Graph Constraint Evaluation over Partial Models by Constraint Rewriting

Oszkár Semeráth and Dániel Varró

Budapest University of Technology and Economics
MTA-BME Lendület Research Group on Cyber-Physical Systems
McGill University, Department of Electrical and Computer Engineering
{semerath,varro}@mit.bme.hu

Abstract. In the early stages of model driven development, models are frequently incomplete and partial. Partial models represent multiple possible concrete models, and thus, they are able to capture uncertainty and possible design decisions. When using models of a complex modeling language, several well-formedness constraints need to be continuously checked to highlight conceptual design flaws for the engineers in an early phase. While well-formedness constraints can be efficiently checked for (fully specified) concrete models, checking the same constraints over partial models is more challenging since, for instance, a currently valid constraint may be violated (or an invalid constraint may be respected) when refining a partial model into a concrete model.
In this paper we propose a novel technique to evaluate well-formedness constraints on partial models in order to detect if (i) a concretization may potentially violate or (ii) any concretization will surely violate a well-formedness constraint to help engineers gradually to resolve uncertainty without violating well-formedness. For that purpose, we map the problem of constraint evaluation over partial models into a regular graph pattern matching problem over complete models by semantically equivalent rewrites of graph queries.

1 Introduction

Model-Driven Engineering (MDE) is a widely used technique in many application domains such as automotive, avionics or other cyber-physical systems [36]. MDE facilitates the use of models in different phases of design and on various levels of abstraction. These models enable the automated synthesis of various design artifacts (such as source code, configuration files, documentation) and help catch design flaws early by model validation techniques. Model validation highly depends on repeatedly checking multiple design rules and well-formedness constraints captured in the form of graph constraints [21,3,17] over large (graph) models to highlight violating model elements to systems engineers.

During the early phase of development as well as in case of software product line engineering, the level of uncertainty represented in the models is still high, which gradually decreases as more and more design decisions are made.

To support uncertainty during modeling, a rich formalism of partial models has been proposed in [10] which marks model elements with four special annotations (namely, may, set, variable and open) with well defined semantics. During the design, these partial models can then be concretized into possible design candidates [28,30].

However, evaluating well-formedness constraints over partial models is a challenging task. While existing graph pattern matching techniques provide efficient support for checking well-formedness constraints over regular model instances [17,20,34,7], SMT/SAT solvers have been needed so far to evaluate the same constraints over partial models, which have major scalability problems [30].

Our objective is to evaluate well-formedness constraints over partial models by graph pattern matching instead of SAT/SMT solving, which poses several conceptual challenges. First, a single node in a graph constraint may be matched to zero or more nodes in a concretization of a partial model. Moreover, graph constraints need to be evaluated over partial models with open world semantics as new elements may be added to the model during concretization.

In the paper, we propose (i) a new partial modeling formalism based on 3-valued logic [16], (ii) a mapping of a popular partial modelling technique called MAVO [10] into 3-valued partial models, and (iii) and novel technique that rewrites the original graph constraints (to be matched over partial models) into two graph constraints to be matched on 3-valued partial models. One constraint will identify matches that *must* exist in all concretizations of the partial model while the other constraint will identify matches that *may* exist. Although the complexity of the pattern increases by the proposed rewrite, we can still rely upon efficient existing graph pattern matching techniques for evaluating the constraints, which is a major practical benefit. As a result, engineers can detect if concretizations of a partial model will (surely) violate or may (potentially) violate a well-formedness constraint which helps them gradually to resolve uncertainty. Our approach is built on top of mainstream modeling technologies: Partial models are represented in Eclipse Modeling Framework [33] annotated in accordance with [10], well-formedness constraints are captured as graph queries [3].

The rest of the paper is structured as follows: Section 2 summarizes core modeling concepts of partial models and queries in the context of a motivating example. Section 3 provides an overview on 3-valued partial models with a graph constraint evaluation technique. Section 4 provides initial scalability evaluation of the approach, Section 5 overviews related approaches available in the literature. Finally, Section 6 concludes the paper.

2 Preliminaries

2.1 Motivating example: Validation of Partial Yakindu Statecharts

Yakindu Statechart Tools [37] is an industrial integrated development environment (IDE) for developing reactive, event-driven systems captured by statecharts using a combined graphical and textual syntax.

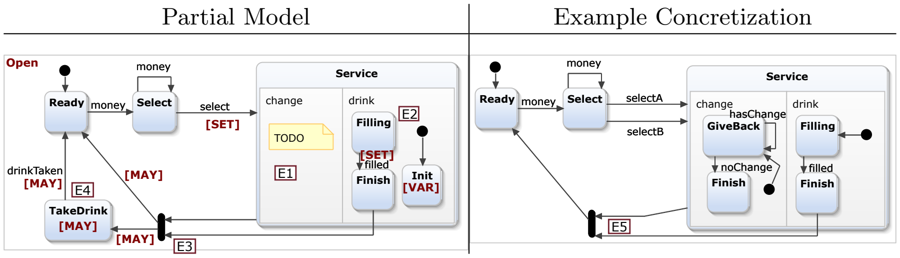


Fig. 1: A partial statechart model and a sample concretization.

A partial model of a coffee machine is illustrated on the left part of Figure 1 together with a sample concrete model on the right. Initially, the machine starts in state Ready and after inserting coins by money events, a drink can be selected in state Select. While multiple concrete drink options may be available in the concrete model (like selectA and selectB), but in the partial model each one is represented by a generic select event. After the selection, the machine starts filling coffee, and gives back the change in state Service. The change management region is missing in the partial model, while a drink preparation region already contains some details. As the developer is uncertain about the initial state in this region, a placeholder state Init is created. In the partial model, it is undecided if it is required to wait until the previous drink is taken (in state TakeDrink), or the machine can enter its initial Ready state immediately.

These uncertainties are captured by special annotations introduced in [10] such as may (elements can be omitted), var (elements that can be merged), set (representing sets of elements) or open (new elements can be added).

The Yakindu IDE checks several well-formedness rules on the statecharts:

C_1 Each region shall have exactly one entry, which has a transition to a state in the same region.

C_2 The target and source states of a synchronization shall be contained in the same parent state.

Both constraints can be defined (e.g. in OCL [21] or graph constraints [3]) and checked over complete models, but our paper focuses on detecting (potential and certain) conceptual errors (marked by E1-4 in Figure 1) in partial models.

