JOT
JOURNAL OF
OBJECT TECHNOLOGY

From Constraints to Commands: Graph Predicate Differentiation in 4-Valued First-Order Logic

Attila Ficsor, Inez Anna Papp, Kristóf Marussy, and Oszkár Semeráth
Department of Artificial Intelligence and Systems Engineering, Budapest University of Technology and Economics, Hungary

ABSTRACT During the early phase of modeling, our knowledge about the model under development is frequently partial. Partial modeling technologies enable the explicit representation of partial knowledge, which allows developers to consider design alternatives, and manage uncertainties and potential contradictions. However, partial modeling is challenging, as models need to adhere to design rules and well-formedness constraints, which are complex logic predicates. Therefore, during manual and automated model refinement, it is important to exclude design alternatives that violate such constraints as early as possible to reduce the number of invalid design decisions.

In this paper, we propose a technique to translate well-formedness constraints to propagation rules, which are logic reasoning steps that can exclude invalid design alternatives automatically. Thus, propagation rules can reduce the number of apparent design decisions to improve the productivity of development. Moreover, by applying propagation rules during automated generation of design alternatives, the proposed technique can improve the performance of logic solvers significantly. We integrated the proposed technique to the Refinery graph generation framework, and illustrated the performance improvement with case studies.

KEYWORDS Partial modeling, Graph generation, Graph predicates, Graph transformation, Logic inference

1. Introduction

1.1. Context: Modeling with uncertainty

In the early stages of design, knowledge about models is often incomplete. Despite this, development processes and environments typically demand fully specified models, forcing designers to encode unknown aspects with comments, placeholders, or to make premature design decisions. More recently, modern environments have begun to incorporate mechanisms for handling uncertainty in models in multiple different ways. *Advanced knowledge bases* such as TypeDB (Dorn & Pribadi 2023) support open-world semantics, allowing models to remain flexible in the face of incomplete information. *Software product lines* provide variability modeling techniques that capture alternative design choices and deferred decisions, allowing uncertainty to be expressed as configurable features (Liang et al. 2015). And finally, *partial modeling environments* are designed to enable explicit representation of uncertainty, either through specialized annotations (Famelis et al. 2012; Troya et al. 2021) or by accommodating unknown values directly.

With advanced tool support, modeling environments enable the documentation of uncertainty during development, help the developer by analyzing the feasibility of design alternatives, and potentially enumerate different configurations.

1.2. Constraint evaluation in partial models

During development, models must adhere to the design rules of their domain, which are commonly expressed as well-formedness constraints (Bergmann et al. 2015; Marussy et al. 2024). Consequently, even in uncertain modeling, developers must ensure that the represented model alternatives remain well-formed and comply with all domain rules. Likewise, automated analyses and configuration generation can only operate on models that satisfy these constraints. Without incorporating

JOT reference format:
Attila Ficsor, Inez Anna Papp, Kristóf Marussy, and Oszkár Semeráth. *From Constraints to Commands: Graph Predicate Differentiation in 4-Valued First-Order Logic.* Journal of Object Technology. Vol. 25, No. 3, 2026.
 http://dx.doi.org/10.5381/jot.2026.25.3.a4

well-formedness constraints into uncertainty modeling, developers cannot differentiate between design decisions that lead infeasible designs.

It is therefore crucial to assess the impact of well-formedness constraints on uncertain models, enabling the elimination of design alternatives that inevitably violate at least one constraint. The goal of this paper is to introduce a technique that automatically refines uncertain models by evaluating the impact of well-formedness constraints. Through this evaluation, the precision of uncertain models can be systematically improved, while design alternatives that necessarily violate constraints are eliminated automatically.

1.3. Contributions and added value

In this paper, we propose a technique that automatically analyzes well-formedness constraints defined as graph predicates and translates them into a group of graph rewriting rules. These rules can be applied on the representation of partial models directly, thereby resolving uncertainties that would otherwise lead to violations of well-formedness constraints. As a result, partial models can be refined automatically without restricting the range of valid design options. Our contributions are :

- We extended a SAT solver algorithm (Wittocx et al. 2013) to 4-valued first-order graph predicate differentiation technique to support quantifiers ($\forall$, $\exists$), transitive closure (R^+) in relevant cases, aggregations (counting the size of sets), and uninterpreted node existence and equivalences (Semeráth et al. 2018; Marussy et al. 2024).
- We used this technique to translate well-formedness constraints, into 4-valued partial model refinement rules.
- We implement this algorithm within the Refinery framework (Marussy et al. 2024), where rules can be efficiently executed using an incremental graph transformation engine (Bergmann et al. 2015).
- We demonstrated that the use of the transformations deliver significant improvements in reducing uncertainty during partial modeling.
- We integrate the algorithm into a graph generation protocol and show that it substantially enhances the performance of design alternative enumeration.
- We empirically evaluate our approach across multiple case studies.

Therefore, our approach derives refinement rules that can be efficiently applied to partial models, thereby reducing uncertainty during the modeling process. By systematically eliminating invalid alternatives, the approach provides a more precise view of the valid design space in partial modeling. Furthermore, we demonstrate that our technique significantly decreases runtime requirements during model generation.

2. Theoretical Background

2.1. Motivating example: railway network

As a case study, we are using a simple metamodel and well-formedness constraints based on the Train Benchmark (Szárnyas et al. 2018). The Train Benchmark is designed to assess the scalability of validating well-formedness constraints over large graph models. The benchmark defines a metamodel and well-formedness constraints in the railway domain. A simplified version of the metamodel is illustrated in Listing 1 in the language of Refinery (Marussy et al. 2020), which specification language closely mimics the XCore syntax ("Xcore" 2025) of Eclipse Modeling Framework ("Eclipse Modeling Framework" 2025). In our simplified version (illustrated in Figure 1a) we focus on the physical connection between the track elements (`TrackElement`), which can be either `Segment`s or `Switch`es. `TrackElement`s are joined with the `connectsTo` reference to describe the physical topology of the network. Moreover, references `straight` and `diverging` denotes the two possible positions of a `Switch`.

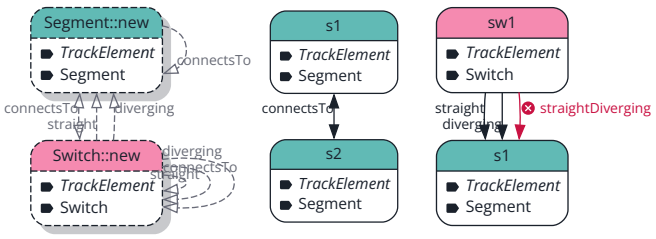
```
abstract class TrackElement {
    TrackElement[1..3] connectsTo
        opposite connectsTo
}
class Segment extends TrackElement.
class Switch extends TrackElement {
    TrackElement straight
    TrackElement diverging
}
```

Listing 1 Metamodel of the simplified railway network in the language of Refinery

In our case study, we consider models where a `Switch` is connected to the same `TrackElement` with both its `straight` and `diverging` relation, to be erroneous. We express this rule as an `error` (a predicate that should be false in a valid model) called `straightDiverging` (illustrated in Listing 2). It searches for a pair of model elements if there is a `straight` and a `diverging` relation pointing from one to the other. An example where this predicate is true is illustrated in Figure 1c.

```
error straightDiverging(switch, target) <->
    straight(switch, target),
    diverging(switch, target).
error noDiverging(switch) <->
    Switch(switch),
    !diverging(switch,_).
```

Listing 2 Constraints of the simplified railway network in the language of Refinery



(a) Simplified metamodel P_1

(b) Concrete model P_2

(c) Model P_3 matching error predicate

Figure 1 Example models

The `noDiverging` `error` states that a model is incorrect if

a **Switch** element does not have a **diverging** relation. From this, we can deduce that a *TrackElement* that cannot have a **diverging** relation can only be a Segment. Furthermore, if there is only one possible *TrackElement* that can be the **diverging** path of a **Switch**, than it must be set as such, otherwise the model would contain an error.

The goal of this paper is to introduce similar reasoning steps and apply it to all error predicates to deduce further information about our models.

2.2. 4-valued logic

In this paper, we use 4-valued logic to explicitly represent unfinished, partial models, as well as errors and inconsistencies in such models. This section provides semantic foundations for our definitions of unit propagation, building on the semantics of the specification language of Refinery (Marussy et al. 2020), which is based on the inconsistency-tolerant Belnap-Dunn 4-valued logic (Belnap 1977; Kamide & Omori 2017). Belnap-Dunn 4-valued logic uses $\mathbb{B}^4 := \{\texttt{true}, \texttt{false}, \texttt{unknown}, \texttt{error}\}$ as its set of truth values, which, in addition of the usual **true** and **false** values, introduces the **unknown** value is for unspecified or unknown properties, and the **error** value for signaling inconsistencies (potentially caused by error predicates).

Two partial orders can be defined over 4-valued logic values (Figure 2f). *Information order*, also called *refinement* relation (denoted by $\succcurlyeq$) expresses the gathering of information as new facts are learned during the refinement of partial models. Facts with **unknown** logical value can be refined to either **true** or **false** while a change to **error** signifies an inconsistency.

