The Semantics of Partial Model Transformations

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

***Abstract*—Model transformations are traditionally designed to operate on models that do not contain uncertainty. In previous work, we have developed partial models, i.e., models that explicitly capture uncertainty. In this paper, we study the transformation of partial models. We define the notion of correct *lifting* of transformations so that they can be applied to partial models. For this, we encode transformations as *transfer predicates* and describe the mechanics of applying transformations using logic. We demonstrate the approach using two example transformations (addition and deletion) and outline a method for testing the application of transformations using a SAT solver. Reflecting on these preliminary attempts, we discuss the main limitations and challenges and outline future steps for our research on partial model transformation.**

I. Introduction and Motivating Example

Software modelers often face the challenging task of working in the presence uncertainty. This involves having to deal with potentially large sets of design alternatives, in situations where there is not enough information to make fully informed decisions. However, existing modeling methodologies, languages and tools rarely address uncertainty in an explicit way.

In particular, model transformation languages, libraries and engines usually assume the existence of a single, unambiguous input model. Uncertainty is therefore implicitly treated as an undesirable property. In fact, in the face of uncertainty, modelers are forced either (a) to transform each alternative model separately or (b) to remove uncertainty entirely before work can continue. The first option is clearly intractable when the set of alternatives is large. The second option requires modelers to make provisional decisions that artificially remove uncertainty from their artifacts. This increases the risk of having to undo their work when previously unknown information becomes available. Even worse, it can mean committing too early to design decisions that cannot be reversed without significant costs, when in fact the modeler may still find it desirable to keep many alternative options open for consideration. [9]

In previous work [5], we introduced *partial models*. Partial models concisely express *sets* of possible models that can result from removing uncertainty. Yet partial models can be developed directly and used as first-class artifacts in software development. In this paper, we present our preliminary work on transforming partial models.

In [11], we introduced a class of transformations that is particular to partial models, namely, *refinement*, aimed to reduce the level of uncertainty. The aim of this paper is different – we want to apply ordinary transformations, otherwise written for classical models, to partial models. We propose to achieve this by *lifting*, i.e., adapting, the semantics of ordinary transformations to partial models.

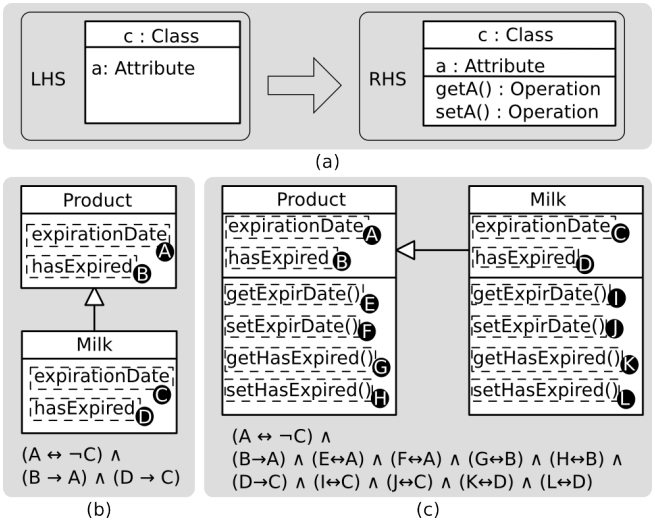

Figure 1. (a) Refactoring transformation rule R_1 for adding getters and setters. (b) Initial partial model M_1 of our motivating example. (c) Resulting partial model M_2 after applying R_1 to M_1.

We now present a motivating example. The partial modeling approach can be applied to any modeling language, but in this example we use a simplified version of UML class diagrams. We assume the scenario where a modeler is facing uncertainty regarding a fragment of the class diagram for an application for a food company.

At the start of our scenario, the modeler has come up with the partial model M_1, shown in Figure 1(b) (see Section II for details about notation and semantics). M_1 captures a basic hierarchy: a class `Milk` that extends a more general class `Product`. The modeler does not yet know if the company plans to use the application for non-perishables as well. She is therefore unclear as to whether `expirationDate` should be an attribute of all products or just for milk.[1] At the same time, she is yet undecided as to whether an attribute `hasExpired`, derived from `expirationDate`, would be useful. The four possible alternative designs (also called *concretizations*), $m_{1,1} - m_{1,4}$, that this partial model encodes are shown in Figures 2(a-d).

