Acta Cybernetica **00** (0000) 1–**??**.

Evaluating Well-Formedness Constraints on Incomplete Models

Oszkár Semeráth* and Dániel Varró*

Abstract

In modern modeling tools used for model-driven development, the validation of several well-formedness constraints is continuously been carried out by exploiting advanced graph query engines to highlight conceptual design flaws. However, while models are still under development, they are frequently partial and incomplete. Validating constraints on incomplete, partial models may identify a large number of irrelevant problems. By switching off the validation of these constraints, one may fail to reveal problematic cases which are difficult to correct when the model becomes sufficiently detailed.

Here, we propose a novel validation technique for evaluating well-formedness constraints on incomplete, partial models with *may* and *must* semantics, e.g. a constraint without a valid match is satisfiable if there is a completion of the partial model that may satisfy it. To this end, we map the problem of constraint evaluation over partial models into regular graph pattern matching over complete models by semantically equivalent rewrites of graph queries.

Keywords: Partial models, Model Validation, Graph Patterns

1 Introduction

Context In Model-Driven Development (MDD), models are the main design artifacts, from which documentation, system configuration, or even source code can be automatically generated. MDD is widely used in industry in various domains including business modeling, avionics and automotive [38] as it provides early validation and advanced automation. When developing complex systems, multiple design rules and well-formedness constraints have to be checked repeatedly over large (graph) models [4] in order to ensure the validity of models throughout the entire design process starting from an early stage of design.

During development, the level of uncertainty represented in the models gradually decreases until all critical design decisions have been made. However, certain constraints can only be checked at the right level of abstraction, i.e. after some

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

design decisions have already been made. When a new constraint is violated, engineers may need to rethink some parts of the system and reiterate on some previous design decisions taken earlier. The uncertainty, which is inherently present in high-level initial models, make design decisions drawn from them especially risky.

Problem statement Partial models have been introduced in [14] to formally capture uncertainty in various design phases. Existing techniques for partial models allow a modeler to explicitly express model uncertainty, or assess possible design candidates [31], but they do not provide sufficient support for the evaluation of well-formedness constraints over partial models. While there is efficient tool support for defining and checking well-formedness constraints and design rules over regular model instances by using graph pattern matching, the evaluation of the same constraints have only been addressed by SMT/SAT solving tools, which have major scalability problems as the size of the models starts to grow.

Objective Our goal is to evaluate well-formedness constraints over incomplete, partial models by graph pattern matching instead of SAT/SMT solving. The key conceptual challenge is that while existing model query and transformation tools evaluate graph constraints over models with closed world semantics (i.e. the model is assumed to be complete), evaluating constraints on partial models necessitates an open world semantics, as new elements may be added to the model later on, which may satisfy (or violate) the constraints. Instead of proposing dedicated graph pattern matching algorithm that operates over partial models, we rewrite the original graph constraints (to be matched with open world semantics over partial models) into an equivalent constraint (to be matched with traditional closed world semantics over regular models).

Contribution Here we present a novel technique for evaluating well-formedness constraints on partial models. Our technique uses the partial snapshots [32] to represent incomplete partial models, and it is compatible with EMF (Eclipse Modeling Framework) [36] which is the de facto industrial modeling standard in MDD. Well-formedness constraints are captured as graph queries using the pattern language of VIATRA [4]. From an input graph query, our approach generates an extended, but semantically equivalent graph query.

Added value Using this technique, design rules specified for concrete models can be automatically checked for incomplete, unfinished models to (i) detect invalid elements, and (ii) identify invalid design options during the development. Additionally, this technique can be used to (iii) enumerate all possible ways to correct currently invalid partial instance model, or (iv) list all options to inject errors by extending a currently valid model. Lastly, our approach uses approximations and an efficient graph query engine, which enables the use of this technique directly on an existing modeling environment. Performing such an analysis was unfeasible previously with logic solvers [33].

As a long term benefit, model generation techniques [33, 31] can be supported efficiently, as we can reuse existing graph pattern matching tools like the VIATRA framework to evaluate well-formedness constraints over incomplete, partial models, just as we do for for regular, complete models.

Structure of the Paper The rest of the paper is structured as follows: In Section 2 we summarize the necessary modeling concepts and introduces a case study for partial models. In Section 3 we provide an overview on how to evaluate model queries with open-world semantics, then in Section 4 we show how graph patterns can be transformed to their open-world equivalent. In Section 5 we provide comparison with other approaches in the literature. Lastly, in Section 6 we draw conclusion and make suggestions for future work.

2 Preliminaries

2.1 Motivating example: Yakindu Statecharts

Yakindu Statecharts Tools [39] is an industrial integrated modeling environment developed by Itemis AG for the development of reactive, event-driven systems based on the concept of statecharts captured in combined graphical and textual syntax. Yakindu simultaneously supports static validation of well-formedness constraints as well as the simulation of (and code generation from) statechart models. Examples in this paper are illustrated by the validation of partial Yakindu models.

The behavior model of a sample coffee machine is illustrated in Figure 1. In **Step I**, an initial statechart is developed, highlighting the key phases of the operation. Initially, the machine starts in state Ready. Then, after inserting coins, a drink can be selected in state Select. After the selection the machine start filling the coffee, and gives back the change in state Service. When the drink is ready, and the change is returned, the coffee machine goes back to the initial state.

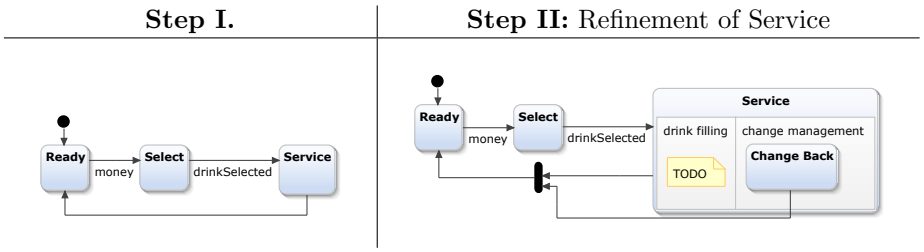


Figure 1: Example statechart model under developement.

Next, in **Step II** of Figure 1, the developer makes the first steps to refine the Service state by adding two new regions, namely one that manages the filling of the drink (drink filling), and one that manages the change (change management). When

both regions are ready, then the two regions synchronize to the initial state. At present, the model in **Step II** is only an unfinished partial model.