The *information merge* operator $\curlywedge$, also called *meet* operator, merges 4-valued truth values where contradictory information results in **error**. The *join* operator $\curlyvee$ joins 4-valued truth values where contradictory information results in **unknown**. Other operations on 4-valued truth values, *negation* ($!$), *and* ($\wedge$) and *or* ($\vee$) are extensions of the usual logic operators. Two modalities, **may** and **must** are also defined as operators on 4-valued truth values. The **may** modality of a truth value is true if it can be refined to **true** false otherwise. The **must** modality of a truth value is true if is a refinement of **true** false otherwise. The truth tables of all operators is shown in Figure 2.

2.3. Partial models

Partial models represent uncertainty and inconsistency in the models explicitly using 4-valued logic. Mathematically, partial models are represented as logic structures defined on a signature.

Definition 2.1 (Signature) *A signature is a pair* $\langle \Sigma, \alpha \rangle$*, where*

- $\Sigma = \{S_1, \ldots, S_s, \texttt{exists}, \texttt{equals}\}$ *is a set of symbols* $S_1, \ldots, S_s$*, a special unary symbol* **exists** *denoting the existence of objects and a special binary symbol* **equals** *denoting the equivalence of two objects,*
- $\alpha : \Sigma \to \mathbb{N}$ *is the arity function* ($\alpha(\texttt{exists}) = 1$ *and* $\alpha(\texttt{equals}) = 2$*).*

Example 2.1 *Signature of the case study:* $\Sigma = \{$*TrackElement*, **Segment**, **Switch**, **connectsTo**, **diverging**, **straight**, **straightDiverging**, **exists**, **equals**$\}$. α *is 1 for* *TrackElement*, **Segment**, **Switch** *and* **exists**, *and 2 for the other symbols.*

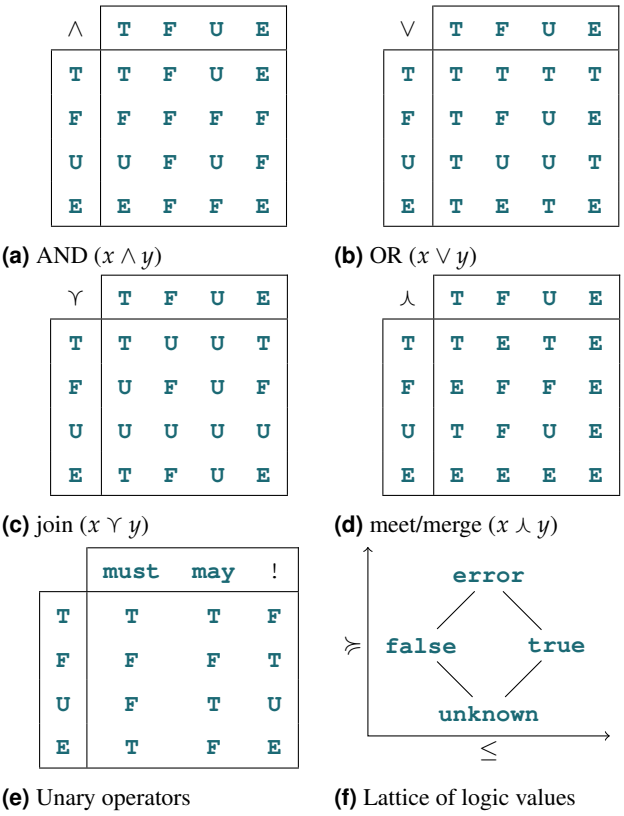


Figure 2 Operations and logic values in 4-valued logic

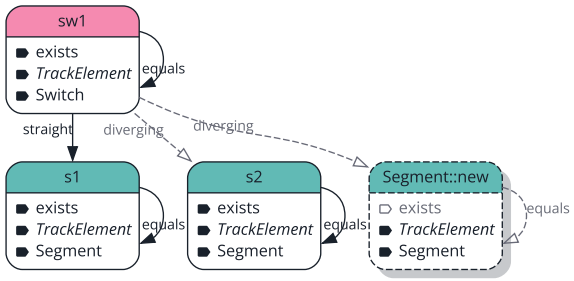
Then we can introduce partial models as logic structures with 4-valued logic interpretation for all symbols.

Definition 2.2 (Partial Model) *A partial model* $P = \langle \mathcal{O}_P, \mathcal{I}_P \rangle$ *is a logic structure, where:*

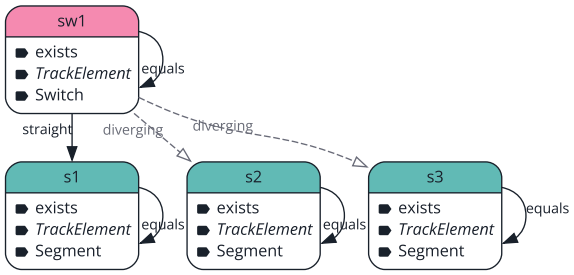
- $\mathcal{O}_P$ *is the finite set of objects in the model,*
- $\mathcal{I}_P$ *gives 4-valued logic interpretation for each symbol* $S \in \Sigma$ *as* $\mathcal{I}_P(S) : \mathcal{O}_P^{\alpha(S)} \to \mathbb{B}^4$

Example 2.2 *A partial model can be seen in Figure 3a, which contains a single* **Switch** *element called* `sw1` *and two* **Segment** *elements,* `s1` *and* `s2`*. We know that these exist, which is indicated by the solid marker next to the* **exists** *keyword and also by the solid outline of the elements. They are also equal to themselves, which means they each represent a single model element. There may also be further* **Segment** *elements in the model that are not specified yet. This uncertainty is shown by the* **Segment**`::new` *element, which is a* **Segment**, *but its existence is uncertain, indicated by the empty marker next to* **exists** *and the dashed outline. It also may not equal itself, meaning it can represent multiple* **Segment** *elements.*

The `sw1` **Switch** *has its* **straight** *direction set to* `s1`*. Its* **diverging** *relation however is not set yet, it can point to either* `s2`*, or a new* **Segment**, *which we have not created yet.*



(a) Partial model



(b) Partial model after refinement

Figure 3 Partial model refinement

2.4. Partial model refinement

We define refinement on a partial model as changing truth values with respect to truth-value refinement relation $\succcurlyeq$. During refinement, uncertain **unknown** values can be refined to either **true** or **false**, while **true** or **false** values can be refined to **error**.

Definition 2.3 (Partial Model Refinement) *A partial model $P = \langle \mathcal{O}_P, \mathcal{I}_P \rangle$ is refined to a partial model $Q = \langle \mathcal{O}_Q, \mathcal{I}_Q \rangle$ (which is denoted $P \succcurlyeq Q$) if there is a refinement function between the objects of P and Q such as $r : \mathcal{O}_P \to 2^{\mathcal{O}_Q}$ that respects the information ordering:*

- *For each n-ary symbol $S \in \Sigma$, each object $p_1, \ldots, p_n \in \mathcal{O}_P$, and for each refinement $q_1 \in ref(p_1), \ldots, q_n \in ref(p_n)$:*

$$\mathcal{I}_P(S)(p_1, \ldots, p_n) \succcurlyeq \mathcal{I}_Q(S)(q_1, \ldots, q_n)$$

- *All objects in Q are refined from an object in P, and existing objects $p \in \mathcal{O}_P$ must have a non-empty refinement.*

Example 2.3 *The partial model in Figure 3b is a refinement of the partial model in Figure 3a. The elements* `sw1`, `s1`, *and* `s2` *did not change, but* **`Segment::new`** *was refined to a concrete element* `s3`*. This new element exists, equals itself, and it still may be the* **`diverging`** *path of* `sw1`*. This means, that the* **`exists`** *and* **`equals`** *interpretation values for this element were refined from* **`unknown`** *to* **`true`***. The names of the element (e.g.* **`Segment`**`::new` *or* `s3`*) are only added to make the models easier to understand, they are not part of the partial model, therefore their change does not affect the refinement relation.*

When all the uncertainty is resolved in a partial model, we get a concrete model, which represents a concrete design alternative. In our notation, concrete models are a special subset of partial models.

Definition 2.4 (Concrete models and concretization) *A partial model M is* concrete *if M contains no* **unknown** *or* **error** *values. A partial model M is the* concretization *of P if $P \succcurlyeq M$ and M is concrete.*

2.5. Graph predicates

In this paper we use first order logic predicates defined with a logic expression in disjunctive normal form (DNF). Each predicate is constructed with variables $v_1, v_2, \ldots$, which can be used as parameters $p_1, p_2, \ldots$, or symbol (;), and symbol (,), equivalence and unequivocal symbols ($=$ and $\neq$), negation symbol (!), transitive closure ($+$), and aggregation (**`count`**$\{\cdot\} > c$ or **`count`**$\{\cdot\} < c$). Moreover, we introduce two modalities to represent different levels of certainty: **`must`** and **`may`**.

