JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

A specification language for consistent model generation based on partial models

Kristóf Marussy*†, Oszkár Semeráth*†, Aren A. Babikian‡, and Dániel Varró*†‡
*MTA-BME Lendület Cyber-Physical Systems Research Group, Hungary
†Budapest University of Technolgy and Economics, Dept. of Measurement and Information Systems, Hungary
‡McGill University, Dept. of Electrical and Computer Engineering, Canada

ABSTRACT Automated graph generation has become a key component in many testing and benchmarking scenarios. For example, modeling tool qualification can be effectively supported by the direct synthesis of well-formed graph models as test inputs, systematic testing of cyber-physical systems requires different test environment models, and different optimization and design-space exploration approaches require the best models with respect to an objective function.

In this paper, we propose a novel specification language for partial models used in consistent graph model generation. The language includes constructs to uniformly capture initial, intermediate and final results of the generation by combining partial models, graph predicates and model metrics with mutual dependencies between them. The formal semantics of the language is defined by using 4-valued Belnap-Dunn logic that explicitly marks inconsistent model elements as part of the partial model. The use of our language is illustrated in the context of a complex case study defined by NASA researchers.

KEYWORDS Partial models, Model generation, 4-valued logic.

1. Introduction

Quality assurance of critical software-intensive systems frequently relies on the automated synthesis of test data to reduce conceptual gaps in the test cases. When testing domain-specific modeling tools, or autonomous cyber-physical systems in model-based systems and software engineering scenarios, test data takes the form of typed and attributed graph models. Automated model generators are key technologies to address the needs of such testing scenarios.

A model generator needs to derive consistent models where each model needs to satisfy (or deliberately violate) a set of constraints captured in the form of OCL constraints or graph predicates. Logic solvers (like SMT-solvers, SAT-solvers, CSP-solvers) have been frequently providing precise foundations for such model generators in tools like Alloy, USE, UMLtoCSP, Formula, etc. However, a recently emerging family of model generators, like PLEDGE (Soltana et al. 2020) or the Viatra Solver (Semeráth et al. 2018), addresses the consistent model generation challenge directly on the level of graph models by sophisticated search strategies (like multi-objective optimization or SAT-solving algorithms) and powerful abstractions provided by 3-valued partial models and partial model refinement (Varró et al. 2018). To fine-tune the model generation process, iterative and incremental approaches (Semeráth et al. 2016) are proposed where models obtained as output in a previous run can be used as inputs (e.g. required or forbidden model fragments) in subsequent runs.

The vast majority of existing model generators adapted popular industrial languages and technologies (like EMF, OCL) as their input specification. On the other hand, custom domain-specific specification languages like Alloy have also become popular due to its precise semantics. However, when generat-

JOT reference format:
Kristóf Marussy, Oszkár Semeráth, Aren A. Babikian, and Dániel Varró. *A specification language for consistent model generation based on partial models*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution - No Derivatives 4.0 International (CC BY-ND 4.0)
http://dx.doi.org/10.5381/jot.2020.19.3.a12
JOT reference format:
Kristóf Marussy, Oszkár Semeráth, Aren A. Babikian, and Dániel Varró. *A specification language for consistent model generation based on partial models*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution - No Derivatives 4.0 International (CC BY-ND 4.0)
http://dx.doi.org/10.5381/jot.2020.19.3.a12

ing models in an iterative and incremental way along partial model refinement (Varró et al. 2018; Semeráth et al. 2018), this *enforces a black-box view on model generation* where the input and output models are concrete instance models, while all intermediate steps of the generation operate on abstract partial models as candidate solutions (some of which cannot be represented as regular instance models in EMF). Similarly, complex rewriting techniques are necessitated to approximately evaluate constraints over partial models (Semeráth & Varró 2017). Furthermore, the development of novel algorithms became very complicated due to the conceptual mismatch between concrete models and partial models.

The main objective of the current paper is to provide foundations for a more *grey-box view of model generation by providing a high-level language for specifying model generation problems*. The main purpose of this language is three-fold.

1. On the one hand, it should be *modeling technology independent* to capture graph-based model generation problems that arise for non-EMF models (e.g. OWL ontologies, graph databases, GraphQL and JSON data).

2. On the conceptual level, it should enable to *capture partial models as first-class citizens* as inputs, outputs, and the intermediate state of model generation. As such, any intermediate result of any model generation step can be used as-is for a subsequent model generation run, thus the internal states of model generation can become transparent by serializing the partial models in that format.

3. Finally, we wish to provide a mathematically precise intermediate language for describing well-formedness constraints and model metrics, again in a technology-independent way. For that purpose, we propose a combined notation influenced by logic programming and functional programming.

The main contribution of the paper is to propose a specification language for model generation problems that uses partial models, graph predicates and metrics as core language elements. A precise semantics of this core language is defined by a combination of 4-valued logic (for structural domain elements) and abstract interpretation over intervals (for attributes). Moreover, we also define some higher-level language elements as syntactic sugars which are semantically mapped back to core elements. We introduce this language on a sequence of examples for a model generation problem introduced by researchers from the NASA JPL in Herzig et al. (2017).

2. Case study

The design synthesis of *interferometry mission (IM) architectures* has been introduced in (Herzig et al. 2017) as a complex challenge for early mission planning for space missions of NASA where a target architecture consists of collaborating satellites (of different size and capabilities) and radio communication links between them. Each mission architecture contains multiple spacecrafts that impose a challenging design task. Constraints and mission objectives used in the current paper were defined by Herzig et al. (2017).

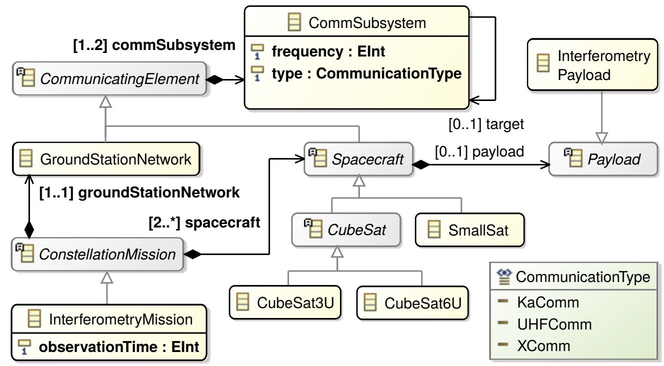


Figure 1 Metamodel of interferometry constellation missions.

A metamodel for interferometry constellation missions is shown in Figure 1 in an EMF notation. An `InterferometryMission` consists of communicating `CommElements`, which are equipped with `CommSubsystem` subsystems (i.e. antennas with different communication frequencies) via their `commSubsystem` references for `KaComm`, `UHFComm` and `XComm` bands. `Spacecraft` of different sizes, including cube satellites `CubeSat3U` and `CubeSat6U`, as well as small satellites `SmallSat`, may carry interferometry `Payloads` (photo sensors), and must be able to reach the `GroundStationNetwork` via radio links (to send interferometry sensor data) denoted by the `target` references.

The constraints of a consistent interferometry mission architecture capture that

- each `Spacecraft` may only have a single transmitting `commSubsystem` (the other `commSubsystem`, if present, may only receive);
- all `Spacecraft` must have a communication path to the `GroundStationNetwork`;
- there must not be any communications loops (communication paths from a `CommunicatingElement` to itself);
- `CommunicatingElement` instances may only communicate if they share the same radio band (`KaComm`, `XComm` or `UHFComm`) and frequency; and
- `CommSubsystem` instances of certain radio bands require specific `Satellite` types (e.g. only `SmallSat` instances can use the `KaComm` band).

A set of objective functions is defined in order to quantitatively assess each IM architecture. Such objectives include

- the cost of equipment in the constellation, including `Spacecraft`, `CommSubsystems` and `Payloads`, which is subject to economies of scale regarding multiple pieces of equipment of the same type;
- the duration of the mission, which is characterized by the observation time and the time required to downlink the gathered images to the `GroundStationNetwork`;
- the coverage of the observation area achieved by the interferometry `Payloads` during the observation time.

3. Syntax of partial models

The goal of partial models is to explicitly represent uncertainty in models by introducing a *4-valued logic representation* where a single partial model covers a set of concrete instance models. Informally, a specific *structural model element* (i.e. a domain object or link) can surely exist (a truth value of `true`), cannot exist (a truth value of `false`), or it may be uncertain if the model element exists (unspecified or unknown). The latter is signified by the `unknown` truth value (Kleene et al. 1952) in accordance with Reps et al. (2004); Varró et al. (2018); Semeráth & Varró (2017). If a model element is inconsistent, then a special `error` value is assigned to it (Marussy et al. 2018). Moreover, *data objects* can be used as attribute values to represent quantitative information. The core constructs of our specification language enable capturing such partial models.

Informally, our specification language allows to define graph-based partial models by using assertions where an assertion may state what is true, what is false or what is unknown related to the model. Well-formedness constraints of a domain can be defined by predicates in a Prolog-style language. Furthermore, graph metrics (e.g. objective functions, model metrics) can also be defined in combination with predicates in the style of functional programming languages (e.g. Erlang or ML).

3.1. Core syntax for partial models

3.1.1. Language literals The partial modeling language includes the following literals: $\langle id \rangle$ defines identifiers in the model, $\langle integer \rangle$ and $\langle real \rangle$ defines numbers, and $\langle string \rangle$ defines strings of characters with escape symbol. In general, this follows a notation similar to programming languages like Java.

$$
\begin{aligned}
\langle \textit{id-fragment} \rangle &::= (\texttt{a} \cdots \texttt{z} \mid \texttt{A} \cdots \texttt{Z} \mid \texttt{\_}) \\
&\quad\; (\texttt{a} \cdots \texttt{z} \mid \texttt{A} \cdots \texttt{Z} \mid \texttt{\_} \mid \texttt{0} \cdots \texttt{9})* \\
\langle \textit{id} \rangle &::= \langle \textit{id-fragment} \rangle \; (\texttt{::}\langle \textit{id-fragment} \rangle)* \\
\langle \textit{integer} \rangle &::= (\texttt{-})? \; (\texttt{0} \cdots \texttt{9})+ \\
\langle \textit{real} \rangle &::= (\texttt{-})? \; (\texttt{0} \cdots \texttt{9})+ \; (\texttt{.}(\texttt{0} \cdots \texttt{9})+)?
\end{aligned}
$$

Example 3.1. `Spacecraft` and `Spacecraft::new` are identifiers, `3000` is an integer and `2.71828` is a real number.

3.1.2. Objects: domain/data, named/unnamed A partial model describes partial models with objects and relations between them. Objects include primitive data objects (e.g. integer and real numbers) or non-primitive domain objects. In either case, an object is either *named* or *unnamed*. A named object can be differentiated from the other named objects by using an identifier, while unnamed objects do not have designated identities. As such, replacing a named object with another object results in a different partial model, while replacing an unnamed object with another (otherwise identical) unnamed object will not change the partial model.

$$
\begin{aligned}
\langle \textit{object-id} \rangle &::= \langle \textit{named-obj-id} \rangle \mid \langle \textit{unnamed-obj-id} \rangle \\
\langle \textit{named-obj-id} \rangle &::= \texttt{'}\langle \textit{char} \rangle\texttt{+'} \\
\langle \textit{unnamed-obj-id} \rangle &::= \langle \textit{id} \rangle
\end{aligned}
$$

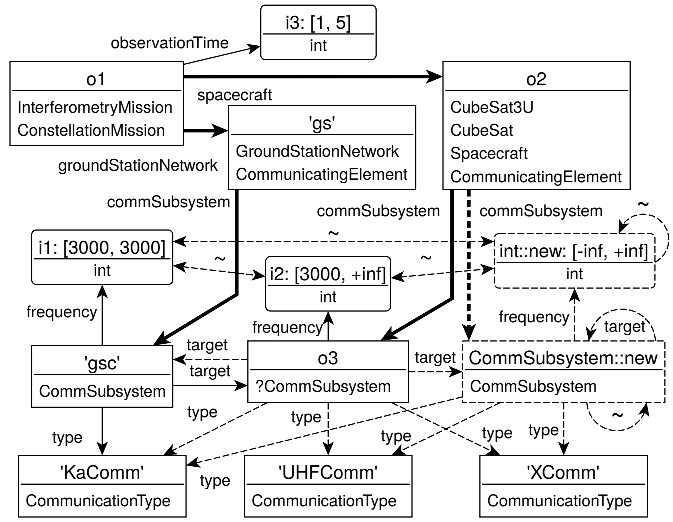


Figure 2 Graphical representation of a partial model.

Example 3.2. Figure 2 shows a partial model using graphical notation. Types (unary predicates) are written inside the boxes for the objects (`?` stands for uncertain types). Edges (binary predicates) are depicted as arrows (dashed arrows stand for uncertain edges and thick arrows stand for containment relations). The special $\sim$ edge denotes object equality.

The named object `'gsc'` represents a special antenna from a `GroundStation` named `'gs'`, and `o3` is an unnamed object representing another antenna. Here, a named object is used for `'gsc'` to differentiate its role from other antennas. The partial model lists three possible `CommunicationType` constants (based on the enumerated type in Figure 1) with named objects `'KaComm'`, `'UHFComm'`, and `'XComm'`. Here, we used named objects for all enumeration constants as we want to keep the identity of each type (i.e. it matters whether `'gsc'` has type `'KaComm'` or `'UHFComm'`). Figure 2 also depicts two data objects `i1` and `i2` that represent in the model (describing frequencies). Finally, the object `CommSubsystem::new` by itself represents all the potential new `CommSubsystem` objects and `int::new` represents all the integer data objects that may be added to the model.

3.1.3. Relation assertions Using the core syntax, a partial model is defined with *relation* and *value assertions* over the objects. Relation assertion defines a 4-valued truth-value (one of `true`, `false`, `unknown` and `error`) for an n-ary relation for a tuple of objects. Besides the usual truth values `true` and `false`, `unknown` represents that the given information is not specified in the model (thus it can be either `true` or `false` in generated models), while `error` represent inconsistent information (i.e. `true` *and* `false` at the same time). From a model generation perspective, a partial model with any `unknown` values means that the model is not yet finished (i.e. it is not yet a regular concrete model), while any `error` value makes the model inconsistent, thus no consistent instance models can be generated by model refinement from the partial model.

⟨*assertion*⟩ ::= ⟨*ground-rel-use*⟩`:` ⟨*logic-value*⟩`.` ▷ relation assertion

⟨*ground-rel-use*⟩ ::= ⟨*relation-id*⟩`(`(⟨*ground-term*⟩(`,` ⟨*ground-term*⟩)∗)?`)` ▷ relation atom with ground arguments

⟨*ground-term*⟩ ::= ⟨*object-id*⟩ ▷ partial model node

⟨*logic-value*⟩ ::= `true` | `false` | `unknown` | `error` ▷ 4-valued logic values

Combined with named and unnamed objects, unary relations are sufficiently expressive to uniformly represent simple node labels, more complex type systems, enumerated types, and primitive types (like integers).

Example 3.3. The following code excerpt shows some unary relation assertions from the partial model in Figure 2.

```
1  CommSubsystem('gsc'): true.
2  Spacecraft('gsc'): false.
3  Payload('gsc'): false. % ...
4  CubeSat3U(o3): true.
5  CubeSat(o3): true.
6  Spacecraft(o3): true.
7  CommunicatingElement(o3): true.
8  CommType('KaComm'): true.
9  CommType('UHFComm'): true.
10 CommType('XComm'): true.
11 int(i1): true. int(i2): true.
12 int(int::new): true.
```

Lines 1–3 of the example denote that it is `true` that the object `'gsc'` has the type `CommSubsystem`, but it does not have other types like `Spacecraft` and `Payload`. Complex type systems can be formalized by listing the combination of all type predicates. For example, the types of the object `o3` are listed in lines 4–7. An enumerated type can be formalized by listing all instances (like in lines 8–10), and specifying that all other objects does not have that type. Finally, primitive data objects are also represented by type predicates, like in lines 11–12.