There are several well-formedness constraints (aka design rules) in the development environment to validate statecharts. Two examples are the following:

1. Each region needs to have exactly one entry, which has a transition to a state in the same region.
2. The target and source states of a synchronization have to be contained in the same parent state.

Both constraints are defined for complete models, and while they can be evaluated on partial models, it provides less relevant information during development:

1 The first constraint is invalid for regions drink filling and change management as they do not have initial states. However, as design decisions about internal states and entries have not yet been made, these are not real errors of the model, but a direct consequence of the incomplete model.

2 However, the synchronization in **Step II** synchronizes states that are not parallel, thus it is a design flaw that will be present in any possible completion.

2.2 Metamodels, Models, Partial Models

A metamodel defines the main concepts, relations and the basic structure of a domain-specific language (DSL). In this paper, domain models are captured by the Eclipse Modeling Framework (EMF) [36], which is frequently used in industrial modeling tools.

An extract of metamodel for Yakindu statecharts describing the state graph is illustrated in Figure 2. A state machine consists of Regions, which in turn contain states (called Vertexes) and Transitions. An abstract state Vertex is further refined into RegularStates (like State) and PseudoStates like Entry and Synchronization states. Note that we intentionally kept the generalization hierarchy unchanged and we simplified the original metamodel by by removing certain elements.

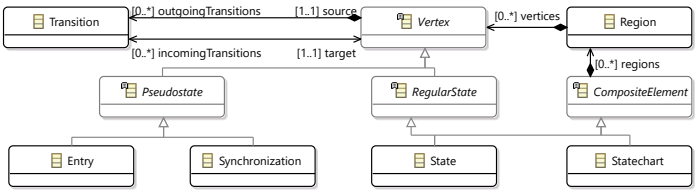


Figure 2: Simplified Yakindu state graph metamodel.

Metamodel elements are mapped to a set of logic relations as defined in [32, 20], which are revisited below:

- **Classes (CLS)**: An *EClass* captures some core concepts in a DSL. In EMF, *EClass*es can be instantiated to *EObjects*, where the set of objects of a model M is denoted by $\mathsf{Objects}_M$. If o is an instance of a type C, it is denoted by $\mathsf{C}(o)$.
- **References (REF)**: *EReference*s between classes S and T capture a binary relation $R(S,T)$ of the metamodel. When two objects o and t are in a relation R, an *EReference* is instantiated leading from o to t denoted by $\mathsf{R}(o,t)$.
- **Attributes (ATT)**: *EAttributes* enrich a class C with values of predefined primitive types like integers and strings, etc. by binary relations $A(C,V)$. If an object o stores a value v as attribute A it is denoted as $\mathsf{A}(o,v)$.

Further structural restrictions implied by a metamodel (and formalized in [32]) include (1) **Generalization (GEN)**, which expresses the fact that a more specific (child) class has every structural feature of the more general (parent) class, (2) **Type compliance (TC)** requires that for any relation $\mathsf{R}(o,t)$, its source and target objects o and t need to have compliant types, (3) **Abstract (ABS)**: If a class is defined as *abstract*, it is not allowed to have direct instances, (4) **Multiplicity (MUL)** of structural features can be limited with upper and lower bounds in the form of "lower..upper" and (5) **Inverse (INV)**, which states that two parallel references of opposite direction always occur in pairs. Finally EMF instance models are often expected to be arranged into a *containment hierarchy*, which is a directed tree along references marked in the metamodel as containment (e.g. regions or vertices).

Model M is a valid instance of a metamodel *Meta* (denoted by $M \models \mathit{Meta}$) if (i) all classes, references and attributes are defined in *Meta* and (ii) satisfies the structural constraints $(1)-(5)$ [32].

Partial models Partial (or incomplete) models arise when we are still planning to add more elements to a model, thus certain constraints can be violated. P is a partial model of a (partial or complete) model M (denoted as $P \subseteq M$) if M contains P as a submodel, so M can be created from P by adding model elements to P, Formally: $P \subseteq M$ holds if (i) P satisfies structural constraints $(1)-(4)$ above with the possible exception of lower multiplicity in **(MUL)** and (ii) there is an injective morphism $f : \mathsf{Objects}_P \to \mathsf{Objects}_M$ which satisfies the following constraints:

1. For each object o_1 and o_2: $o_1 = o_2 \Leftrightarrow f(o_1) = f(o_2)$.
2. For each class C and each object $o \in \mathsf{Objects}_P : \mathsf{C}(o) \Leftrightarrow \mathsf{C}(f(o))$.
3. For each reference R and each object pair $s,t \in \mathsf{Objects}_P : \mathsf{R}(s,t) \Rightarrow \mathsf{R}(f(s),f(t))$.
4. Finally, for each attribute type A and attribute value v and each object $o \in \mathsf{Objects}_P : \mathsf{A}(o,v) \Rightarrow \mathsf{A}(f(o),f(v))$

In a standard EMF model it is not specified which part of the model is complete, and which one is under development. Therefore, a partial model can be extended

in any part: new references or attribute values can be added to any objects, or new child objects of any type can be added to the model, as long as structural constraints (1) – (4) are not violated.

2.3 Graph Patterns

Well-formedness constraints are often captured by graph patterns (GP) [37, 4], which is an expressive formalism used for various purposes in model-driven development alternatively for standard OCL constraints [23]. A graph pattern is a graph-like structure representing several conditions (or constraints) matched against an instance model.

In the VIATRA pattern language, a **graph pattern** $q(p_1, \ldots, p_n) = def$ is defined by a name q and symbolic parameters $p_1, \ldots, p_n$, and constraints over the parameters (captured by *body*). A pattern may have multiple bodies with *constraints*, and may introduce additional local variables beside the parameters. The constraints available in the pattern language include:

- *Classifier constraint:* checks whether a variable is an instance of a class, i.e. checks whether $\mathsf{C}(o)$ true.
- *Path constraint:* requires a specific reference, an attribute, or a path of reference and attribute sequence between two variables.
- *Equality constraint:* specifies that two variables have to be mapped to the same model element.
- *Pattern call constraint:* enables the composition of multiple patterns. The positive pattern call refers to another pattern and specifies that the called pattern must be satisfied in the context of the actual parameters. Additionally, a pattern may be composed negatively (`neg` keyword), which means that the target negative pattern is not allowed to have a valid match along the actual parameters. Finally, it is possible to compute the transitive closure of a two-parameter pattern by the + symbol.
- *Count expression:* counts the number of matches of a pattern.
- *Check constraint or eval:* evaluates a specific attribute expression on the variables of the pattern and accepts matches only if the result of an attribute condition is true.

