Automated Generation of Consistent Models with Structural and Attribute Constraints

Oszkár Semeráth
Budapest University of Technology and Economics; MTA-BME Lendület Cyber-Physical Systems Res. Group
Budapest, Hungary
semerath@mit.bme.hu

Aren A. Babikian
McGill University
Montreal, Canada
aren.babikian@mail.mcgill.ca

Anqi Li
McGill University
Montreal, Canada
anqi.li2@mail.mcgill.ca

Kristóf Marussy
Budapest University of Technology and Economics; MTA-BME Lendület Cyber-Physical Systems Res. Group
Budapest, Hungary
marussy@mit.bme.hu

Daniel Varró
McGill University;
Budapest University of Technology and Economics; MTA-BME Lendület Cyber-Physical Systems Res. Group
Montreal, Canada
daniel.varro@mcgill.ca

ABSTRACT

Automatically synthesizing consistent models is a key prerequisite for many testing scenarios in autonomous driving or software tool validation where model-based systems engineering techniques are frequently used to ensure a designated coverage of critical cornercases. From a practical perspective, an inconsistent model is irrelevant as a test case (e.g. false positive), thus each synthetic model needs to simultaneously satisfy various structural and attribute well-formedness constraints. While different logic solvers or dedicated graph solvers have recently been developed, they fail to handle either structural or attribute constraints in a scalable way.

In the current paper, we combine a structural graph solver that uses partial models with an SMT-solver to automatically derive models which simultaneously fulfill structural and attribute constraints while key theoretical properties of model generation like completeness or diversity are still ensured. This necessitates a sophisticated bidirectional interaction between different solvers which carry out consistency checks, decision, unit propagation, concretization steps. We evaluate the scalability and diversity of our approach in the context of three complex case studies.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; • **Mathematics of computing → Solvers**.

KEYWORDS

model generation, partial modeling, SMT-solvers

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410962

ACM Reference Format:
Oszkár Semeráth, Aren A. Babikian, Anqi Li, Kristóf Marussy, and Daniel Varró. 2020. Automated Generation of Consistent Models with Structural and Attribute Constraints. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3365438.3410962

1 INTRODUCTION

The automated generation of test data is a core challenge for the effective testing of many software engineering applications. For many system-level testing scenarios e.g. for autonomous driving [32] or for taxation systems [64], such test data takes the form of a (typed and attributed) graph model. Moreover, when complex design or simulation tools are used in model-based systems engineering of safety-critical cyber-physical systems, those software tools themselves need to be tested with the same level of scrutiny as the underlying critical system itself (as part of software tool qualification [46]), otherwise the output of those tools cannot be trusted without further tests. Since such design and simulation tools frequently represent the underlying models internally as graphs, the quality assurance of tools also highly depends on the automated synthesis of consistent graph models as test data.

However, the automated synthesis of such consistent graph-based models that satisfy (or deliberately violate) a set of well-formedness constraints is a very challenging task. While various underlying logic solvers like SAT, SMT (Satisfiability Modulo Theories) or CSP (Constraint Satisfaction Problem) solvers have been repeatedly used for such purposes in tools, like in USE [16, 17], UML2CSP [11], Formula [28], various theorem provers [4], or Alloy [25] thanks to many favorable theoretical properties (e.g. soundness or completeness) such solvers primarily excel in detecting inconsistencies and not in deriving models used as test cases. As such, the use of logic solver based model generators is frequently hindered in practical testing scenarios by the lack of scalability [56, 64] (i.e. models with limited size can be generated) and diversity [27, 55] (i.e. models with identical structures are derived).

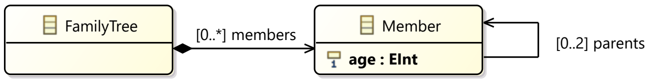


Figure 1: Metamodel of a Family Tree domain

Recent model generators [56, 63, 64] have successfully improved on scalability by lifting the model synthesis problem on the level of graph models by using meta-heuristic search [63] possibly combined with an SMT-solver [64]. Alternatively, partial model refinement [56] can be used as search strategy while efficient query/constraint evaluation engines [66, 68] validate the constraints during state space exploration. However, there are also important restrictions imposed by these tools such as lack of completeness [63, 64] or lack of attribute handling [56] in constraints.