We further assume that the modeler wants to refactor the model to add getter and setter methods. For that she wants to

[1] We disregard the obvious modeling solution for illustration purposes.

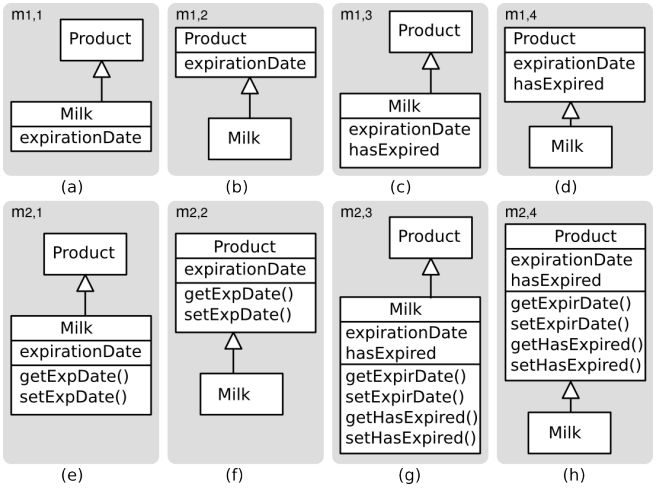

Figure 2. (a-d) The four concretizations of M_1. (e-h) Concretizations of M_2, i.e., transformed versions of (a-d) respectively after the addition of getters and setters.

use a simple rewrite rule R_1 (in the remainder of the paper, we refer to it as *An Example Adding Rule*) which is shown in Figure 1(a). The left-hand side (LHS) of the rule can match an attribute of a class. The matched sub-model is then replaced with the right-hand side (RHS) of the rule, which contains the newly added getter and setter operations.

Applying a rule to a single classical model is straightforward, both in terms of desired results and of mechanics. But what does it mean to apply a transformation rule to a partial model? We break down this general question into two specific ones: (Q1) What should the result of applying a transformation to a partial model be? (Q2) What is the mechanics of applying a rule to a partial model?

Regarding (Q1), we describe the intuition as follows: the partial model M_1 in Figure 1(b) encodes exactly the four models $m_{1,1} - m_{1,4}$ in Figure 2(a-d). Applying the rule R_1, using the "standard" graph transformation approach [3] to each alternative classical model $m_{1,i}$ in produces a new model $m_{2,i}$, respectively shown in Figures 2(e-h).[2] Models $m_{2,1} - m_{2,4}$ are encoded by the partial model M_2 shown in Figure 1(c). Therefore, the application of the rule R_1 directly to M_1 should result in M_2. We formalize this intuition as a correctness criterion in Section IV.

Regarding (Q2), we have opted to not change the transformation itself, but rather redefine the semantics of rule application for partial models, using logic. In Section II, we describe how both classical and partial models can be encoded in propositional logic. In Section III-A, we outline a method to also encode the rules as *transfer predicates* and in Section III-B, we describe how to apply such encoded rules on partial models.

[2]We assume a graph transformation engine that applies the rule in parallel to all matching sites. For example, in the model $m_{1,3}$ in Figure 2(c), the rule matches simultaneously both attributes, resulting in the model $m_{2,3}$ in Figure 2(g).

We conclude our motivating example by introducing a second example transformation. After applying R_1 to M_1 and producing M_2, the modeler realizes that applying this rule had the undesired effect of also adding setter methods for the derived attribute `hasExpired`. For illustration purposes, we assume that derived attributes follow the naming convention "`hasX`". Based on this, the modeler defined the transformation R_2, shown in Figure 3(a) (in the remainder of the paper, we refer to it as *An Example Deleting Rule*). The rule matches classes that have both a derived attribute and its respective setter and deletes this setter. Applying R_2 to M_2 should result in a new partial model M_3, shown in Figure 3(b), that has exactly the four concretizations $m_{3,1} - m_{3,4}$, shown in Figures 2(c-f).