A **match** m of pattern $q(p_1, \ldots, p_n) = def$ over model M maps each symbolic parameter p_i to a model element (object, enum literal or primitive) from the target model M. A match is valid if it satisfies each constraint in a body of the pattern def: $\forall m : M \models \bigvee_{body \in def} body(m(params))$. A **partial match** is a partial function m^p where only a subset of parameters is mapped to model elements.

A match of a pattern is marked by $M, m \models p$. Furthermore, we use $[M, m \models p]$ as a predicate over matches, which is evaluated to true if m is a valid match of p in M, and $\neg[M, m \models p]$ means m is not a valid match of p.

```
1  pattern entryInRegion(r, e) {
2    Entry(e);
3    Region.vertices(r,e);
4  }
5  pattern noEntry(r) {
6    Region(r);
7    neg find entryInRegion(r,_e);
8  }
9  pattern multipleEntry(r) {
10   find entryInRegion(r, e1);
11   find entryInRegion(r, e2);
12   e1 != e2;
13 }
```

$$\begin{aligned}
&\mathsf{entryInRegion}(r, e) := \\
&\qquad \mathsf{Entry}(e) \wedge \mathsf{vertices}(r, e) \\
&\mathsf{noEntry}(r) := \forall _e : \\
&\qquad \mathsf{Region}(r) \wedge \neg \mathsf{entryInRegion}(r, _e) \\
&\mathsf{multipleEntry}(r) := \exists e_1, e_2 : \\
&\qquad \mathsf{entryInRegion}(r, e_1) \wedge \\
&\qquad \mathsf{entryInRegion}(r, e_2) \wedge \\
&\qquad e_1 \neq e_2
\end{aligned}$$

Figure 3: Examle validation patterns

The main task of a graph query engine is to evaluate graph pattern over a model by gradually extending partial matches to a complete match. When a graph pattern is evaluated on a model, one distinguishes between zero or more matches: in the first case the query evaluates to false, while in the second case, it evaluates to true.

Example Three graph patterns are illustrated in Figure 3 along with their mathematical formalization. Pattern *entryInRegion* collects all entry nodes e of a region r, and constructed from a classifier constraint ($\mathsf{Entry}(e)$) and a path constraint ($\mathsf{vertices}(r, e)$). Pattern *noEntry* identifies regions r without an entry node by negative composition (call) to *entryInRegion* (using a negative predicate $\neg\mathsf{entryInRegion}(r, \_e)$). Lastly, pattern *multipleEntry* highlights regions r with more than a single entry node.

3 Matching Graph Queries on Incomplete Models

When evaluating a graph query over an incomplete partial model, the pattern matching may have multiple outcomes: a graph pattern *may*, *must* or *cannot* have a match on a partial model depending on whether the partial model can possibly be extended by adding model elements to fulfill the graph condition of the pattern.

Below, we introduce a novel graph rewriting technique for evaluating standard queries on incomplete, partial models. There are several challenges related to this matching problem. First, it has to cover an infinite number of possible instance models, which causes decidability issues [32], and necessitates the use of approximation techniques [33]. Also, the result set has to be calculated efficiently, as checking how a pattern may or may not match can cause a combinatorial explosion even for small partial models.

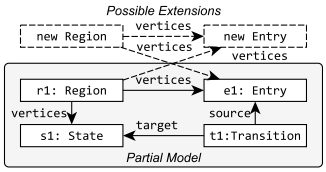
3.1 Partial Matches on Incomplete Models

When matching patterns on a complete model, the matches refer to real objects of the model in accordance with closed world semantics. However, when only a partial model is to be matched, a match may refer to currently non-existing objects, which are created in a later state of the development. Therefore, we rely upon projections of a full match of a complete model M to partial matches of partial model P where undefined elements refer to new elements added to M later.

A **projection of a match** on a partial model P is defined by a function π_P, which maps fully defined matches of an extended models M ($P \subseteq M$) to a partial match of P by excluding parameter bindings that are unavailable in P (but they remain defined for all parameters which maps an element of P to a parameter). Formally, $\pi_P(m) = m'$ if

$$m'(param) = \begin{cases} m(param) & \text{if } m(param) \in \mathsf{Objects}_P \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example For example, Figure 4a shows a partial model of a simple statechart with a single State, an Entry in a Region and a Transition. Some possible extensions are marked with a dashed line: either new Regions or new Entries can be added to the model, or they can be linked with vertices references.



(a) Example partial model with possible extensions.

```
pattern entryInRegion(r:Region, e:Entry) {
  Region.vertices(r,e);
}
```

Matches	r:Region	e:Entry
m_1	r1	e1
m_2	r1	new Entry
m_3	new Region	e1
m_4	new Region	new Entry

(b) Pattern *entryInRegion* and its possible matches.

Figure 4: Pattern matching for partial models

The graph pattern *entryInRegion* collects regions with their entry states. Figure 4b also enumerates the possible kind of matches:

m_1 is a complete match where all parameters are bound to an existing object of the partial model (e.g. r1 and e1) which satisfy the condition of the pattern.

m_2, m_3 are incomplete matches which represent a possible satisfaction of the pattern between some existing and some future objects. For example, m_2 represents a case where a new Entry needs to be added to r1 region with a new vertices reference to form a match.

m_4 is a symbolic match on a completely new partition of a model (where all of the elements are new), which represents all other matches that are independent from the partial model.

The use of symbolic values and match projection in the context of a partial model allows one to cover an infinite number of possible matches. In the following, we formalize when a pattern *must*, *may* or *cannot* be matched on a partial model.

3.2 Must and May Modality of Graph Queries

Now we will introduce two kinds of modalities to calculate possible matches of pattern p on partial model P. A *must-match* (denoted as $\square[P, m \models p]$) is a match which exists in every possible M extension of P (including P itself). Formally:

$$\square[P, m \models p] := \forall M : [P \subseteq M] \Rightarrow M, m \models p$$

A *may-match* (denoted as $\Diamond[P, m \models p]$) represents a case where a m is a projected match of an extension of P (even if the match does not exist directly in P). Formally:

$$\Diamond[P, m \models p] := \exists M : P \subseteq M \wedge M, m' \models p \wedge m = \pi_P(m')$$

Example Figure **??** collects the must and may-matches of pattern `entryInRegion` shown in Figure 4b, when it is matched on the partial model of Figure 4a. In this example, the r1 and e1 pair is the only must-match: this match will exist in any possible extension of the partial model, i.e. it cannot be removed by adding new elements. The may-matches, as always, also contain the must-match, and additionally contain two cases with new elements: a new entry in r1 may create a new match, and a completely new region with an entry may also create matches. However, match m_3 requires the creation of an additional Region to the existing Entry is not a may-match, because the pattern requires a vertices containment reference and the creation of a second parent to e1, which would violate the containment hierarchy.

must	r:Region	e:Entry
m_1	r1	e1

missing	r:Region	e:Entry
m_4	new Region	e1

may	r:Region	e:Entry
m_1	r1	e1
m_2	r1	new Entry
m_3	new Region	new Entry

Table 1: Must and may-matches of an example model query on partial model.