Definition 2.5 (Syntax of literal) *A literal is given in one of the following formats:*

- $L_l^c = MXST(v_1, \ldots, v_a)$*, where:*
  - *M is the modality of the literal, which can be* **`must`**`,` **`may`***, or partial (no keyword).*
  - *X is the polarity of the literal, which is either positive (no keyword) or negative (!).*
  - *T is the transitivity of the literal, which can be transitive closure ($+$), or no transitivity (no keyword).*
- *L_l^c is an aggregation and comparison to a constant value c:* COMP(**`count`**$\{S(v_1, \ldots, v_a)\}$, c)*, where* COMP *is a comparison operation from Table 1.*
- *L_l^c is an equivalence check between two variables v_i and v_j: $v_i{=}v_j$ or $v_i{\neq}v_j$.*

where S is a symbol with arity $\alpha(S) = a$ and $v_1, \ldots, v_a$ are variables used in the literals.

Then, a predicate can be constructed with literals in disjunctive normal form.

Definition 2.6 (Syntax of predicate) *An n-ary logic symbol S is defined as a predicate in the format of* **`pred`** *$S(p_1, \ldots, p_n) \leftrightarrow \varphi$, where φ is a logic expression disjunctive normal form $\varphi = C_1; \ldots; C_c; \ldots; C_n$, and each clause C_c is given as a conjunction of literals: $C_c = L_1^c, \ldots, L_l^c, \ldots, L_m^c$.*

Example 2.4 (Syntax of predicate) *Examine the* **`reachable`** *predicate in Listing 3:*

This predicate defines a binary logic symbol **`reachable`***, with two clauses in its logic expression, each a conjunction of two literals. The first literal of both clauses are $L_1^1 = L_1^2 =$*

COMP	Syntax	Rewriting
LESS	$a<b$	
GREATER	$a>b$	
LESSEQ	$a\leqslant b$	$a<b+1$
GREATEREQ	$a\geqslant b$	$a>b-1$
EQ	$a=\!=b$	$a>b-1, a<b+1$
NOTEQ	$a\neq b$	$a<b; a>b$

Table 1 Syntax and rewriting of integer comparison operators

`TrackElement(source)`, *where the modality is* **must**, *the polarity is positive, the symbol is* `TrackElement` *with arity of* 1, *there is no transitivity, and the only variable is* $v_1 =$ `source`.

```
pred reachable(source, target) ⟷
    TrackElement(source),
    source == target
;
    TrackElement(source),
    connectsTo+(source, target).
```

Listing 3 Predicate for connected `TrackElement` objects

L_2^1, *the second literal of the first clause is a positive equivalence check, with* $v_1 =$ `source` *and* $v_2 =$ `target`.

Finally, the second literal of the second clause is L_2^1, *which is a positive transitive closure of the* **connectsTo** *symbol with* **must** *modality and variable assignments* $v_1 =$ `source` *and* $v_2 =$ `target`.

Definition 2.7 (Variable binding) *A* variable binding *of a predicate* **pred** $S(p_1,\ldots,p_n) \leftrightarrow \varphi$ *with parameters* $\{p_1,\ldots,p_n\}$ *on a partial model* $P = \langle \mathcal{O}_P, \mathcal{I}_P \rangle$ *is a function* $Z : \{p_1,\ldots,p_n\} \to \mathcal{O}_P$ *mapping the parameters to objects.*

Definition 2.8 (Semantics of predicates) *The semantics of a predicate* **pred** $S(p_1,\ldots,p_n) \leftrightarrow \varphi$ *on a partial model* $P = \langle \mathcal{O}_P, \mathcal{I}_P \rangle$ *with variable binding* $Z : \{p_1,\ldots,p_n\} \to \mathcal{O}_P$ *is denoted as* $[\![\varphi]\!]_P^Z$ *and defined as:*

$$[\![C_1; \ldots; C_n]\!]_P^Z := [\![C_1(p_1,\ldots,p_n)]\!]_P^Z \vee \ldots \vee [\![C_n(p_1,\ldots,p_n)]\!]_P^Z$$

$$[\![C(p_1,\ldots,p_n)]\!]_P^Z := [\![\langle \textit{positive binding} \rangle \wedge (L_1^C, \ldots, L_m^C)]\!]_P^Z,$$

where $\langle$*positive binding*$\rangle$ *is a conjunction of* **exists**(p) *for all unbound positive variables* $p \neq p_1,\ldots,p_n$, *and*

$$[\![L_1^C, \ldots, L_n^C]\!]_P^Z := [\![L_1^C]\!]_P^Z \wedge \ldots \wedge [\![L_n^C]\!]_P^Z.$$

A logic literal *is defined as:*

$$[\![MXST(v_1,\ldots,v_a)]\!]_P^Z := MXT\mathcal{I}_P(S)(Z(v_1),\ldots,Z(v_a)) \vee \langle \textit{negative binding} \rangle,$$

where

- *modality (***may** X *or* **must** X*) and*
- *polarity (*X *or* $!X$*) is defined according to Figure 2,*
- *and transitivity (*$\mathrm{S}{+}(v_1, v_2)$*) means that there is a sequence of objects* $o_1,\ldots,o_n \in \mathcal{O}_P$ *where* $Z(v_1) = o_1$, $Z(v_2) = o_n$, *and* $\bigwedge [\![S(a,b)]\!]_{a\mapsto o_i, b\mapsto o_{i+1}}^P$,
- *and* $\langle$*negative binding*$\rangle$ *is a disjunction of* **!exists**(p) *for all unbound negative variables introduced in the literal.*

An aggregation literal *is defined as:*

$$[\![\mathrm{COMP}(\textbf{count}\{S(v_1,\ldots,v_a)\}, \mathrm{c})]\!]_P^Z := \mathrm{COMP}(\{o_1,\ldots,o_n \in \mathcal{O}_P | [\![S(v_1,\ldots,v_a)]\!]_P^Z\}, \mathrm{c})$$

An equivalence literal *is defined as:*

$$[\![p == q]\!]_P^Z := \mathcal{I}(\textbf{equals})(p,q).$$

Based on the predicates, we can introduce validation and refinement rules.

2.6. Refinement rules

Definition 2.9 (Syntax of transformation rules) *A transformation rule is defined as*

rule $R(p_1,\ldots,p_a) \longleftrightarrow \varphi \Longrightarrow S(p_1,\ldots,p_a) : T$. *where:*

- $R(p_1,\ldots,p_a) \longleftrightarrow \varphi$ *is a predicate definition, and*
- $S(p_1,\ldots,p_a) : T$ *is an action, where:*
  - S *is a symbol with arity* $\alpha(S) = a$,
  - $p_1,\ldots,p_a$ *are the parameter variables, and*
  - $T \in \mathbb{B}^4$ *is a truth-value.*

Definition 2.10 (Semantics of transformation rules) *A transformation rule* **rule** $R(p_1,\ldots,p_a) \longleftrightarrow \varphi \Longrightarrow S(p_1,\ldots,p_a) : T$. *applied on a partial model* P *with match* m *produces a partial model* Q, *if* $[\![\varphi]\!]_P^m =$ **true**, *and there is a relation* $r : \mathcal{O}_P \to \mathcal{O}_Q$, *where*

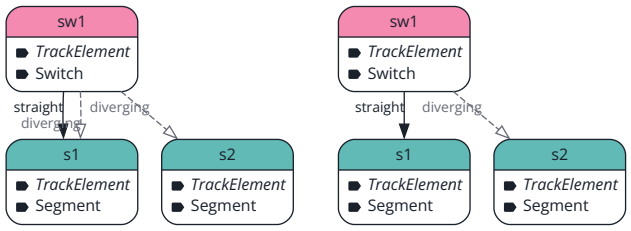
- $\mathcal{I}_Q(S)(r(m(p_1)),\ldots,r(m(p_a))) = \mathcal{I}_P(S)(m(p_1),\ldots,m(p_a)) \curlywedge T$
- *and for all other combination of* S' *symbols and* $o_1,\ldots,o_n$ *objects* $\mathcal{I}_Q(S')(r(o_1),\ldots,r(o_n)) = \mathcal{I}_P(S')(o_1,\ldots,o_n)$.

When we apply a refinement rule R on a partial model P it produces a more refined partial model Q by resolving uncertainty.

Example 2.5 *Take the* `removeDiverging` *refinement rule below as an example. It states, that when it is applied to a pair of elements* `switch` *and* `trackElement`, *the* **diverging** *relation between them will be set to* **false**. *This rule can be applied to model elements that satisfy its precondition.*

```
rule removeDiverging(switch, trackElement) ⟷
  Switch(switch),
  TrackElement(trackElement)
  must straight(switch, trackElement),
  may diverging(switch, trackElement),
  !must diverging(switch, trackElement)
⟹
  !diverging(switch, trackElement).
```

In model P shown in Figure 4a, model elements `sw1` *and* `s1` *satisfy the precondition of* `removeDiverging`*. If we apply the rule to model P with match* $\{$`sw1`, `s1`$\}$*, the* **`unknown`** *value of the* **`diverging`** *relation is refined to* **`false`***, resulting in model Q shown in Figure 4b.*



(a) Partial model P **(b)** Partial model Q

Figure 4 Application of refinement rule `removeDiverging`

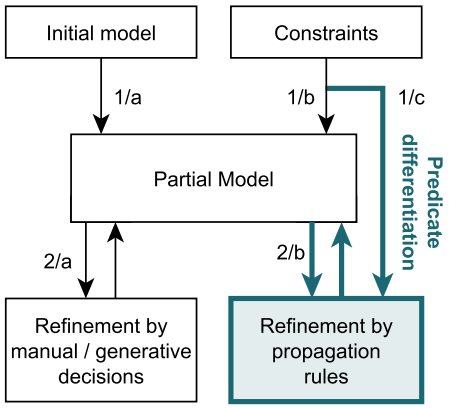


Figure 5 Overview of the Propagation

3. Overview

3.1. Functional overview

The functional overview of our approach is shown in Figure 5. A partial modeling environment is first initialized with a partial model that captures the available knowledge about the design (1/a). This initial model may originate from an existing solution or from an empty model in which most values are set to **`unknown`**. In addition, the domain specifies a set of well-formedness constraints, expressed as predicates, that must hold for all concrete models (1/b). Those predicates are continuously evaluated on the partial model to find and report violations.