To summarize, the paper makes the following contributions: (a) We define the notion of lifting transformations to partial models and outline a way to test it. (b) We provide a method for defining semantics of lifting transformations for partial models using logical encodings in the form of transfer predicates. (c) We demonstrate the approach by applying it to two transformation examples and discuss the limitations and challenges of lifting (Section V).

II. Partial Models

In this section, we use the motivating example from Section I to briefly explain the notation and semantics of partial models. Their formal definition, semantics, encodings and representation are described in detail in [6].

The partial model M_1 in Figure 1(b), called a *May Model* [11], summarizes the four alternative concretizations in Figure 2(a-d) compactly and exactly. Each element in M_1 is annotated as True, False or Maybe [8]. A Maybe annotation indicates an optional element, such as the attributes labeled with the letters A to D in Figure 1(b). These are the model elements that differ between the concretizations and are depicted graphically using dashed lines.

As more information becomes available, the annotations of these elements can change from Maybe to True or False (i.e., be included or excluded from the model) or remain Maybe. This process is a form of *"point-wise" refinement* [11] that only affects the degree of explicated uncertainty without introducing additional elements. A *concretization* of a partial model is a model where all the Maybe elements have been annotated with True or False. All concretizations of a partial model are classical and are obtained via point-wise refinement. We denote the set of concretizations of a partial model M by $[M]$.

May models are accompanied by *May Formulas* constraining the allowable configurations of the Maybe elements and thus defining the set of concretizations of the partial model. For example, the May formula for model M_1 is $\Phi^{may}_{M_1} = (A \leftrightarrow \neg C) \wedge (B \rightarrow A) \wedge (D \rightarrow C)$. A particular valuation of $\Phi^{may}_{M_1}$, namely, $\langle A = \mathsf{True}, B = C = D = \mathsf{False}\rangle$, corresponds to the classical model $m_{1,2}$, shown in Figure 2(b).

Every classical model is a partial model where all terms are annotated by True or False and the May formula is True.

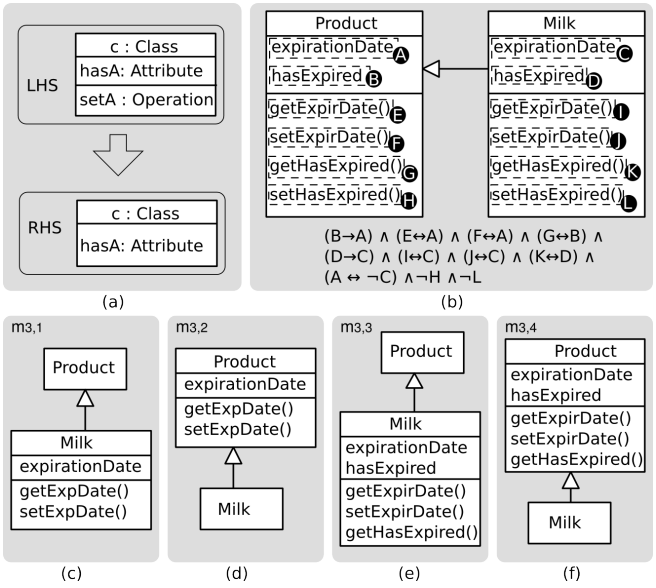

Figure 3. (a) Refactoring transformation rule R_2 for removing setters of derived attributes. (b) The partial model M_3 resulting from applying R_2 to M_2. (c-f) Concretizations of M_3.

A partial model (and thus every classical model as well!) can be encoded entirely as a propositional formula [6]. This is done by conjoining its May Formula with a conjunction of the model's True and False elements. For the model M_1, this propositional encoding is the conjunction $\Phi_{M_1} = \texttt{Product} \wedge \texttt{Milk} \wedge \texttt{gen\_Milk\_Product} \wedge \ldots \wedge \Phi^{may}_{M_1}$, where `gen_Milk_Product` represents the generalization between `Milk` and `Product`.[3] Moreover, the propositional formula for a partial model can be obtained by computing a disjunction of encodings for each of its concretizations.