Combination of may- and must-matches Table 2 summarizes the possible combinations of may- and must-matches. If the partial model can be extended to a valid model (so $\exists M : P \subseteq M$), a must-match implies a may-match ($\square[P, m \models p] \Rightarrow$

$\Diamond[P, m \models p]$), so a must and may-match simply means a must-match. If a match is a must-match, but not a may-match it means inconsistency (marked with an X). Because $P \subseteq P$ is true, if there is an actual match on the partial model, then, by definition, the must-matches and may-matches have to contain this match. Cases where this condition does not hold are marked with an X. A may-match but not must-match combination has the weakest consequence; namely a pattern might or might not be satisfied in an extension of the partial model. And finally, if an m is not a may-match it means that it cannot be matched in any possible extension of the model.

	$\Diamond[P, m \models p]$		$\neg\Diamond[P, m \models p]$	
	$[P, m \models p]$	$\neg[P, m \models p]$	$[P, m \models p]$	$\neg[P, m \models p]$
$\Box[P, m \models p]$	$\forall M : P \subseteq M$ $\Rightarrow [M, m \models p]$ (Constant match)	X	X	X
$\neg\Box[P, m \models p]$	$[P, m \models p]$, $\exists M : P \subseteq M$ $\wedge\neg[M, m \models p]$ (Possibly disappearing match)	$\neg[P, m \models p]$, $\exists M : P \subseteq M$ $\wedge[M, m \models p]$ (Possibly appearing match)	X	$\forall M : P \subseteq M$ $\Rightarrow \neg[M, m \models p]$ (Impossible match)

Table 2: Combination of concrete, must- and may-matches.

Here, it is necessary to examine the semantics of must- and may-matches in the context of well-formedness constraints. If a pattern has a must-match of an ill-formedness pattern, then it marks an invalid model that cannot be repaired by adding new elements. If a partial model has a may-match ($\Diamond[P, m \models p]$), and the pattern has an actual match on the partial model ($[P, m \models p]$) it means that the model is currently ill-formed, but it might be repaired by adding new elements. Otherwise, if there is a may-match ($\Diamond[P, m \models p]$), but there no actual matches ($\neg[P, m \models p]$), it means that the model is currently well-formed, but the may-match highlights possible ways to inject errors.

4 Rewriting Model Queries with Open-World Semantics

Next, we introduce a novel technique that is able to produce must- and may-matches from existing model queries that are defined on complete models. The approach is based on a graph pattern rewriting technique that creates over- and under-approximated patterns to evaluate must-matches and may-matches.

4.1 Approximation of Must- and May-Matches

Unfortunately, to decide whether a provisional match of pattern p on a partial model P can be extended to a complete match for a model M where $P \subseteq M$ is a complicated problem. In general, it requires complex logic analysis [32], which scales poorly with respect to the size of the partial model [33], and may have undecidability issues. Therefore, we will introduce approximations to tackle this problem.

In our approach, must- and may-matches are approximated with regular pattern matching problems that can be efficiently evaluated on complete and partial models. A pattern matching problem $P^U, m^U \models p^U$ *under-approximates* a must-matching problem, if it satisfies the following constraint:

$$[P^U, m^U \models p^U] \quad \Rightarrow \quad \Box[P, m \models p]$$

Similarly, a pattern matching problem $P^O, m^O \models p^O$ *over-approximates* a may-matching problem, if the following constraint is satisfied:

$$\Diamond[P, m \models p] \quad \Rightarrow \quad [P^O, m^O \models p^O]$$

These formulae define a conservative approximation of must- and may-matches over partial models (see Table 3) with two potential inaccuracies: (1) a must-match might not be detected in a partial model, or (2) the unsatisfiability of a match might not be proved. In our approach, these inaccurate cases are collected in the same category as the may-matches. In other words, the match result is approximated in the direction of may-matches, which also collects the unknown cases. This is a safe compromise in most application areas like model validation.

	$[P^O, m^O \models p^O]$	$\neg[P^O, m^O \models p^O]$
$[P^U, m^U \models p^U]$	$\Box[P, m \models p]$	X
$\neg[P^U, m^U \models p^U]$	$\Diamond[P, m \models p]$ + Unknown	$\neg\Diamond[P, m \models p]$

Table 3: Consequences of approximated matches.

In our approach, must- and may-matches are produced in four steps:

1. Approximated models P^U and P^O are created from the original partial model P (Section 4.2).
2. Approximated patterns p^U and p^O are constructed from p (Section 4.4) with the help of uncertain model indexers (Sections 4.3).
3. The approximated pattern matching problems are executed as normal pattern matching problems.
4. The matches of the approximated pattern matching problem m^U, m^O are interpreted as may- and must-matches of the original pattern p (Section 4.5).

In the following, we provide a constructive way for creating approximated models and patterns for a must/may matching problem, along with a technique for transforming over- and under-approximated matches back to must- and may-matches.

4.2 Representation of Partial Models

Now we will show how approximated models P^O and P^U can be created from a partial model P which will serve as a basis for evaluating approximated patterns over it later on. In our approach, standard Ecore [36] models are used to define partial models enriched with special annotations on model elements which are adapted from [14]. In our current implementation, both P^O and P^U are created in the same way from a PartialModel, which is illustrated in Figure 5.

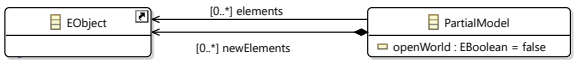


Figure 5: Helper metamodel for partial models.

This partial model representation has three important characteristics. These are:

- The reference elements represents the objects of the partial model. Our technique incorporates objects only if they are referred by the partial model with this reference. Hence it is easy to temporarily exclude elements from an existing model to create experimental variations of the model without modifying it.
- A partial model contains additional symbolic (newly created) elements along the newElements reference. A new element in a may-match represents a new object that needs to be created (in a proper context) in order to satisfy the condition of the match. Typically, a partial model contains a new element for each concrete (non-abstract) object of the metamodel. However, the analysis can be restricted by removing objects, thus forbidding the creation of this kind of object.
- The semantic interpretation of the partial model can be switched between open-world and close-world by setting the value of the attribute openWorld.