E1 marks that an entry state is missing from region change, thus violating C_1. However, as the model is under construction, it can be repaired in a later stage. The other region (marked by E2) already contains an entry state, thus the WF constraint is currently satisfied, but it can potentially be violated in a future refinement by connecting it to a state located in a different region. E3 shows evidence of an invalid synchronization of parallel states Finish and its parent Service violating C_2. This error will be present in *all possible concretizations* (or completions) of the partial model, e.g. as E5 in Figure 1. Finally, E4 marks a possible error for synchronizing two target states that are not parallel (TakeDrink and Ready if all may elements are preserved).

2.2 Metamodels and Instance Models

A domain-specific (modeling) language (DSL) is typically defined by a *metamodel* and several *well-formedness constraints*. A metamodel defines the main concepts and relations in a domain, and specifies the basic graph structure of the models. In this paper, domain models are captured by the Eclipse Modeling Framework (EMF) [33], which is widely used in industrial modeling tools including Yakindu statecharts.

A metamodel defines a vocabulary $\Sigma = \{\mathtt{C}_1, \ldots, \mathtt{C}_n, \mathit{exist}, \mathtt{R}_1, \ldots, \mathtt{R}_m, \sim\}$ where a unary predicate symbol $\mathtt{C}_i$ $(1 \leq i \leq n)$ is defined for each *EClass*, and a binary predicate symbol $\mathtt{R}_j$ $(1 \leq j \leq m)$ is derived for each *EReference*. Moreover, we define a unary *exist* predicate to denote the existence of an object in a given model, while $\sim$ denotes an equivalence relation over objects. For a set of unique *Id* constants, $id_1, \ldots, id_k$, $k \in \mathbb{Z}^+$ well-formed terms can be constructed as $\mathtt{C}(id_i)$, $\mathit{exist}(id_i)$, $\mathtt{R}(id_i, id_j)$ and $id_i \sim id_j$, where $1 \leq i, j \leq k$. For space considerations, we omit the precise handling of attributes from this paper, which could be introduced accordingly.

An *instance model* can formally be represented as a logic structure $M = \langle \mathit{Obj}_M, \mathcal{I}_M \rangle$ where Obj_M is the finite, nonempty set of individuals in the model (i.e. the objects), and $\mathcal{I}_M$ provides interpretation for all constants in *Id* and predicate symbols in Σ as follows:

- the interpretation of a constant id_i (denoted as $\mathcal{I}_M(\mathtt{id}) : \mathit{Id} \to \mathit{Obj}_M$) is an element from Obj_M;
- the 2-valued interpretation of a unary predicate symbol $\mathtt{C}_i$ (and similarly *exist*) is defined in accordance with the existence of objects in the EMF model and denoted as $\mathcal{I}_M(\mathtt{C}_i) : \mathit{Obj}_M \to \{1, 0\}$;
- the 2-valued interpretation of a binary predicate symbol $\mathtt{R}_j$ (and also $\sim$) is defined in accordance with the links in the EMF model and denoted as $\mathcal{I}_M(\mathtt{R}_j) : \mathit{Obj}_M \times \mathit{Obj}_M \to \{1, 0\}$;

For a notational shortcut, we will use $\mathit{exist}_M(x)$ instead of $\mathcal{I}_M(\mathit{exist})(x)$ and $x \sim_M y$ in place of $\mathcal{I}_M(\sim)(x, y)$ (where x and y could be variables or constants). For a *simple* instance model M we assume the following S1-4 properties:

S1 $\forall o \in \mathit{Obj}_M$: $(o \sim_M o) > 0$ (reflexive).
S2 $\forall o \in o_1, o_2 \in \mathit{Obj}_M$: $(o_1 \sim_M o_2) = (o_2 \sim_M o_1)$ (symmetric)
S3 $\forall o_1, o_2 \in \mathit{Obj}_M$: $(o_1 \neq o_2) \Rightarrow (o_1 \sim_M o_2 < 1)$ (unique objects)
S4 $\forall o \in \mathit{Obj}_M$: $\mathit{exist}_M(o) > 0$ (model does not contain not existing objects)

Which means that in a simple instance model M, $\sim$ is the same as $=$, and for all objects o existence predicate is always evaluated to true: $\mathit{exist}_M(o) = 1$.

2.3 Graph Patterns as Logic Formulae

In many industrial modeling tools, WF constraints are captured either by standard OCL constraints [21] or alternatively, by graph patterns (GP) [17,20,3],

$$
\begin{array}{rlrl}
\llbracket \mathtt{C}(v) \rrbracket_Z^M & := \mathcal{I}_M(\mathtt{C})(Z(v)) & \llbracket \mathtt{R}(v_1, v_2) \rrbracket_Z^M & := \mathcal{I}_M(\mathtt{R})(Z(v_1), Z(v_2)) \\
\llbracket exist(v) \rrbracket_Z^M & := \mathcal{I}_M(exist)(Z(v)) & \llbracket v_1 \sim v_2 \rrbracket_Z^M & := \mathcal{I}_M(\sim)(Z(v_1), Z(v_2)) \\
\llbracket \varphi_1 \land \varphi_2 \rrbracket_Z^M & := min(\llbracket \varphi_1 \rrbracket_Z^M, \llbracket \varphi_2 \rrbracket_Z^M) & \llbracket \varphi_1 \lor \varphi_2 \rrbracket_Z^M & := max(\llbracket \varphi_1 \rrbracket_Z^M, \llbracket \varphi_2 \rrbracket_Z^M) \\
\llbracket \neg \varphi \rrbracket_Z^M & := 1 - \llbracket \varphi \rrbracket_Z^M & &
\end{array}
$$

$$
\llbracket \exists v : \varphi \rrbracket_Z^M := max\{\llbracket exist(x) \land \varphi \rrbracket_{Z, v \mapsto x}^M : x \in Obj_M\}
$$

$$
\llbracket \forall v : \varphi \rrbracket_Z^M := min\{\llbracket \neg exist(x) \lor \varphi \rrbracket_{Z, v \mapsto x}^M : x \in Obj_M\}
$$

Fig. 2: Semantics of graph logic expressions (defined inductively)

which provide an expressive formalism. A graph pattern (or constraint) captures structural conditions over an instance model as paths in a graph. In order to have a unified and semantically precise handling of evaluating graph patterns for regular and partial models, we use a tool-independent logic representation (which was influenced by [35,25]) that covers the key features of several concrete graph pattern languages.