Binary relation assertions can represent a wide range of structures like references (links, edges) and attributes.

Example 3.4. In Figure 2, communication paths between the satellites are represented by the `target` relation.

```
1 target('gsc', o3): true.
2 target(o3, 'gsc'): unknown.
3 target('gsc', 'gsc'): false.
```

In the partial model, it is known that object `'gsc'` is communicating with object `o3` (denoted with solid edges in Figure 2 and `target('gsc', o3): true` in line 1). Potential communication relations are denoted with `unknown` values, e.g. in line 2, the `target` link between `o3` and `'gsc'` is `unknown`. Finally, a partial model can represent the absence of a relation with the truth value `false`, e.g. in line 3.

3.1.4. Exists and equals

A partial model uses two special relations: the unary relation `exists` assigns a truth-value for the existence of an object, while the binary relation `equals` assigns a truth-value for the equivalence of two objects. These special relations can also take `unknown` truth values to represent abstract graph structures.

Example 3.5. The following lines illustrate different combinations of `exists` and `equals` from Figure 2.

```
1 exists(o3): true. equals(o3, o3): true.
2 equals(o1, o3): false. equals(o2, o3): false.
3 exists(CommSubsystem::new): unknown.
4 equals(CommSubsystem::new, CommSubsystem::new):
     unknown.
5 equals(i1, i2): unknown. equals(i2, i3): unknown.
6 equals(i1, i3): unknown.
```

Lines 1–2 show a *concrete* object `o3` which is existing, and it is equal to itself and different from others. Next, lines 3–4 show a *multi-object* `CommSubsystem::new` that may or may not `exist` (i.e. 0.. multiplicity) and its `unknown` equivalence with itself denotes that the object may represent multiple different objects (i.e. ..* multiplicity). Finally, lines 5–6 represent the uncertain equivalence of the three integers `i1`, `i2` and `i3`. These data objects may possibly be merged with each other.

3.1.5. Value assertions

In a concrete model, each data object (e.g. integer) has a concrete value defined by a single assignment. However, in a partial model, the value of a data object is defined by one or more closed interval assertions, where an interval can possibly be `empty` or infinite (`inf`). If several of such assertions exist, then the intersection (conjunction) of such intervals is taken.

⟨*assertion*⟩ ::= ⋯ | ⟨*ground-term*⟩`:` ⟨*interval*⟩`.` ▷ value assertion

⟨*interval*⟩ ::= `[`⟨*lower-bound*⟩`,` ⟨*upper-bound*⟩`]` ▷ non-empty interval literal

| `empty` ▷ empty interval literal

⟨*lower-bound*⟩ ::= ⟨*real*⟩ | `-inf` ▷ finite or infinite lower bound

⟨*upper-bound*⟩ ::= ⟨*real*⟩ | `+inf` ▷ finite or infinite upper bound

Example 3.6. The partial model in Figure 2 illustrates three `int` objects. The value of `i1` is set to 3000, which is defined by the interval `[3000, 3000]`. For `i2` we define only that the value is greater than or equal to 3000, thus we use interval `[3000, +inf]`. Finally, by setting an interval of `int::new` we define the potential range of all the new integers.

```
i1: [3000, 3000]. i2: [3000, +inf].
int::new: [-inf, +inf].
```

3.2. Syntactic sugar for partial models

Since partial models can be defined by the users as an initial input (seed) model, we also provide a simplified notation with syntactic shortcuts for the most frequently used constructs.

3.2.1. Simplified assertions

First, relation assertions can be simplified with the prefixes `?` and `!`, where `?` stands for the truth-value `unknown`, `!` stands for the truth-value `false` and the lack of a prefix stands for `true`. Shortcuts can also be introduced for data objects by referring to them with their value. As a result, relations in partial models can be defined by Prolog-style clauses while still keeping the option of using many-valued logic and abstract domains.

Original	Translated
⟨*ground-rel-use*⟩.	⟨*ground-rel-use*⟩: `true`.
`!`⟨*ground-rel-use*⟩.	⟨*ground-rel-use*⟩: `false`.
`?`⟨*ground-rel-use*⟩.	⟨*ground-rel-use*⟩: `unknown`.
(⟨*int*⟩)	`int(literals::`⟨*id*⟩`).` `literals::`⟨*id*⟩`:` ⟨*int*⟩.
(⟨*real*⟩)	`real(literals::`⟨*id*⟩`).` `literals::`⟨*id*⟩`:` ⟨*real*⟩.
(⟨*interval*⟩)	`literals::`⟨*id*⟩`:` ⟨*interval*⟩.

Table 1 Translations for brief relation assertions.

⟨*assertion*⟩ ::= ⋯ | (`?` | `!`)? ⟨*ground-rel-use*⟩.
▷ brief relation assertion

⟨*ground-term*⟩ ::= ⋯ | ⟨*int*⟩ | ⟨*real*⟩ | ⟨*interval*⟩
▷ integer, real or interval literal

Translations of the constructs defined above to the core syntax are shown in Table 1.

Example 3.7. Lines 1–3 of the following code excerpt show three examples for a positive, negative and uncertain assertions, respectively. Line 4 uses 3000 as a data object with the constant value 3000, while line 5 uses the interval `[3000, +inf]`.

```
1 target('gsc',o3).
2 !target('gsc','gsc').
3 ?target(o3,'gsc').
4 frequency('gsc', 3000).
5 frequency(o3, [3000, +inf]).
```

3.2.2. Any and default The complete definition of a partial model needs to assert truth values for all relations. This would require a large number of assertions, which is impractical. Therefore, we introduce two mechanisms to assert truth-values over a range of objects. First, assertions can be expressed for *any object* using the symbol `*`.

Additionally, *missing truth values* of relations can be asserted by using the `default` keyword. This allows discharging with the assumption that *everything unspecified is* `unknown` (i.e. open-world semantics). For example, we can avoid explicitly enumerating the types an object *does not* have by setting the `default` logic value for each type to `false`((i.e. closed-world semantics). Conversely, setting `default` to `error` means every possible value of the relation must be explicitly enumerated, even `unknown` values.

⟨*ground-term*⟩ ::= ⋯ | `*` ▷ all partial model nodes

⟨*default-assertion*⟩ ::= `default` ⟨*assertion*⟩

Table 2 shows the translations for these constructs.

Example 3.8. By defining the `default` value of `CommSubsystem` as `false` in line 1, we may omit the statement `CommSubsystem(o1): false`. However, logic values differing from `false` must be state explicitly.

Original	Translated
`(*)`	the containing statement is copied for all possible objects
`default` ⟨*assertion*⟩	⟨*assertion*⟩ if no other logic value was defined previously

Table 2 Translations for any and default assertions.

```
1 default CommSubsystem(*): false.
2 CommSubsystem('gsc').
3 ?CommSubsystem(o3).
```

4. Graph predicates and metrics

Building on the core notions for describing partial models from section 3, in this section, we will introduce *graph predicates* and *metrics*. Graph predicates capture derived relations and well-formedness constraints, while metrics allow numerical computations, including constraints on numerical values, as well as objective functions such as model size and cost.

4.1. Graph predicates

We partition relation symbols (i.e. ⟨*relation-id*⟩) into *base relation* symbols and *defined predicate* symbols.

The base relations include built-in relations, such as `int` and `real`, as well as any types, references and attributes that comprise the model. Any relation symbol appearing in the partial model without an associated predicate definition is considered to be a base relation symbol. The truth values of these relations can be specified arbitrarily by assertions.

In contrast, one may create defined predicate symbols by *predicate defintions*. The body of the definition may be evaluated on the model to compute a logic values. Thus assertions involving defined predicate symbols are *constraints* on the computed logic values. In the following, we will discuss the syntax of predicate definitions.

4.1.1. Predicate definitions Predicate definitions specify queries (or constraints) as Datalog-like expressions over relation symbols. The name and parameters of the predicate are separated by the `:=` symbol from the body of the predicate definition. The body is specified in a disjunctive normal form. Literals in the body of predicate definition may refer to (n-ary) relation symbols, including both defined (interpreted) predicate symbols and base (uninterpreted) relation symbols, metric checking literals (to check whether the value of a metric lies within an interval using the `in` operator), the transitive closures of binary relations, as well as negations thereof. However, as a specific limitation, recursive definitions (when a predicate definition refers to itself directly or indirectly via relations and metric use) are disallowed.

When using relations (and metrics) in predicate definitions, only variables (e.g. `x`) are allowed as arguments. Variables in a body of a predicate that do not appear in the parameter list of the predicate definition are implicitly considered to be existentially quantified. As an exception, for variables that solely appear in a single negative literal, the existential quantifier is moved inside the negation in order to match the quantification semantics of the VIATRA query language (Varró et al. 2016).

⟨*predicate-def*⟩ ::= ⟨*pred-def-core*⟩
▷ basic form of predicate definition without any modifiers

⟨*pred-def-core*⟩ ::=
⟨*predicate-id*⟩`(`(⟨*param*⟩(`,` ⟨*param*⟩)∗)?`)` `:-` ⟨*disjunction*⟩`.`

⟨*param*⟩ ::= ⟨*var-id*⟩ ▷ parameter variable

⟨*disjunction*⟩ ::= ⟨*conjunction*⟩ (`;` ⟨*conjunction*⟩)∗

⟨*conjunction*⟩ ::= ⟨*literal*⟩ (`,` ⟨*literal*⟩)∗

⟨*literal*⟩ ::= ⟨*atom*⟩ ▷ positive atom
| `!`⟨*atom*⟩ ▷ negative atom

⟨*atom*⟩ ::= ⟨*logic-value*⟩ ▷ 4-valued logic constant
| ⟨*relation-use*⟩ ▷ relation application
| ⟨*relation-id*⟩`+(`⟨*term*⟩`,` ⟨*term*⟩`)`
▷ transitive closure
| ⟨*relation-id*⟩`*(`⟨*term*⟩`,` ⟨*term*⟩`)`
▷ reflexive transitive closure
| ⟨*metric-use*⟩ `in` ⟨*interval*⟩ ▷ metric check
| ⟨*term*⟩ `==` ⟨*named-object-id*⟩ ▷ find by name

⟨*relation-use*⟩ ::= ⟨*relation-id*⟩`(`(⟨*term*⟩(`,` ⟨*term*⟩)∗)?`)`

⟨*metric-use*⟩ ::= ⟨*metric-id*⟩`(`(⟨*term*⟩(`,` ⟨*term*⟩)∗)?`)`

Example 4.1. Consider the following two predicate definitions:

```
directCommunicationLink(from, to) :-
    Spacecraft(from), CommunicatingElement(to),
    commSubsystem(from, fromComm),
    target(fromComm, toComm),
    commSubsystem(to, toComm).
noLinkToGroundStation(s) :-
    Spacecraft(s), g == 'gs',
    !directCommunicationLink+(s, g).
```

The binary predicate `directCommunicationLink` matches pairs of `Spacecraft` objects `from` and `CommunicatingElement` objects `to` such that there is a direct communication path between the antennas of the two elements. The variables `fromComm` and `toComm` of the antennas are (implicitly) existentially quantified.

The `noLinkToGroundStation` selects `Spacecraft` objects where there is no indirect communication path to the named `'gs'` object, where indirect paths are defined as the transitive closure of the `directCommunicationLink` relation.

4.1.2. Combining metrics and predicates

As a special feature of our language, metrics (see subsection 4.2) and graph predicates can mutually depend on each other. On the one hand, a predicate can check if the value calculated by a metric is within a specific interval (*metric value check* atom). Reversely, predicates can serve as conditions in aggregation expressions (e.g. to count the number of matches of a predicate) and in conditionals.

4.1.3. Type annotations

In addition to the core syntax above, several language constructs help define predicates as syntactic sugar. Firstly, unary predicates may appear as *type annotations* in parameter. For each type annotation, a call to the predicate used as type is added to each disjunctive case of the predicate body, which results in a notation reminiscent of typed object-oriented programming languages.

⟨*param*⟩ ::= · · · | ⟨*relation-id*⟩ ⟨*var-id*⟩ ▷ typed parameter

Example 4.2. Predicate definition `directCommunicationLink` from the previous example can be more succinctly written as follows (where `Spacecraft` and `CommunicatingElement` are introduced as types):

```
directCommunicationLink(Spacecraft from,
        CommunicatingElement to) :-
    commSubsystem(from, fromComm),
    target(fromComm, toComm),
    commSubsystem(to, toComm).
```

4.1.4. Error predicates

The `error` keyword may be used to introduce *error predicates*, which should never hold in a consistent model. This is equivalent to asserting that the predicate is `false` everywhere in the model.

⟨*predicate-def*⟩ ::= · · · | `error` ⟨*pred-def-core*⟩
▷ error predicate definition

Example 4.3. To designate `noLinkToGroundStation` as an error predicate, one may define it as

```
error noLinkToGroundStation(s) :-
    Spacecraft(s), g == 'gs',
    !directCommunicationLink+(s, g).
```

which is equivalent to define `noLinkToGroundStation` without the `error` keyword and asserting that

```
noLinkToGroundStation(*): false.
```

For convenience, we also allow error predicates without a predicate name. These definitions are filled in with a newly generated, unique name before translation into assertions.

⟨*unnamed-err*⟩ ::=
`error(`(⟨*param*⟩(`,` ⟨*param*⟩)∗)?`)` `:-` ⟨*disjunction*⟩`.`
▷ unnamed error predicate definition

Example 4.4. The unnamed error predicate definition

```
error(s) :- Spacecraft(s),
    g == 'gs', !directCommunicationLink+(s, g).
```

is equivalent to a definition with a generated unique name

```
error unnamed0001(s) :- Spacecraft(s),
    g == 'gs', !directCommunicationLink+(s, g).
```

4.1.5. Functional predicates

We introduce the `functional` keyword to express that the last parameter of an n-ary relation ($n \geq 2$) is functionally dependent on its previous parameters, i.e. for any possible binding of the first $n-1$ parameters, there is at most only a single possible value for the nth parameter for which the predicate evaluates to `true` in a consistent model. This is achieved by automatically

Original	Translated
`m() =:=` ⟨*real*⟩	`m() in [`⟨*real*⟩`,` ⟨*real*⟩`]`
`m() =!=` ⟨*real*⟩	`m() in [!(`⟨*real*⟩`,` ⟨*real*⟩`])`
`m() <=` ⟨*real*⟩	`m() in [-inf,` ⟨*real*⟩`]`
`m() <` ⟨*real*⟩	`!(m() in [`⟨*real*⟩`, +inf])`
`m() >=` ⟨*real*⟩	`m() in [`⟨*real*⟩`, +inf]`
`m() >` ⟨*real*⟩	`!(m() in [-inf,` ⟨*real*⟩`])`

Table 3 Translations of comparison operators.

adding a new `error` predicate for each `functional` predicate definition. A (base) relation may be marked as `functional` to imply the same kind of constraint. In that case, the definition of the predicate is omitted, only writing the `functional` keyword and the name of the (base) relation.

⟨*predicate-def*⟩ ::= ⋯ | `functional` ⟨*pred-def-core*⟩
▷ functional predicate definition

⟨*functional-decl*⟩ ::= `functional` ⟨*base-relation-id*⟩ `.`
▷ functional base relation declaration

Example 4.5. When one designates `directCommunicationLink` as `functional` by writing

```
functional directCommunicationLink(
        Spacecraft from,
        CommunicatingElement to) :-
    commSubsystem(from, fromComm),
    target(fromComm, toComm),
    commSubsystem(to, toComm).
```

then each `from` object may have at most one `to` object associated with it. Thus the following `error` definition is generated:

```
error directCommunicationLinkNotFunctional(
        Spacecraft from) :-
    !equals(to1, to2),
    directCommunicationLink(from, to1),
    directCommunicationLink(from, to2).
```

4.1.6. *Comparison operators*

We allow comparison operators in addition to the `in` keyword in expressions checking the value of a metric.