4.3 Open-World Indexing of Model Elements

The construction of the over- and under-approximated patterns is separated into two layers: (1) first an *uncertain property indexer* collects the possible variations of model properties (like objects and references), and (2) an *uncertain pattern*

layer that combines those uncertain properties into over- and under-approximated patterns. There we present the uncertain property indexer.

There are four kinds of basic properties of a partial model: (1) what the objects are (i.e. $\mathsf{Objects}_P$) (2) what the type of objects in the model is (C(o)) (3) where references are between the objects (R(s,t)), and (4) what the attribute values are in the model (A(o,v)). In the following, we describe how these basic properties can be over- and under-approximated by appropriate graph patterns.

Objects First, the objects of an extended model are either objects that are present in the partial model (referred by elements) or newly created objects (contained by newElements). However, for newly created elements referred in projected matches it is not decidable that the new objects are different or equal. Therefore the equivalence of the objects needs to be over- and under-approximated. Figure 6 illustrates a pattern for which under- and over-appriximating the equivalence of two objects, which are matched by `mustEqual` and `mayEqual` patterns. Objects e_1 and e_2 must be equals in all possible extensions of P^U, if they are equal in the partial model. However, two objects may be equal if they are represented by the same prototype object.

```
pattern mustEqual(p,e1,e2) {
  PartialModel.elements(p,e1);
  e1 == e2;
}
pattern mayEqual(p,e1,e2) {
  PartialModel.openWorld(p,true);
  PartialModel.newElements(p,e2);
  e1 == e2;
} or {
  find mustEqual(p,e1,e2);
}
```

Figure 6: Approximation of equivalences

```
pattern mustType_C(p,e) {
  C(e); PartialModel.elements(p,e);
}
pattern mayType_C(p,e) {
  C(e);
  PartialModel.newElements(p,e);
  PartialModel.openWorld(p,true);
} or {
  find mustType_C(p,e);
}
```

Figure 7: Approximation of type predicates

Types of objects It can be decided whether an object o must be of class C by simply evaluating the predicate $\mathsf{C}(o)$, which is a safe under-approximation. However, in the case of partial models, the set of instances of class C may include some elements from the symbolic instances with compatible types. Figure 7 illustrates the schema of patterns that match objects which are necessary or possible instances of class C.

References and attributes The indexing of possible and necessary references and attributes is a more complex case, as illustrated in the pattern template in Figure 8. First, in Figure 8, R(s,t) must be true if the reference is present in the partial model, so it is a safe under-approximation. However, possible pairs of

objects included in an over-approximation of R(s,t) have to satisfy several structural constraints (see Section 2.2 for more details):

1. **Type compliance:** If there is a reference between a source class S and a target class T, then a possible reference can only be instantiated between possible instances of S and T, so if the types are not correct, R(s,t) is excluded from the over-approximation.
2. **Upper multiplicity:** If there is an upper multiplicity defined for the reference (in the form of min..max), then the number of outgoing references must be less than max in order to possibly create a new reference. If this constraint fails, R(s,t) is excluded from the over-approximation.
3. **Upper multiplicity of inverse:** Similarly, if the reference S has an inverse reference I with an upper bound max, then the number of incoming R references to the target (which is the same as the number of outgoing I references from the target) must be less than max in order to possibly create a new reference.
4. **Containment reference, multiple parent:** There are two ways of violating the containment hierarchy with an additional reference. The first case is when an additional parent is created for an object. So, if there is a containment reference to a target object, then it is not possible to add another containment reference.
5. **Containment reference, circular containment:** Another way of violating the containment hierarchy is to create a circle with a new containment reference. Therefore it is not possible to create a containment reference between object s and t if there is a path of containments from t to s.
6. **Inverse of a containment reference:** If the inverse of R is a containment reference, then a may match has to satisfy **multiple parent** and **circular containment** rules.

In the following, we describe how the previous properties indexed with *may* and *must* modalities can be combined to create approximated patterns.

4.4 Transforming Approximated Patterns

A pattern p defines several structural constraints on a match m of model M; it is satisfied when then the pattern condition holds, which is denoted by $[M, m \models p]$. The condition of a pattern is defined as a disjunction of pattern bodies, which is defined by a set of constraints that has to be simultaneously satisfied:

$$[M, m \models p] \Leftrightarrow \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} constraint(m, var)$$

Our approach is to create an over- and under-approximated version of the pattern condition by using the previously over- and under-approximated versions of

```
pattern mustReference_R(p, s, t) {
  PartialModel.elements(p, s);PartialModel.elements(p, t); S.R(s, t);
}
pattern mayReference_R(p, s, t) {
  // The partial model is Open World
  PartialModel.openWorld(p,true);
  // There are s and t with the correct types
  find mayType_S(p,s); find mayType_T(p,t);
  // Upper multiplicity of R allows the addition of a new reference
  numberOfExistingReferences == count find mustReference_R(p, s, _);
  check(numberOfExistingReferences < {upper multiplicity of R});
  // Upper multiplicity of the inverse reference I allows the addition
  numberOfOppositeReferences == count find mustReference_I(p, _, t);
  check(numberOfOppositeReferences < {upper multiplicity of I});
  // If R is a containment relation, the new reference cannot create
  //  1. Multiple parents
  neg find mustContains(p,_,t);
  //  2. Circle in the containment hierarchy
  neg find mustTransitiveContains(p,t,s);
  // If I is the inverse of R, and R is a containment relation,
  // then the new reference cannot create
  //  1. Multiple parents
  neg find mustContains(p,s,_);
  //  2. Circle in the containment hierarchy
  neg find mustTransitiveContains(p,s,t);
} or {
  find mustReference_R(p, s, t);
}

// Support patterns, where R_1 ... R_n are containment references
pattern mustContains(p,s,t)
  {mustReference_R_1(p,s,t);} or ... or {mustReference_R_n(p,s,t);}
pattern mustTransitiveContains(p,s,t) {
  find mustContains+(p,s,t);}
```

Figure 8: An approximation of reference predicates

atomic constraints. So an over-appriximated version of a pattern is created by over-approximating each constraint in it, which creates a valid overapproximation of the pattern:

$$\Diamond[P, m \models p] = \Diamond\Big[P, m \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} constraint(m, var)\Big] \Rightarrow$$

$$\Big[P^O, m^O \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} [constraint(m, var)]^O\Big] = [P^O, m^O \models p^O]$$