Syntax. Syntactically, a graph pattern is a first order predicate $\varphi(v_1, \ldots, v_n)$ over (object) variables. A graph formula φ can be inductively constructed (see Figure 2) by using class and relation predicates $\mathtt{C}(v)$ and $\mathtt{R}(v_1, v_2)$, equivalence check $=$, standard first order logic connectives $\neg$, $\lor$, $\land$, and quantifiers $\exists$ and $\forall$.

Semantics. A predicate $\varphi(v_1, \ldots, v_n)$ can be evaluated on model M along a variable binding Z, which is a mapping $Z : \{v_1, \ldots, v_n\} \to Obj_M$ from variables to objects in M. The truth value of φ can be evaluated over model M and Z (denoted by $\llbracket \varphi(v_1, \ldots, v_n) \rrbracket_Z^M$) in accordance with the semantic rules defined in Figure 2. Note that *min* and *max* takes the numeric minimum and maximum values of 0 and 1, and the rules follow the construction of standard first order logic formulae as used in [35,25].

A variable binding Z where the predicate φ is evaluated to 1 over M is often called a *pattern match*, formally $\llbracket \varphi \rrbracket_Z^M = 1$. Otherwise, if there are no bindings Z to satisfy a predicate, i.e. $\llbracket \varphi \rrbracket_Z^M = 0$ for all Z, then the predicate φ is evaluated to 0 over M. Graph query engines like [3,5] can retrieve (one or all) matches of a pattern over a model. When using graph patterns for validating WF constraints, a match of a pattern usually denotes a violation, thus the corresponding graph formula needs to capture the erroneous case.

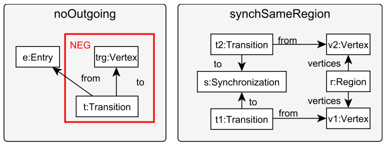
Example. To capture the erroneous case as a pattern match, the WF constraints C_1 and C_2 from the Yakindu documentation need to be reformulated as follows:

φ_{1a} There is an entry state without an outgoing transition.

φ_{1b} There is an entry state with a transition to a vertex in a different region.

φ_2 The target and source states of a synchronization are contained in different regions of the same parent state.

Graph patterns and the corresponding logic formulae for φ_{1a} and φ_2 are depicted in Figure 3. With a negative condition (marked by NEG) in `noOutgoing`,



$\texttt{noOutgoing}(e) :=$
$\texttt{Entry}(e) \wedge \neg\exists t, trg : \texttt{from}(t, e) \wedge \texttt{to}(t, trg)$
$\texttt{synchSameRegion}(s) := \exists t_1, t_2, v_1, v_2, r : \texttt{Synchronization}(s) \wedge \texttt{from}(t_1, v_1) \wedge \texttt{to}(t_1, s) \wedge \texttt{from}(t_2, v_2) \wedge \texttt{to}(t_2, s) \wedge \texttt{vertices}(r, v_1) \wedge \texttt{vertices}(r, v_2) \wedge \neg t_1 \sim t_2$

Fig. 3: Sample graph patterns for statecharts with their equivalent logic formula

Entry states can be detected without any outgoing transitions. Moreover pattern synchSameRegion searches for synchronizations between vertices v_1 and v_2 which are in the same region.

3 Formalism of 3-Valued Partial Models

Partial modeling [10,15,29] is a generic technique to introduce uncertainty into instance models. Semantically, one abstract partial model represents a range of possible instance models, which are called *concretizations*. During the development, the level of uncertainty can be gradually reduced by *refinements*, which results in partial model with less concretizations. In the following we present a novel 3-valued partial modeling formalism, and give a method to evaluate graph patterns on it.

3.1 Properties of 3-Valued Logic

In this paper 3-valued logic [16,25] is used to explicitly represent unspecified or unknown properties of the models with a third $1/2$ logic value (beside 1 and 0 which means a value must be true or false). During a refinement, $1/2$ properties are gradually refined to either 0 or 1. This refinement is defined by an information ordering relation $X \sqsubseteq Y$, which specifies that either $X = 1/2$ and Y refined to a more specific 1 or 0, or $X = Y$.

$$X \sqsubseteq Y := (X = 1/2) \vee (X = Y)$$

Information ordering $X \sqsubseteq Y$ has two important properties: first, if we know that $X = 1$ then it can be deduced that Y must be 1, and secondly, if $Y = 1$ then $X \geq 1/2$ (i.e. 1 or $1/2$). Those two properties will be used to approximate possible values of a concrete model by checking the property on a partial model.

3.2 Partial Models based on 3-Valued Logic

In this paper we propose a generic, 3-valued partial modeling formalism. A partial model of the same vocabulary $\Sigma = \{\texttt{C}_1, \ldots, \texttt{C}_n, exist, \texttt{R}_1 \ldots \texttt{R}_m, \sim\}$ is a 3-valued logic structure $P = \langle Obj_P, \mathcal{I}_P \rangle$, where Obj_P is the finite set of symbolic objects, and $\mathcal{I}_P$ provides 3-valued interpretation for all constants in Id and predicate symbols in Σ.

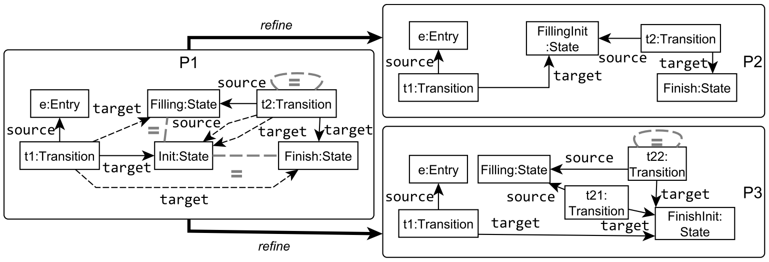


Fig. 4: Example 3-valued partial model with refinements

Uncertain Types. $\mathcal{I}_P$ gives a 3-valued interpretation to each *EClass* symbol $\mathtt{C}_i$ in Σ: $\mathcal{I}_P(\mathtt{C}_i) : \mathit{Obj}_P \to \{1, 0, 1/2\}$, where an $1/2$ value represents a case where it is unknown if an object has a type `C` or not.

Uncertain References. $\mathcal{I}_P$ gives a 3-valued interpretation to each *EReference* symbol $\mathtt{R}_j$ in Σ: $\mathcal{I}_P(\mathtt{R}_j) : \mathit{Obj}_P \times \mathit{Obj}_P \to \{1, 0, 1/2\}$. An uncertain $1/2$ value represent possible references.