The set of all variables in such a propositional encoding is called the *scope* or *vocabulary* of the partial model. It is possible to *embed* a partial model into a larger scope, i.e., one that contains elements not present in the original. The new elements simply appear negated in the propositional encoding. For example, M_1 can be embedded into the scope of M_2, in which case its propositional encoding becomes $\Phi_{M_1} \wedge \neg E \wedge \ldots \wedge \neg L$.

III. Partial Transformations

In the motivating scenario presented in Section I, the modeler uses graphical rules to express transformations. Traditionally, rules and their applications are defined as follows:

Definition 1: A *transformation rule* R is a tuple $\langle$LHS, RHS$\rangle$, where the typed, attributed graphs LHS, RHS are respectively called the *left-hand* and *right-hand* sides of the rule.[4]

Definition 2: A rule $R = \langle$LHS, RHS$\rangle$ is *applied* to a model M by finding all *matches* of LHS in M and replacing all *matched sites* with RHS, to produce the output model N. That is, a rule R can be applied to an input model M to produce a new model N. We denote this as $M \overset{R*}{\Longrightarrow} N$.

While inspired from algebraic graph transformation [3], we adopt a purely logic-based perspective of model transformation. A transformation is accordingly captured as a *transfer predicate* that relates elements of the input and the output models. In this section, we outline our approach for encoding and applying transformation rules as transfer predicates.

A. Representing Transformations

A transformation rule can be encoded in logic:

Definition 3: Given a rule $R = \langle$LHS,RHS$\rangle$, where $\mathbf{\Phi_L}, \mathbf{\Phi_R}$ are propositional formula patterns, that logically encode the rule's LHS and RHS respectively, the *transfer predicate* $\mathcal{R}(R)$ encodes the relationship between the variables of $\mathbf{\Phi_L}$ and those of $\mathbf{\Phi_R}$ as follows: $\mathcal{R}(R) = (\mathbf{\Phi_L} \rightarrow \mathbf{\Phi_R}) \wedge (\neg\mathbf{\Phi_L} \rightarrow \mathbf{\Phi}_{NE})$, where $\mathbf{\Phi}_{NE}$ describes the case where the rule has "no effect". Matching is done by unifying the variables of $\mathbf{\Phi_L}, \mathbf{\Phi_R}$ with those of the model at each particular site.

That is, the rule says that if the LHS applies, then in the result (denoted by primed variables), RHS should hold. Otherwise, values of variables are left unchanged.

Systematic construction of transfer predicates is still work in progress. In the following, we outline the manual construction of transfer predicates for the two rules of the motivating scenario.

An Example Adding Rule. The rule R_1, shown in Figure 1(a), is a refactoring rule that adds getters and setters for attributes in classes that don't have them. Overall, the rule has four propositional variables: a class c, an attribute a, a getter g and a setter s. The rule's LHS explicitly refers to the first two variables, c and a whereas the variables g, s are implicitly set to False (i.e., the rule does not match sites where the getter and the setter already exist). The rule's RHS sets the variables of g', s' depending on the value of a.

A matched site is captured by $\mathbf{\Phi_L} = c \wedge a \wedge \neg g \wedge \neg s$. It is evident that *at each matched site*, the transfer predicate includes the expression $\mathbf{\Phi'_R} = (c' \leftrightarrow c) \wedge (a' \leftrightarrow a) \wedge (g' \leftrightarrow a) \wedge (s' \leftrightarrow a)$[5]. For each model element x that is not matched by the rule, the transfer predicate is $\mathbf{\Phi}_{NE} = (x' \leftrightarrow x)$.

An Example Deleting Rule. The rule R_2, shown in Figure 3(a), deletes the setters for derived attributes, assuming that the naming convention "`hasX`" is followed. This rule has three propositional variables: a class c, a derived attribute h and a setter s. The RHS of the rule implicitly sets s' to False.

A matched site is captured by $\mathbf{\Phi_L} = c \wedge h \wedge s$. At each matched site, the transfer predicate is $\mathbf{\Phi'_R} = (c' \leftrightarrow c) \wedge (h' \leftrightarrow h) \wedge \neg s'$. For each model element x that is not matched by the rule, the transfer predicate is again $\mathbf{\Phi}_{NE} = (x' \leftrightarrow x)$.