⟨*atom*⟩ ::= ⋯ | ⟨*metric-use*⟩ ⟨*comp-op*⟩ ⟨*real*⟩
▷ comparison of metric value

⟨*comp-op*⟩ ::= `=:=` | `=!=` | `<=` | `<` | `>=` | `>`

The comparisons can be readily translated back into a check expression with the `in` keyword as shown in Table 3.

4.2. Model metrics

Analogously to predicate definitions that allow computing logic values from (partial) models, we introduce *metric* definitions to compute *numerical* values.

4.2.1. *Metric definitions*

Metric definitions are structured similarly to predicate definitions (using the `:=` operator instead of the `:-` operator). The body of the metric definition is a *metric expression*. Basic elements of metric expressions include numerical constants, terms referencing parameters, other variables or named objects, applications of other metrics, as well as elementary algebraic operators. While we currently only specify a small set of elementary operators (addition, subtraction, multiplication, division, exponentiation and the floor function), the language is easily extensible with additional operators (e.g. logarithms). Similarly to predicate definitions, direct or indirect recursion between metric definitions is not allowed.

Evaluation of the metric expression may fail with a non-number result if an illegal operation, e.g. division by zero is performed. Otherwise, the result of the expression evaluation will be the value of the defined metric. When a term (variable or named object) appears in a metric expression, it must refer to a data node in the graphs and will evaluate to the numeric value bound to the data node. Plain terms not referring to data nodes cause expression evaluation to fail. In contrast with predicate definitions, metric definitions do not quantify over their free variable existentially.

⟨*metric-def*⟩ ::=
⟨*metric-id*⟩ `(`(⟨*param*⟩(`,` ⟨*param*⟩)∗)?`)` `:=` ⟨*metric-expr*⟩ `.`
▷ metric definition

⟨*metric-expr*⟩ ::= ⟨*real*⟩ ▷ number literal
| ⟨*term*⟩ ▷ value of data node
| ⟨*metric-use*⟩ ▷ metric application
| ⟨*unary-op*⟩ ⟨*metric-expr*⟩ ▷ unary operation
| ⟨*metric-expr*⟩ ⟨*binary-op*⟩ ⟨*metric-expr*⟩
▷ binary operation
| ⟨*metric-expr*⟩ `as int` ▷ cast (floor function)
| `(`⟨*metric-expr*⟩`)` ▷ parenthesized expression

⟨*binary-op*⟩ ::= `+` | `-` | `*` | `/` | `^`

⟨*unary-op*⟩ ::= `+` | `-`

Interaction between metrics and relations is provided by conditional, switch and aggregate expressions, which allow the value of a metric to be determined by the values of base relations or defined predicates.

4.2.2. *Conditionals and switch expressions*

A conditional (`if`) expression contains a single relation use as its condition. If the condition evaluates to `true`, the metric expression in the `then` branch is evaluated, else the `else` branch is evaluated.

⟨*metric-expr*⟩ ::= ⋯ | `if` ⟨*relation-use*⟩ `then` ⟨*metric-expr*⟩
`else` ⟨*metric-expr*⟩
▷ conditonal expression

A switch expression contains multiple relation applications as conditions. If exactly one condition evaluates to `true` and the rest are `false`, the corresponding metric expression (separated from the condition by the `->` operator) is evaluated. Otherwise, the switch expression will fail.

⟨*metric-expr*⟩ ::= · · · | ⟨*relation-use*⟩ `->` ⟨*metric-expr*⟩
(`;` ⟨*relation-use*⟩ `->` ⟨*metric-expr*⟩)∗
▷ switch expression

4.2.3. *Aggregations* Aggregation operators allow aggregating the values of metrics over relations. The aggregation is formed by selecting a metric to be aggregated and using a relation serving as the condition. One can introduce new (unbound) variables for a metric expression only in such relation uses. For each binding of the fresh variables in the condition that evaluate it to `true`, the metric expression is evaluated and the results are aggregated.

The `sum`, `min` and `max` aggregations operators take the sum, minimum and maximum of the aggregated values, respectively. The `single` operation is special: if exactly one value is aggregated, i.e. the condition evaluates to `true` for exactly one variable binding, it returns the single value of the aggregated metric expression. However, if there are zero or too many values to aggregate, the evaluation fails. This allows traversing the model along functional dependencies to extract values from data objects, which mimics navigating along relations and attributes in OCL.

⟨*metric-expr*⟩ ::= · · ·
| ⟨*aggr-op*⟩ `{` ⟨*metric-use*⟩ `|` ⟨*relation-use*⟩ `}`
▷ aggregation expression

⟨*aggr-op*⟩ ::= `sum` | `min` | `max` | `single`

4.2.4. *Metric assertions* Similarly to relation and object value assertions, a partial model may specify the values of metric by *metric assertions*. In a partial model, the value of the metric is an interval. In each concrete model, the value of the metric must lie in the interval specified in the partial model.

⟨*assertion*⟩ ::= · · · | ⟨*ground-metr-use*⟩`:` ⟨*interval*⟩`.`
▷ metric assertion

⟨*ground-metr-use*⟩ ::=
⟨*metrc-id*⟩`(`(⟨*ground-term*⟩(`,` ⟨*ground-term*⟩)∗)?`)`
▷ metric atom with ground arguments

Example 4.6. The following metrics are an excerpt from the calculation of the costs of an `InterferometryMission`:

```
1  missionCost(m) :=
2      sum { spacecraftCost(s) | spacecraft(m, s) }
3      + 100000.0 * single { t |
4          observationTime(m, t) }.
5  spacecraftCost(s) :=
6      basePrice(s) * (kindCount(s) ^ (-0.25)) +
7      payloadCost(s) + commSubsysCost(s).
8  kindCount(s) :=
9      CubeSat3U(s) -> sum { 1 | CubeSat3U(_s2) }
10   ; CubeSat6U(s) -> sum { 1 | CubeSat6U(_s2) }
11   ; SmallSat(s)  -> sum { 1 | SmallSat(_s2) }.
12 missionCost(o1): [0.0, 50000000.0].
```

In lines 1–4, the `missionCost` metric describes the overall cost of the mission, which is the sum of the individual costs of the spacecraft connected to it via the `spacecraft` relation, plus $100,000 per each hour of observation. The time of observation is the value bound to the single data object associated to the mission `m` via the `observationTime` relation.

In lines 5–7, `spacecraftCost` calculates the costs associated with a satellite `s`, which is comprised of the unit cost, the cost of the payload and the cost of the associated communications subsystems. Thanks to economies of scale, the unit cost decreases as the number of spacecraft of the same kind is increased. Subsequent calculations except `kindCount` were omitted from the code example for brevity.

In line 8–11, `kindCount` calculates the number of spacecraft of the same kind as `s`. This is achieved by a switch expression, which finds the type of the spacecraft, and aggregation expressions, which count spacecraft of a given type.

Lastly, line 12 contains an assertion, which restricts the `InteferometryMission` `o1` to be no more expensive than $50,000,000.

4.2.5. *Syntactic sugar for defining metrics* Several syntactic shortcuts are provided for defining metrics. First, unary relations can be placed before parameter names to serve as type predicates, similarly to predicate definitions. This is equivalent to introducing a switch expression with a single case, and ensures that the evaluation of the metric fails if the parameter does not satisfy the given relation.

Example 4.7. To restrict the domain of `spacecraftCost` to `Spacecraft` objects, we may write

```
spacecraftCost(Spacecraft s) :=
    basePrice(s) * (kindCount(s) ^ (-0.25)) +
    payloadCost(s) + commSubsysCost(s).
```

which is equivalent to

```
spacecraftCost(s) := Spacecraft(s) ->
    basePrice(s) * (kindCount(s) ^ (-0.25) +
    payloadCost(s) + commSubsysCost(s)).
```

Moreover, we introduce a dedicated operator `count` to count objects satisfying some relations. This replaces the summation of constant `1` values to enhance readability.

⟨*metric-expr*⟩ ::= · · · | `count` `{` ⟨*relation-use*⟩ `}`
▷ count aggregation expression

Example 4.8. With the `count` operator, `kindCount` can be more concisely written as

```
kindCount(s) :=
    CubeSat3U(s) -> count { CubeSat3U(_s2) }
  ; CubeSat6U(s) -> count { CubeSat6U(_s2) }
  ; SmallSat(s)  -> count { SmallSat(_s2) }.
```

Lastly, there is a shorthand for the joint use of `functional` relations and the `single` aggregation operator. Relation symbols that are `functional` can be used as if they were metrics by omitting their last argument. The value of the expression is the value bound to the data object appearing as the single possible value of the omitted argument such that the relation evaluates to `true`. In other words, we may write `single` `{` $\mathtt{m}_n$ `|` ⟨*relation-id*⟩`(`$\mathtt{m}_1$`,` · · · `,` $\mathtt{m}_{n-1}$`,` $\mathtt{m}_n$`)` `}` as ⟨*relation-id*⟩`(`$\mathtt{m}_1$`,` · · · `,` $\mathtt{m}_{n-1}$`)` instead. This allows mimicking object-oriented programming languages more closely, where values of attributes (here represented as `functional` binary relations) can be accessed directly from objects.

⟨*metric-expr*⟩ ::= · · · | ⟨*relation-use*⟩ ▷ functional application

Example 4.9. Because observationTime is a functional relation, the missionCost metric can access its value directly:

```
missionCost(m) :=
    sum { spacecraftCost(s) | spacecraft(m, s) }
    + 100000.0 * observationTime(m).
```

4.3. Scopes

Scope constraints are used to restrict the number of objects represented by multi-objects in a model. While scopes can be expressed with metrics using the count operator, we provide a dedicated facility for scope definitions to separate this concern from the rest of the partial model.

Scope definitions refer to an unary relation acting as a type predicate. They may specify lower or upper bounds or the exact number of objects satisfying the type predicate. In contrast with other model generators, such as Alloy (D. Jackson 2002), there are no restrictions on the unary relations for which scopes can be defined. In particular, they may be arbitrarily overlapping. However, contradictory scope constants, just like contradictory error predicates, lead to partials models without any corresponding consistent concrete model.

⟨*scope-decl*⟩ ::= scope ⟨*relation-id*⟩ ⟨*comp-op*⟩ ⟨*int*⟩ .
▷ scope declaration

Example 4.10. The following scope constraints specify that the model should contain between 16 and 32 domain objects, while exactly 12 objects should be Spacecraft instances.

```
scope domain >= 16.
scope domain <= 32.
scope Spacecraft =:= 12.
```

Without using the notation for scopes, this could have alternatively been written as

```
numberOfDomainObjects() := count { domain(_o) }.
numberOfDomainObjects(): [16, 32].
numberOfSpacecraft() := count { Spacecraft(_s) }.
numberOfSpacecraft(): [12, 12].
```

4.4. Containment hierarchy

Containment constraints often occur in industrial modeling environments such as UML (Rumbaugh et al. 2004), SysML (Friedenthal et al. 2008) and EMF ("Eclipse Modeling Framework" 2019). While containment constraints can be specified as error predicates or as assertions, we provide a language-level facility to enable an easy definition. The advantages of this approach are twofold:

- Handwritten containment constraints as predicate definition can frequently grow very large due to the need to specify all containment relations in one place.
- Model generators can leverage explicitly signaled containment information to increase their efficiency.

Binary base relations can be designated as containment relations using the containment keyword. Unary base relations can be designated as containment roots using the root keyword.

⟨*containment-decl*⟩ ::= containment ⟨*base-relation-id*⟩ .
▷ containment relation declaration

⟨*root-decl*⟩ ::= root ⟨*base-relation-id*⟩ .
▷ root relation declaration

To satisfy the containment hierarchy constraint in a concrete model, for every object o exactly one of the following constraints must hold:

- o is a data object.
- o has a unary base predicate marked as a *root* that evaluates to true (i.e. t is a root object).
- relation(c, o) evaluates to true for exactly one containment relation relation and object c.

Additionally, the containment relations must span a forest, i.e. there may be no loops along containment relations.

Example 4.11. Consider the root and containment declarations

```
root ConstellationMission.
containment spacecraft.
containment commSubsystem.
```

Each object must either be a data object, a root object, or be connected to some other (containing) object with a spacecraft or a commSubsystem reference. In addition, spacecraft and commSubsystem references jointly form a forest. We can alternatively write this as error patterns as follows:

```
1  containmentRelation(c, o) :-
2      spacecraft(c, o); commSubsystem(c, o).
3  error cyclicContainment(o) :-
4      containmentRelation+(o, o).
5  numberOfContainers(o) :=
6      count { spacecraft(_c, o) } +
7      count { commSubsystem(_c, o) }.
8  rootRelation(o) :- ConstellationMission(o).
9  containedObject(o) :-
10     domain(o), !rootRelation(o).
11 error wrongNumberOfContainers(o) :-
12     containedObject(o),
13     numberOfContainers(o) =!= 1
14   ; !containedObject(o),
15     containmentRelation(_c, o).
```

Lines 1–2 define the predicate containmentRelation, which matches pairs of objects in any containment relations. The error predicate cyclicContainment in lines 3–4 asserts that cycles in this relation are an error, i.e. containment edges form a forest. Lines 5–7 count a the number of containers than an object o has. Note that instead of counting containmentRelation edges, we sum the counts of the different containment relations. This allows detecting parallel edges (of different containment relations) multiple times, which are erroneous. In line 9–10, the containedObject predicate matches the objects which need exactly one adjacent containment edge. Any other objects must not have a container. This constraint is formalized by the wrongNumberOfContainers error predicate in lines 11–15.

5. Mapping from domain-specific modeling technologies

Most existing model generators rely upon popular modeling technologies to define the core domain concepts in the form of a metamodel. To illustrate the power of abstractions in our

Original	**Translated**
abstract class ⟨*relation-id*⟩ { ··· }	Abstract classes do not have to be translated explicitly.
class ⟨*relation-id*⟩ { ··· } (extends ⟨*supertypes*⟩)?	?exists(⟨*relation-id*⟩::new). ?equals(⟨*relation-id*⟩::new, ⟨*relation-id*⟩::new). ⟨*relation-id*⟩(⟨*relation-id*⟩::new). For each *supertype*: ⟨*supertype-id*⟩(⟨*relation-id*⟩::new). For each *incompatible* type: ⟨*other-type-id*⟩(⟨*relation-id*⟩::new).
enum ⟨*relation-id*⟩ { ⟨*object-id*$_1$⟩, ··· ,⟨*object-id*$_k$⟩ }	For each enum literal i: ⟨*relation-id*⟩('⟨*object-id*$_i$⟩'). For each literals $i \neq j$: !equals('⟨*object-id*$_i$⟩', '⟨*object-id*$_j$⟩'). default !⟨*relation-id*⟩(*).

Table 4 Translations for Xcore classes and enums.

specification language, we demonstrate how the concrete syntax of Xcore ("Xcore" 2020) can be mapped into partial models.

Xcore introduces a type system for models described with a `class` hierarchy, denoting the generalization relation with `extends` keyword and selecting classes that does not have direct instances with the `abstract` keyword. The class hierarchy is extended by enumerated types with predefined instances (with the `enum` keyword). References and attributes are defined over the strict type hierarchy with multiplicity constraints (denoted with `[`*lower*`,` *upper*`]`) and containment notation (denoted with the `contains` keyword).