Uncertain Equivalence. $\mathcal{I}_P$ gives a 3-valued interpretation for the equivalence relation $\mathcal{I}_P(\sim) : \mathit{Obj}_P \times \mathit{Obj}_P \to \{1, 0, 1/2\}$. An uncertain $1/2$ value relation between two objects means that the object might be equals and they can be potentially merged. For an object o where $o \sim_P o = 1/2$ it means that the object may represent multiple different objects, and can be split later on.

Uncertain Existence. $\mathcal{I}_P$ gives a 3-valued interpretation for the existence relation $\mathcal{I}_P(\mathit{exist}) : \mathit{Obj}_P \to \{1, 0, 1/2\}$, where an $1/2$ value represents objects that may be removed from the model.

The simplicity requirements S1-4 defined on page 4 are also assumed on partial models, which, in this case, allow uncertain $1/2$ equivalences and existence.

Figure 4 illustrates three partial models, where P_1 shows a submodel of the coffee machine from Figure 1. The objects are represented with nodes labelled with a unique name of its class. Solid and dashed lines represent references with 1 value and $1/2$ references respectively, and missing edges represent 0 values. For example, in $P1$ state *Init* must be the `target` of transition $t1$, and *Filling* and *Finish* are potential targets. Uncertain $1/2$ equivalences are also marked by dashed line with an = symbol. In $P1$ this means that state *Init* may be merged to states *Filling* and *Finish*, or $t2$ may be split into multiple objects between *Filling* and *Finish*.

3.3 Refinement and Concretization

By resolving some uncertain parts, a partial model P can be refined to a more concrete partial model Q (denoted as $P \rightsquigarrow Q$). A refinement is defined by a function $\mathit{refine} : \mathit{Obj}_P \to 2^{\mathit{Obj}_Q}$, which maps each object of P to a set of objects in the refined partial model Q. A valid refinement *refine* respects the information order of type, reference, equivalence and existence predicates:

- for each class $\mathtt{C}$ and for each $p \in \mathit{Obj}_P$ and $q \in \mathit{refine}(p)$: $[\![\mathtt{C}(p)]\!]^P \sqsubseteq [\![\mathtt{C}(q)]\!]^Q$.
- for each reference $\mathtt{R}$ and for each $p_1, p_2 \in \mathit{Obj}_P, q_1 \in \mathit{refine}(p_1), q_2 \in \mathit{refine}(p_2)$: $[\![\mathtt{R}(p_1, p_2)]\!]^P \sqsubseteq [\![\mathtt{R}(q_1, q_2)]\!]^Q$
- for each $p_1, p_2 \in P, q_1 \in \mathit{refine}(p_1), q_2 \in \mathit{refine}(p_2)$: $[\![p_1 \sim p_2]\!]^P \sqsubseteq [\![q_1 \sim q_2]\!]^Q$
- for each $p \in \mathit{Obj}_P$ and $q \in \mathit{refine}(p)$: $[\![\mathit{exist}(p)]\!]^P \sqsubseteq [\![\mathit{exist}(q)]\!]^Q$
- for each $p \in \mathit{Obj}_P$ if $[\![\mathit{exist}(p)]\!]^P = 1$ then $\mathit{refine}(p)$ is not empty

Figure 4 illustrates two partial models $P2$ and $P3$ as possible refinements of $P1$. $P2$ represents a refinement scenario where $Init$ and $Filling$ are mapped to the same objects $FillingInit$, and the equivalence between the two objects are refined to 1 from $1/2$. Simultaneously, the possible equivalence between $FillingInit$ and $Finish$ must be refined to 0 to satisfy the information order, because $[\![Filling \sim Finish]\!]^{P1}$ was 0. In $P2$ the equivalence on $\mathtt{Transition}$ $t2$ is refined to 1 from $1/2$, and mapped to a single object. $P3$ represents another valid refinement, where the $Init$ and $Finish$ are merged, and $t2$ is refined into two different objects $t21$ and $t22$, where $t22$ still represents a set of objects.

If a refinement resolves all uncertainty, and there are no $1/2$ values in a partial model $P = \langle \mathit{Obj}_P, \mathcal{I}_P \rangle$, and P is simple, then P represents a concrete (simple) instance model $M = \langle \mathit{Obj}_M, \mathcal{I}_M \rangle$ where $\mathit{Obj}_M = \mathit{Obj}_P$ and $\mathcal{I}_M = \mathcal{I}_P$, which is called concretization and also marked with $P \rightsquigarrow M$. As $P2$ in Figure 4 does not contain any $1/2$ values, it can be interpreted as concretization of $P1$.

3.4 Evaluating Predicates on 3-Valued Partial Models

The main goal of this paper is to evaluate graph patterns on partial models in order to check possible matches on all possible concretizations. Evaluating a graph query over a partial model may have multiple outcomes: a pattern *may* ($1/2$), *must* (1) or *cannot* (0) have a match depending on whether the partial model can possibly be concretized in a way to fulfill the condition of the pattern.

Syntax. The same syntax is used for defining predicates on partial models as for concrete models, therefore the same well-formedness constraints can be used to check the correctness of partial models as for instance models.

Semantics. The predicates are evaluated in two steps: first, some expression rewriting is required to resolve *implicit equivalence checks*, then the rewritten pattern can be *directly evaluated* on the partial model. Implicit equivalence means that a match has to substitute the same value for each occurrence of a single pattern variable. For example in predicate $\mathtt{noOutgoing}(e)$ in Figure 3 the expression $\mathtt{from}(t, e) \wedge \mathtt{to}(t, trg)$ implicitly states that the value of the two t is the same. Our technique requires the explicit notation of equivalences, which can be achieved by rewriting each variable occurrence (except for those in equality constraints) to a new variable, and explicitly defining the equivalence between the new variables, by creating a logically equivalent expression. For example, the previous expression is changed to $\mathtt{from}(t_1, e) \wedge \mathtt{to}(t_2, trg) \wedge t_1 \sim t_2$.