[3] We have omitted the ownership relationships between attributes and classes for brevity.

[4] We simplify the problem by ignoring negative application conditions [3]

[5] The operator $\leftrightarrow$ is interpreted as equality.

B. Applying Transformations

When using the logical representations of models and transfer predicates, rule *application* is defined as:

Definition 4: A rule $R = \langle LHS, RHS \rangle$, applied to a (partial) model M to produce the output model N, corresponds to the equation $\Phi'_N = \mathcal{R}(R, M, N) \wedge \Phi_M$, where $\mathcal{R}(R, M, N)$ is a *transfer predicate*, expressed in first-order logic, that associates the (primed) elements of N with the (unprimed) elements of M. The transfer predicate $\mathcal{R}(R, M, N)$ is constructed by unifying the predicate $\mathcal{R}(R)$ with each site in M and N.

Multiple applications of rules can be achieved by the construction: $\Phi^n = \Phi^{n-1} \wedge \mathcal{R}(R_n, M^{n-1}, M^n)$.

In this paper, we assume for simplicity that the vocabulary of the output model is known at the time of rule application. With this assumption, the Transfer Predicate is propositionalized over the union of the vocabularies of M and N.

Rule Application for Classical Models: Example. We illustrate Definition 4 by creating the transfer predicate for applying the rule R_1 to the model $m_{1,1}$ in Figure 2(a). The construction is done manually by doing the matching and grounding over the union of the vocabularies of $m_{1,1}$ and the model $m_{2,1}$ in Figure 2(e). The rule matches the variables $\langle c, a, g, s \rangle$ at the matching site $\langle \texttt{Milk}, C, I, J \rangle$. Thus, after simplification, the grounded transfer predicate becomes

$$\begin{aligned}\mathcal{R}(R_1, m_{1,1}, m_{2,1}) \quad &= (\texttt{Milk}' \leftrightarrow \texttt{Milk}) \wedge (C' \leftrightarrow C) \wedge (I' \leftrightarrow C) \wedge \\ &(J' \leftrightarrow C) \wedge (\texttt{Product}' \leftrightarrow \texttt{Product}) \wedge \\ &(\texttt{gen\_Milk\_Product}' \leftrightarrow \texttt{gen\_Milk\_Product})\end{aligned}$$

The last two clauses of the conjunction are the elements in the model that are unaffected by the transformation. Given the propositional encoding $\Phi_{m_{1,1}}$ of the model $m_{1,1}$, the production $m_{1,1} \overset{R_1\,*}{\Longrightarrow} m_{2,1}$ is consequently encoded as

$$\Phi_{m_{2,1}} = \mathcal{R}(R_1, m_{1,1}, m_{2,1}) \wedge \Phi_{m_{1,1}}$$

This mechanism for applying transformations is directly applicable to the transformation of partial models. The only difference is that partial models have somewhat more complex encoding as propositional formulas. We illustrate on two examples below.

Example Adding Rule (Cont'd). We begin by encoding the production $M_1 \overset{R_1\,*}{\Longrightarrow} M_2$. Using the union of the vocabularies of the models M_1, M_2, shown in Figures 1(b-c), we manually perform the matching and propositionalize the transfer predicate at the possible matching sites in M_1. For the variables $\langle c, a, g, s \rangle$, these are four matches: $\{\langle \texttt{Product}, A, E, F \rangle, \langle \texttt{Product}, B, G, H \rangle, \langle \texttt{Milk}, C, I, J \rangle, \langle \texttt{Milk}, D, K, L \rangle\}$. The only element unaffected by the rule in this case is the generalization `gen_Milk_Product`). After simplifying and removing duplicates, the transfer predicate becomes:

$$\begin{aligned}\mathcal{R}(R_1, M_1, M_2) \quad &= (\texttt{Product}' \leftrightarrow \texttt{Product}) \wedge (\texttt{Milk}' \leftrightarrow \texttt{Milk}) \wedge \\ &(A' \leftrightarrow A) \wedge (B' \leftrightarrow B) \wedge (C' \leftrightarrow C) \wedge \\ &(D' \leftrightarrow D) \wedge (E' \leftrightarrow A) \wedge (F' \leftrightarrow A) \wedge \\ &(G' \leftrightarrow B) \wedge (H' \leftrightarrow B) \wedge (I' \leftrightarrow C) \wedge \\ &(J' \leftrightarrow C) \wedge (K' \leftrightarrow D) \wedge (L' \leftrightarrow D) \wedge \\ &(\texttt{gen\_Milk\_Product}' \leftrightarrow \texttt{gen\_Milk\_Product})\end{aligned}$$