Definition 3.1 (Valid model) *Let $\varphi_1, \ldots, \varphi_n$ denote a set of error predicates we use as well-formedness constraints. A partial (or concrete) model P is* invalid *if there is a predicate φ_i and variable binding Z where:*

$$\texttt{true} \succcurlyeq [\![\varphi_i]\!]_P^Z \textit{ (i.e., the value is either } \texttt{true} \textit{ or } \texttt{error}\textit{)}$$

A model is valid *if it is not invalid.*

In our approach, we introduce a technique called *predicate differentiation* that automatically derives groups of refinement rules from well-formedness constraints (1/c). These rules act as *propagation rules* that refine partial models while preserving all well-formed and valid design alternatives.

Definition 3.2 (Propagation rule) *Let $\varphi_1, \ldots, \varphi_n$ denote a set of error predicates. A propagation rule R is a* refinement rule *if for all partial model P and all application of R on P producing Q, and all well-formed concrete model M ($[\![\varphi_1]\!]^M =$ `false`$, \ldots, [\![\varphi_n]\!]^M =$ `false`)*

$$[M \textit{ is concretization of } P] \Leftrightarrow [M \textit{ is concretization of } Q]$$

During the development of partial models, designers progressively resolve uncertainty by adding information (2/a), thereby moving the model closer to a concrete design. Alternatively, generative processes can automatically explore design alternatives and synthesize candidate models.

The novel propagation rules proposed in the paper (2/b) supports this process by inferring whether a given relation must always be present or can never be present in the model. Rather than relying on repeated trial-and-error insertions, propagation rules apply logical inference to determine the required state of each connection in advance. If a rule deduces that a relation must hold, it is automatically inserted into the model; if it determines that a relation cannot hold without violating a constraint, it is systematically excluded from further consideration.

This proactive reasoning significantly reduces redundant generation attempts and improves both the efficiency and the consistency of the model construction process. In the following, we illustrate our approach with a case study.

3.2. Case study: propagation rules

Consider the **`straightDiverging`** error predicate defined in our case study:

```
error straightDiverging(switch, target) ⟷
  straight(switch,target),
  diverging(switch,target).
```

This error predicate contains a single clause with two literals, therefore we can derive two propagation rules from it. The `straightDivergingC1L1` rule performs propagation based on the first literal of the first clause, **`straight`**`(switch, target)`:

```
propagation rule
    straightDivergingC1L1(p1, p2) ⟷
  must diverging(switch, target),
  may straight(p1, p2),
  !must straight(p1, p2),
  p1 = switch,
  p2 = target
⟹
  !straight(p1, p2).
```

This rule specifies that whenever a **`diverging`**`(switch, target)` relationship exists between a switch and a target element, a **`straight`**`(p1, p2)`

relationship cannot also exist for the same pair of elements. The `p1 == switch` and `p2 == target` equivalence relations in the connection predicate ensure that the constraint is applied to the same elements.

The second propagation rule, `straightDivergingC1L2`, is the complement of the previous one propagated by the second literal of the first clause (C1L2):

```
propagation rule
    straightDivergingC1L2(p1, p2) ⟷
  must straight(switch, target),
  may diverging(p1, p2),
  !must diverging(p1, p2),
  p1 == switch,
  p2 == target
⟹
  !diverging(p1,p2).
```

This rule states that if a **straight**`(switch, target)` connection exists, then the corresponding **diverging**`(p1, p2)` connection for the same pair must not exist. Together, the two rules enforce mutual exclusivity between the straight and diverging relations of a switch, ensuring that the model remains structurally consistent and that no switch connects to the same target through both paths simultaneously.

Consider the following partial model, containing a **Switch** and three **Segment** elements, where the **straight** and **diverging** relations are unknown (Figure 6), they can be connected to either **Segment**. The **connectsTo** relations and some of the assertions of the model are excluded for simplicity.

```
Switch(sw1).
Segment(s1).
Segment(s2).
Segment(s3).
```

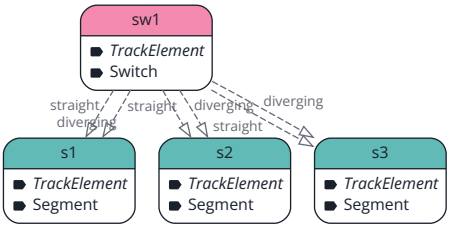
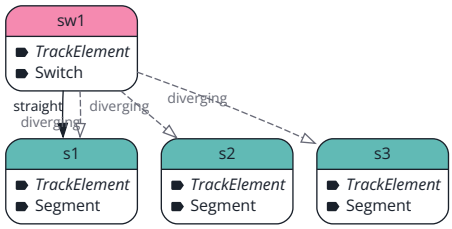


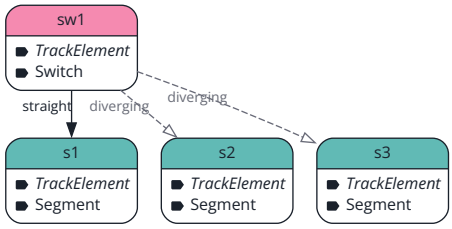
Figure 6 Initial partial model

During the model editing, for example in the generation process, we decide to refine the model by adding **straight**`(sw1, s1)`. The result can be seen in Figure 7a. The precondition of the `straightDivergingC1L2` propagation rule is true for this partial model, so in the next propagation step, this rule is applied. In the resulting model in Figure 7b, the diverging path of the `switch` cannot be `s1`.

In the next step, we decide that `s2` cannot be the **diverging** path of `sw1` and get the model in Figure 8a. Here, as we can see, the only option as the **diverging** path of `sw1` is `s3`. From the **noDiverging** **error**, we derive the following **propagation rule**:



(a) Partial model before propagation



(b) Partial model after propagation

Figure 7 Application of **propagation rule** `straightDivergingC1L2`

```
propagation rule
    noDivergingC1L2(p1, p2) ⟷
  must Switch(switch),
  count{may diverging(p1, _)} == 1,
  may diverging(p1, p2),
  !must diverging(p1, p2),
  p1 == switch
⟹
  diverging(p1, p2).
```

The precondition of this **propagation rule** is true for `sw1` and `s3`, so applying it will add the **diverging**`(sw1, s3)` relation, resulting in the model in Figure 8b, where there is no more uncertainty.

In the following, we present the proposed graph predicate differentiation algorithm.

4. Graph Predicate Differentiation

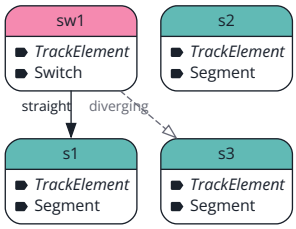
The goal of graph predicate differentiation is to automatically derive propagation rules from error predicates.

4.1. Overview of the differentiation

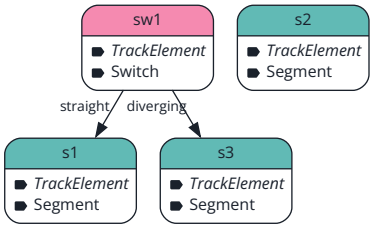
In general, predicates are defined in the form of $\varphi \Leftrightarrow \bigvee_c \bigwedge_l L_l^c$, where L_l^c denotes the l-th literal in the c-th clause. If the aforementioned predicate φ is an error predicate, it means that in a valid model cannot satisfy all the literals in any of the clauses; at least one literal must be evaluated to **false** for all objects in the model. The general strategy is to capture this property using implication normal form (like proposed in (Wittocx et al. 2013)):

all condition is **true** but L_l^c for some objects $\Rightarrow$ L_l^c is **false**

Therefore, propagation rules can update and refine the model and enforce the implications of the error predicates. Our rewriting strategy is based on the following steps:



(a) Model before propagation



(b) Model after propagation

Figure 8 Application of `propagation rule noDivergingC1L2`

1. All error predicates φ are collected.
2. For all error predicates φ a propagation rules are constructed for all literal L_l^c with predicate differentiation.
3. During partial modeling, all propagation rules are applied as long as possible.

In the following we describe the details of the rewriting of error predicates to refinement rules.

4.2. Predicate rewriting

Before rewriting a predicate, comparisons LESSEQ, GREATEREQ, EQ, and NOTEQ, are substituted according to the rules in Table 1, and the new predicates are represented in their disjunctive normal form.