We have constructed the semantic derivation rules of 2-valued logic in Figure 2 to evaluate 3-valued logic by using a numeric value of the unknown symbol

$1/2$, resulting in a 3-valued logic (similar to [16,25]). Therefore the same derivation rules can be used to evaluate the rewritten (but logically equivalent) rules on partial model. Additionally, the truth value of the expression follows the $\sqsubseteq$ information ordering, which has two important consequences:

Theorem 1 (Forward concretization). *If $[\![\varphi]\!]^P = 1$ in a partial model P, then $[\![\varphi]\!]^Q = 1$ in each partial model Q where $P \rightsquigarrow Q$, and $[\![\varphi]\!]^M = 1$ in a each M concretization where $P \rightsquigarrow M$. Similarly, if $[\![\varphi]\!]^P = 0$, then $[\![\varphi]\!]^Q = [\![\varphi]\!]^M = 0$.*

Theorem 2 (Backward concretization). *If $[\![\varphi]\!]^M = 1$ in a concrete model M, then $[\![\varphi]\!]^P \geq 1/2$ in a partial model P where $P \rightsquigarrow M$. Similarly, if $[\![\varphi]\!]^M = 0$ then $[\![\varphi]\!]^P \leq 1/2$.*

Therefore, if an error predicate evaluates to 1, then it identifies an invalid partial model that cannot be repaired in any concretization. If it evaluates to $1/2$ it highlights possible ways to inject errors. And finally, a 0 value can prove that an error cannot occur in the concretizations.

This approach provides a conservative approximation for 1 and 0 values, where inaccurate cases are considered as $1/2$. In other words, the match result is approximated in the direction of $1/2$, which also includes the unknown cases. That is a safe compromise in many application areas such as model validation.

3.5 Rewriting Predicates to Must and May Predicates

In the previous section we defined the resolution rules for evaluating a graph predicate over a 3-valued partial model, which can result in three possible values: 1, $1/2$, or 0. However, traditional query engines support only 2-valued pattern evaluation on 2-valued models. Therefore, to utilise efficient graph pattern matching engines like introduced in [3], we introduce a predicate rewriting technique to calculate 3-valued predicate using two 2-valued predicates called *must* and *may* predicates, and combining the into 3 logic value. A predicate $\mathit{must}(\varphi)$ is a must predicate of φ, if $[\![\mathit{must}(\varphi)]\!]^P_Z = 1$ when $[\![\varphi]\!]^P_Z = 1$, otherwise $[\![\varphi_{\mathit{must}}]\!]^P_Z = 0$. Similarly a predicate $\mathit{may}(\varphi)$ is a may predicate of φ, if $[\![\mathit{may}(\varphi)]\!]^P_Z = 1$ when $[\![\varphi]\!]^P_Z \geq 1/2$, otherwise $[\![\varphi_{\mathit{may}}]\!]^P_Z = 0$.

In the following, we give expression rewriting rules (illustrated in Figure 5) to create may and must predicates from a predicate. First, atomic expressions $\mathtt{C}(v)$, $\mathtt{R}(v_1, v_2)$, $\mathit{exist}(v)$ and $v_1 \sim v_2$ are replaced by $(\varphi \geq 1/2)$ and $(\varphi = 1)$ 2-valued predicates in order to round $1/2$ values up or down for maximizing the result for $\mathit{may}(\varphi)$ predicates, or to minimize the result for $\mathit{may}(\varphi)$ predicates.

Secondly, as the lower part of Figure 5 describes, *may* and *must* predicates are constructed from complex expression φ by recursively rewriting all subexpressions. It is important to note that the rewriting rule of the negated expression $\neg\varphi$ changes the modality of the inner expression from *may* to *must* and vica versa. Figure 6 illustrates the rewriting steps of an example graph previously introduced in Figure 3 into a may predicate.

Finally, $\mathit{may}(\varphi)$ and $\mathit{must}(\varphi)$ predicates are traditional 2-valued predicates, whose can be combined to encode 3 possible truth values:

$$\begin{array}{cc}
\multicolumn{2}{c}{\text{May and Must rewriting of atomic expressions}} \\
\hline
\mathit{may}(\texttt{C}(v)) := (\texttt{C}(v) \geq 1/2) & \mathit{must}(\texttt{C}(v)) := (\texttt{C}(v) = 1) \\
\mathit{may}(\texttt{R}(v_1, v_2)) := (\texttt{R}(v_1, v_2) \geq 1/2) & \mathit{must}(\texttt{R}(v_1, v_2)) := (\texttt{R}(v_1, v_2) = 1) \\
\mathit{may}(\mathit{exist}(v)) := (\mathit{exist}(v) \geq 1/2) & \mathit{must}(\mathit{exist}(v)) := (\mathit{exist}(v) = 1) \\
\mathit{may}(v_1 \sim v_2) := (v_1 \sim v_2 \geq 1/2) & \mathit{must}(v_1 \sim v_2) := (v_1 \sim v_2 = 1) \\
\multicolumn{2}{c}{\text{May and Must rewriting of complex predicates}} \\
\hline
\mathit{may}(\varphi_1 \wedge \varphi_2) := \mathit{may}(\varphi_1) \wedge \mathit{may}(\varphi_2) & \mathit{must}(\varphi_1 \wedge \varphi_2) := \mathit{must}(\varphi_1) \wedge \mathit{must}(\varphi_2) \\
\mathit{may}(\varphi_1 \vee \varphi_2) := \mathit{may}(\varphi_1) \vee \mathit{may}(\varphi_2) & \mathit{must}(\varphi_1 \vee \varphi_2) := \mathit{must}(\varphi_1) \vee \mathit{must}(\varphi_2) \\
\mathit{may}(\neg\varphi) := \neg\mathit{must}(\varphi) & \mathit{must}(\neg\varphi) := \neg\mathit{may}(\varphi) \\
\multicolumn{2}{c}{\mathit{may}(\exists v : \varphi) := \exists v : \mathit{may}(\mathit{exist}(v)) \wedge \mathit{may}(\varphi)} \\
\multicolumn{2}{c}{\mathit{must}(\exists v : \varphi) := \exists v : \mathit{must}(\mathit{exist}(v)) \wedge \mathit{must}(\varphi)} \\
\multicolumn{2}{c}{\mathit{may}(\forall v : \varphi) := \forall v : \mathit{may}(\neg\mathit{exist}(v)) \vee \mathit{may}(\varphi)} \\
\multicolumn{2}{c}{\mathit{must}(\forall v : \varphi) := \forall v : \mathit{must}(\neg\mathit{exist}(v)) \vee \mathit{must}(\varphi)}
\end{array}$$