Using the propositional encoding Φ_{M_1} of the partial model M_1, the production $M_1 \overset{R_1\,*}{\Longrightarrow} M_2$ is thus encoded as

$$\Phi_{M_2} = \mathcal{R}(R_1, M_1, M_2) \wedge \Phi_{M_1}$$

Example Deleting Rule (Cont'd). We now demonstrate the application of rule R_2 to M_2. Using the vocabulary of M_2 and manually doing the match for $\langle c, h, s \rangle$, we get two matching sites $\{\langle \texttt{Product}, B, H \rangle, \langle \texttt{Milk}, D, L \rangle\}$. The transfer predicate becomes

$$\begin{aligned}\mathcal{R}(R_2, M_2, M_3) \quad &= (\texttt{Product}' \leftrightarrow \texttt{Product}) \wedge (\texttt{Milk}' \leftrightarrow \texttt{Milk}) \wedge \\ &(B' \leftrightarrow B) \wedge (D' \leftrightarrow D) \wedge \neg H' \wedge \neg L' \wedge \Phi_{NE}\end{aligned}$$

Here we used Φ_{NE} to denote a conjunction of terms of the form $(X' \leftrightarrow X)$ for each model element that was not matched by the rule, such as `expirationDate`, `getHasExpired`, etc. The production $M_2 \overset{R_2\,*}{\Longrightarrow} M_3$ is encoded as

$$\Phi_{M_3} = \mathcal{R}(R_2, M_2, M_3) \wedge \Phi_{M_2}$$

using the propositional encoding Φ_{M_2} of the partial model M_2.

In this section, we showed how to encode an outcome of applying a transformation to models, partial and classical. This gives as *semantics of partial transformations* and answers question Q2 posted in Section I. Our treatment was based on knowing what the outcome of the transformation is supposed to be, since the output model was an integral part of computing the transfer predicate. In the next section, we discuss what such output model should be (question Q1).

IV. Towards Verification of Correctness

In this section, we define a criterion that lifted transformations should follow and check correctness of adding and deleting rules in our running example.

Correctness Condition. Partial models are intended to be exact representations of sets of models and lifted transformations must preserve this. This means that applying a lifted transformation R to a partial model M should be equivalent to applying its classical version to each of the concretizations of M and building a partial model from the result. We refer to this principle as the *Correctness Criterion* for lifting transformations. More formally:

Definition 5: Given a rule R, a partial model M with a set of concretizations $[M] = \{m_1, \ldots, m_n\}$, and the set $U = \{m'_i | \forall m_i \in [M] \cdot m_i \overset{R\,*}{\Longrightarrow} m'_i\}$, the production $M \overset{R\,*}{\Longrightarrow} N$ is *correct* iff the set of concretizations of the resulting partial model M' satisfies the condition $[N] = U$.

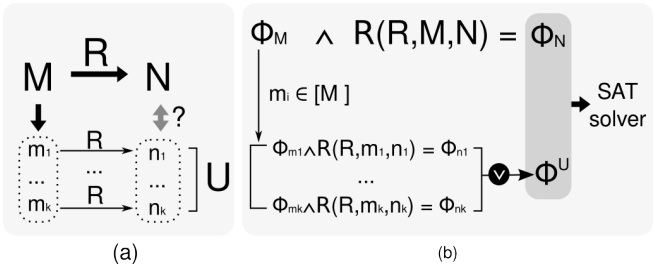

Figure 4. Testing the rule application $M \overset{R*}{\Longrightarrow} N$ for correctness of lifting: (a) model view; (b) logical view.