The rewriting strategy of a predicate φ is separated into three steps:

- *Precondition* describes the condition when a change in the truth-value of a literal L_l^c could change the truth-value of φ.
- *Action* describes the rewriting that ensures the change in the truth-value in L_l^c.
- *Connection* describes the connection between the precondition and the postcondition and supports the execution of the implication as graph rewriting.

First, precondition is defined as follows.

Definition 4.1 (Propagation precondition) *For a predicate φ the* propagation precondition *on clause c and literal l is denoted as $pre_l^c[\varphi]$, and defined as:*

$$pre_l^c[\varphi] := \texttt{must}L_1^c, \ldots, \texttt{must}L_{l-1}^c, \texttt{must}L_{l+1}^c, \ldots, \texttt{must}L_m^c$$

where the c-th clause of φ is

$$C_c = L_1^c, \ldots, L_{l-1}^c, L_l^c, L_{l+1}^c, \ldots, L_m^c.$$

Then, the connection predicate for literal L_l^c depends on the type of the predicate.

Definition 4.2 (Connection predicate) *For a predicate φ, clause c, literal l (where the l-th literal of the c-th clause of φ is L_l^c), and parameters $\langle p_1, \ldots, p_a \rangle$, the connection predicate is denoted as $con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle]$ and defined as follows:*

Existentially quantified: *if $L_l^c = XS(v_1, \ldots, v_a)$ and $v_1, \ldots, v_a$ are existentially quantified in φ, then: $con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] := \texttt{may}S(p_1, \ldots, p_a), v_1{=}p_1, \ldots, v_a{=}p_a$.*

Universally quantified: *if $L_l^c = \texttt{!}S(p_1, \ldots, p_a)$ and at least one of the parameters $\langle p_1, \ldots, p_a \rangle$ is universally quantified, then: $con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] := \texttt{count}\{\texttt{may}S(u(p_1), \ldots, u(p_a))\} = 1, \texttt{may}S(p_1, \ldots, p_a), \; v_1{=}p_1, \ldots, v_a{=}p_a$, where $u(p) = p$ if p was existentially quantified, otherwise $u(p)$ is a new unused universally quantified variable.*

Positive transitive closure: *if $L_l^c = S{+}(v_i, v_j)$, then: $con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] := \texttt{must}S\texttt{*}(v_1, p_1), \texttt{may}S(p_1, p_2), \texttt{must}S\texttt{*}(p_2, v_2)$, where: $S\texttt{*}(\texttt{x},\texttt{y}) \longleftrightarrow S{+}(x, y)$; $\texttt{x==y}$.*

Negative transitive closure: *This rewriting rule is not implemented, as transitive closure with exclusion cannot be expressed in Refinery.*

Aggregation and comparison: *if $L_l^c = \text{COMP}(\texttt{count}\{S(v_1, \ldots, v_a)\}, c)$, then: $con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] := \texttt{count}\{MS(p_1, \ldots, p_a)\}{=}c, \texttt{may}S(p_1, \ldots, p_a), \; v_1{=}p_1, \ldots, v_a{=}p_a$, where*

$$M = \begin{cases} \texttt{may}, & \text{if COMP is LESS} \\ \texttt{must}, & \text{if COMP is GREATER} \end{cases}$$

If we want the propagation rule to block its own repeated application, then the connection predicate is extended with $\texttt{!}\texttt{must}S(p_1, \ldots, p_a)$.

Definition 4.3 (Propagation Action) *For a predicate φ clause c, literal l (where the l-th literal of the c-th clause of φ is L_l^c), and parameters $\langle p_1, \ldots, p_a \rangle$ the* propagation action *is denoted as $post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle]$, and defined as follows:*

- *if the polarity of L_l^c is positive or* COMP *is* LESS*, then $post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] := S(p_1, \ldots, p_a) : \texttt{false}$*
- *if the polarity of L_l^c is negative or* COMP *is* GREATER*, then $post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] := S(p_1, \ldots, p_a) : \texttt{true}$.*

Then the propagation rule with predicate derivation is constructed in the following form.

Definition 4.4 (Propagation rule) *The* propagation rule *of a predicate φ with clause c and literal l is defined as the following:*

$$\texttt{rule}\ R(p_1, \ldots, p_a) \longleftrightarrow$$

$$pre_l^c[\varphi], con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] \Longrightarrow post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle].$$

Where $pre_l^c[\varphi]$ is a propagation precondition, $con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle]$ is a connection predicate, and $post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle]$ is a propagation action.

4.3. Soundness and Completeness

The proposed predicate differentiation rules produce a group of partial rewriting rules. In the following we would like to review some important properties of the rules generated in this way.

Definition 4.5 (Soundness and completeness of refinement) *Let $\varphi_1, \ldots, \varphi_n$ denote error predicates, and R a refinement rule. R is* sound/complete*, if for all partial model P and all application of R on P producing Q, and all well-formed concrete model M ($[\![\varphi_1]\!]^C = \texttt{false}, \ldots, [\![\varphi_n]\!]^M = \texttt{false}$)*

$$[P \succcurlyeq M] \quad \Leftarrow \quad [Q \succcurlyeq M] \quad \textit{(soundness)}$$
$$[P \succcurlyeq M] \quad \Rightarrow \quad [Q \succcurlyeq M] \quad \textit{(completeness)}$$

Then, if a rewriting rule is sound, it will not lead to a design alternative that was not available from the original partial model P, and if a rewriting rule is complete, it will not exclude any valid potential design alternatives available from P. We review the two properties separately.

Soundness ($\Leftarrow$) We will prove that a propagation rule is a refinement relation. The application of a refinement rule R with parameter binding m on partial model P producing Q is defined with a relation $r : \mathcal{O}_P \to \mathcal{O}_Q$. Since $\mathcal{I}_Q(S)(r(m(p_1)), \ldots, r(m(p_a))) = \mathcal{I}_P(S)(m(p_1)), \ldots, m(p_a))) \curlywedge T$ and for all truth-value X it is true that $X \succcurlyeq (X \curlywedge T)$ for any T, then r provides a refinement relation as well. Since $P \succcurlyeq Q$ and $Q \succcurlyeq M$ then $P \succcurlyeq M$.

Completeness ($\Rightarrow$) We will prove that for each concrete model M, if $P \succcurlyeq M$ and not $Q \succcurlyeq M$ then model M violates at least one well-formedness constraint: the one used to derive the propagation rule. Therefore, no valid refinement is excluded.

Let L_l^c denote the l-th literal of the c-th clause of the error predicate φ from which the propagation rule is derived with refinement relation r. Since the rule is applied on P, we know that the precondition of the rule matches on P with a parameter binding Z. For each concrete model M where $P \succcurlyeq M$ and not $Q \succcurlyeq M$ we know that the action of the propagation rule ($post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle]$) is false.

We need to show that if the *precondition* and the *connection* is `true` (with Z), the propagation *action* is `false` (with $r \circ Z$), then it implies that the error predicate is `true`:

$$pre_l^c[\varphi], con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle], \, !post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] \Rightarrow \varphi.$$

As $C^c = (pre_l^c[\varphi], L_l^c)$ and $C^c \Rightarrow \varphi$, we can show the *connection* and the negated *action* implies L_l^c:

$$con_l^c[\varphi, \langle p_1, \ldots, p_a \rangle], \, !post_l^c[\varphi, \langle p_1, \ldots, p_a \rangle] \Rightarrow L_l^c$$

In the following we will show that for each type of literal L_l^c listed in Definition 4.2:

Existentially quantified: if $\langle p_1, \ldots, p_a \rangle$ is existentially quantified, then $\texttt{may}S(p_1, \ldots, p_a), v_1{=}p_1, \ldots, v_a{=}p_a$ and $S(p_1, \ldots, p_a) = \texttt{true}$ ($S(p_1, \ldots, p_a) = \texttt{false}$) entails that $S(v_1, \ldots, v_a)$ is `true` (`false`).

Universally quantified: $\langle p_1, \ldots, p_a \rangle$ is universally quantified $\texttt{count}\{\texttt{may}S(u(p_1), \ldots, u(p_a))\} = 1, \texttt{may}S(p_1, \ldots, p_a), v_1{=}p_1, \ldots, v_a{=}p_a$ and ensures, that there is a single potential match for $S(v_1, \ldots, v_a)$, so $S(p_1, \ldots, p_a)$ entails $!S(v_1, \ldots, v_a)$ is true.

Positive transitive closure: $\texttt{must}S_\star(v_1, p_1), \texttt{may}S(p_1, p_2), \texttt{must}S_\star(p_2, v_2)$ and $S(p_1, p_2) = \texttt{true}$ ensures $S_+(v_1, v_2)$ as (1) $S_\star(v_1, p_1), S(p_1, p_2) \Rightarrow S_+(v_1, p_2)$ and (2) $S_+(v_1, p_2), S(p_2, v_2) \Rightarrow S_+(v_1, v_2)$.