Fig. 5: May and Must rewriting rules for graph predicates

$$\begin{array}{rl}
\text{original pattern: } \texttt{noOutgoing}(e) := \texttt{Entry}(e) \wedge \neg\exists t, \mathit{trg} : \texttt{from}(t, e) \wedge \texttt{to}(t, \mathit{trg}) & \\
\mathit{may}(\texttt{noOutgoing}(e)) := \mathit{may}(\texttt{Entry}(e)) \wedge \mathit{may}(\neg\exists t, \mathit{trg} : \texttt{from}(t, e) \wedge \texttt{to}(t, \mathit{trg})) = & \\
= \mathit{may}(\texttt{Entry}(e)) \wedge \neg\exists t, \mathit{trg} : & \mathit{must}(\mathit{exist}(t)) \wedge \mathit{must}(\mathit{exist}(\mathit{trg})) \wedge \\
& \mathit{must}(\texttt{from}(t, e)) \wedge \mathit{must}(\texttt{to}(t, \mathit{trg})) = \\
= (\texttt{Entry}(e) \geq 1/2) \wedge \neg\exists t, \mathit{trg} : & (\mathit{exist}(t) = 1) \wedge (\mathit{exist}(\mathit{trg}) = 1) \wedge \\
& (\texttt{from}(t, e) = 1) \wedge (\texttt{to}(t, \mathit{trg}) = 1)
\end{array}$$

Fig. 6: Example graph pattern rewriting

- If $[\![\mathit{must}(\varphi)]\!]^P_Z = 1$ and $[\![\mathit{may}(\varphi)]\!]^P_Z = 1$ then $[\![\varphi]\!]^P_Z = 1$
- If $[\![\mathit{must}(\varphi)]\!]^P_Z = 0$ and $[\![\mathit{may}(\varphi)]\!]^P_Z = 1$ then $[\![\varphi]\!]^P_Z = 1/2$
- If $[\![\mathit{must}(\varphi)]\!]^P_Z = 0$ and $[\![\mathit{may}(\varphi)]\!]^P_Z = 0$ then $[\![\varphi]\!]^P_Z = 0$

3.6 Transforming MAVO Uncertainty to 3-Valued Partial Models

MAVO uncertainty (which stands for May-Abstract-Variable-Open world) is a well-known and user-friendly partial modeling formalism [10,28,27] with several use-cases and tool support [4]. In the following we present a mapping of MAVO partial models to 3-valued partial models, enabling the evaluation of graph constraints on it.

MAVO specifies partial models with a concrete instance model B called base model, and introduces uncertainty annotations on the objects and references of B. The transformation starts with the mapping of the base model, then the annotations are transformed separately.

Base Model. First, a logic structure $P = \langle \mathit{Obj}_P, \mathcal{I}_P \rangle$ of Σ is created from the base model B, where $\mathit{Obj}_P := \mathit{Obj}_B$, and $\mathcal{I}_P := \mathcal{I}_B$.

Mapping of May. In MAVO, *may* annotation marks uncertainty about the *existence* of an object or reference. For each object o marked by *may*, uncertain existence can be expressed by $\mathcal{I}_P(\mathit{exist})(o) := 1/2$. For each reference $\texttt{R}$, holds that if a link between o_1 and o_2 is marked by *may*, then $\mathcal{I}_P(\texttt{R})(o_1, o_2) := 1/2$.

Mapping of Abstract. Abstract objects marked by *set* annotation marks uncertainty about the number of elements represented by an object. In 3-valued partiality, this can be represented by uncertain equivalence: for each object o marked by *set*, $\mathcal{I}_P(\sim)(o,o) := 1/2$.

Mapping of Variable. *var* annotation marks uncertainty about the distinctness of an object from another (which is not necessarily marked by *var*). In MAVO, objects with the same type are compatible for merging. Additionally, a *var* annotation implicitly specifies that the incoming and outgoing references of the compatible objects may be added to each other. For example, in the partial model in Figure 1, each incoming reference to $Init$ may be redirected to $Filling$ upon a merge. So for each object o_1 marked by *var*, and for each object o_2 with the same class predicate values ($\mathcal{I}_P(\texttt{C})(o_1) = \mathcal{I}_P(\texttt{C})(o_2)$ for each C) holds that:

- $\mathcal{I}_P(\sim)(o_1,o_2) := 1/2$, meaning that o_1 and o_2 may be merged
- for each incoming reference R from another object *src* to o_1 holds that: if $\mathcal{I}_P(\texttt{R})(src,o_2) = 0$ then $\mathcal{I}_P(\texttt{R})(src,o_2) := 1/2$. The outgoing references are handled similarly.

Mapping of Open. *open* is a *global property* of a MAVO partial model which marks uncertainty about the completeness of the model. If a model is *open*, then it can be extended by new objects and references in a refinement. Otherwise, only the existing elements can be resolved. In 3-valued partial models, this can be represented in the following way:

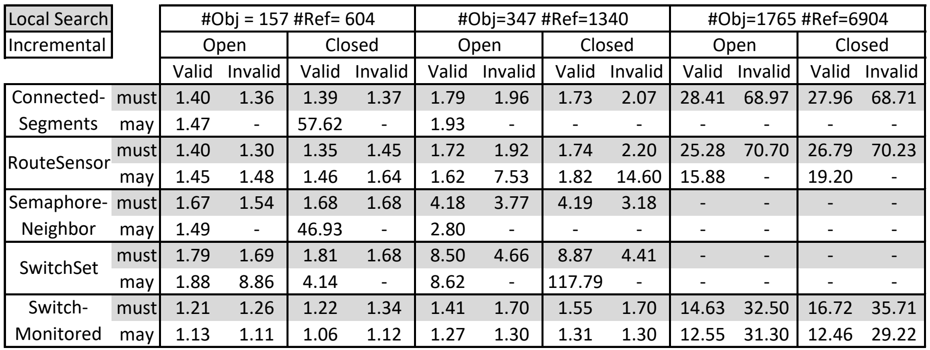
- a new object *other* is added to Obj_P, which represents the new objects.
- $\mathcal{I}_P(\sim)(other,other) = 1/2$, so *other* represent a set of objects.
- $\mathcal{I}_P(exist)(other) = 1/2$, so new objects are not necessarily added.
- for each class C: $\mathcal{I}_P(\texttt{C})(other) = 1/2$, so *other* represents all types.
- for each reference R and each object pair o_1, o_2: if $\mathcal{I}_P(\texttt{R})(o_1,o_2) = 0$, then $\mathcal{I}_P(\texttt{R})(o_1,o_2) := 1/2$. Therefore new references can be added.

Cleaning of the Partial Model. During the translation of uncertainty annotations, new $1/2$ references are added to the partial model without considering the structural constraints imposed by the target metamodel. Therefore, in order to exclude malformed instances from the analysis, when a $1/2$ reference is added during the translation, (1) the ending types, (2) the multiplicity, (3) the containment hierarchy and (4) possible inverse relations are checked. If a possible reference would violate a structural constraint, then it is not added to P, so the precision of the approach can be increased by excluding invalid extensions only.