Alternatively, the resulting model N is obtained by building a partial model from the set U. Note that this correctness criterion holds independently of how we choose to encode productions!

Testing Correctness. Using this criterion, we can *test* whether a production $M \overset{R*}{\Longrightarrow} N$ is correct with respect to lifting. Testing is done in three steps: (1) create the model N by applying the rule, (2) create the set U by appling the rule to each concretization m of M, (3) compare the sets $[N]$ and U. The process is summarized in Figure 4(a). The reason why we call this testing rather than verification is that we can only establish that the production is correct *w.r.t. an input model* M.

In our motivating example, correctness of our encoding of the production $M_1 \overset{R_1*}{\Longrightarrow} M_2$ can be tested by checking whether the set of concretizations $[M_2]$ is the same as the set of models that is constructed by individually transforming the set of concretizations $[M_1]$. Indeed, by individually transforming the set $\{m_{1,1}, m_{1,2}, m_{1,3}, m_{1,4}\}$ of concretizations of M_1, shown in Figures 2(a-d), we arrive to the set of models $\{m_{2,1}, m_{2,2}, m_{2,3}, m_{2,4}\}$, shown in Figures 2(e-h), which is exactly the set $[M_2]$.

Towards Tool Support. Our encoding of transformations using the transfer predicate allows us to automatically check correctness of transformations using a SAT solver.

In particular, given a production $M \overset{R*}{\Longrightarrow} N$, we compare the sets of valuations of the following two formulas: (a) the rule application formula $\Phi_N = \mathcal{R}(R, M, N) \wedge \Phi_M$ and (b) the formula $\Phi^U = \Phi_{m'_1} \vee \ldots \vee \Phi_{m'_k}$. Φ^U encodes the set U of individually transformed concretizations. Each such concretization m'_i is encoded by the formula $\Phi_{m'_i} = \mathcal{R}(R, m_i, m'_i) \wedge \Phi_{m_i}$. The process is summarized in Figure 4(b). We illustrate this verification approach using our example.

In order to check the correctness of the production $M_1 \overset{R_1*}{\Longrightarrow} M_2$, we first construct the formula $\Phi_{M_2} = \mathcal{R}(R_1, M_1, M_2) \wedge \Phi_{M_1}$, as discussed in Section III-B. Subsequently, we create the formula Φ_2^U which represents the set of transformed concretizations $\{m_{2,1}, m_{2,2}, m_{2,3}, m_{2,4}\}$, that are shown in Figures 2(e-h).

We used these formulas as inputs to MathSAT4 [1], running the SAT solver in ALLSAT mode. Comparing the valuations produced by MathSAT4 for each formula, we verified that the two formulas have the same sets of possible valuations, thus satisfying the correctness criterion.

Similarly, we tested the production $M_2 \overset{R_2*}{\Longrightarrow} M_3$. We compared the valuations of the formulas $\Phi_{M_3} = \mathcal{R}(R_2, M_2, M_3) \wedge \Phi_{M_2}$ and Φ_3^U. The latter was constructed similarly to Φ_2^U above for the models $\{m_{3,1}, m_{3,2}, m_{3,3}, m_{3,4}\}$, shown in Figures 3(c-f). Using MathSAT4, we verified that the sets of valuations of these formulas coincide, thus satisfying the correctness criterion.

For these small examples, the runtimes of the SAT solver were negligible. As discussed in Section V, we do expect scalability issues for transformations of partial models with larger sets of concretizations and/or more sites of application. The overall process is expensive, due to the fact that in order to do a test, we have to enumerate and transform each concretization of the input model.

V. Summary and Discussion

Transforming partial models is an important component of the research agenda for partial models that we presented in [5], which aims to build a comprehensive framework for explicitly handling uncertainty in the software engineering lifecycle. In Section I, we presented the problem of transforming partial models in terms of two basic questions: (Q1) what should the outcome look like, and (Q2) what mechanics would produce such an outcome. We have outlined our preliminary thoughts on this: In Section IV, we defined a correctness criterion to answer (Q1). In Section III, we presented our approach to addressing (Q2), using a logic-based approach for defining the semantics of transformations that takes into account partial models. We further experimented with two examples of transformations, an additive and a deleting transformation. In this section, we draw from this experience to outline the major challenges that we have identified and discuss future steps.