Example 5.1. The following example captures a fragment of the metamodel in Figure 1.

```
abstract class CommunicatingElement {
  contains CommSubsystem[1, 2] commSubsystem
}
class CubeSat3U extends CommunicatingElement { }
class CommSubsystem {
  refers CommSubsystem[0, 1] target
  CommunicationType[1, 1] type
  real[1, 1] frequency
}
enum CommunicationType { KaComm, UHFComm, XComm }
```

`CommunicatingElement` is defined as an `abstract class`, which has a concrete subclass `CubeSat3U`. Each `CommunicatingElement` `contains` one or two `CommSubsystem` instances, which has a non-containment reference to another `CommSubsystem`, a reference to an `enum CommunicationType`, and a `real frequency` value.

In order to map Xcore to partial models, each non-abstract class can be mapped to partial model object ⟨*relation-id*⟩`::new`, which represents all potential newly generated instances, and defines its valid type combination (based on the inheritance relations). With this mapping, `abstract` classes are also represented correctly.

Enumerated types with enum literals ⟨*id*$_1$⟩, ··· ,⟨*id*$_n$⟩ can mapped to partial model objects by

– defining a named-object for each literal '⟨*id*$_i$⟩',
– setting the type of each object '⟨*id*$_i$⟩' to `true` for the enum type and `false` for each other types and
– specifying that no other object has the enum type.

These translations are illustrated in Table 4.

The type hierarchy imposes global structural well-formedness constraints over all objects of the partial model. This is enforced by the following structural predicates:

– If an object is an instance of class `C` then it is an instance of all supertypes `Sup_1`, ··· , `Sup_n`.

```
error(C o) :- !Sup_1(o). % ...
error(C o) :- !Sup_n(o).
```

– If an object is an instance of class `C`, then it is not an instance of any incompatible class `I_1`, ··· , `I_n` that is not connected to `C` via a directed `extends` path.

```
error(C o) :- I_1(o). % ...
error(C o) :- I_n(o).
```

– If an object is an instance of an abstract type `A`, then it must be an instance of a concrete subtype `Sub_1`, ··· , `Sub_n`.

```
error(A o) :- !Sub_1(o), /*...*/, !Sub_n(o).
```

Each reference and attribute `R` is mapped to a relation with default value `unknown` between the objects, and `false` between objects with incompatible types `I_1` and `I_2` (including the `::new` objects derived from the classes).

```
default R(*, *): unknown.
% For all incompatible objects i_1, i_2:
!R(i_1, i_2).
```

References and attributes also impose extra global structural constraints as follows.

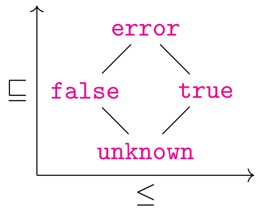
– Type of a relation `R` between classes `S` and `T` is enforced by the following constraint:

```
error(s, t) :- R(s, t), !S(s).
error(s, t) :- R(s, t), !T(t).
```

– If a reference `R` has a `contains` label, then it is a containment relation.

```
containment R.
```

– The multiplicity constraint `[`*lower*`,` *upper*`]` is translated into two error predicates that match when the multiplicity is outside of the given range.



⊕	u	f	t	e
u	u	f	t	e
f	f	f	e	e
t	t	e	t	e
e	e	e	e	e

(a) Lattice of logic values and information merge

$\neg^4$	
u	u
f	t
t	f
e	e

$\vee^4$	u	f	t	e
u	u	u	t	t
f	u	f	t	e
t	t	t	t	t
e	t	e	t	e

$\wedge^4$	u	f	t	e
u	u	f	u	f
f	f	f	f	f
t	u	f	t	e
e	f	f	e	e

(b) Truth tables of logical connectives

Figure 3 Four-valued Belnap–Dunn logic

```
countOfR(S o) := count { R(o, _) }.
error(S o) :- countOfR(o) < lower.
error(S o) :- countOfR(o) > upper.
```

– If the *upper* multiplicity bound is `1` and the target type `T` is `real` or `int`, the attribute `R` is marked as functional for easy use in metric definitions.

```
functional R.
```

Similar mappings can be provided for other modeling technologies like OWL for description logic, GraphQL or languages used in graph database technologies.

6. Semantics of partial models

6.1. Representing incomplete and inconsistent models

6.1.1. Four-valued logic In this paper, we utilize 4-valued logic to explicitly represent unfinished, partial (*paracomplete*) models, as well as errors and inconsistencies (*paraconsistency*) arising during the evaluation of computations over such models. This section provides semantic foundations for our specification language based on the *inconsistency-tolerant* Belnap-Dunn 4-valued logic (Belnap 1977; Kamide & Omori 2017), which can reason about runtime errors caused by undefined arithmetic operations, such as division by zero (McKubre-Jordens & Weber 2012).

Belnap-Dunn 4-valued logic contains the usual false `false` and true `true` truth values, the *unknown* `unknown` value introduced for unspecified or unknown properties, and the *inconsistent* `error` value that signals inconsistencies and computation failures. The subset {`false`, `true`, `unknown`} of logic values can express partial, but consistent information. Conversely, the subset {`false`, `true`, `error`} expresses possibly inconsistent, but complete information.

Two partial orders can be defined over 4-valued logic values (Figure 3a). *Information order* (denoted by $\sqsubseteq$) expresses the gathering of information as new facts are learned during the refinement of partial models. Facts with `unknown` logical value can be set to either `true` or `false`, while a change to `error` signifies an inconsistency or failure. This order is defined as

$$(X \sqsubseteq Y) \Leftrightarrow [(X = \texttt{unknown}) \vee (X = Y) \vee (Y = \texttt{error})].$$

The second partial order is *implication order*, which defined as

$$(X \leq Y) \Leftrightarrow [(X = \texttt{false}) \vee (X = Y) \vee (Y = \texttt{true})]$$

and serves as a generalization of logical implication. We will write $X \sqsubset Y$ and (resp. $X < Y$) when $X \sqsubseteq Y$ (resp. $X \leq Y$) and $X \neq Y$ hold.

The *information merge* operator $\oplus$ merges 4-valued truth values where contradictory information results in `error`. Other operations on 4-valued truth values $\neg^4$, $\vee^4$, and $\wedge^4$ are extensions of the usual logic operators $\neg$, $\vee$, and $\wedge$. Their truth tables (see Figure 3b) correspond with their classical counterparts for {`false`, `true`} inputs.

Semantically, `unknown` truth value represents potential `true` or `false` (or `error`) values, and the semantic is chosen to cover all of those options. On the other hand, `error` is often unintuitive, but it allows the precise and explicit localization of inconsistencies within models (Belnap 1977; Chechik et al. 2011). For example, we may see that if $X =$ `error` and $Y =$ `unknown`, then $X \vee^4 Y =$ `true`, because the *only* way for our logical inference to result in a consistent truth value is to eventually learn that Y is `true`. Should Y turn out to be `false`, the inconsistent `error` value will be propagated.

6.1.2. Interval arithmetic We use interval arithmetic to represent unfinished or inconsistent numerical values. A closed, possibly infinite interval $iv \in \mathbb{IV} \subsetneq 2^{\mathbb{R}}$ of real numbers denotes a set of possible numerical values. The empty interval $\emptyset \in \mathbb{IV}$ denotes a missing value or a result of failed computation.

The operators $+^\sharp$, $-^\sharp$, $\cdot^\sharp$, $/^\sharp$, $\uparrow^\sharp$, $\Sigma^\sharp$, $\min^\sharp$ and $\max^\sharp$ refer to the interval arithmetic (Kulisch 2009) versions of the usual $+$ (addition), $-$ (subtraction), $\cdot$ (multiplication), $/$ (division), $\uparrow$ (exponentiation), Σ (summation), min and max operations over real numbers, respectively. The $\sqcup$ symbol denotes the *join* (smallest interval containing both intervals) of two intervals, while $\cap$ denotes interval intersection. Additionally, we define a special (directed) multiplication operator, which properly propagates the number of errors when multiplying a value with a number of matches.

$$iv_1 \triangleright iv_2 = \begin{cases} iv_1 \cdot^\sharp iv_2 & \text{if } iv_2 \neq \emptyset, \\ [0,0] & \text{if } 0 \in iv_1 \text{ and } iv_2 = \emptyset, \\ \emptyset & \text{if } 0 \notin iv_1 \text{ and } iv_2 = \emptyset. \end{cases}$$

6.2. Partial models

First, we provide an algebraic definition for partial models. For that purpose, one needs to establish a signature and a logic structure defined over it.

Definition 1. A *signature* $\langle \Sigma, \alpha \rangle$ is collection of relation and metric *symbols* $\Sigma = \Sigma_R \cup \Sigma_N$ and an *arity* function $\alpha \colon \Sigma \to \mathbb{N}$. Σ_R is the finite set of *relation symbols*, which includes

– $\mathtt{R}_1, \ldots, \mathtt{R}_k$, which are the *base relation symbols*;

- $\mathtt{P}_1, \dots, \mathtt{P}_\ell$, which are the *predicate symbols*;
- $\mathtt{'N}_1\mathtt{'}, \dots, \mathtt{'N}_t\mathtt{'}$, which are the *names* of named objects;
- $\mathtt{exists}$, which is the *existence relation symbol* with $\alpha(\mathtt{exists}) = 1$;
- $\mathtt{equals}$, which is the *equality relation symbol* with $\alpha(\mathtt{equals}) = 2$;
- $\mathtt{domain}$, designating *domain objects* with $\alpha(\mathtt{domain}) = 1$;
- $\mathtt{data}$, designating *data objects* with $\alpha(\mathtt{data}) = 1$;
- $\mathtt{real}$, designating *real number data objects* with $\alpha(\mathtt{real}) = 1$;
- $\mathtt{int}$, designating *integer data objects* with $\alpha(\mathtt{int}) = 1$.

Σ_M is the finite set of *numeric symbols* that is disjoint from Σ_S and includes

- $\mathtt{M}_1, \dots, \mathtt{M}_u$, which are the *metric symbols*;
- $\mathtt{value}$, designating the *numeric value* of data objects with $\alpha(\mathtt{value}) = 1$.

Definition 2. Given a signature $\langle \Sigma, \alpha \rangle$, a *partial model* is a logic structure $P = \langle \mathcal{O}_P, \mathcal{I}_P, \mathcal{V}_P \rangle$, where

- $\mathcal{O}_P$ is the finite set of objects in the model;
- $\mathcal{I}_P$ gives a 4-valued logic interpretation for each relation symbol $r \in \Sigma_R$ $\mathcal{I}_P(r)\colon (\mathcal{O}_P)^{\alpha(r)} \to \{\mathtt{false}, \mathtt{true}, \mathtt{unknown}, \mathtt{error}\}$;
- $\mathcal{V}_P$ gives a numeric value interpretation for each metric symbol $m \in \Sigma_M$ as $\mathcal{V}_P(m)\colon (\mathcal{O}_P)^{\alpha(m)} \to \mathbb{IV}$.

We define a concept of regularity for partial models (for detailed definitions, see Appendix C). The relevance of the concept is that each intermediate model retrieved by a graph solver is assured to be regular by applying a set of propagation rules. Naturally, when an initial partial model is provided by the user, it may not be regular, but one can still apply those propagation rules to regularize it. Note that the regularity of a partial model is different from its consistency, which will be defined in the sequel.

Definition 3. A partial model P is *regular* if it is structurally regular (Definition 11), naming regular (Definition 12) and data regular (Definition 13).

Concrete instance models are special form of partial models where all structural predicates are resolved to $\mathtt{true}$ or $\mathtt{false}$ values (i.e. $\mathtt{unknown}$ and $\mathtt{error}$ values are not used) and all intervals representing data values uniquely contain a single value (i.e. when an attribute value is 1 then it is represented as an interval $[1, 1]$).

Definition 4. A regular partial model P is *concrete*, if

- for each relation symbol $r \in \Sigma_R$ which is either a base relation symbol $\mathtt{R}_1, \dots, \mathtt{R}_k$, an object name $\mathtt{'N}_1\mathtt{'}, \dots, \mathtt{'N}_t\mathtt{'}$ or a built-in relation symbol $\mathtt{exists}$, $\mathtt{equals}$, $\mathtt{domain}$, $\mathtt{data}$, $\mathtt{real}$, $\mathtt{int}$, the interpretation $\mathcal{I}_P(r)$ contains $\mathtt{true}$ and $\mathtt{false}$ values only;
- for each object $o \in \mathcal{O}_P$ satisfying $\mathcal{I}_P(\mathtt{real})(o) = \mathtt{true}$, $\mathcal{V}_P(\mathtt{value})(o) = [x, x]$ for some real number $x \in \mathbb{R}$;
- for each object $o \in \mathcal{O}_P$ satisfying $\mathcal{I}_P(\mathtt{int})(o) = \mathtt{true}$, $\mathcal{V}_P(\mathtt{value})(o) = [x, x]$ for some whole number $x \in \mathbb{Z}$.

6.3. Compatibility and inconsistency of partial models

To incorporate the definitions of graph predicates and metrics, we define a respective theory (set of axioms). As such, we can focus only on semantic interpretations of partial models which are compatible with the actual definitions. For example, if a particular metric computes a value as a result, then the interpretation of the data object stored in the partial model should actually retrieve the respective value.

Definition 5. Given a signature $\langle \Sigma, \alpha \rangle$, a *theory* T is a set of axioms

$$
\begin{aligned}
T = \{ &\mathtt{P}_1(v^{(1)}_{\mathtt{P}_1}, \cdots, v^{(\alpha(\mathtt{P}_1))}_{\mathtt{P}_1}) \Leftrightarrow \varphi_{\mathtt{P}_1}, \dots, \\
&\mathtt{P}_\ell(v^{(1)}_{\mathtt{P}_\ell}, \cdots, v^{(\alpha(\mathtt{P}_\ell))}_{\mathtt{P}_\ell}) \Leftrightarrow \varphi_{\mathtt{P}_\ell}, \\
&\mathtt{M}_1(v^{(1)}_{\mathtt{M}_1}, \cdots, v^{(\alpha(\mathtt{M}_1))}_{\mathtt{M}_1}) \equiv \mu_{\mathtt{M}_1}, \dots, \\
&\mathtt{M}_u(v^{(1)}_{\mathtt{M}_u}, \cdots, v^{(\alpha(\mathtt{M}_u))}_{\mathtt{M}_u}) \equiv \mu_{\mathtt{M}_u} \qquad \},
\end{aligned}
$$

where there is a single axiom for each predicate symbol $\mathtt{P}_i$ and each metric symbol $\mathtt{M}_i$. The variables $v^{(j)}_{\mathtt{P}_i}$ and $v^{(j)}_{\mathtt{M}_i}$ correspond to the parameters of the predicate $\mathtt{P}_i$ or the metric $\mathtt{M}_i$, and are free in the predicate definition body $\varphi_{\mathtt{P}_i}$ or metric expression $\mu_{\mathtt{M}_i}$, respectively.

6.3.1. Semantics of predicates and metrics

A defined predicate $\mathtt{P}(v_1, \dots, v_n) \Leftrightarrow \varphi$ can be evaluated on a partial model P along a variable binding $Z\colon \{v_1, \dots, v_n\} \to \mathcal{O}_P$ (denoted as $[\![\varphi]\!]^P_Z$), which can result in four truth values: $\mathtt{true}$, $\mathtt{false}$, $\mathtt{unknown}$ or $\mathtt{error}$. The inductive rules for evaluating the semantics of a logic expression are illustrated in Table 5. We use the notation $Z' = (Z, u \mapsto o)$ to extend the binding Z into a new binding Z: $Z'(v) = o$ if $v = u$, $Z(v)$ otherwise.

A metric $\mathtt{M}(v_1, \dots, v_n) \equiv \mu$ can also be evaluated on a partial model P along variable binding $Z\colon \{v_1, \dots, v_n\} \to \mathcal{O}_P$ (denoted as $(\!|\mu|\!)^P_Z$), resulting in a possibly infinite interval. The empty interval signifies that the evaluation of the metric has failed, e.g. due to division by zero or other undefined algebraic operations. The inductive rules capturing the semantics of a metric expression are illustrated in Table 6.