In this paper, we propose a model generation technique which can automatically derive consistent graph models that satisfy both structural and attribute constraints. For that purpose, the structural constraints are satisfied along partial model refinement (like [56]) while attribute constraints are satisfied by repeatedly calling the Z3 SMT-solver [12] (like [64]). However, as a conceptual extension to preceding work, we define refinement units (in analogy with an abstract DPLL procedure modulo theories (Davis–Putnam–Logemann–Loveland) [39] or SMT-solvers [38]) with consistency checking, decision, unit propagation and concretization steps to enable a bidirectional interaction between a graph solver and an SMT-solver where a decision in one solver can be propagated to the other solver and vice versa. In particular,

- We define 3-valued logic semantics for evaluating structural and attribute constraints over partial models.
- We propose qualitative abstractions to uniformly represent attribute constraints as (structural) relations in a model.
- We define a mapping from attribute constraints to an SMT-problem interpreted by an SMT-solver.
- We propose a generic model generation process with bidirectional interaction between structural and attribute solvers. We present the detailed description of an attribute solver for numeric (e.g. *int* or *double*) constraints.
- We evaluate a prototype implementation of the approach on three case studies to assess scalability and diversity properties of model generation.

As a main added value with respect to existing results, our approach provides good scalability for automatically generating consistent models with structural and attribute constraints while still providing completeness and diversity.

2 PRELIMINARIES

2.1 Running example

We illustrate the challenges of handling both structural and attribute constraints in model generation for a simple domain of family trees with a metamodel shown in Figure 1 and well-formedness (WF) constraints defined by graph patterns (in the VQL language [67, 68]) listed in Figure 2. This domain is intentionally chosen to contain only few domain concepts, while it can demonstrate all key technical challenges of constraint evaluation.

```
private pattern memberHasParent(m: Member) { Member.parents(m, _); }
@Constraint pattern twoMembersHaveNoParent(m1: Member, m2: Member) {
    FamilyTree.members(f, m1); FamilyTree.members(f, m2); m1 != m2;
    neg find memberHasParent(m1); neg find memberHasParent(m2);
}
@Constraint pattern negativeAge(m: Member) {
    Member.age(m,mAge); check(mAge<0);
}
@Constraint pattern parentTooYoung(child: Member, parent: Member) {
    Member.parents(child, parent);
    Member.age(child, c); Member.age(parent, p); check(p <= (c + 12));
}
```

Figure 2: Structural & numerical constraints in Family Tree

A `FamilyTree` contains `Member`s with an integer `age` attribute. `Members` are related to each other by `parents` relations. The *violating cases* of the three WF constraints are defined by VQL graph patterns that all consistent family tree models need to respect:

- `twoMembersHaveNoParent`: There is at most one member in a family tree without a parent;
- `negativeAge`: All age attributes of family members are non-negative numbers;
- `parentTooYoung`: There must be more than 12 years of difference between the `age` attribute of a parent and a child.

In this paper, we use color coding to separate logic and numeric reasoning. The first constraint is a structural constraint (i.e. only navigation along object references) while the second constraint is a numerical constraint which accesses the `age` attribute (`mAge`) of a family member `m` and checks if the value of age is negative. The third constraint contains both structural and numerical clauses which mutually depend on each other. (1) If a new `parents` reference is created between two family members then a new numerical constraint needs to be enforced between the respective age attributes (logic→numeric dependency). (2) If the age attribute of two family members is already determined then a new `parents` reference may (or must not) be added between them (numeric→logic dependency).

Our paper investigates how to generate consistent models in the presence of such mutual dependencies between structural and attribute constraints. The bidirectional interaction is exemplified in the paper for numerical attributes, but the conceptual framework is applicable to attributes of other domains (e.g. strings, bitvectors) assuming the existence of an underlying solver (e.g. SMT-solver) for the background theory of the respective attribute. Our model generation framework semantically relies on model refinement carried out by 3-valued constraint evaluation over partial models.