In Section IV, we used the correctness criterion to test the lifting of our example transformations, using SAT. Certain important limitations arose from these examples:

1) Testing the production, e.g. $M_1 \overset{R_1*}{\Longrightarrow} M_2$ requires the construction of the formula Φ_2^U that encodes the individually transformed concretizations of M_1. Essentially, this amounts to the explicit construction and enumeration of all the concretizations of M_2. This corresponds to *thorough* checking [7] and can be very expensive, especially if the set of concretizations of M_1 is large or if there are many possible matching sites.
2) The criterion can only be used to *test* the application of a lifted transformation for specific input and output models. We should be able to *verify* that a transformation is correctly lifted, regardless of specific inputs and outputs.

We are currently working on alternative approaches to verifying lifted transformations compositionally. For this, we aim to use an approach centered on proving transformations correct, based on identifying *proof obligations*, similar to the

approach we developed in [10] for verifying the special class of uncertainty-reducing partial model refinements.

A logic-based lifting semantics for partial model transformation was introduced in Section III and illustrated on our two examples. Our preliminary attempts point us to identify the following challenges:

1) We manually constructed the transfer predicates for the particular examples we presented. Similarly, rule application was also done manually (matching and propositionalization of the transfer predicates). We are currently working on systematizing the creation of transfer predicates using First-Order Logic. Given a classical transformation, we want to create the transfer predicate using a predefined set of reusable modular predicates, such as MatchNode() AddNode(), DeleteEdge(), etc. This will allow us to support more sophisticated features of model transformation, such as Negative Application Conditions. At the same time, this will help streamline the lifting process and form the basis for effective tool support for automatic matching and propositionalization.
2) In our examples, we required knowledge of the vocabulary of the output model prior to rule application, especially in the case of additive transformations. This is, naturally, not always the case in real-world settings. A similar problem occurs with deleting transformations: deleted elements remain in the vocabulary, albeit negated. We are working on a more systematic approach to expand and contract vocabularies and, more specifically, on ways to treat transformations that affect changes to the vocabulary.
3) Applying transformations using the transfer predicate results in a formula that is in terms of the original model in the form of "unprimed variables". Converting this to a representation that is useful to the user requires us to construct an expression that is solely expressed in terms of the resulting model. To do this, unprimed variables need to be factored out, which can be expensive.

An alternative approach, that would address the last concern, but also provide a more general solution, would be to define partial model transformations in terms of the Double Pushout (DPO) approach [2]. In [3], it is proven that the DPO can be applied to any Adhesive High-Level Replacement Category (AHLR) [4]. We have made some initial attempts at formalizing partial models as an AHLR, with the biggest challenge being the identification of the proper morphisms.

In conclusion, in this paper we have outlined the problem of partial model transformation and identified its main parameters in terms of correctness and mechanics. We have illustrated our preliminary approach in two examples of basic additive and deleting transformations. Using this experience we were able to identify the main challenges and outline next steps.

REFERENCES

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

[2] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Lowe. Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach. In *Handbook of graph grammars and computing by graph transformation*, pages 163–245. World Scientific Publishing Co., Inc., 1997.

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

[4] H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive High-Level Replacement Categories and Systems. *Graph Transformations*, 3256:144–160, 2004.

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

[6] M. Famelis, M. Chechik, and R. Salay. "Partial Models: Towards Modeling and Reasoning with Uncertainty". In *Proceedings of ICSE'12*, 2012. To appear.

[7] A. Gurfinkel and M. Chechik. "How Thorough Is Thorough Enough?". In *Proceedings of CHARME'05*, pages 65–80, 2005.

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

[9] M. Petre. "Insights from Expert Software Design Practice". In *Proceedings of FSE'09*, 2009.

[10] R. Salay, M. Chechik, and J. Gorzny. "Towards a Methodology for Verifying Partial Model Refinements", 2012. submitted.

[11] R. Salay, M. Famelis, and M. Chechik. "Language Independent Refinement using Partial Modeling". In *Proceedings of FASE'12*, 2012.