6.3.2. Compatibility and inconsistency

The proper alignment of the truth values and metric intervals assigned by the semantic interpretation of a partial model and the definition of predicates and metrics (in a theory) is characterized by the concept of compatibility. Informally, a partial model P is compatible with a theory, if the truth values and metric intervals assigned by the definition and the interpretation are identical.

Definition 6. Given a signature $\langle \Sigma, \alpha \rangle$ along with a regular partial model P and a theory T, P *is compatible with* T, if

- for all predicate symbols $\mathtt{P} \in \Sigma_R$ and for all objects $o_1, \dots, o_{\alpha(\mathtt{P})} \in \mathcal{O}_P$, $[\![\varphi_{\mathtt{P}}]\!]^P_Z = \mathcal{I}_P(\mathtt{P})(o_1, \dots, o_{\alpha(\mathtt{P})})$, where $Z = v^{(1)}_{\mathtt{P}} \mapsto o_1, \dots, v^{(\alpha(\mathtt{P}))}_{\mathtt{P}} \mapsto o_{\alpha(\mathtt{P})}$; and

$[\![\mathtt{R}(v_1,\ldots,v_n)]\!]^P_Z$	$:=$	$\mathcal{I}_P(\mathtt{R})(Z(v_1),\ldots,Z(v_n))$
$[\![v \texttt{ == } \texttt{'N'}]\!]^P_Z$	$:=$	$\mathcal{I}_P(\texttt{'N'})(Z(v))$
$[\![\mathtt{R+}(v_1,v_2)]\!]^P_Z$	$:=$	$[\![\mathtt{R}(v_1,v_2)]\!]^P_Z \vee^4 [\![\mathtt{R}(v_1,m_1),\mathtt{R}(m_1,v_2)]\!]^P_Z \vee^4 \cdots \vee^4 [\![\mathtt{R}(v_1,m_1),\ldots,\mathtt{R}(m_n,v_2)]\!]^P_Z$, where $n = \lvert\mathcal{O}_P\rvert$
$[\![\mathtt{R*}(v_1,v_2)]\!]^P_Z$	$:=$	$\mathcal{I}_P(\mathtt{equals})(Z(v_1),Z(v_2)) \vee^4 [\![\mathtt{R+}(v_1,v_2)]\!]^P_Z$
$[\![\mathtt{!}\varphi]\!]^P_Z$	$:=$	$\bigwedge^4_{o_1,\ldots,o_m\in\mathcal{O}_P} \left(\neg^4[\![\mathtt{exists}(u_1)]\!]^P_{u_1\mapsto o_1} \vee^4 \cdots \vee^4 \neg^4[\![\mathtt{exists}(u_m)]\!]^P_{u_m\mapsto o_m} \vee^4 \neg^4[\![\varphi]\!]^P_{Z,u_1\mapsto o_1,\ldots,u_m\mapsto o_m}\right)$, where φ has free variables $u_1,\ldots,u_m$
$[\![\varphi_1,\ldots,\varphi_n]\!]^P_Z$	$:=$	$\bigvee^4_{o_1,\ldots,o_m\in\mathcal{O}_P} \left([\![\mathtt{exists}(u_1)]\!]^P_{u_1\mapsto o_1} \wedge^4 \cdots \wedge^4 [\![\mathtt{exists}(u_m)]\!]^P_{u_m\mapsto o_m} \wedge^4 [\![\varphi_1]\!]^P_{Z,u_1\mapsto o_1,\ldots,u_m\mapsto o_m} \wedge^4 \cdots \wedge^4 [\![\varphi_n]\!]^P_{Z,u_1\mapsto o_1,\ldots,u_m\mapsto o_m}\right)$, where $\varphi_1,\ldots,\varphi_n$ have free variables $u_1,\ldots,u_m$
$[\![\varphi_1\mathtt{;}\ldots\mathtt{;}\varphi_n]\!]^P_Z$	$:=$	$[\![\varphi_1]\!]^P_{Z,u_1\mapsto o_1,\ldots,u_m\mapsto o_m} \vee^4 \cdots \vee^4 [\![\varphi_n]\!]^P_{Z,u_1\mapsto o_1,\ldots,u_m\mapsto o_m}$
$[\![\mathtt{M}(v_1,\ldots,v_n)\ \mathtt{in}\ iv]\!]^P_Z$	$:=$	$\begin{cases} \mathtt{unknown} & \text{if } iv' \cap iv \neq \emptyset \text{ and } iv' \not\subseteq iv, \quad \mathtt{error} \quad \text{if } iv' = \emptyset, \\ \mathtt{false} & \text{if } iv' \neq \emptyset \text{ and } iv' \cap iv = \emptyset, \quad \mathtt{true} \quad \text{if } iv' \neq \emptyset \text{ and } iv' \subseteq iv \end{cases}$ where $iv' = \mathcal{V}_P(\mathtt{M})(Z(v_1),\ldots,Z(v_n))$

Table 5 Inductive semantics of graph predicates.

– for all metric symbols $\mathtt{M} \in \Sigma_M$ and for all objects $o_1,\ldots,o_{\alpha(\mathtt{M})} \in \mathcal{O}_P$, $(\!|\varphi_{\mathtt{M}}|\!)^P_Z = \mathcal{V}_P(\mathtt{M})(o_1,\ldots,o_{\alpha(\mathtt{M})})$, where $Z = v^{(1)}_{\mathtt{M}} \mapsto o_1,\ldots,v^{(\alpha(\mathtt{M}))}_{\mathtt{M}} \mapsto o_{\alpha(\mathtt{P})}$.

Using the 4-valued interpretation, inconsistencies for a domain object in a partial model are recorded in the partial model itself by using the `error` values. Similarly, inconsistencies for a data object are identified when the interval of possible values for the data object is the empty set. Note that an inconsistent model cannot be concrete.

Definition 7. A regular partial model P is *inconsistent* if

– for some relation symbol $r \in \Sigma_R$ which is either a base relation symbol $\mathtt{R}_1,\ldots,\mathtt{R}_k$, an object name $\texttt{'N}_1\texttt{'},\ldots,\texttt{'N}_t\texttt{'}$ or a built-in relation symbol `exists`, `equals`, `domain`, `data`, `real` or `int`, the interpretation $\mathcal{I}_P(r)$ contains at least one `error` value; or
– there is an object $o \in \mathcal{O}_P$ with $\mathcal{I}_P(\mathtt{data})(o) \sqsupseteq \mathtt{true}$ and $\mathcal{V}_P(\mathtt{value})(o) = \emptyset$.

6.4. Model generation and optimization by refinement and concretization

The generation of consistent (concrete) models is driven by a series of refinement and concretization steps where the level of uncertainty in partial models is gradually reduced. When all uncertainties are resolved and there is still no inconsistency in the model then a concrete model is obtained.

Definition 8. A refinement from P to Q (denoted as $P \sqsubseteq Q$) is defined by a refinement function between the objects of the partial model $ref : \mathcal{O}_P \to 2^{\mathcal{O}_Q}$ which respect the information ordering of 4-valued logic values and intervals:

– For each relation symbol $r \in \Sigma_R$, for each object $p_1,\ldots,p_{\alpha(r)} \in \mathcal{O}_P$ and for each corresponding object $q_1 \in ref(p_1),\ldots,q_{\alpha(r)} \in ref(p_{\alpha(r)})$,

$$\mathcal{I}_P(r)(p_1,\ldots,p_{\alpha(r)}) \sqsubseteq \mathcal{I}_Q(r)(q_1,\ldots,q_{\alpha(r)}).$$

– For each metric symbol $m \in \Sigma_M$, for each object $p_1,\ldots,p_{\alpha(m)} \in \mathcal{O}_P$ and for each corresponding object $q_1 \in ref(p_1),\ldots,q_{\alpha(m)} \in ref(p_{\alpha(m)})$,

$$\mathcal{V}_P(m)(p_1,\ldots,p_n) \supseteq \mathcal{V}_Q(m)(q_1,\ldots,q_n).$$

– All objects in Q are refined from an object in P:

$$\mathcal{O}_Q = \bigcup_{p\in\mathcal{O}_P} ref(p).$$

– Existing objects $p \in \mathcal{O}_P$ cannot disappear, i.e. they must have non-empty refinements:

$$[\mathcal{I}_P(\mathtt{exists})(p) \sqsupseteq \mathtt{true}] \Rightarrow [ref(p) \neq \emptyset].$$

Next, we formally define the task of (consistent) model generation along partial models. Given a model generation task, a *complete model generator* outputs some model $Q \in$ *solutions*(P,T) if *solutions*(P,T) is non-empty. Otherwise, it provides a proof of the unsatisfiability of the task.

Definition 9. A *model generation task* consists of a signature $\langle\Sigma,\alpha\rangle$ along with a regular partial model P and a theory T. The

$\llparenthesis x \rrparenthesis^P_Z := [x, x]$, where $x \in \mathbb{R}$	$\llparenthesis \mathtt{M}(v_1, \ldots, v_n) \rrparenthesis^P_Z := \mathcal{V}(\mathtt{M})(Z(v_1), \ldots, Z(v_n))$
$\llparenthesis v \rrparenthesis^P_Z := \mathcal{V}(\mathtt{value})(Z(v))$	$\llparenthesis \langle \mathit{unary\text{-}op} \rangle \mu \rrparenthesis^P_Z := \langle \mathit{unary\text{-}op} \rangle^\sharp \llparenthesis \mu \rrparenthesis^P_Z$ (e.g. +, -)
$\llparenthesis \mu \ \mathtt{as\ int} \rrparenthesis^P_Z := \begin{cases} [\lfloor a \rfloor, \lfloor b \rfloor] & \text{if } \llparenthesis \mu \rrparenthesis^P_Z = [a, b], \\ \emptyset & \text{otherwise} \end{cases}$	$\llparenthesis \mu_1 \langle \mathit{binary\text{-}op} \rangle \mu_2 \rrparenthesis^P_Z := \llparenthesis \mu_1 \rrparenthesis^P_Z \langle \mathit{binary\text{-}op} \rangle^\sharp \llparenthesis \mu_2 \rrparenthesis^P_Z$ (e.g. +, -, *, /, ^)

$$\llparenthesis \mathtt{if}\ \varphi\ \mathtt{then}\ \mu_1\ \mathtt{else}\ \mu_2 \rrparenthesis^P_Z := \begin{cases} \llparenthesis \mu_1 \rrparenthesis^P_Z & \text{if } [\![\varphi]\!]^P_Z = \mathtt{true}, & \llparenthesis \mu_2 \rrparenthesis^P_Z & \text{if } [\![\varphi]\!]^P_Z = \mathtt{false}, \\ \llparenthesis \mu_1 \rrparenthesis^P_Z \sqcup \llparenthesis \mu_2 \rrparenthesis^P_Z & \text{if } [\![\varphi]\!]^P_Z = \mathtt{unknown}, & \emptyset & \text{if } [\![\varphi]\!]^P_Z = \mathtt{error} \end{cases}$$

$$\llparenthesis \varphi_1 \ \mathtt{->}\ \mu_1 \mathtt{;} \ldots \mathtt{;} \varphi_n \ \mathtt{->}\ \mu_n \rrparenthesis^P_Z := \begin{cases} \emptyset & \text{if } \mathit{branches}(\mathtt{error}) \neq \emptyset \text{ or } |\mathit{branches}(\mathtt{true})| \geq 2, \\ \llparenthesis \mu_i \rrparenthesis^P_Z & \text{if } \mathit{branches}(\mathtt{error}) = \emptyset \text{ and } \mathit{branches}(\mathtt{true}) = \{i\}, \\ \bigsqcup_{i \in \mathit{branches}(\mathtt{unknown})} \llparenthesis \mu_i \rrparenthesis^P_Z & \text{otherwise,} \end{cases}$$

where $\mathit{branches}(X) = \{i = 1, \ldots, n \mid [\![\varphi_i]\!]^P_Z = X\}$

For aggregation operators $\mathtt{sum}$, $\mathtt{min}$, $\mathtt{max}$ and $\mathtt{single}$ $\llparenthesis \langle \mathit{aggregation\text{-}op} \rangle \mathtt{\{}\ \mu\ \mathtt{|}\ \varphi\ \mathtt{\}} \rrparenthesis^P_Z$, we define the auxiliary functions

$$\mathit{matches}_\varphi(X) = \{Z' = (Z, u_1 \mapsto o_1, \ldots, u_m \mapsto o_m) \mid \mathcal{I}_P(\mathtt{exists})(o_1) \wedge^4 \cdots \wedge^4 \mathcal{I}_P(\mathtt{exists})(o_1) \wedge^4 [\![\varphi]\!]^P_{Z'} = X\},$$

$$\mathit{matchCount}_\varphi(Z') = \begin{cases} \mathit{count}(Z'(o_1)) \cdot^\sharp \ldots \cdot^\sharp \mathit{count}(Z'(o_m)) & \text{if } Z' \in \mathit{matches}_\varphi(\mathtt{true}), \\ \mathit{count}(Z'(o_1)) \cdot^\sharp \ldots \cdot^\sharp \mathit{count}(Z'(o_m)) \sqcup [0, 0] & \text{otherwise} \end{cases}$$

$$\mathit{count}(o) = \begin{cases} [0, 1] & \text{if } \mathcal{I}_P(\mathtt{exists})(o) = \mathtt{unknown} \text{ and } \mathcal{I}_P(\mathtt{equals})(o, o) = \mathtt{true}, \\ [1, 1] & \text{if } \mathcal{I}_P(\mathtt{exists})(o) = \mathtt{true} \text{ and } \mathcal{I}_P(\mathtt{equals})(o, o) = \mathtt{true}, \\ [0, +\infty] & \text{if } \mathcal{I}_P(\mathtt{exists})(o) = \mathtt{unknown} \text{ and } \mathcal{I}_P(\mathtt{equals})(o, o) = \mathtt{unknown}, \\ [1, +\infty] & \text{if } \mathcal{I}_P(\mathtt{exists})(o) = \mathtt{true} \text{ and } \mathcal{I}_P(\mathtt{equals})(o, o) = \mathtt{unknown}, \\ \emptyset & \text{if } \mathcal{I}_P(\mathtt{exists})(o) = \mathtt{error} \text{ or } \mathcal{I}_P(\mathtt{equals})(o, o) = \mathtt{error}, \end{cases}$$

where φ has free variables $u_1, \ldots, u_m$.

$$\llparenthesis \mathtt{sum}\ \mathtt{\{}\ \mu\ \mathtt{|}\ \varphi\ \mathtt{\}} \rrparenthesis^P_Z := \begin{cases} \emptyset & \text{if } \mathit{matches}_\varphi(\mathtt{error}) \neq \emptyset\,, \\ \sum^\sharp_{Z' \in \mathit{matches}_\varphi(\mathtt{unknown}) \cup \mathit{matches}_\varphi(\mathtt{true})} \mathit{matchCount}_\varphi(Z') \triangleright \llparenthesis \mu \rrparenthesis^P_{Z'} & \text{otherwise} \end{cases}$$

$$\llparenthesis \mathtt{min}\ \mathtt{\{}\ \mu\ \mathtt{|}\ \varphi\ \mathtt{\}} \rrparenthesis^P_Z := \begin{cases} \emptyset & \text{if } \mathit{matches}_\varphi(\mathtt{error}) \neq \emptyset, \\ \min^\sharp \{iv_1, iv_2\} & \text{otherwise} \end{cases}$$

where $iv_1 = \min^\sharp_{Z' \in \mathit{matches}_\varphi(\mathtt{unknown}), \llparenthesis \mu \rrparenthesis^P_{Z'} \neq \emptyset} \llparenthesis \mu \rrparenthesis^P_{Z'}$ and $iv_2 = \min^\sharp_{Z' \in \mathit{matches}_\varphi(\mathtt{true})} \llparenthesis \mu \rrparenthesis^P_{Z'}$

$$\llparenthesis \mathtt{max}\ \mathtt{\{}\ \mu\ \mathtt{|}\ \varphi\ \mathtt{\}} \rrparenthesis^P_Z := \begin{cases} \emptyset & \text{if } \mathit{matches}_\varphi(\mathtt{error}) \neq \emptyset, \\ \max^\sharp \{iv_1, iv_2\} & \text{otherwise} \end{cases}$$

where $iv_1 = \max^\sharp_{Z' \in \mathit{matches}_\varphi(\mathtt{unknown}), \llparenthesis \mu \rrparenthesis^P_{Z'} \neq \emptyset} \llparenthesis \mu \rrparenthesis^P_{Z'}$ and $iv_2 = \max^\sharp_{Z' \in \mathit{matches}_\varphi(\mathtt{true})} \llparenthesis \mu \rrparenthesis^P_{Z'}$

$$\llparenthesis \mathtt{single}\ \mathtt{\{}\ \mu\ \mathtt{|}\ \varphi\ \mathtt{\}} \rrparenthesis^P_Z := \begin{cases} \emptyset & \text{if } \mathit{matches}_\varphi(\mathtt{error}) \neq \emptyset \text{ or } |\mathit{matches}_\varphi(\mathtt{true})| \geq 2, \\ \llparenthesis \mu \rrparenthesis^P_{Z'} & \text{if } \mathit{matches}_\varphi(\mathtt{error}) = \emptyset \text{ and } |\mathit{matches}_\varphi(\mathtt{true})| = \{Z'\}, \\ \bigsqcup_{Z' \in \mathit{matches}_\varphi(\mathtt{unknown})} \llparenthesis \mu \rrparenthesis^P_{Z'} & \text{otherwise} \end{cases}$$

Table 6 Inductive semantics of metric expressions.

solutions of the model generation task are

$$solutions(P, T) = \{Q \mid P \sqsubseteq Q \text{ and } Q \text{ is concrete and it is compatible with } T\}.$$

As a corollary, if a partial model P is inconsistent, then its refinements Q ($P \sqsubseteq Q$) cannot be concrete. Hence, when searching for a concrete, consistent refinement of P, model generators can abandon inconsistent partial models Q without compromising the completeness of model generation.