4 Scalability Evaluation

We carried out an initial scalability evaluation[1] of our approach using 3 models (with 157, 347 and 1765 objects, respectively) and 5 queries available from the

[1] A detailed description at `https://github.com/FTSRG/publication-pages/wiki/Graph-Constraint-Evaluation-over-Partial-Models-by-Constraint-Rewriting`.



Local Search		#Obj = 157 #Ref= 604				#Obj=347 #Ref=1340				#Obj=1765 #Ref=6904			
Incremental		Open		Closed		Open		Closed		Open		Closed	
		Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid
Connected-Segments	must	1.40	1.36	1.39	1.37	1.79	1.96	1.73	2.07	28.41	68.97	27.96	68.71
	may	1.47	-	57.62	-	1.93	-	-	-	-	-	-	-
RouteSensor	must	1.40	1.30	1.35	1.45	1.72	1.92	1.74	2.20	25.28	70.70	26.79	70.23
	may	1.45	1.48	1.46	1.64	1.62	7.53	1.82	14.60	15.88	-	19.20	-
Semaphore-Neighbor	must	1.67	1.54	1.68	1.68	4.18	3.77	4.19	3.18	-	-	-	-
	may	1.49	-	46.93	-	2.80	-	-	-	-	-	-	-
SwitchSet	must	1.79	1.69	1.81	1.68	8.50	4.66	8.87	4.41	-	-	-	-
	may	1.88	8.86	4.14	-	8.62	-	117.79	-	-	-	-	-
Switch-Monitored	must	1.21	1.26	1.22	1.34	1.41	1.70	1.55	1.70	14.63	32.50	16.72	35.71
	may	1.13	1.11	1.06	1.12	1.27	1.30	1.31	1.30	12.55	31.30	12.46	29.22

Table 1: Evaluation time of validation patterns on partial models (in sec)

open TrainBenchmark [32]. We generated randomly assigned MAVO annotations for 5% of the model elements (e.g. with 7, 17, 88 uncertainties respectively). We evaluated the performance of (1) each graph query individually for (2) both may- and must-patterns (*may/must*) using (3) two pattern matching strategies (*incremental/local-search*) with (4) open world or closed world assumption (*open/closed*) after an optional (5) fault injection step (*valid/invalid*) to introduce some constraint violations. We measured the execution time for evaluating the queries in seconds with a timeout of 2 minutes using a laptop computer (CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10 Pro). Our experiments were executed 10 times and the median of execution time is reported in Table 1 (table entries with a dash denote a timeout). We

Our main observations can be summarized as follows:

- *Pattern matching over partial models is complex.* To position our experimental results, it is worth highlighting that most solutions of the Train Benchmark [32] evaluate *graph queries for regular models very fast* (scales up to millions of objects) for all these cases thus pattern matching over partial models must likely be in a different complexity class.
- *Fast inconsistency detection for must-matches.* The detection of a must-match over partial models is fast for both case of closed world and with open world assumption, especially, when using local-search graph pattern matching. It is also in line with previous observations in [29] using SMT-solvers.
- *Scalable detection of may-matches with closed world assumption.* Our approach may identify potential inconsistencies (i.e. may-matches) over partial models with closed world semantics containing more than 1500 objects using incremental pattern matching. It is more than one order of magnitude increase compared to previous results reported in [12,10] using Alloy.
- *Full match set of may-matches and open world is impractical.* As a negative result, calculating the full match set of graph patterns for may-matches *and* open world assumption frequently resulted in a timeout for models over 160 objects due to the excessively large size of the match set. For practical

analysis, we believe that *open* annotation in MAVO should be restricted to be defined in the context of specific model elements.

- *Selection of graph pattern matching strategy.* In case of timeouts, we observed that large match sets caused problems for an incremental evaluation strategy while the lack of matches caused problems for local-search strategy.

5 Related Work

Analysis of Uncertain/Partial Models. Uncertain models [10] provide user-friendly languages for defining partial models. Such models document semantic variation points generically by annotations on a regular instance model. Most analysis of uncertain models focuses on the generation of possible concrete models or the refinement of partial models. Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer and its back-end SAT solvers [28,27], or refined by graph transformation rules [26].

Approaches [11,12] analyse possible matching and execution of model transformation rules on partial models by using a SAT solver (MathSAT4) or by automated graph approximation (referred to as "lifting"), or by graph query engines [**?**]. The main difference is that their approach inspects possible partitions of a finite concrete model while we instead aim at (potentially infinite number of) extensions of a partial model. As a further difference, we use existing graph query engine instead of a SAT solver, which has a very positive effect on scalability (17 objects and 14 may annotations reported in [12] vs. over 1700 objects with 88 MAVO annotations in our paper).

Verification of Model Transformations. There are several formal methods that aim to evaluate graph patterns on abstract graph models (by either abstract interpretation [23,24], or predicate abstraction [25]) in order to detect possibly invalid concretizations. Those techniques typically employ techniques called pre-matching to create may-matches that are further analyzed. In [22] graph constraints are mapped to a type structure in order to differentiate objects that satisfy a specific predicate from objects that do not which could be used in our technique to further increase the precision of the matches.

In the previous cases an abstract graph similarly represents a range of possible models, and graph patterns are evaluated on abstract models to analyze their concretization. However, all of those technique expect a restricted structure in the abstract model, which is not available in partial models that are created by the developer.

Logic Solver Approaches. There are several approaches that map a (complete) initial instance model and WF constraints into a logic problem, which are solved by underlying CSP/SAT/SMT-solvers. In principle, the satisfaction of well-formedness constraints over a partial model (i.e. may- and must-matches) could be reformulated also using these techniques, although the same challenge has not been addressed so far. Complete frameworks with standalone specification languages include Formula [15] (which uses Z3 SMT- solver [19]), Alloy [14] (which relies on SAT solvers) and Clafer [1] or a combination of solvers [29].

There are several approaches to validate models enriched with OCL constraints [13] by relying upon different back-end logic-based approaches such as constraint logic programming [9,8], SAT-based model finders (like Alloy) [31,18], first-order logic [2] or higher-order logic [6]. As a common issue of such SAT/SMT-based approaches, the scalability is limited to small models.