2.2 Domain-specific partial models

Domain specification. We formalize the concepts in a target domain $\langle \Sigma, \alpha \rangle$ using an algebraic representation with a signature Σ and an arity function $\alpha : \Sigma \to \mathbb{N}$. Such a signature $\Sigma = \{\mathtt{T}_1, \ldots, \mathtt{T}_n, \mathtt{R}_t, \ldots, \mathtt{R}_r, \mathtt{P}_1, \ldots, \mathtt{P}_p, \mathtt{A}_1, \ldots, \mathtt{A}_a, \varepsilon, \sim\}$ can be easily derived from metamodeling formalisms like EMF [65].

- Unary predicate symbols $\{\mathtt{T}_1, \ldots, \mathtt{T}_t\}$ (with $\alpha(\mathtt{T}_i) = 1$) are defined for each *EClass* and *EEnum* in the domain (e.g. `FamilyTree`), `Bool` denotes the *EBoolean* type, `Int` denotes integer numbers types like *EInt* or *EShort*, etc.
- Binary predicate symbols $\{\mathtt{R}_1, \ldots, \mathtt{R}_r\}$ (with $\alpha(\mathtt{R}_i) = 2$) are defined for each *EReference* and *EAttribute* in the metamodel.

For example, `parents` represent the *parent* reference between two *Members*, and `age` represents the *age* attribute relation between a *Member* and an *EInt*.

- Structural predicate symbols $\{\mathtt{P}_1, \ldots, \mathtt{P}_p\}$ are n-ary predicates derived from graph queries (with $\alpha(\mathtt{P}_i) = n$ equal to the number of formal parameters of a graph query); e.g. `parentTooYoung` becomes a binary predicate symbol.
- Attribute predicate symbols $\{\mathtt{A}_1, \ldots, \mathtt{A}_a\}$ represent n-ary predicates derived from attribute (*check*) expressions of queries (with $\alpha(\mathtt{A}_i) = n$); e.g. $\mathtt{check}_{p \le c+12}(c, p)$ is a binary attribute predicate with parameters c and p.
- A special unary symbol ε denotes the *existence* of objects.
- A special binary symbol $\sim$ denotes the *equivalence* relation between two objects, which can be represented explicitly.

Partial models. Partial models are frequently used to explicitly represent uncertainty in models [13, 50], which is particularly relevant for intermediate steps of a model generation process. We use 3-valued partial models where the traditional truth values *true* (1) and *false* (0) are extended with a third truth value 1/2 to denote *unknown* structural parts of the model [19, 47, 57]. Similarly, we extend the domain of traditional numeric values (e.g. 1 or 2.1) with ? to denote an *unknown numeric value*.

Definition 2.1. Given a signature $\langle \Sigma, \alpha \rangle$, a *numerical partial model* is a logic structure $P = \langle O_P, \mathcal{I}_P, \mathcal{V}_P \rangle$ where:

- O_P is the finite set of objects in the model,
- $\mathcal{I}_P$ gives a 3-valued logic interpretation for each symbol $\mathtt{s} \in \Sigma$ as $\mathcal{I}_P(\mathtt{s}) : (O_P)^{\alpha(\mathtt{s})} \to \{0, 1, 1/2\}$,
- $\mathcal{V}_P$ gives a numeric value interpretation for each object in the model: $\mathcal{V}_P : O_P \to \mathbb{R} \cup \{?\}$.

Note that this definition uniformly handles domain objects (e.g. `Member`) and data objects (e.g. `Int`), which is frequently the case in object-oriented languages. Next, we capture some regularity restrictions to exclude irrelevant (irregular) partial models:

Definition 2.2. A partial model $P = \langle O_P, \mathcal{I}_P, \mathcal{V}_P \rangle$ is *regular*, if it satisfies the following conditions:

R1 $\forall o \in O_P : \mathcal{I}_P(\varepsilon)(o) > 0$ (non-existing objects are omitted)
R2 $\forall o \in O_P : \mathcal{I}_P(\sim)(o, o) > 0$ ($\sim$ is reflexive)
R3 $\forall o_1, o_2 \in O_P : \mathcal{I}_P(\sim)(o_1, o_2) = \mathcal{I}_P(\sim)(o_2, o_1)$ ($\sim$ is symmetric)
R4 $\forall o_1, o_2 \in O_P : (o_1 \not\equiv o_2) \Rightarrow \mathcal{I}_P(\sim)(o_1, o_2) < 1$ (if two objects are different, then they cannot be equivalent)
R5 $\forall o \in O_P : [(\mathcal{I}_P(\mathtt{Int})(o) = 0) \vee (\mathcal{I}_P(\mathtt{Real})(o) = 0)] \Rightarrow [\mathcal{V}_P(o) = ?]$ (domain objects do not have a value)
R6 $\forall o \in O_P : [\mathcal{V}_P(o) \neq ?] \Rightarrow [(\mathcal{I}_P(\mathtt{Int})(o) = 1) \vee (\mathcal{I}_P(\mathtt{Real})(o) = 1)]$ (objects with values are numbers)
R7 $\forall o \in O_P : [\mathcal{I}_P(\mathtt{Int})(o) = 1)] \Rightarrow [(\mathcal{V}_P(o) = ?) \vee (\mathcal{V}_P(o) \in \mathbb{N})]$ (only natural numbers are bound to `Int` objects)

Example 2.3. Figure 6 illustrates partial models. In **State 1**, we have three concrete objects (where ε and $\sim$ are 1): `FamilyTree` $f1$ and a `Member` $m1$, and an unbound `Int` data object $a1$ (with ? value). The partial model also contains an abstract *"new objects"* node that represents multiple potential new nodes (using 1/2 values for ε and for $\sim$, denoted by dashed border), and a *"new integers"* node representing the potential new integers. In Figure 6, predicates with value 1 are denoted by solid lines (as for the `member` edge between $f1$ and $m1$ in **State 1**) and predicates with value 1/2 are denoted by dashed lines (like the potential `parents` edge in **State 1**).

2.3 Refinement and concretization

During model generation, the level of uncertainty in partial models will be gradually reduced by refinements. In a refinement step, uncertain 1/2 values can be refined to either 1 or 0, or unbound values ? are refined to concrete numerical values. This is captured by an information ordering relation $X \sqsubseteq_L Y := (X = 1/2) \vee (X = Y)$ where an $X = 1/2$ is either refined to another value Y, or $X = Y$ remains equal. An information ordering can be defined between numerical values x and y similarly $x \sqsubseteq_N y := (x = ?) \vee (x = y)$.

A refinement from partial model P to partial models Q is a mapping that respects both information ordering relations ($\sqsubseteq_L / \sqsubseteq_N$).

Definition 2.4. A refinement from regular partial model P to regular partial model Q (denoted as $P \sqsubseteq Q$) is defined by a refinement function between the objects of the partial model $ref : O_P \to 2^{O_Q}$ which respect the information ordering:

- For each n-ary symbol $s \in \Sigma$, each object $p_1, \ldots, p_n \in O_P$, and for each refinement $q_1 \in ref(p_1), \ldots, q_n \in ref(p_n)$:
$$\mathcal{I}_P(s)(p_1, \ldots, p_n) \sqsubseteq_L \mathcal{I}_Q(s)(q_1, \ldots, q_n).$$
- For each object $p \in O_P$ and its refinement $q \in ref(p)$:
$$\mathcal{V}_P(p) \sqsubseteq_N \mathcal{V}_Q(q).$$
- All objects in Q are refined from an object in P, and existing objects $p \in O_P$ must have a non-empty refinement.

Model generation is carried out along refinements by eventually resolving all uncertainties to obtain a concrete model.

Definition 2.5. A regular (see Definition 2.2) partial model P is *concrete*, if (a) $\mathcal{I}_P$ does not contain 1/2 values, and (b) $\mathcal{V}_P$ does not contain ? values for integer and real data objects (for object o where $\mathcal{I}_P(\mathtt{Int})(o) = 1$ or $\mathcal{I}_P(\mathtt{Real})(o) = 1$).

Example 2.6. Figure 6 illustrates several refinement steps. Between **State 0** and **State 1**, *new object* is split into two objects by refining $\sim$ to 0 between *new object* and $m1$, creating a single concrete object $m1$ by refining $\sim$ on $m1$ to 1. Simultaneously, type `Member` is refined to 1, `FamilyTree` refined to 0, and reference predicate `members` from $f1$ to $m1$ is refined to 1. Eventually, the value of data object $a1$ is refined from ? to 2 in **State 4.2**.