An *optimizing model generator* aims to find a consistent model $Q \in optimal(P, T, \mathtt{M})$ optimal with respect to a given objective $\mathtt{M}$ if such a model exists. As such, conceptually, a model generator can also handle model optimization challenges with appropriate modifications of the search strategy (e.g. using a branch-and-bound strategy).

Definition 10. A *single-objective model optimization task* is a model generation task P, T over a signature $\langle \Sigma, \alpha \rangle$ along with a 0-ary metric symbol $\mathtt{M} \in \Sigma$ ($\alpha(\mathtt{M}) = 0$) serving as the *objective* to be maximized. The *optimal solutions* of the task are

$$optimal(P, T, \mathtt{M}) = \{Q \mid Q \in solutions(P, T) \text{ and } \forall Q' \in solutions(P, T)\colon \mathcal{V}_Q(\mathtt{M})() \geq \mathcal{V}_{Q'}(\mathtt{M})()\}.$$

Note that the definition of a model generation task complies with (but generalizes) the previous formalization of the model generation problem in Varró et al. (2018). As such, existing model generation techniques and tools (Semeráth et al. 2018, 2020) remain applicable in our context. On the other hand, these existing approaches do not support an optimizing model generator, which is to be targeted in future work.

7. Related work

Several software engineering and verification approaches are based on the automated generation or extension of graph-based models. In the following, we are summarizing those techniques and their relation with our partial modeling language.

7.1. Model generation for domain-specific languages

Several approaches map a model modeling artifacts into a logic problem solved by an underlying backend solver. There are several approaches aiming to validate standardized engineering models enriched with OCL constraints (Gogolla et al. 2005) by relying upon different back-end logic-based approaches such as constraint logic programming (Cabot et al. 2007, 2008; Büttner & Cabot 2012), SAT-based model finders (Shah et al. 2009; Anastasakis et al. 2010; Büttner et al. 2012; Kuhlmann et al. 2011; Soeken et al. 2010; Semeráth et al. 2017, 2016), CSP solvers (González et al. 2012) first-order logic (Beckert et al. 2002), constructive query containment (Queralt et al. 2012), higher-order logic (Brucker & Wolff 2007; Grönniger et al. 2009), rewriting logic (Clavel & Egea 2008), or genetic algorithms (Soltana et al. 2017, 2020).

Partial snapshots, metamodels and a large fragment of constraints, defined either as OCL constraints ("Object Constraint Language, v2.4" 2014) or VIATRA queries (Varró et al. 2016), can be uniformly represented as first order constraints (Semeráth et al. 2017). As a main conceptual difference, this approach operates directly on a mathematically precise formalism, but, for the sake of reusability, it avoids direct integration with existing modeling tools.

7.2. Partial modeling

Partial models are similar to uncertain models, which offer a rich specification language (Famelis et al. 2012; Salay & Chechik 2015) amenable to analysis. They provide a more intuitive, user-friendly language compared to our mathematical notation, but without handling additional predicates and metrics. With respect to expression power, Semeráth & Varró (2017) presented a rewriting techniques that maps uncertain models to (3-valued) partial models. Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer (Salay et al. 2012), or refined by graph transformation rules (Salay et al. 2015).

In some extended approaches, like Famelis et al. (2013), it is possible to analyze predicates and executions of model transformation rules on partial models by using a SAT solver (MathSAT4) or by automated graph approximation (referred to as "lifting"), or by graph query engines (Semeráth & Varró 2017). As a key difference, our approach carries out model refinement while simultaneously evaluating predicates and metrics.

7.3. Feature modeling

Feature modeling is an engineering formalism to define the possible combination of selected features from a large selection of atomic features (Lee et al. 2002; Griss et al. 1998). While configurations (i.e. a valid combination of features) can be represented as a graph model (where each object is named, and only edges and the `exists` relation have `unknown` values), not all graph generation problems can be represented as a feature modeling problem.

Clafer (Bak et al. 2010) is a lightweight structural modeling language used for feature modeling with minimalistic syntax and rich semantics equivalent to first-order relational logic. The specification language supports structural modeling, constraints (well-formedness constraints are written in their own language, which is said to be equivalent to FOL) and also partial configurations. Partial configurations are like partial snapshots in our approach: instance models with undefined attributes and features that can be the basis of model completion. DSL specification given in Clafer are validated using the Clafer Tools (Antkiewicz et al. 2013) that supports various tasks for domain engineering, like consistency checking and instance model generation based on backend reasoners like Alloy or Choco (Liang 2012).

7.4. Symbolic approaches

Certain techniques use abstract (or symbolic) graphs for analysis purposes. A tableau-based reasoning method is proposed for graph properties (Schneider et al. 2017; Pennemann 2008; Al-Sibahi et al. 2016), which automatically refines solutions based on WF constraints, and handles the state space in the form of a resolution tree as opposed to a partial model. When scalability evaluation is included, these techniques demonstrated to derive only small graphs (< 10 objects).

7.5. Shape analysis

Another group of symbolic approaches introduces sophisticated type graphs to uniformly represent objects with similar properties. Reps et al. (2004); Ferrara et al. (2012); Gopan et al. (2004) introduce predicate abstraction techniques for graphs using 3-valued logic, which is used as a theoretical basis for our model generation technique. In those approaches, concretization is used to materialize (typically small) counter-examples for designated safety properties in a graph transformation system. However, their focus is to support model checking of abstract graph transformation systems, which can evaluate complex trajectories, but do not scale in the size of the models.

A similar neighborhood-based abstract interpretation technique is introduced by Rensink & Distefano (2006), where each abstract object is represents objects with similar neighborhood (up to a given range).

Handling numeric (integer or real) variables and constraints in model generation scenarios requires their abstract interpretation through numerical abstract domains (Miné 2004; Singh et al. 2018). Numerical abstract domains may be used to summarize object attributes in value analysis of heap programs (Magill et al. 2007; McCloskey et al. 2010; Ferrara et al. 2012). Summarized dimensions (Gopan et al. 2004) were introduced to succinctly represent attributes of a potentially unbounded set of objects via multi-objects. This approach enables attribute handling in three-valued partial models, and allows checking for refinements by abstract subsumption (Anand et al. 2009). But these approaches do not generate graph models.

7.6. Specification frameworks

Complete frameworks with standalone specification languages include Formula (E. K. Jackson et al. 2011), which uses the Z3 SMT-solver (de Moura & Bjørner 2008), Alloy (D. Jackson 2002), which relies on a similar relational logic (Torlak & Jackson 2007) and SAT-solvers like Sat4j (Le Berre & Parrain 2010), and Clafer (Bak et al. 2013), which uses reasoners like Alloy.

As a main difference, our specification language is capable of uniformly representing various model generation and optimization tasks, any initial or intermediate state of model generation, and the concrete models generated as result. As such, it is especially suitable for support iterative model generation, where the output of one run can be used as the input of the next run.

8. Conclusions and future work

In this paper, we have proposed a novel specification language for partial models to be used for consistent graph model generation. While a novel class of model generators derive consistent instance models along a refinement calculus of partial models, we intend to uniformly represent any intermediate state of the generation (not only the initial and final states). Key features of our specification language have been presented along a series of examples in the context of a complex case study proposed by NASA researchers. As a major novelty, our language seamlessly integrates partial models with graph predicates and graph metrics, which can mutually depend on each other. We define precise semantics for our language using a 4-valued interpretation based on Belnap–Dunn logic. On a practical note, a parser and textual editor has been implemented using the Xtext technology.

The proposed language can serve as a user-friendly front-end for model generation tasks used in a series of past papers (Semeráth et al. 2018, 2020) as well as in other modeling challenges. However, no support exists currently for model optimization tasks, which provides the main direction of our future work.

References

Al-Sibahi, A. S., Dimovski, A. S., & Wasowski, A. (2016). Symbolic execution of high-level transformations. In *Sle 2016* (p. 207-220). Springer.

Anand, S., Păsăreanu, C. S., & Visser, W. (2009). Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.*, *11*(1), 53-67.

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2010). On challenges of model transformation from UML to Alloy. *Softw. Syst. Model.*, *9*(1), 69-86.

Antkiewicz, M., Bak, K., Murashkin, A., Olaechea, R., Liang, J., & Czarnecki, K. (2013). Clafer tools for product line engineering. In *Splc.* Tokyo, Japan.

Bak, K., Czarnecki, K., & Wasowski, A. (2010, 10/2010). Feature and meta-models in clafer: Mixed, specialized, and coupled. In *3rd international conference on software language engineering.* Eindhoven, The Netherlands. doi: 10.1007/978-3-642-19440-5_7

Bak, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., & Wasowski, A. (2013). Clafer: unifying class and feature modeling. *Softw. Syst. Model.*, 1-35.

Beckert, B., Keller, U., & Schmitt, P. H. (2002). Translating the Object Constraint Language into First-order Predicate Logic. In *Proc. verify, workshop at floc.*

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

Brucker, A. D., & Wolff, B. (2007). *The HOL-OCL tool.* (http://www.brucker.ch/)

Büttner, F., & Cabot, J. (2012). Lightweight string reasoning for OCL. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, & D. S. Kolovos (Eds.), *Ecmfa 2012* (Vol. 7349, p. 244-258). Springer.

Büttner, F., Egea, M., Cabot, J., & Gogolla, M. (2012). Verification of ATL transformations using transformation models and model finders. In *Icfem* (p. 198-213). Springer.

Cabot, J., Clarisó, R., & Riera, D. (2007). UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Ase 2017* (p. 547-548). ACM.

Cabot, J., Clariso, R., & Riera, D. (2008, April). Verification of UML/OCL class diagrams using constraint programming. In *Software testing verification and validation workshop, 2008. icstw '08. ieee international conf. on* (p. 73-80).

Chechik, M., Nejati, S., & Sabetzadeh, M. (2011). A relationship-based approach to model integration. *Innova-*

tions in Systems and Software Engineering, *8*(1), 3-18. doi: 10.1007/s11334-011-0155-2
Clavel, M., & Egea, M. (2008). *The ITP/OCL tool.* (http://maude.sip.ucm.es/itp/ocl/)
de Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and algorithms for the construction and analysis of systems, 14th international conference (tacas 2008)* (Vol. 4963, p. 337-340). Springer.
Eclipse Modeling Framework [Computer software manual]. (2019). (http://www.eclipse.org/emf)
Famelis, M., Salay, R., & Chechik, M. (2012). Partial models: Towards modeling and reasoning with uncertainty. In *ICSE* (pp. 573–583). IEEE Computer Society.
Famelis, M., Salay, R., Di Sandro, A., & Chechik, M. (2013). Transformation of models containing uncertainty. In *International conference on model driven engineering languages and systems* (pp. 673–689).
Ferrara, P., Fuchs, R., & Juhasz, U. (2012). TVAL+: TVLA and value analyses together. In *Sefm 2012* (Vol. 7504, p. 63-77). Springer.
Friedenthal, S., Moore, A., & Steiner, R. (2008). *A practical guide to sysml: Systems modeling language*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
Gogolla, M., Bohling, J., & Richters, M. (2005). Validating UML and OCL models in USE by automatic snapshot generation. *Softw. Syst. Model.*, *4*, 386-398.
González, C. A., Büttner, F., Clarisó, R., & Cabot, J. (2012). EMFtoCSP: a tool for the lightweight verification of EMF models. In *Formsera 2012* (p. 44-50).
Gopan, D., DiMaio, F., Dor, N., Reps, T., & Sagiv, M. (2004). Numeric domains with summarized dimensions. In *Tacas 2004* (Vol. 2988, p. 512-529). Springer.
Griss, M. L., Favaro, J., & d'Alessandro, M. (1998). Integrating feature modeling with the rseb. In *Proceedings. fifth international conference on software reuse (cat. no. 98tb100203)* (pp. 76–85).
Grönniger, H., Ringert, J. O., & Rumpe, B. (2009). System model-based definition of modeling language semantics. In *Forte* (Vol. 5522, p. 152-166). Springer.
Herzig, S. J. I., Mandutianu, S., Kim, H., Hernandez, S., & Imken, T. (2017). Model-transformation-based computational design synthesis for mission architecture optimization. In *Ieee aerospace conference.* IEEE.
Jackson, D. (2002). Alloy: a lightweight object modelling notation. *Trans. Softw. Eng. Methodol.*, *11*(2), 256–290.
Jackson, E. K., Levendovszky, T., & Balasubramanian, D. (2011). Reasoning about metamodeling with formal specifications and automatic proofs. In *Model driven engineering languages and systems* (pp. 653–667). Springer.
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
Kleene, S. C., De Bruijn, N., de Groot, J., & Zaanen, A. C. (1952). *Introduction to metamathematics* (Vol. 483). van Nostrand New York.
Kuhlmann, M., Hamann, L., & Gogolla, M. (2011). Extensive validation of OCL models by integrating SAT solving into USE. In *Tools '11* (Vol. 6705, p. 290-306).
Kulisch, U. W. (2009). Complete interval arithmetic and its implementation of the computer. In *Numerical validation in current hardware architectures* (Vol. 5492, p. 7-26). Springer. doi: 10.1007/978-3-642-01591-5_2
Le Berre, D., & Parrain, A. (2010). The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*(2-3), 59–64.
Lee, K., Kang, K. C., & Lee, J. (2002). Concepts and guidelines of feature modeling for product line software engineering. In *International conference on software reuse* (pp. 62–77).
Liang, J. (2012, 12/2012). Solving clafer models with choco. (GSDLab-TR 2012-12-30).
Magill, S., Berdine, J., Clarke, E., & Cook, B. (2007). Arithmetic strengthening for shape analysis. In *Sas 2007* (Vol. 4634, p. 419-436). Springer.
Marussy, K., Semeráth, O., & Varró, D. (2018). Incremental view model synchronization using partial models. In *Models 2018* (p. 323-333).
McCloskey, B., Reps, T., & Sagiv, M. (2010). Statically inferring complex heap, array, and numeric invariants. In *Sas 2010* (Vol. 6337, p. 71-99). Springer.
McKubre-Jordens, M., & Weber, Z. (2012). Real analysis in paraconsistent logic. *J. Phil. Logic*, *41*(5), 901-922. doi: 10.1007/s10992-011-9210-6
Miné, A. (2004). *Weakly relational numerical abstract domains* (Unpublished doctoral dissertation).
Object Constraint Language, v2.4 [Computer software manual]. (2014, February).
Pennemann, K.-H. (2008). Resolution-like theorem proving for high-level conditions. In *Icgt 2008* (Vol. 5214, p. 289-304). Springer.
Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012). OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. *Data Knowl. Eng.*, *73*, 1-22.
Rensink, A., & Distefano, D. (2006). Abstract graph transformation. *Electronic Notes in Theoretical Computer Science*, *157*(1), 39–59.
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).
Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified modeling language reference manual, the (2nd edition)*. Pearson Higher Education.
Salay, R., & Chechik, M. (2015). A generalized formal framework for partial modeling. In A. Egyed & I. Schaefer (Eds.), *Fundamental approaches to software engineering* (Vol. 9033, p. 133-148). Springer Berlin Heidelberg.
Salay, R., Chechik, M., Famelis, M., & Gorzny, J. (2015). A methodology for verifying refinements of partial models. *Journal of Object Technology*, *14*(3), 3:1–31.
Salay, R., Famelis, M., & Chechik, M. (2012). Language independent refinement using partial modeling. In *FASE* (pp. 224–239). Springer.
Schneider, S., Lambers, L., & Orejas, F. (2017). Symbolic model generation for graph properties. In *Fase 2017* (Vol.