And similarly, an under-approximated version of a pattern is created when each constraint is replaced by an under-approximated one:

$$\Box[P, m \models p] = \Box[M, m \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} constraint(m, var)] \Leftarrow$$

$$[P^U, m^U \models p^U] = [M^U, m^U \models \bigvee_{\text{bodies}} \exists var : \bigwedge_{\text{constraints}} [constraint(m, var)]^U],$$

```
1  // Original pattern
2  pattern noEntry(r : Region) {
3    neg find entryInRegion(r, _);
4  }
5
6  // Must version
7  pattern mustPattern_noEntry(p,r)
8  {
9    find mustType_Region(p,r);
10   neg find mayPattern_entryInRegion
11     (p,r,_);
12 }
13 // May version
14 pattern mayPattern_noEntry(p,r)
15 {
16   find mayType_Region(p,r);
17   neg find mustPattern_entryInRegion
18     (p,r,_);
19 }
20 // Original pattern
21 pattern multipleEntry(r) {
22   find entryInRegion(r, e1);
23   find entryInRegion(r, e2);
24   e1 != e2; }
25 // Must version
26 pattern mustPattern_multipleEntry(p,r){
27   find mustPattern_entryInRegion(
28     p,r,e1);
29   find mustPattern_entryInRegion(
30     p,r,e2);
31   neg find mayEqual(p,e1,e2); }
32 // May version
33 pattern mayPattern_multipleEntry(p,r){
34   find mayPattern_entryInRegion(
35     p,r,e1);
36   find mayPattern_entryInRegion(
37     p,r,e2);
38   neg find mustEqual(p,e1,e2); }
```

Figure 9: Examle under- and overapproximated patterns

where $[constraint]^U$ symbolizes the under-approximated (must) version of constraint, and $[constraint]^O$ denotes the over-approximated (may) version of it. In the latter modality, the variables and parameters can be bound to symbolic objects, which is handled by our open world indexing implementation by matching newElements prototypes. The only remaining task is to replace constraints of the original match by calling the must or may variant of the corresponding constraint. In Figure 9, there are two patterns with under- and over-approximated patterns:

- **Pattern:** Newly created patterns are prefixed with `mustPattern_` or `mayPattern_` and add a parameter `p` for the PartialModel object. For instance, as shown in Figure 9, `mustPattern_noEntry(p,r)` is the under-approximated version of `noEntry(r)`.
- **Classifier Constraint:** We replace all C`(e)` constraints by new pattern calls to `find mustType_`C`(p,e)` or `find mayType_`C`(p,e)`.
- **Path Constraint:** We split path expressions into single reference and attribute constraints, and replace all occurrences of a single R`(s,t)` constraint by either `find mustReference_`R`(p,s,t)` or `find mayReference_`R`(p,s,t)`.
- **Equality Constraints:** in the case of equality constraints, replace all `a==b` by `find mustEqual(p,a,b)` or `find mayEqual(p,a,b)`. However, in the case of inequality (`a!=b`), the modality has to be changed to the dual:
  - $[$`a!=b`$]^U$ is replaced by `neg find mayEqual(p,a,b)`
  - $[$`a!=b`$]^O$ is replaced by `neg find mustEqual(p,a,b)`

  For example, the constraint `e1!=e2` in line 24 of Figure 9 expresses the fact, that the two entries are different. In the must version of the pattern, it is replaced by `neg find mayEqual(p,e1,e2)`, which excludes matches where the

two entries can be mapped to the same element. However, in the may version of the pattern, it is replaced by `neg find mustEqual(p,e1,e2)`, which exclude matches only where it is certain that the two matches are equal.

- **Pattern Call Constraints:** There are different rules to map pattern calls:
  - **Positive call:** a positively called pattern `find` *ref* `(par)` is mapped to `find mustPattern_`*ref*`(p,par)` or `find mayPattern_`*ref*`(p,par)`. For instance, `find entryInRegion(r, e1);` in line 22 of Figure 9 is replaced by `find mustPattern_entryInRegion(p,r,e1)` in the must variant of the pattern.
  - **Negative call:** In the case of a negative pattern call (`neg find`), the modality of the called pattern has to be changed to the opposite:
    - $[$`neg find` *ref* `(par)`$]^U$ replaced by `neg find mayPattern_`*ref*`(p,par)`
    - $[$`neg find` *ref* `(par)`$]^O$ replaced by `neg find mustPattern_`*ref*`(p,par)`

    For example, the negative pattern call `neg find entryInRegion(r,_)` in line 3 of Figure 9 is transformed to a negative may-pattern call `neg find mayPattern_entryInRegion(p,r,_)` constraint in the must version of the pattern, which forbids all possible matches where the entry can be in the checked region. Conversely, in the may version, it is transformed to `neg find mustPattern_entryInRegion(p,r,_)`, which filters the matches only where it is certain that there is an entry in the region.
  - **Transitive closure:** A transitive closure call `find` *ref*`+(par)` is transformed to either an under-approximated `find mustPattern_`*ref*`+(p,par)` or an over-approximated `find mayPattern_`*ref*`+(p,par)`.
- **Count find:** The number of occurrences of a pattern may be under- and over-approximated in several cases by replacing `C==count find` *ref* `(par)` by `C<=count find mustPattern_`*ref*`(par)` or `C>=count find mayPattern_`*ref*`(par)`. However, as the count of the matches (i.e. the value of `C`) can be used in other constraints, this constraint is supported only when `C` is a constant integer.
- **Eval, check:** In those features language that cannot be supported automatically, under- and over-approximated versions of the embedded expressions need to be created manually.

4.5 Interpretation of approximated matches

Calculating approximated matches The over- and under-approximated versions of matches can be matched on partial models (as regular instance models) by traditional graph pattern matching approaches. For this purpose, we have been using the incremental graph pattern matching engine of the VIATRA framework [4]. Since the approximated patterns are more complex than the original pattern, we expect that matching partial models will be slower. According to our initial

experiments, we expect a quadratic decrease in performance - which is still more efficient than using SAT/SMT solvers for the same purpose.

Interpreting approximated matches The calculated matches of approximated patterns on partial models need to be interpreted as must- and may-matches by projecting them onto the original partial model ($m = \pi_P(m^O)$ and $m = \pi_P(m^U)$). This projection is not computationally intensive since matches are only filtered but never extended.