Aggregation and comparison, LESS **:** if $\texttt{count}\{\texttt{may}S(p_1, \ldots, p_a)\}{=}c, \texttt{may}S(p_1, \ldots, p_a), v_1{=}p_1, \ldots, v_a{=}p_a$ it means that the potential number of matches of S is c, and those potential matches include $\langle p_1, \ldots, p_a \rangle$. Therefore, if $S(p_1, \ldots, p_a) = \texttt{false}$, then the number will be LESS than c.

Aggregation and comparison, GREATER **:** if $\texttt{count}\{\texttt{must}S(p_1, \ldots, p_a)\}{=}c, \texttt{may}S(p_1, \ldots, p_a), v_1{=}p_1, \ldots, v_a{=}p_a$ it means that the certain number of matches of S is c, excluding a potential match $\langle p_1, \ldots, p_a \rangle$. Therefore, if $S(p_1, \ldots, p_a) = \texttt{true}$, then the number will be GREATER than c.

5. Evaluation

We carried out an experimental evaluation of our approach to address the following research questions:

- RQ1 What is the impact of the proposed unit propagation technique on parsing partial models?
  - RQ1.1 How much does the proposed unit propagation technique reduce the uncertainty in models?
  - RQ1.2 What is the runtime impact of parsing and propagation in partial models?
- RQ2 How does the proposed unit propagation technique affect the performance of graph generation?
  - RQ2.1 What is the runtime impact of the automated propagation on graph generation?
  - RQ2.2 How is the change in runtime distributed between different steps of graph generation?

5.1. Domains

Since there are no well-established performance benchmarks for model generation and partial modeling, we selected three domains from different application areas, each with different complexity levels. Additionally, we created two simplified versions of the most complex domain to analyze the effect of complexity on the results. The selected domains are summarized below. **Railway network (TB1):** This is a simple railway network with segments and switches. This domain is the physical architecture of the railways at the most fundamental level. This domain includes three classes, one of which is abstract, and six error predicates. **Simplified Trainbenchmark (TB2):** This middle-sized railway is an extension of TB1, introducing two new classes, regions and sensors. There are also six new error predicates in addition to the six defined in TB1. **Trainbenchmark (TB3):** The Trainbenchmark domain (Szárnyas et al. 2018) can be used to model routes on railway networks,

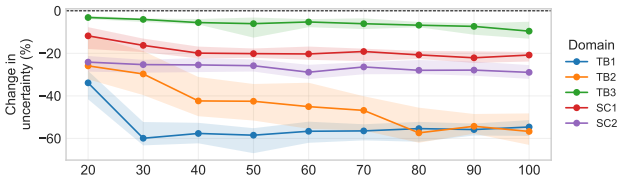


Figure 9 Uncertainty change with propagation rules

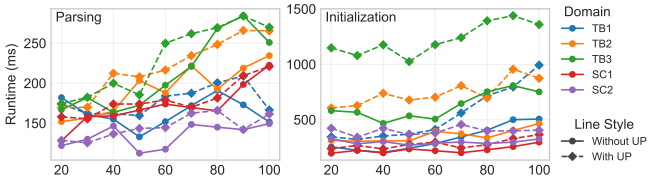


Figure 10 Loading and parsing time of partial models

including safety aspects. It contains a total of 9 classes and 16 error predicates. This domain is the most complex among the selected ones. **Simple statechart (SC1):** This simple statechart domain-specific language (Famelis et al. 2012) can be used to design simple statecharts for peer-to-peer file-sharing protocols. Its metamodel consists of two classes and there are four error predicates defined. Some states and all the allowed transition labels are predefined. **Yakindu (SC2):** Itemis CREATE (itemis 2025) (formerly Yakindu) is an industrial modeling environment for statecharts. This example represents the generation of test cases for the Yakindu Statecharts modeling tool. The metamodel includes 12 classes, and the generation is constrained by 10 error predicates.

For each domain, a model of size n contains at least n nodes, with a margin of 20% for its maximal size. E.g. when generating a model with size 30, it will have at least 30 and at most 36 nodes. Some of the domains have additional scope constraints on the number of certain types of nodes. The input files and raw measurement data supporting the results of this study are available in our GitHub repository at https://github.com/ftsrg/paper-ecmfa26-unitprop-materials (Ficsor et al. 2026).

All measurements for were executed on a virtual machine on an Intel Xeon (Skylake) processor, using 2 cores at 2000 MHz, and 16 GiB RAM.

5.2. Measurement setup for RQ1

To evaluate the effect of propagation rules on the parsing of partial models, we generated 30 models per domain and size, from 20 to 100 nodes with increments of 10. Partial models are created by enabling the extension of existing valid partial models with further elements (i.e., adding new edges and nodes to existing graphs). For TB2 and TB3, we disable the further editing of the existing tracks, so no existing segments can be directly joined together, but new segments can be added to connect them. Finally, we randomly remove an additional 10% of the remaining facts, introducing further uncertainty. This simulates the development of models by engineers, who may want to add further details to their designs. In order to quantify the uncertainty in the models, we count the unknown values in the interpretations of the relations defined in the metamodels. We also measure the time it takes to parse the models and calculate the interpretations.

5.3. Measurement results for RQ1

The results for RQ1.1 are shown in Figure 9. It visualizes the median change in the number of unknown values in the model after the unit propagation (UP), as a ratio of all unknown values before the propagation.

For RQ1.2, the measure results are shown in Figure 10. The chart shows the median parsing and model initialization times of models with and without the use of propagation rules. The data is plotted on a logarithmic scale. The total runtime of parsing and initialization is written above the data.

5.4. Research answer for RQ1

Figure 9 shows that the application of propagation rules reduced the uncertainty in valid models in all domains and sizes. For TB3 the reduction in uncertainty was 3–10%, for SC1 and SC2 12–29% and for TB1 and TB2 26–57%. The reduction is consistent across model sizes within a domain, with a small increase as the model grows. This means, that in some model editing scenarios an engineer would only need to consider less than half of the possibilities, when using the propagation rules.

As we can see in Figure 10, the parsing of the files is always less than 300 milliseconds, with a maximum of 56 ms increase when using propagation rules. The model initialization on the other hand takes 18–152% more time when using propagation rules. In the cases of the SC1 and SC2 the difference in initialization time was only 35–158 milliseconds, while for TB, TB2, and TB3 the difference was 90–710 ms. This shows a greater tradeoff in the runtime for reducing uncertainty with the application of the propagation rules.

RA1.1: The use of propagation rules can decrease the uncertainty in partial models by up to 57%, but for some problems it may have a smaller effect.

RA1.2: The runtime of parsing partial models is minimally increased by the propagation rules. The model initialization step slows down due to the introduction of propagation rules, in some cases more than doubling the runtime.

5.5. Measurement setup for RQ2

This measurement aims at determining the effect of propagation rules on the runtime of model generation. We measured the generation time of a single model of a given size with and without the generation and application of propagation rules.

We measured the model generation time in the five domains with five model sizes. The generation for SC2 was faster than the other domains, so in this case, we used a size increment of 20, while in the other domains we used an increment of 10 between the model sizes. Model generation for size 10 in TB3 is an unsatisfiable problem, therefore we measured the time it took to determine the unsatisfiability.

Each test case was executed 30 times to account for variability in execution time. A timeout of 120 seconds was set for each

	Parse (ms)		Init (ms)		Generate (ms)	
Use UP	False	True	False	True	False	True
SC1	100	94	154	192	1333	874
SC2	64	60	206	216	3069	1532
TB1	86	82	114	130	35726	14886
TB2	95	88	188	385	14673	34489
TB3	67	91	327	755	39070	6076

Table 2 Median execution times for 30 objects

test case to prevent excessively long runs. We also accounted for warm-up effects and memory handling of the Java virtual machine (JVM) with 5 to 20 warm-up runs and 200 ms of wait between runs to allow the garbage collection to complete.

5.6. Measurement results for RQ2

Figure 11 shows the execution times of the model generations for RQ2.1. For each domain, two diagrams show the generation times of the successful generations, with and without the use of unit propagation (UP) rules. Each plot displays the results for each of the five model sizes with a box plot. The ratio of runs that timed out is shown in red.

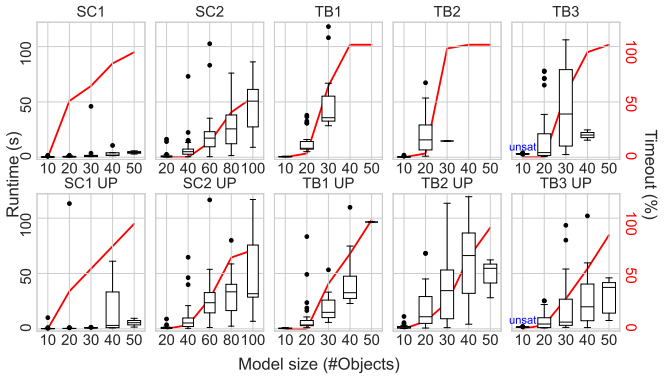


Figure 11 Model generation with and without propagation

For RQ2.2, Table 2 shows the median execution time of each step for the successful model generation runs with size 30.

5.7. Research answer for RQ2

We can read from Figure 11, that with the exception of SC2, for all domains and model sizes, the ratio of timed out runs decreased with the use of propagation rules. Although the runtime for the successful generations may stay the same or even increase in some cases, accounting for the increased success rate indicates an improvement in all measurements.