10202, p. 226-243). Springer.
Semeráth, O., Barta, Á., Horváth, Á., Szatmári, Z., & Varró, D. (2017). Formal validation of domain-specific languages with derived features and well-formedness constraints. *Softw. Syst. Model*, *16*(2), 357-392.
Semeráth, O., Farkas, R., Bergmann, G., & Varró, D. (2020). Diversity of graph models and graph generators in mutation testing. *Int. J. Softw. Tools Technol. Transf.*, *22*(1), 57–78.
Semeráth, O., Nagy, A. S., & Varró, D. (2018). A graph solver for the automated generation of consistent domain-specific models. In *ICSE* (pp. 969–980). ACM.
Semeráth, O., & Varró, D. (2017). Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In *ICMT* (pp. 138–154).
Semeráth, O., Vörös, A., & Varró, D. (2016). Iterative and incremental model generation by logic solvers. In *FASE* (pp. 87–103). Springer.
Shah, S. M. A., Anastasakis, K., & Bordbar, B. (2009). From UML to Alloy and back again. In *Modevva '09: Proceedings of the 6th international workshop on model-driven engineering, verification and validation* (pp. 1–10). ACM.
Singh, G., Püschel, M., & Vechev, M. (2018). A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.*, *2*(POPL). (Article no. 2)
Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., & Drechsler, R. (2010). Verifying UML/OCL models using boolean satisfiability. In *Date'10* (p. 1341-1344). IEEE.
Soltana, G., Sabetzadeh, M., & Briand, L. C. (2017). Synthetic data generation for statistical testing. In *ASE* (pp. 872–882).
Soltana, G., Sabetzadeh, M., & Briand, L. C. (2020, April). Practical constraint solving for generating system test data. *ACM Trans. Softw. Eng. Methodol.*, *29*(2).
Torlak, E., & Jackson, D. (2007). Kodkod: A relational model finder. In *Tacas* (p. 632-647). Springer.
Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., & Ujhelyi, Z. (2016). Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling*, *15*(3), 609–629.
Varró, D., Semeráth, O., Szárnyas, G., & Horváth, Á. (2018). Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Graph transformation, specifications, and nets - in memory of hartmut ehrig* (Vol. 10800, pp. 285–312). Springer.
Xcore [Computer software manual]. (2020). (https://wiki.eclipse.org/Xcore)

About the authors

Kristóf Marussy is a PhD student at the Department of Measurement and Information Systems at Budapest University of Technology and Economics. He is also a research assistant at the MTA Lendület Cyber-Physical Systems Research Group. His research interest include the modeling and analysis of extra-functional properties of cyber-physical systems, and the synthesis of reliable architectures. He participated in research visits at the University of L'Aquila and McGill University. You can contact the author at marussy@mit.bme.hu or visit https://inf.mit.bme.hu/en/members/marussyk.

Oszkár Semeráth is a research fellow at Budapest University of Technology and Economics and MTA Lendület Cyber-Physical Systems Research Group. His research focuses on modeling tools, logic solvers and graph generation, he is the main developer of the VIATRA Solver graph generator framework. His results were published in a book chapter, 4 journal papers with impact factor, in 17 conference papers, and won IEEE/ACM best paper award at the MODELS 2013 conference. You can contact the author at semerath@mit.bme.hu or visit https://inf.mit.bme.hu/en/members/semeratho.

Aren A. Babikian is a PhD student at the McGill University. His research focuses on using model generation techniques for the safety assurance of cyber-physical system and their design tools. He recently published a related paper as first author at the FASE 2020 conference. You can contact the author at aren.babikian@mcgill.ca or visit http://arenbabikian.github.io.

Dániel Varró is a full professor of software engineering at McGill University and at Budapest University of Technology and Economics. He is also a research chair of the MTA Lendület Cyber-Physical Systems Research Group. He is a co-author of more than 170 scientific papers with seven Distinguished Paper Awards, and two Most Influential Paper Award. He serves on the editorial board of the Software and Systems Modeling (Springer) and Journal of Object Technology journals. He has been a program committee co-chair of FASE 2013, ICMT 2014, SLE 2016 and MODELS 2021 conferences. He is a co-founder of the VIATRA model query and transformation framework, and IncQuery Labs Ltd., a technology-intensive Hungarian company. You can contact the author at daniel.varro@mcgill.ca.

A. Grammar of the configuration language

Below we provide a grammar for our proposed configuration language in a form that suitable for illustrative purposes, but is not context-free. For a slightly more complex form that is suitable for implementing a parser, we refer to our implementation available at https://github.com/viatra/VIATRA-Generator.

```
⟨id-fragment⟩ ::= (a ··· z | A ··· Z | _)
                  (a ··· z | A ··· Z | _ | 0 ··· 9)∗

⟨id⟩ ::= ⟨id-fragment⟩ (::⟨id-fragment⟩)∗

⟨integer⟩ ::= (-)? (0 ··· 9)+

⟨real⟩ ::= (-)? (0 ··· 9)+ (.(0 ··· 9)+)?

⟨object-id⟩ ::= ⟨named-obj-id⟩ | ⟨unnamed-obj-id⟩

⟨char⟩ ::= any Unicode character except '

⟨named-obj-id⟩ ::= '⟨char⟩+'

⟨unnamed-obj-id⟩ ::= ⟨id⟩

⟨relation-id⟩ ::= ⟨builtin-rel-id⟩          ▷ built-in relation
                | ⟨base-rel-id⟩             ▷ base relation
                | ⟨predicate-id⟩            ▷ defined predicate

⟨builtin-rel-id⟩ ::= exists | equals
                   | domain | data | int | real

⟨base-rel-id⟩ ::= ⟨id⟩

⟨predicate-id⟩ ::= ⟨id⟩

⟨metric-id⟩ ::= ⟨id⟩

⟨assertion⟩ ::= ⟨ground-rel-use⟩: ⟨logic-value⟩.
                    ▷ relation assertion
              | ⟨ground-term⟩: ⟨interval⟩.
                    ▷ value assertion
              | (? | !)? ⟨ground-rel-use⟩.
                    ▷ brief relation assertion
              | ⟨ground-metr-use⟩: ⟨interval⟩.
                    ▷ metric assertion

⟨ground-rel-use⟩ ::=
    ⟨relation-id⟩((⟨ground-term⟩(, ⟨ground-term⟩)∗)?)
                    ▷ relation atom with ground arguments

⟨ground-term⟩ ::= ⟨object-id⟩               ▷ partial model node
                | ⟨int⟩ | ⟨real⟩ | ⟨interval⟩
                    ▷ integer, real or interval literal
                | *                         ▷ all partial model nodes

⟨logic-value⟩ ::= true | false | unknown | error
                    ▷ 4-valued logic values

⟨interval⟩ ::= [⟨lower-bound⟩,⟨upper-bound⟩]
                    ▷ non-empty interval literal
             | empty                        ▷ empty interval literal

⟨lower-bound⟩ ::= ⟨real⟩ | -inf

⟨upper-bound⟩ ::= ⟨real⟩ | +inf

⟨default-assertion⟩ ::= default ⟨assertion⟩

⟨predicate-def⟩ ::= ⟨pred-def-core⟩
                    ▷ basic form of predicate definition without any modifiers
                  | error ⟨pred-def-core⟩
                    ▷ error predicate definition
                  | functional ⟨pred-def-core⟩
                    ▷ functional predicate definition

⟨pred-def-core⟩ ::=
    ⟨predicate-id⟩((⟨param⟩(, ⟨param⟩)∗)?) :- ⟨disjunction⟩.

⟨param⟩ ::= ⟨var-id⟩                        ▷ parameter variable
          | ⟨relation-id⟩ ⟨var-id⟩          ▷ typed parameter

⟨disjunction⟩ ::= ⟨conjunction⟩(; ⟨conjunction⟩)∗

⟨conjunction⟩ ::= ⟨literal⟩(, ⟨literal⟩)∗

⟨literal⟩ ::= ⟨atom⟩                        ▷ positive atom
            | !⟨atom⟩                       ▷ negative atom

⟨atom⟩ ::= ⟨logic-value⟩                    ▷ 4-valued logic constant
         | ⟨relation-use⟩                   ▷ relation application
         | ⟨relation-id⟩+(⟨term⟩, ⟨term⟩)
                    ▷ transitive closure
         | ⟨relation-id⟩*(⟨term⟩, ⟨term⟩)
                    ▷ reflexive transitive closure
         | ⟨metric-use⟩ in ⟨interval⟩
                    ▷ metric check
         | ⟨term⟩ == ⟨named-object-id⟩
                    ▷ find by name
         | ⟨metric-use⟩ ⟨comp-op⟩ ⟨real⟩
                    ▷ comparison of metric value

⟨relation-use⟩ ::= ⟨relation-id⟩((⟨term⟩(, ⟨term⟩)∗)?)

⟨metric-use⟩ ::= ⟨metric-id⟩((⟨term⟩(, ⟨term⟩)∗)?)

⟨unnamed-err⟩ ::=
    error((⟨param⟩(, ⟨param⟩)∗)?) :- ⟨disjunction⟩.
                    ▷ unnamed error predicate definition

⟨functional-decl⟩ ::= functional ⟨base-relation-id⟩.
                    ▷ functional base relation declaration

⟨comp-op⟩ ::= =:= | =!= | <= | < | >= | >

⟨metric-def⟩ ::=
    ⟨metric-id⟩((⟨param⟩(, ⟨param⟩)∗)?) := ⟨metric-expr⟩.
                    ▷ metric definition

⟨metric-expr⟩ ::= ⟨real⟩                    ▷ number literal
                | ⟨term⟩                    ▷ value of data node
                | ⟨metric-use⟩              ▷ metric application
                | ⟨unary-op⟩ ⟨metric-expr⟩
                    ▷ unary operation
```

| ⟨*metric-expr*⟩ ⟨*binary-op*⟩ ⟨*metric-expr*⟩
▷ binary operation

| ⟨*metric-expr*⟩ `as int`
▷ cast (floor function)

| `(`⟨*metric-expr*⟩`)`
▷ parenthesized expression

| `if` ⟨*relation-use*⟩ `then` ⟨*metric-expr*⟩ `else` ⟨*metric-expr*⟩
▷ conditonal expression

| ⟨*relation-use*⟩ `->` ⟨*metric-expr*⟩ (`;` ⟨*relation-use*⟩ `->` ⟨*metric-expr*⟩)*
▷ switch expression

| ⟨*aggr-op*⟩`{`⟨*metric-use*⟩`|`⟨*relation-use*⟩`}`
▷ aggregation expression

| `count {` ⟨*relation-use*⟩ `}`
▷ count aggregation expression

| ⟨*relation-use*⟩ ▷ functional application

⟨*binary-op*⟩ ::= `+` | `-` | `*` | `/` | `^`

⟨*unary-op*⟩ ::= `+` | `-`

⟨*aggr-op*⟩ ::= `sum` | `min` | `max` | `single`

⟨*ground-metr-use*⟩ ::=
⟨*metrc-id*⟩`(`(⟨*ground-term*⟩(`,` ⟨*ground-term*⟩)*)?`)`
▷ metric atom with ground arguments

⟨*scope-decl*⟩ ::= `scope` ⟨*relation-id*⟩ ⟨*comp-op*⟩ ⟨*int*⟩`.`
▷ scope declaration

⟨*containment-decl*⟩ ::= `containment` ⟨*base-relation-id*⟩`.`
▷ containment relation declaration

⟨*root-decl*⟩ ::= `root` ⟨*base-relation-id*⟩`.`
▷ root relation declaration

B. Example configuration file

To showcase the usability of our configuration language, we present a formalization of the case study from section 2 below.

```
% Xcore-style metamodel definition

class InterferometryMission {
  contains GroundStationNetwork[1, 1]
    groundStationNetwork
  contains Spacecraft[2, +inf] spacecraft
  int observationTime
}

abstract class CommunicatingElement {
  contains CommSubsystem[1, 2] commSubsystem
}

class GroundStationNetwork extends
    CommunicatingElement {}

abstract class Spacecraft extends
    CommunicatingElement {
  contains Payload[0, 1] payload
}

abstract class CubeSat extends Spacecraft {}
class CubeSat3U extends CubeSat {}
class CubeSat6U extends CubeSat {}
class SmallSat extends Spacecraft {}

abstract class Payload {}
class InterferometryPayload extends Payload {}

class CommSubsystem {
  refers CommSubsystem[0, 1] target
  refers CommunicationType[1, 1] type
  int frequency
}

enum CommunicationType {
  KaBand,
  XBand,
  UHFBand
}

% Well-formedness constraints (excerpt)

functional directCommunicationLink(
        Spacecraft from,
        CommunicatingElement to) :-
    commSubsystem(from, fromComm),
    target(fromComm, toComm),
    commSubsystem(to, toComm).

error noLinkToGroundStation(Spacecraft s) :-
    g == 'gs', !directCommunicationLink+(s, g).

error communicationLoop(
        CommunicatingElement e) :-
    directCommunicationLink+(e, e).

error cubeSatWithKaAntenna(CubeSat s) :-
    commSubsystem(s, comm), ka == 'KaComm',
    type(comm, ka).

% Cost metric

missionCost(InterferometryMission m) :=
    sum { spacecraftCost(s) | spacecraft(m, s) }
        + 100000.0 * observationTime(m).

spacecraftCost(Spacecraft s) :=
    basePrice(s) * (kindCount(s) ^ (-0.25))
        + payloadCost() + commSubsysCost().

basePrice(Spacecraft s) :=
    CubeSat3U(s) ->  250000.0
  ; CubeSat6U(s) ->  750000.0
  ; SmallSat(s)  -> 3000000.0.

kindCount(Spacecraft s) :=
    CubeSat3U(s) -> count { CubeSat3U(_) }
  ; CubeSat6U(s) -> count { CubeSat6U(_) }
  ; SmallSat(s)  -> count { SmallSat(_) }.

hasPayload(Spacecraft s) := payload(s, _).

payloadCost(Spacecraft s) :=
    if hasPayload(s) then 50000.0 else 0.0.

hasAdditionalCommSubsys(Spacecraft s) :=
    commSubsystem(s, comm1),
    commSubsystem(s, comm2),
    !equals(comm1, comm2).

% The cost of the first CokmSubsystem is
% included in the price of the Spacecraft.
```

```
commSubsysCost(Spacecraft s) :=
    if hasAdditionalCommSubsys(s)
        then 100000.0 else 0.0.

% Objects and relations

InterferometryMission(o1).
observationTime(o1, [1, 5]).
GroundStationNetwork('gs').
groundStationNetwork(o1, 'gs').
CommSubsystem('gsc').
commSubsystem('gs', 'gsc').
type('gsc', 'KaComm').
frequency('gsc', 3000).
!commSubsystem('gsc', CommSubsystem::new).
CubeSat3U(o2).
spacecraft(o1, o2).
?CommSubsystem(o3).
commSubsystem(o2, o3).
frequency(o3, [3000, +inf]).
missionCost(o1): [0.0, 500000000.0].

% Scope definitions

scope domain >= 16.
scope domain <= 32.
scope Spacecraft =:= 12.
```