4.6 Validation of the approach

In order to validate our approach, we developed automated transformations to (1) rewrite models into partial models, and (2) rewrite graph patterns into approximated patterns. We executed these transformations in the context of Yakindu statecharts (i.e. the running example of our paper).

Evaluating must and may matches on a partial model is always decidable, and the problem has polynomial time complexity (as it can be reduced to a subgraph isomorphism problem [27]). In contrast, logic analysis of graph patterns and partial models is, in general, undecidable [32]. Also, industrial graph query engines like [4] are able to efficiently evaluate graph queries over large models with thousands of objects. In our example, the execution time of this compilation step (i.e. the rewrite and execution of transformations) was negligible. Still, our previous experience showed that advanced SAT and SMT solvers failed to solve partial model analysis with the same metamodel and size [33].

Besides the abov, we systematically developed a test set with full metamodel coverage [6] and full inclusion of well-formedness constraints defined for Yakindu statecharts, which consists of prototype graph patterns. This test set is listed in the Appendix, and the generated output and further details are available in GitHub[1]. We generated both a may- and a must-approximation from the graph pattern of the designated constraint, and we evaluated these patterns on sample prototypical instance models. The correctness of the generation was established by manually inspecting the retrieved result set of the may and must patterns. In the future, we intend to carry out a more systematic performance evaluation of our approach.

5 Related Work

5.1 Analysis of Uncertain Models

Partial models are a subclass of *uncertain models*, which offer a rich specification language [14, 28] amenable to analysis. Uncertain models provide a more expressive language compared to partial models, but without handling additional WF constraints. Such models document semantic variation points generically via by means of annotations on a regular instance model. Most uncertain model analysis

[1] `https://github.com/FTSRG/publication-pages/wiki/Evaluating-Well-Formedness-Constraints-on-Partial-Model`

approaches focus on the generation of possible concrete models or the refinement of partial models. A potential concrete models compliant with an uncertain model may be synthesized by the Alloy Analyzer and its back-end SAT solvers [31, 30], or be refined by graph transformation rules [29].

The most similar approaches [15, 16] analyze the possibility of model transformation rule matching and execution on partial models by using a SAT solver (MathSAT4) or by automated graph approximation (referred to as "lifting"). The main difference is that in their approach inspects possible partitions of a concrete model (so checking all possible P-s for a given M with $P \subseteq M$), instead of possible extensions of a partial model. Therefore, it cannot be used to support a mostly incremental development process with growing models.

There is an extensive tool support for editing and analyzing advanced uncertain models [5]. In contrast, our approach may be applied on (unfinished) EMF models directly [4] in their own editors, to indicate must- and may-matches of ill-formedness constraints as errors and warnings.

5.2 Verification of Model Transformations

Besides uncertain models, there are several formal methods available that seek to evaluate graph patterns on abstract graph models (either abstract interpretation[25] or predicate abstraction [26]) in order to detect possible concretizations matches. Those techniques typically employ similar techniques called pre-matching to create may-matches that are further analyzed.

5.3 Logic Solver Approaches

There are several approaches available that map partial models and WF constraints into a logic problem, which are solved by underlying SAT/SMT-solvers. With these techniques the implication between a partial model and the satisfaction of a well-formedness constraint can be directly evaluated in order to reason about must- and may-matches.

Complete frameworks with standalone specification languages include Formula [20] (which uses the Z3 SMT- solver [13]), Alloy [19] (which relies on SAT solvers like Sat4j[22]) and Clafer [2] (using backend reasoners like Alloy).

There are several approaches which seeks to validate standardized engineering models enriched with OCL constraints [17] by relying upon different back-end logic-based approaches such as constraint logic programming [11, 10, 8], SAT-based model finders (like Alloy) [34, 1, 9, 21, 35], first-order logic [3], constructive query containment [24], higher-order logic [7, 18], and rewriting logics [12]. Partial snapshots and WF constraints can be uniformly represented as constraints [32].

The scalability of all these approaches are limited to small models / counterexamples. Furthermore, these approaches are either a priori bounded (where the search space needs to be explicitly restricted) or they have decidability issues [33].

6 Conclusions and Future Work

Here we presented a novel validation technique that is able to check unfinished, partial models with well-formedness constraints that are defined for fully defined models. The outcome of searching a malformed model partitions can be either a must-match (which detects whether a partial model already contains a conceptual flaw), a may-match (which means that a possible extension may become invalid, and highlights all possible threats), or no match (which proves that the partial model already satisfies a validation rule). The approach is based on an approximation technique that reduces must- and may-matching to regular pattern matching problems, which can be executed by graph query engines.

As a future work we are planning to integrate our pattern matching technique with advanced partial modeling formalisms like [14] to increase the expression power of the analysis. Also, we intend to carry out a systematic performance evaluation, and experiment with guiding incremental model generators processes [33] by evaluating must- and may-matches on intermediate solutions.

References

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[2] Kacper Bak, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, pages 1–35, 2013.

[3] B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *Proc. of the VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[4] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for emf models. In *Fourth International Conference on Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 167–182. Springer, 2011.

[5] Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors. *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. IEEE Computer Society, 2015.

[6] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *17th International Symposium on Software Reliability Engineering, 2006. ISSRE '06.*, pages 85–94, Nov 2006.

[7] A. D. Brucker and B. Wolff. The HOL-OCL tool, 2007. `http://www.brucker.ch/`.

[8] Fabian Büttner and Jordi Cabot. Lightweight string reasoning for OCL. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings*, volume 7349 of *LNCS*, pages 244–258. Springer, 2012.

[9] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. Verification of ATL transformations using transformation models and model finders. In *14th International Conf. on Formal Engineering Methods,ICFEM'12*, pages 198–213. LNCS 7635, Springer, 2012.

[10] J. Cabot, R. Clariso, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conf. on*, pages 73–80, April 2008.

[11] Jordi Cabot, Robert Clarisó, and Daniel Riera. 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)*, pages 547–548, New York, NY, USA, 2007. ACM.

[12] M. Clavel and M. Egea. The ITP/OCL tool, 2008. `http://maude.sip.ucm.es/itp/ocl/`.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering*, pages 573–583, Piscataway, NJ, USA, 2012. IEEE Press.

[15] Michalis Famelis, Rick Salay, and Marsha Chechik. The semantics of partial model transformations. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 64–69. IEEE Press, 2012.

[16] Michalis Famelis, Rick Salay, Alessio Di Sandro, and Marsha Chechik. Transformation of models containing uncertainty. In *International Conference on Model Driven Engineering Languages and Systems*, pages 673–689. Springer, 2013.

