $M \models \Phi_P$ and therefore, $M \in [P]$.

A.3 Proof of Prop. 3

Let $R(P,P')$ and $R'(P',P'')$ be two MAVO mappings that are valid refinements. Let $FO(P) = \langle \Sigma_P, \Phi_P \rangle$, $FO(P') = \langle \Sigma_{P'}, \Phi_{P'} \rangle$ and $FO(P'') = \langle \Sigma_{P''}, \Phi_{P''} \rangle$ be the FO encodings as defined in Section 2. We show that conditions Ref1 and Ref2 hold for the composed mapping $R' \circ R$.

Ref1: Since $R'$ is a valid refinement, it satisfies Ref1 and $P''$ is satisfiable. Thus, $R' \circ R$ also satisfies Ref1.

Ref2: Since $R$ and $R'$ are valid refinements, Ref2 holds,

$$\Phi_{P'} \Rightarrow R(\Phi_P)$$  \hspace{1cm} (2)$$
$$\Phi_{P''} \Rightarrow R'(\Phi_{P'})$$  \hspace{1cm} (3)$$
Applying to $R'(\Phi_P \Rightarrow R(\Phi_P))$ which is equivalent to $R'(\Phi_P \Rightarrow R'(R(\Phi_P)))$ since the mapping translation only affects predicate symbols (See Fig. 8). Combining this with (3) yields $\Phi_P \Rightarrow R'(R(\Phi_P))$. Therefore, Ref2 holds for $R' \circ R$.

Since both conditions Ref1 and Ref2 hold for $R' \circ R$, it is a valid refinement.

A.4 Proof of Prop. 4

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

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

Therefore, when $\alpha$ is annotated with epc, $R^#(P^#_{LHS}, P^#_{RHS})$ 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 $P^#_{LHS}$ is annotated with mpc, then it must have the same annotation in $P^#_{RHS}$ (by definition of a simple extension), and the first constraint in cases (1) and (3) no longer applies.

Therefore, for any annotation on $\alpha$, $R^#(P^#_{LHS}, P^#_{RHS})$ satisfies all the syntactic refinement conditions and Ref1 holds.
About the authors

Rick Salay is currently a NECSIS Research Associate in the Department of Computer Science at the University of Toronto. He received a B.A.Sc. and M.A.Sc. in Systems Design Engineering from University of Waterloo (1991) and a Ph.D. in Computer Science from the University of Toronto (2010). His research focus is on developing formal theories about non-formal concepts such as modeler intent and modeler uncertainty in order to provide a foundation for tool support that will help software engineering practitioners. He regularly serves on programme committees for software engineering conferences and workshops. Prior to his Ph.D., he had a 15 year career in advanced software product development holding various senior software design roles, most recently as chief architect at InSystems Technologies Inc. (now Oracle). Contact him at rsalay@cs.toronto.edu, or visit http://www.cs.toronto.edu/~rsalay/.

Marsha Chechik is currently Professor and Bell University Labs Chair in Software Engineering in the Department of Computer Science at the University of Toronto. She is also Vice Chair of the Department. She received her Ph.D. from the University of Maryland in 1996. Prof. Chechik’s research interests are in the application of formal methods to improve the quality of software. She has authored over 100 papers in formal methods, software specification and verification, computer security and requirements engineering. In 2002-2003, Prof. Chechik was a visiting scientist at Lucent Technologies in Murray Hill, NY and at Imperial College, London UK, and in 2013 – at Stonybrook University. She is a member of IFIP WG 2.9 on Requirements Engineering and an Associate Editor of Journal on Software and Systems Modeling. She is has been an associate editor of IEEE Transactions on Software Engineering 2003-2007, 2010-2013. She regularly serves on program committees of international conferences in the areas of software engineering and automated verification. Marsha Chechik is a Program Committee Co-Chair of TACAS’16. She has been a PC Co-Chair of the 2014 International Conference on Automated Software Engineering (ASE), Co-Chair of the 2008 International Conference on Concurrency Theory (CONCUR), PC Co-Chair of the 2008 International Conference on Computer Science and Software Engineering (CASCOn), and PC Co-Chair of the 2009 International Conference on Formal Aspects of Software Engineering (FASE). She is a Member of ACM SIGSOFT and the IEEE Computer Society. Contact her at chechik@cs.toronto.edu, or visit http://www.cs.toronto.edu/~chechik/.
Michalis Famelis has been a graduate student in the Department of Computer Science at the University of Toronto since 2008, working on his PhD thesis since 2010. He has been continuously involved in software modeling research since his undergraduate diploma thesis, in the National Technical University of Athens in 2007. In his current work, Michalis is studying the management of design uncertainty in software models, via the use of partial modeling techniques. He has served as co-chair and program committee member of the workshop on Model Driven Engineering Verification and Validation (MoDeVVa) since 2012 and has been a program committee member of the 2013 Doctoral Symposium at the conference on Model Driven Engineering Languages and Systems (MODELS) and the 2013 Workshop on Domain-Specific Modeling (DSM). Contact him at famelis@cs.toronto.edu, or visit http://www.cs.toronto.edu/~famelis/.

Jan Gorzny received his bachelor’s degree in computer science and combinatorics from the University of Waterloo in 2011, and a M.Sc. in computer science from the University of Toronto in 2013. He is currently pursuing a M.Sc. in algorithmic graph theory at the University of Victoria. His research interests include formal methods and model-driven engineering, as well as graph theory and discrete mathematics. Contact him at jgorzny@uvic.ca, or visit http://www.math.uvic.ca/~jgorzny/.

Acknowledgments  We thank reviewers of this paper for their thoughtful feedback. We are also grateful to the reviewers of our original VOLT’12 submission as well as the attendees of the VOLT’12 workshop for discussions and suggestions for improvement of this work.