C. Regularity of partial models

Definition 11. A partial model $P = \langle \mathcal{O}_P, \mathcal{I}_P, \mathcal{V}_P \rangle$ is *structurally regular* if it satisfies the following conditions:

S1. $\forall o \in \mathcal{O}_P \colon \mathcal{I}_P(\texttt{exists})(o) > \texttt{false}$
$\triangleright$ non-existing objects are omitted

S2. $\forall o \in \mathcal{O}_P \colon \mathcal{I}_P(\texttt{equals})(o, o) > \texttt{false}$
$\triangleright$ the equals relation is reflexive

S3. $\forall o_1, o_2 \in \mathcal{O}_P \colon$
$\mathcal{I}_P(\texttt{equals})(o_1, o_2) = \mathcal{I}_P(\texttt{equals})(o_2, o_1)$
$\triangleright$ equals is symmetric

S4. $\forall o_1, o_2 \in \mathcal{O}_P \colon$
$(o_1 \not\equiv o_2) \Rightarrow \mathcal{I}_P(\texttt{equals})(o_1, o_2) < \texttt{true}$
$\triangleright$ if two objects are different, then they cannot be equal

S5. $\forall o \in \mathcal{O}_P \colon \mathcal{I}_P(\texttt{domain})(o) = \neg^4 \mathcal{I}_P(\texttt{data})(o)$
$\triangleright$ objects are partitioned into domain and data objects

Definition 12. A partial model $P = \langle \mathcal{O}_P, \mathcal{I}_P, \mathcal{V}_P \rangle$ is *naming regular* if it satisfies the following conditions:

N1. $\forall o \in \mathcal{O}_P, \texttt{'N'} \in \Sigma \colon \mathcal{I}_P(\texttt{'N'})(o) \leq \mathcal{I}_P(\texttt{domain})(o)$
$\triangleright$ named objects must be domain objects

N2. $\forall o \in \mathcal{O}_P, \texttt{'N'} \in \Sigma \colon \mathcal{I}_P(\texttt{'N'})(o) \neq \texttt{unknown}$
$\triangleright$ names cannot be uncertain

N3. $\forall o_1, o_2 \in \mathcal{O}_P, \texttt{'N'} \in \Sigma \colon [\mathcal{I}_P(\texttt{'N'})(o_1) \sqsupseteq \texttt{true} \wedge$
$\mathcal{I}_P(\texttt{'N'}) \sqsupseteq \texttt{true}] \Rightarrow (o_1 \equiv o_2)$
$\triangleright$ any name 'N' may only belong to a single object

N4. $\forall o \in \mathcal{O}_P, \texttt{'N}_i\texttt{'}, \texttt{'N}_j\texttt{'} \in \Sigma \colon [\mathcal{I}_P(\texttt{'N}_i\texttt{'})(o) \sqsupseteq \texttt{true} \wedge$
$\mathcal{I}_P(\texttt{'N}_j\texttt{'})(o) \sqsupseteq \texttt{true}] \Rightarrow (\texttt{'N}_i\texttt{'} = \texttt{'N}_j\texttt{'})$
$\triangleright$ an object cannot have more than one name

Definition 13. A partial model $P = \langle \mathcal{O}_P, \mathcal{I}_P, \mathcal{V}_P \rangle$ is *data regular* if it satisfies the following conditions:

D1. $\forall o \in \mathcal{O}_P \colon$
$[\mathcal{I}_P(\texttt{data})(o) \sqsupseteq \texttt{false}] \Rightarrow [\mathcal{V}_P(\texttt{value})(o) = \emptyset]$
$\triangleright$ only data objects have a numerical value

D2. $\forall o \in \mathcal{O}_P \colon$
$\mathcal{I}_P(\texttt{data})(o) \sqsupseteq \mathcal{I}_P(\texttt{real})(o) \vee^4 \mathcal{I}_P(\texttt{int})(o)$
$\triangleright$ data objects are either real or int

D3. $\forall o \in \mathcal{O}_P \colon [\mathcal{I}_P(\texttt{real})(o) \wedge^4 \mathcal{I}_P(\texttt{int})(o)] < \texttt{true}$
$\triangleright$ real and int are disjoint

D4. $\forall o \in \mathcal{O}_P : [\mathcal{I}_P(\texttt{int})(o) \sqsupseteq \texttt{true}] \Rightarrow$
$[\mathcal{V}_P(\texttt{value})(o) \cap \mathbb{Z} \neq \emptyset \vee \mathcal{V}_P(\texttt{value})(o) = \emptyset]$
$\triangleright$ int objects are bound to whole numbers

D5. $\forall o_1, o_2 \in \mathcal{O}_P, x \in \mathbb{R} \colon$
$[\mathcal{I}_P(\texttt{real})(o_1) \sqsupseteq \texttt{true} \wedge \mathcal{I}_P(\texttt{real})(o_2) \sqsupseteq \texttt{true} \wedge$
$\mathcal{V}_P(\texttt{value})(o_1) = \mathcal{V}_P(\texttt{value})(o_2) = [x, x]] \Rightarrow$
$(o_1 \equiv o_2)$
$\triangleright$ each real number is represented by a unique object

D6. $\forall o_1, o_2 \in \mathcal{O}_P, x \in \mathbb{R} \colon$
$[\mathcal{I}_P(\texttt{int})(o_1) \sqsupseteq \texttt{true} \wedge \mathcal{I}_P(\texttt{int})(o_2) \sqsupseteq \texttt{true} \wedge$
$\mathcal{V}_P(\texttt{value})(o_1) = \mathcal{V}_P(\texttt{value})(o_2) = [x, x]] \Rightarrow$
$(o_1 \equiv o_2)$
$\triangleright$ each int number is represented by a unique object

D. Parsing and serialization

A textual description of a model generation problem simultaneously encodes a signature $\langle \Sigma, \alpha \rangle$, a partial model $P = \langle \mathcal{O}_P, \mathcal{I}_P, \mathcal{V}_P \rangle$ and a theory T. The input language from section 3 can be parsed as follows:

1. First, syntactic sugars are removed from the input by translating them back into core concepts. This simplifies the input for the subsequent steps.

2. Symbols are gathered from the input. Base relations are turned into base relation symbols $\texttt{R}_1, \ldots \texttt{R}_m$, relations with a corresponding predicate definition are turned into predicate symbols $\texttt{P}_1, \ldots, \texttt{P}_\ell$, names of named objects are turned into object name symbols $\texttt{'N}_1\texttt{'}, \ldots, \texttt{'N}_t\texttt{'}$ and metrics are turned into metric symbols $\texttt{M}_1, \ldots, \texttt{M}_u$. Together with the build-in symbols, these form the symbols Σ of the signature. The arity $\alpha(s)$ of each symbol $s \in \Sigma$ is set to the number of arguments in the input text.

3. To obtain $\mathcal{O}_P$, we add an object $o_{\texttt{x}}$ for every unnamed object identifier $\texttt{x}$ and an object $o_{\texttt{'N'}}$ for every object name 'N' in the input problem.

4. We initialize $\mathcal{I}_P$ by setting $\mathcal{I}_P(\texttt{'N'})(o_{\texttt{'N'}}) = \texttt{true}$ for each object name 'N'. For all other $\mathcal{O}_P \ni o' \neq o_{\texttt{'N'}}$, $\mathcal{I}_P(\texttt{'N'})(o') = \texttt{false}$.

5. For all relation symbols $r \in \Sigma_R$ and objects $o_1, \ldots, o_{\alpha(r)} \in \mathcal{O}_P$, we gather all assertions of the form $r(o_1, \cdots, o_{\alpha(r)}) : X.$ and set $\mathcal{I}_P(r)(o_1, \ldots, o_{\alpha(r)}) = \bigoplus \{X \mid r(o_1, \cdots, o_{\alpha(r)}) : X.\}$.
6. For all objects $o \in \mathcal{O}_P$, we gather all value assertions of the form $o \colon iv.$ and set $\mathcal{V}_P(\texttt{value})(o) = \bigcap \{iv \mid (o \colon iv.\}$. If there are no such assertions, we set $\mathcal{V}_P(\texttt{value})(o) = [-\infty, +\infty]$ instead.
7. For all metric symbols $\texttt{M} \in \Sigma_M$ and objects $o_1, \ldots, o_{\alpha(\texttt{M})} \in \mathcal{O}_P$, we gather all value assertions of the form $\texttt{M}(o_1, \cdots, o_{\alpha(\texttt{M})}) : iv.$ and set $\mathcal{V}_P(\texttt{M})(o_1, \ldots, o_{\alpha(\texttt{M})}) = \bigcap \{iv \mid \texttt{M}(o_1, \cdots, o_{\alpha(\texttt{M})}) : iv.\}$. If there are no such assertion, we set $\mathcal{V}_P(\texttt{M})(o_1, \ldots, o_{\alpha(\texttt{M})}) = [-\infty, +\infty]$ instead.
8. Lastly, we gather all predicate and metric definitions into the theory T.

D.1. Regularization

After parsing the textual description of a model generation problem, the resulting partial model P may not necessarily be regular. Hence, a *regularization* procedure is applied to obtain an equivalent partial model P' as a refinement $P \sqsubseteq P'$ that is regular. To simplify presentation, in the steps below, P always refers to the output of the previous step. Thus, P serves as the input of the currently executed step, while P' is the output of the current step.

1. **Naming regularity** conditions N2–4 are satisfied by construction after parsing. To ensure N1, we only need to set $\mathcal{I}_{P'}(\texttt{domain})(o) = \mathcal{I}_P(\texttt{domain})(o) \vee^4 \mathcal{I}_P(\texttt{'N'})(o)$ for all $o \in \mathcal{O}_P$ and $\texttt{'N'} \in \Sigma$.
2. **Structural regularity** is enforced as follows:
   - S1. Surely non-existent ($\mathcal{I}_P(\texttt{exists})(o) = \texttt{false}$) objects are removed.
   - S2. We set $\mathcal{I}_{P'}(\texttt{equals})(o_1, o_2) = \texttt{error}$ for all o_1, $o_2 \in \mathcal{O}_P$ where $\mathcal{I}_P(\texttt{equals})(o_1, o_2) = \texttt{false}$.
   - S3. For all $o_1, o_2 \in \mathcal{O}_P$, we set $\mathcal{I}_{P'}(\texttt{equals})(o_1, o_2) = \mathcal{I}_P(\texttt{equals})(o_1, o_2) \oplus \mathcal{I}_P(\texttt{equals})(o_2, o_1)$ in order to to make $\texttt{equals}$ symmetric.
   - S4. Surely equal objects ($\mathcal{I}_P(\texttt{equals})(o_1, o_2) = \texttt{true}$) objects are merged. While merging objects, relevant interpretations of relational symbols are combined with the $\oplus$ information merge operator ($\mathcal{I}_{P'}(r)(o_{1,2}) = \mathcal{I}_P(r)(o_1) \oplus \mathcal{I}_P(r)(o_2)$). Similarly, interpretations of metric symbols are combined with the $\cap$ interval intersection operator.
   - S5. We set

$$\begin{aligned}\mathcal{I}_{P'}(\texttt{data})(o) = \mathcal{I}_P(\texttt{data})(o) & \\ \oplus \neg^4 \mathcal{I}_P(\texttt{domain})(o) & \\ \oplus [\mathcal{I}_P(\texttt{real})(o) & \\ \vee^4 \mathcal{I}_P(\texttt{int})(o)], & \\ \mathcal{I}_{P'}(\texttt{domain})(o) = \neg^4 \mathcal{I}_{P'}(\texttt{data})(o) &\end{aligned}$$

for all $o \in \mathcal{O}_P$.

3. **Data regularity** is enforced as follows:
   - D1. For all objects $o \in \mathcal{O}_P$ with $\mathcal{I}_P(\texttt{data})(o) \sqsupseteq \texttt{false}$, we set $\mathcal{V}_{P'}(\texttt{value})(o) = \emptyset$.
   - D2. When we enforced S5, D2 was also enforced as a side effect.
   - D3. If $\mathcal{I}_P(\texttt{real})(o) \wedge^4 \mathcal{I}_P(\texttt{int})(o) = \texttt{true}$ for an object $o \in \mathcal{O}_P$, we set all of $\texttt{real}$, $\texttt{int}$, $\texttt{data}$ and $\texttt{domain}$ for o to $\texttt{error}$.
   - D4. If $\mathcal{I}_P(\texttt{int})(o) \sqsupseteq \texttt{true}$ and $\mathcal{V}_P(\texttt{value})(o) \cap \mathbb{Z} = \emptyset$ for some object o, we set $\mathcal{V}_P(\texttt{value})(o) = \emptyset$.

   In order to enforce D5 and D6, we merge $\texttt{real}$ and $\texttt{int}$ objects that are bound to the same numbers (similarly to how we enforced S4).

Each of these steps are partial model refinements, hence the ultimate output P' is a refinement of the initial input P.

D.2. Serializing model generation problems

Serializing a partial model allows us to inspect the internal state of model generators. Moreover, model generation can be paused and its state can be saved to derive a new model generation problem either equivalent to the original one, or extended with new objects, assertions and constraints.

Regular partial models P and theories T over a signature $\langle \Sigma, \alpha \rangle$ can be serialized into a textual description of a model generation problem that, when parsed, gives rise to an identical signature, partial model and theory.

1. For each named object $o \in \mathcal{O}_P$ with name $\texttt{'N'}$ ($\mathcal{I}_P(\texttt{'N'})(o) \sqsupseteq \texttt{true}$), o will be uniquely represented by the name $\texttt{'N'}$. For all other objects, we assign unique identifiers x_o.
2. For each relation symbol $r \in \Sigma_R$ and objects $o_1, \ldots, o_{\alpha(r)}$, an assertion $r(o_1, \cdots, o_{\alpha(r)}) : \mathcal{I}_P(r)(o_1, \ldots, o_{\alpha(r)}).$ is printed, where each o_1 is represented either by its name or by its unique identifier.
3. For each object $o \in \mathcal{O}_P$ with $\mathcal{I}_P(\texttt{data})(o) \sqsupseteq \texttt{true}$, an assertion $o \colon \mathcal{V}_P(\texttt{value})(o).$ is printed.
4. For each metric symbol $\texttt{M} \in \Sigma_M$ and objects $o_1, \ldots, o_{\alpha(\texttt{M})}$, an assertion $\texttt{M}(o_1, \cdots, o_{\alpha(\texttt{M})}) : \mathcal{V}_P(\texttt{M})(o_1, \ldots, o_{\alpha(\texttt{M})}).$ is printed.
5. Lastly, axioms of the theory T are printed as predicate definitions and metric definitions.