[17] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and Systems Modeling*, 4:386–398, 2005.

[18] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 152–166. Springer, 2009.

[19] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[20] Ethan K Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Model Driven Engineering Languages and Systems*, pages 653–667. Springer, 2011.

[21] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into use. In *TOOLS'11 - Objects, Models, Components and Patterns*, volume 6705 of *LNCS*, pages 290–306, 2011.

[22] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[23] The Object Management Group. *Object Constraint Language, v2.0*, May 2006.

[24] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.*, 73:1–22, 2012.

[25] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, 157(1):39–59, 2006.

[26] Thomas W Reps, Mooly Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In *International Conference on Computer Aided Verification*, pages 15–30. Springer, 2004.

[27] Grzegorz Rozenberg. *Handbook of Graph Grammars and Comp.*, volume 1. World scientific, 1997.

[28] Rick Salay and Marsha Chechik. A generalized formal framework for partial modeling. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033 of *LNCS*, pages 133–148. Springer Berlin Heidelberg, 2015.

[29] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. A methodology for verifying refinements of partial models. *Journal of Object Technology*, 14(3):3:1–31, 2015.

[30] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a methodology for verifying partial model refinements. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 938–945. IEEE, 2012.

[31] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 224–239. Springer Berlin Heidelberg, 2012.

[32] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software and Systems Modeling*, pages 1–36, 2015.

[33] Oszkár Semeráth, András Vörös, and Dániel Varró. Iterative and incremental model generation by logic solvers. In *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 87–103, 2016.

[34] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.

[35] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation and Test in Europe, (DATE'10)*, pages 1341–1344. IEEE, 2010.

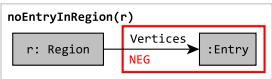
[36] The Eclipse Project. *Eclipse Modeling Framework*. `//www.eclipse.org/emf`.

[37] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, October 2007.

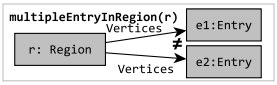
[38] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[39] Yakindu Statechart Tools. *Yakindu*. `http://statecharts.org/`.

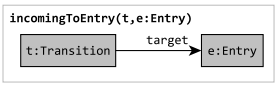
Appendix



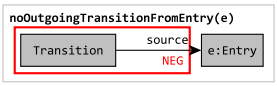

```
pattern entryInRegion(r:Region, e:Entry) {
  Region.vertices(r,e); }
@Constraint pattern noEntryInRegion(r:Region) {
  neg find entryInRegion(r,_); }
```



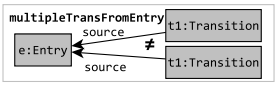

```
@Constraint pattern multipleEntryInRegion(r) {
  find entryInRegion(r,e1); find entryInRegion(r,e2);
  e1 != e2; }
```



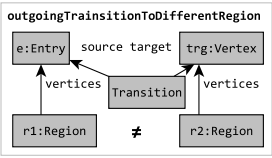

```
pattern transition(t,src,trg) {
  Transition.source(t,src); Transition.target(t,trg); }
@Constraint pattern incomingToEntry(t, e:Entry) {
  find transition(t,_,e); }
```



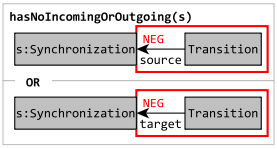

```
@Constraint
pattern noOutgoingTransitionFromEntry(e:Entry) {
  neg find transition(_,e,_);
}
```



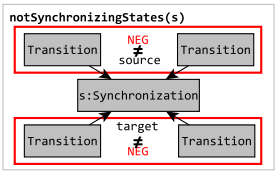

```
@Constraint pattern multipleTransitionFromEntry(e:Entry,t1,t2) {
  find transition(t1,e,_); find transition(t2,e,_);
  t1! == t2;
}
```



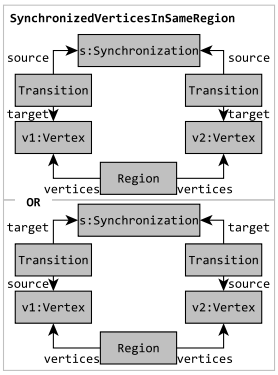

```
@Constraint
pattern outgoingTrainsitionToDifferentRegion(e:Entry,trg,r) {
  find transition(_,e,trg);
  Region.vertices(r1,e);
  Region.vertices(r2,trg);
  r1 != r2;
}
```



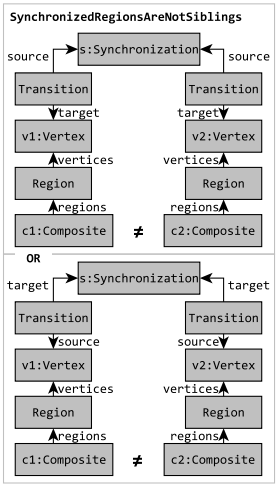

```
@Constraint hasNoIncomingOrOutgoing(s:Synchronization) {
  neg find transition(_,_,s);
} or {
  neg find transition(_,s,_);
}
```




```
private pattern hasMultipleOutgoingTrainsition(v) {
  find transition(_,v,trg1); find transition(_,v,trg2); trg1 != trg2; }
private pattern hasMultipleIncomingTrainsition(v) {
  find transition(_,src1,v); find transition(_,src2,v); src1 != src2; }
@Constraint notSynchronizingStates(s:Synchronization) {
  neg find hasMultipleOutgoingTrainsition(s);
  neg find hasMultipleIncomingTrainsition(s); }
```




```
@Constraint
pattern SynchronizedVerticesInSameRegion(s:Synchronization,v1,v2) {
  find transition(t,v1,s);
  find transition(t,v2,s);
  Region.vertices(r,v1);
  Region.vertices(r,v2);
} or {
  find transition(t,s,v1);
  find transition(t,s,v2);
  Region.vertices(r,v1);
  Region.vertices(r,v2);
}
```




```
@Constraint
pattern SynchronizedRegionsAreNotSiblings(s:Synchronization,v1,v2) {
  find transition(t,v1,s);
  find transition(t,v2,s);
  CompositeElement.regions.vertices(r1,v1);
  CompositeElement.regions.vertices(r2,v2);
  r1 != r2;
} or {
  find transition(t,s,v1);
  find transition(t,s,v2);
  CompositeElement.regions.vertices(r1,v1);
  CompositeElement.regions.vertices(r2,v2);
  r1 != r2;
}
```