As we can see in Table 2, the generation step dominates the total execution time, with the parsing of the input file being consistently low regardless of the domain or the use of propagation rules. The introduction of propagation rules increases the model initialization time by up to 130% in the TB3 experiment, but in most cases this effect is negated by the decrease in the generation time. While in the case of TB2 the median generation time of successful generation also increased, the lower value without the propagation rules is based on a single successful generation. As a result of applying the propagation rules the number of timeouts decreased from 29 to 7 out of 30 according to Figure 11, which results in an overall better performance if our goal is to generate many models in a limited time.

RA2.1: The introduction of propagation rules either decreases the execution time and the number of timeouts in model generation or does not have an effect on them.

RA2.2: While the runtime of model initialization is increased by the propagation rules, the reduction of the generation time and ratio of timeouts compensates for it. The parsing of the input files is independent of the propagation rules, therefore its runtime is not affected.

5.8. Threats to validity

Internal validity Our experiments pertaining to runtime measurements incorporated a warmup phase prior to actual measurements to reduce variance in execution times due to the JVM. Between measurements, we left time for the garbage collection to complete. However, other factors such as background processes on the host machine or JVM optimizations could still have influenced the results. To mitigate these effects, we conducted multiple runs for each test case and reported median values. Nonetheless, some variability may still exist in the measurements.

External validity A potential threat to the external validity of our results is the absence of a head-to-head comparison with alternative graph solvers. To the best of our knowledge, there are currently no existing graph-solvers that implement a comparable functionality. To mitigate this, we conducted an ablation study to isolate the performance gains of our approach. Our experiments were conducted on three external case studies of various complexity, and two simplified variations of the most complex one, further increasing the variability of the domains. Each domain was tested with multiple model sizes to account for different levels of complexity. However, the results may not generalize to all possible domains or configurations. Further studies on additional domains would help to strengthen the external validity of our findings.

Construct validity We measured the execution time of model generation and the reduction of uncertainty in partial models, which are relevant metrics for evaluating the effectiveness of our approach, as it could be used in modeling tools, where performance is important. However, other factors such as memory usage or scalability to larger models were not considered in this study. Future work could explore these aspects to provide a more comprehensive evaluation.

6. Related Work

There are several techniques and tools utilizing logic reasoning and propagation rules. In the following, we are summarizing those techniques and their relation with our graph predicate differentiation approach.

Uncertainty modeling Partial modeling techniques provide expressive languages for representing uncertainty in models (Troya et al. 2021). Techniques such as MAVO (Famelis et al. 2012) offer a domain-independent specification language for annotating existing models with uncertainty (e.g., whether a particular object may or may not exist), and their semantics are amenable to analysis (Saadatpanah et al. 2012). These approaches are particularly useful when analysis begins from an existing, fully specified design. However, such techniques do not inherently address the handling of well-formedness constraints.

Potential concrete models compliant with an uncertain specification can be synthesized using the Alloy Analyzer (D. Jackson 2002) or refined through graph transformation rules (Salay et al. 2015). While scalability remains a challenge in partial modeling, refinement of uncertain models is always decidable.

In contrast, our approach relies on many-valued logic (Belnap 1977; Kamide & Omori 2017) to explicitly represent uncertain and inconsistent parts of a model. Similar notations appear in shape-analysis techniques based on 3-valued logic (Reps et al. 2004; Ferrara et al. 2012) and in modal graph abstraction techniques (Rensink 2004; Rensink & Distefano 2006), where they serve as automated abstraction mechanisms. Related frameworks with standalone specification languages include Formula (E. K. Jackson et al. 2011, 2013), which uses the Z3 SMT solver (De Moura & Bjørner 2008); Alloy, which relies on relational logic (Torlak & Jackson 2007) and SAT solvers such as Sat4j (Berre & Parrain 2011); and Clafer (Bąk et al. 2016), which also employs Alloy-based reasoning.

Comparable reasoning tools exist for feature models (Liang et al. 2015). However, feature modeling focuses on efficiently exploring a fixed set of predefined configuration options. In contrast, partial models operate over graph structures with an unknown number of nodes and edges, where the available design options are not predetermined.

Logic solvers Logic solver-based approaches translate graphs and well-formedness (WF) constraints into logical formulas and use a solver to generate graphs that satisfy these formulas. EMF2CSP/UML2CSP (González Pérez et al. 2012; Cabot et al. 2014) transforms model generation into a constraint-programming problem and relies on an underlying CSP solver to compute solutions. (Hao 2013) reports class-diagram generation with EMF2CSP, where the search is guided by class-diagram–specific metrics such as LCOM and Class Cohesion (Chidamber & Kemerer 1994). ASMIG (Wu et al. 2013) employs the Z3 SMT solver to generate typed and attributed graphs with inheritance. (Cunha et al. 2014) extends the Kodkod relational model finder (Torlak & Jackson 2007) to generate models that resemble a given target model.

In contrast, rather than translating graph-generation problems into SAT/SMT formulations, the logic-solver algorithm in Refinery (Semeráth et al. 2018) adapts core logic-solving techniques directly to graph structures. The closest related work of this paper might be (Wittocx et al. 2013), where transformation rules are used for SAT solving. In this paper, we extended the results from propositional logic to 4-valued first-order graph predicates with quantifiers ($\forall$, $\exists$), transitive closure (R^+) in relevant cases, aggregations (#, counting the size of sets), and uninterpreted node existence and equivalences (Semeráth et al. 2018; Marussy et al. 2024). Uninterpreted `equals` and `exists` relations are especially relevant in partial modeling, as they enable the representation of graph problems in which the size of the model can grow dynamically. Finally, our approach translates deduction rules into graph-transformation rules, enabling efficient execution during model exploration.

Search-based generators. SDG (Soltana et al. 2017) introduces an approach that employs a search-based custom OCL solver to generate synthetic data for statistical testing. The resulting models are multidimensional, locally consistent, and realistic. The study demonstrates scalability by producing a large number of small, unconnected models. However, because the search-based method treats well-formedness (WF) constraints as soft constraints, the likelihood of generating fully consistent models decreases as model size grows.

The work in (Soltana et al. 2019) proposes a hybrid technique that combines a meta-heuristic, search-based OCL solver (Ali et al. 2016) for structural constraints with an SMT solver for attribute constraints, building on the snapshot generator of the USE framework (Gogolla et al. 2007). The generated models are multidimensional, locally consistent, and large, although their realism is limited to type and attribute distributions.

In contrast, search-based approaches modify graphs directly, whereas our approach performs refinement with respect to information ordering (Semeráth & Varró 2017).

Inference in databases Modern graph and knowledge-graph datastores provide useful context for our approach. TypeDB uses a typed hypergraph schema and a type-theoretic query language (TypeQL) (Dorn & Pribadi 2023); earlier versions (2.x) supported rule-based inference, while the current 3.x architecture replaces rules with typed, composable functions acting as views over graph patterns. These mechanisms modularize graph-pattern logic but remain manually authored and two-valued. Recent work has addressed graph consistency, constraint enforcement, and repair, including normalization of property graphs (Skavantzos & Link 2023, 2025), formal frameworks for repairing inconsistent databases (Abriola et al. 2023; Fröhlich et al. 2025), and incremental logic-based graph repairs (Schneider et al. 2019). In contrast, our approach automatically derives propagation commands from error predicates in a 4-valued first-order logic, bridging declarative constraints and operational enforcement.

7. Conclusions

This paper demonstrated that incorporating well-formedness constraints into uncertain modeling can provide significant help in distinguishing feasible design alternatives from those that inevitably lead to invalid solutions. Based on (Wittocx et al. 2013) we introduced a technique that translates graph-based well-formedness predicates into refinement rules capable of automatically improving the precision of partial models. Our implementation within the Refinery framework (Semeráth et al.

2018; Marussy et al. 2024) shows that these rules effectively reduce uncertainty while preserving all valid design options. Empirical evaluations further confirm that the approach significantly enhances the efficiency, reducing the execution time of model generation and design-space exploration.

Automatically calculating propagation rules can be a key feature at supporting a wider range of problems with the Refinery framework, including e.g., Shapes Constraint Language (SHACL) (K Soman 2019) for JSON generation.

Acknowledgments

The project was partially supported by the Doctoral Excellence Fellowship Programme (DCEP), funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation and the Budapest University of Technology and Economics, and the Department of Navy award (N629092412063) issued by the Office of Naval Research. The computational resources were provided by the BME VIK CIRCLE cloud infrastructure, which is developed and operated by BME IK and BME IIT, partially funded by faculty resources.

References

Abriola, S., Cifuentes, S., Pardal, N., & Pin, E. (2023). Data-graph repairs: the preferred approach. In *Proceedings of the 15th alberto mendelzon international workshop on foundations of data management (amw).*

Ali, S., Iqbal, M. Z., Khalid, M., & Arcuri, A. (2016, Dec 01). Improving the performance of OCL constraint solving with novel heuristics for logical operations: a search-based approach. *Empirical Software Engineering*, *21*(6), 2459–2502. doi: 10.1007/s10664-015-9392-6

Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., & Wąsowski, A. (2016). Clafer: unifying class and feature modeling. *Software & Systems Modeling*, *15*(3), 811–845.