DLL62 and

6 Conclusion and Future Work

Conclusions. In this paper, we proposed a technique to evaluate graph queries capturing constraints over partial models. Since a partial model may be extended by the designer in future refinement steps, we defined may- and must-matches of a graph query correspondingly to denote potential and real violations of constraints. We also defined conservative approximations of may- and must-matches by rewriting of graph patterns in accordance with MAVO semantics.

Our initial scalability evaluation using the open Train Benchmark [32] shows that (1) finding real constraint violations over partial models is fast; (2) identifying potential inconsistencies with either open world or closed world assumption may scale for partial models with over 1500 model elements (which is one order of magnitude larger than reported in previous papers).

Future work. Although we motivated our work to check well-formedness constraints over uncertain models, our current results provide a key milestone for an ongoing project, which aims at the automated generation of scalable and consistent domain-specific graph models (aka a graph-based model finder). While metamodels of industrial modeling language often contain several hundreds of classes, existing logic solvers fail to produce an instances model containing over 150 objects, which is a major limitation for industrial use. Since the actual validation of complex graph constraints consumes significant amount of time in existing SAT/SMT-solvers, our current approach (which exploits efficient checking of graph constraints) can nicely complement traditional logic solvers.

Acknowedgement

This paper is partially supporded by MTA-BME Lendület Research Group on Cyber-Physical Systems, and NSERC RGPIN-04573-16 project. Additionally, we would like to thank Gábor Bergmann and the anonymous reviewers for their insightful comments.

References

1. Bak, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., Wasowski, A.: Clafer: unifying class and feature modeling. Software & Systems Modeling pp. 1–35 (2013)
2. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into First-order Predicate Logic. In: Proc. of the VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark (2002)
3. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A Graph Query Language for EMF models. In: Fourth International Conference on Theory and Practice of Model Transformations. LNCS, vol. 6707, pp. 167–182. Springer (June 2011)
4. Bertolino, A., Canfora, G., Elbaum, S.G. (eds.): 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. IEEE Computer Society (2015), http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7174815
5. Biermann, E., Ehrig, K., Ermel, C., Köhler, C., Taentzer, G.: The EMF model transformation framework. In: AGTIVE. pp. 566–567 (2007)
6. Brucker, A.D., Wolff, B.: The HOL-OCL tool (2007), http://www.brucker.ch/
7. Búr, M., Ujhelyi, Z., Horváth, Á., Varró, D.: Local search-based pattern matching features in EMF-IncQuery. In: 8th International Conference on Graph Transformation (2015)
8. Cabot, J., Clariso, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conf. on. pp. 73–80 (April 2008)
9. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07). pp. 547–548 (2007)
10. Famelis, M., Salay, R., Chechik, M.: Partial models: Towards modeling and reasoning with uncertainty. In: Proceedings of the 34th International Conference on Software Engineering. pp. 573–583. IEEE Press, Piscataway, NJ, USA (2012)
11. Famelis, M., Salay, R., Chechik, M.: The semantics of partial model transformations. In: Proceedings of the 4th International Workshop on Modeling in Software Engineering. pp. 64–69. IEEE Press (2012)
12. Famelis, M., Salay, R., Di Sandro, A., Chechik, M.: Transformation of models containing uncertainty. In: International Conference on Model Driven Engineering Languages and Systems. pp. 673–689. Springer (2013)
13. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and Systems Modeling 4, 386–398 (2005)
14. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. 11(2), 256–290 (2002)
15. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about metamodeling with formal specifications and automatic proofs. In: Model Driven Engineering Languages and Systems, pp. 653–667. Springer (2011)
16. Kleene, S.C., De Bruijn, N., de Groot, J., Zaanen, A.C.: Introduction to metamathematics, vol. 483. van Nostrand New York (1952)
17. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of ocl for capturing structural constraints in modelling languages. In: Rigorous Methods for Software Construction and Analysis, pp. 204–218 (2009)
18. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into use. In: TOOLS'11 - Objects, Models, Components and Patterns. LNCS, vol. 6705, pp. 290–306 (2011)

19. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008). LNCS, vol. 4963, pp. 337–340. Springer (2008)
20. Nickel, U., Niere, J., Zündorf, A.: The fujaba environment. In: Proceedings of the 22nd international conference on Software engineering. pp. 742–745. ACM (2000)
21. The Object Management Group: Object Constraint Language, v2.0 (May 2006)
22. Radke, H., Arendt, T., Becker, J.S., Habel, A., Taentzer, G.: Translating essential ocl invariants to nested graph constraints focusing on set operations. In: International Conference on Graph Transformation. pp. 155–170. Springer (2015)
23. Rensink, A., Distefano, D.: Abstract graph transformation. Electronic Notes in Theoretical Computer Science 157(1), 39–59 (2006)
24. Rensink, A., Zambon, E.: Pattern-based graph abstraction. In: Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings. pp. 66–80 (2012)
25. Reps, T.W., Sagiv, M., Wilhelm, R.: Static program analysis via 3-valued logic. In: International Conference on Computer Aided Verification. pp. 15–30 (2004)
26. Salay, R., Chechik, M., Famelis, M., Gorzny, J.: A methodology for verifying refinements of partial models. Journal of Object Technology 14(3), 3:1–31 (2015)
27. Salay, R., Chechik, M., Gorzny, J.: Towards a methodology for verifying partial model refinements. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 938–945. IEEE (2012)
28. Salay, R., Famelis, M., Chechik, M.: Language independent refinement using partial modeling. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering, LNCS, vol. 7212, pp. 224–239. Springer Berlin Heidelberg (2012)
29. Semeráth, O., Barta, A., Horváth, A., Szatmári, Z., Varró, D.: Formal validation of domain-specific languages with derived features and well-formedness constraints. Software and Systems Modeling pp. 1–36 (2015)
30. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: 19th International Conference on Fundamental Approaches to Software Engineering. pp. 87–103 (2016)
31. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation. pp. 1–10. ACM (2009)
32. Szárnyas, G., Semeráth, O., Ráth, I., Varró, D.: The TTC 2015 train benchmark case for incremental model validation. In: 8th Transformation Tool Contest, (STAF 2015). pp. 129–141 (2015)
33. The Eclipse Project: Eclipse Modeling Framework, //www.eclipse.org/emf
34. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., Varró, D.: Emf-incquery: An integrated development environment for live model queries. Science of Computer Programming 98 (02/2015 2015)
35. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 68(3), 214–234 (October 2007)
36. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE software 31(3), 79–85 (2014)
37. Yakindu Statechart Tools: Yakindu, http://statecharts.org/