Belnap, N. D., Jr. (1977). A useful four-valued logic. In *Modern uses of multiple-valued logic* (Vol. 2, p. 5-37). Springer. doi: 10.1007/978-94-010-1161-7_2

Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., & Varró, D. (2015). Viatra 3: A reactive model transformation platform. In *international conference on theory and practice of model transformations* (pp. 101–110).

Berre, D. L., & Parrain, A. (2011). The sat4j library, release 2.2: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, *7*(2-3), 59-64. doi: 10.3233/SAT190075

Cabot, J., Clarisó, R., & Riera, D. (2014, March). On the Verification of UML/OCL Class Diagrams using Constraint Programming. *Journal of Systems and Software*. doi: 10.1016/j.jss.2014.03.023

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476–493.

Cunha, A., Macedo, N., & Guimarães, T. (2014). Target oriented relational model finding. In *Proceedings of the 17th international conference on fundamental approaches to software engineering - volume 8411* (p. 17–31). Berlin, Heidelberg: Springer-Verlag. doi: 10.1007/978-3-642-54804-8_2

De Moura, L., & Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on tools and algorithms for the construction and analysis of systems* (pp. 337–340).

Dorn, C., & Pribadi, H. (2023). Type theory as a unifying paradigm for modern databases. In *Proceedings of the 32nd acm international conference on information and knowledge management* (p. 5238–5239). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3583780.3615999

Eclipse Modeling Framework [Computer software manual]. (2025). https://eclipse.dev/emf/.

Famelis, M., Salay, R., & Chechik, M. (2012). Partial models: Towards modeling and reasoning with uncertainty. In *2012 34th international conference on software engineering (icse)* (pp. 573–583).

Ferrara, P., Fuchs, R., & Juhasz, U. (2012). Tval+: Tvla and value analyses together. In *International conference on software engineering and formal methods* (pp. 63–77).

Ficsor, A., Papp, I. A., Marussy, K., & Semeráth, O. (2026). *Supplementary Material for "From Constraints to Commands: Graph Predicate Differentiation in 4-Valued First-Order Logic"*. GitHub repository, https://github.com/ftsrg/paper-ecmfa26-unitprop-materials. GitHub. (Version 1.0.0)

Fröhlich, N., Meier, A., Pardal, N., & Virtema, J. (2025, 10). A Logic-Based Framework for Database Repairs. In *Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning* (pp. 326–335). doi: 10.24963/kr.2025/32

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, *69*(1), 27 - 34. (Special issue on Experimental Software and Toolkits) doi: 10.1016/j.scico.2007.01.013

González Pérez, C. A., Buettner, F., Clarisó, R., & Cabot, J. (2012, June). EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA).* Zurich, Switzerland.

Hao, W. (2013). *Automated metamodel instance generation satisfying quantitative constraints* (Unpublished doctoral dissertation). National University of Ireland Maynooth.

itemis. (2025). *Create.* https://www.itemis.com/en/products/itemis-create/.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on software engineering and methodology (TOSEM)*, *11*(2), 256–290.

Jackson, E. K., Levendovszky, T., & Balasubramanian, D. (2011). Reasoning about metamodeling with formal specifications and automatic proofs. In *International conference on model driven engineering languages and systems* (pp. 653–667).

Jackson, E. K., Simko, G., & Sztipanovits, J. (2013). Diversely enumerating system-level architectures. In *Proceedings of the 11th acm int. conf. on embedded software* (p. 11).

Kamide, N., & Omori, H. (2017). An extended first-order Belnap-Dunn logic with classical negation. In *Lori 2017* (Vol. 10455, p. 79-93). Springer. doi: 10.1007/978-3-662-55665-8

_6
K Soman, R. (2019). Modelling construction scheduling constraints using shapes constraint language (shacl). In *Ec3 conference 2019* (Vol. 1, pp. 351–358).
Liang, J. H., Ganesh, V., Czarnecki, K., & Raman, V. (2015). Sat-based analysis of large real-world feature models is easy. In *Proceedings of the 19th international conference on software product line* (pp. 91–100).
Marussy, K., Ficsor, A., Semeráth, O., & Varró, D. (2024). Refinery: Graph solver as a service: Refinement-based generation and analysis of consistent models. In *Proceedings of the 2024 ieee/acm 46th international conference on software engineering: Companion proceedings* (p. 64-68). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3639478.3640045
Marussy, K., Semeráth, O., Babikian, A. A., & Varró, D. (2020, October). A specification language for consistent model generation based on partial models. *Journal of Object Technology*, *19*(3), 3:1-22. doi: 10.5381/jot.2020.19.3.a12
Rensink, A. (2004). Canonical graph shapes. In *ESOP* (pp. 401–415). Springer. doi: 10.1007/978-3-540-24725-8_28
Rensink, A., & Distefano, D. (2006). Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, *157*(1), 39 - 59. (Proceedings of the Third International Workshop on Software Verification and Validation (SVV 2005)) doi: 10.1016/j.entcs.2006.01.022
Reps, T. W., Sagiv, M., & Wilhelm, R. (2004). Static program analysis via 3-valued logic. In *International conference on computer aided verification* (pp. 15–30).
Saadatpanah, P., Famelis, M., Gorzny, J., Robinson, N., Chechik, M., & Salay, R. (2012). Comparing the effectiveness of reasoning formalisms for partial models. In *Proceedings of the workshop on model-driven engineering, verification and validation* (pp. 41–46).
Salay, R., Chechik, M., Famelis, M., & Gorzny, J. (2015). A methodology for verifying refinements of partial models. *J. Object Technol.*, *14*(3), 3–1.
Schneider, S., Lambers, L., & Orejas, F. (2019). A logic-based incremental approach to graph repair. In *International conference on fundamental approaches to software engineering* (pp. 151–167).
Semeráth, O., Nagy, A. S., & Varró, D. (2018). A graph solver for the automated generation of consistent domain-specific models. In *Proceedings of the 40th international conference on software engineering* (pp. 969–980).
Semeráth, O., & Varró, D. (2017). Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In *ICMT* (pp. 138–154). doi: 10.1007/978-3-319-61473-1_10
Skavantzos, P., & Link, S. (2023, July). Normalizing property graphs. *Proc. VLDB Endow.*, *16*(11), 3031–3043. doi: 10.14778/3611479.3611506
Skavantzos, P., & Link, S. (2025, February). Third and boyce–codd normal form for property graphs: Foundations, achievements, and normalization. *The VLDB Journal*, *34*(2). doi: 10.1007/s00778-025-00902-2
Soltana, G., Sabetzadeh, M., & Briand, L. C. (2017). Synthetic data generation for statistical testing. In *ASE* (pp. 872–882). doi: 10.1109/ASE.2017.8115698
Soltana, G., Sabetzadeh, M., & Briand, L. C. (2019). Practical model-driven data generation for system testing. *CoRR*, *abs/1902.00397*.
Szárnyas, G., Izsó, B., Ráth, I., & Varró, D. (2018, Oct 01). The train benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, *17*(4), 1365-1393. doi: 10.1007/s10270-016-0571-8
Torlak, E., & Jackson, D. (2007). Kodkod: A relational model finder. In *International conference on tools and algorithms for the construction and analysis of systems* (pp. 632–647).
Troya, J., Moreno, N., Bertoa, M. F., & Vallecillo, A. (2021). Uncertainty representation in software models: a survey. *Software and Systems Modeling*, *20*(4), 1183–1213.
Wittocx, J., Denecker, M., & Bruynooghe, M. (2013, August). Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Logic*, *14*(3). doi: 10.1145/2499937.2499938
Wu, H., Monahan, R., & Power, J. F. (2013, July). Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In *Tase* (p. 175-182). doi: 10.1109/TASE.2013.31
Xcore [Computer software manual]. (2025). Retrieved from https://wiki.eclipse.org/Xcore

About the authors

Attila Ficsor received the Computer Science Engineer degree from the Budapest University of Technology and Economics in 2022. He is currently pursuing a PhD degree with the Critical Systems Research Group. His research interests include model-driven engineering, graph transformation, and autonomous vehicle testing. You can contact the author at ficsor@mit.bme.hu.

Inez Anna Papp is an MSc student at the Budapest University of Technology and Economics. Her academic work focuses on logic propagation in graph-based models. She has presented her research results on this topic at the Student Scientific Conference on two occasions.

Kristóf Marussy is an assistant professor in the Department of Measurement and Information Systems, Budapest University of Technology and Economics, Hungary. His research interests include graph generation, logic solvers, formal verification, and applying these techniques for ensuring the safety and correctness of critical systems. You can contact the author at marussy@mit.bme.hu.

Oszkár Semeráth is an associate professor at the Budapest University of Technology and Economics. His research focuses on modeling technologies, and the application and development of specialized logic solvers for graph generation. He won IEEE/ACM best paper award at the MODELS 2013 conference, Young Researcher Award from the Hungarian Academy of Science, John George Kemeny Award from John von Neumann Computer Society, Amazon Research Award, and the scholarship of the New National Excellence Program three times. You can contact the author at semerath@mit.bme.hu.