2.4 Constraints over partial models

Syntax. Both structural (logical) and numerical constraints can be evaluated on partial models. For each graph pattern we derive a *logic predicate* (LP) defined as $\mathtt{P}(v_1, \ldots, v_n) \Leftrightarrow \varphi$, where φ is a *logic expression* (LE) constructed inductively from the pattern body as follows (assuming the standard precedence for operators).

- if $s \in \Sigma$ is an n-ary predicate symbol (i.e. T, R, P, A, ε or $\sim$) then $s(v_1, \ldots, v_n)$ is a logic expression;
- if φ_1 and φ_2 are logic expressions, then $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, and $\neg\varphi_1$ are logic expressions;
- if φ is a logic expression, and v is a variable, then $\exists v : \varphi$ and $\forall v : \varphi$ are logic expressions.

For each attribute constraint, we derive *attribute predicates* (as helpers) by reification to enable seamless interaction between structural and attribute solvers along a compatibility (if and only if) operator $\Leftrightarrow$ (see Figure 3). In case of numbers, such an attribute predicate is tied to a *numerical predicate* defined as $\mathsf{A}(v_1, \ldots, v_n) \Leftrightarrow \psi$ where ψ is constructed from *numeric expressions*. The expressiveness of those expressions are limited by the background theories of the underlying backend SMT solver. Here we define a core language of basic arithmetical expressions, which is supported by a wide range of numerical solvers:

- each variable v, constant symbol and literal (concrete number) c is a numerical expression,
- if ψ_1 and ψ_2 are numerical expressions, then $\psi_1 + \psi_2$, $\psi_1 - \psi_2$, $\psi_1 \times \psi_2$ and $\psi_1 \div \psi_2$ are numerical expressions.
- if ψ_1 and ψ_2 are numerical expressions then $\psi_1 < \psi_2$, $\psi_1 > \psi_2$, $\psi_1 \geq \psi_2$, $\psi_1 \leq \psi_2$, $\psi_1 = \psi_2$, $\psi_1 \neq \psi_2$ are numerical predicates.

Example 2.7. Graph pattern `parentTooYoung(child, parent)` from Figure 2 is formalized as the following logic predicate:

$$\mathsf{parentTooYoung}(child, parent) \Leftrightarrow \mathsf{parents}(child, parent) \land \mathsf{age}(child, c) \land \mathsf{age}(parent, p) \land \mathsf{check}_{p \leq c+12}(c, p)$$

where $\mathsf{check}_{p \leq c+12}(c, p) \Leftrightarrow p \leq c + 12$ is a numerical predicate.

Later such predicates will help communicate between different solvers, e.g. if $\mathsf{check}_{p \leq c+12}(c_1, p_1)$ is found to be 1 by the graph solver for some members c_1 and p_1 then the numerical predicate $p_1 \leq c_1 + 12$ needs to be enforced by a numerical solver for the respective data objects and vice versa.

Semantics. A logic predicate $\mathsf{P}(v_1, ..., v_n) \Leftrightarrow \varphi$ can be evaluated on a partial model P along a variable binding $Z : \{v_1, \ldots, v_n\} \to O_P$ (denoted as $[\![\varphi]\!]^P_Z$), which can result in three truth values: 1, 0 or $1/2$. The inductive rules of evaluating the semantics of a logic expression is illustrated in Figure 3. Note that *min* and *max* takes the numerical minimum and maximum values of 0, $1/2$ and 1.

A numerical predicate $\mathsf{A}(v_1, \ldots, v_n) \Leftrightarrow \psi$ can also be evaluated on a partial model P along variable binding $Z : \{v_1, \ldots, v_n\} \to O_P$ (denoted as $(\!|\varphi|\!)^P_Z$) with a result of 1, 0 or $1/2$. The inductive rules capturing the semantics of logic expressions are illustrated in Figure 3. Note that $x \langle cmp \rangle y$ means the truth value of numerical comparison $\langle cmp \rangle$ (e.g. $3 < 5$ is 1), while $x \langle op \rangle y$ means the numerical value of the result of an operation $\langle op \rangle$ (e.g. $3 + 5$ is 8).

Constraint approximation. When a predicate is evaluated on a partial model, then the 3-valued semantics of constraint evaluation guarantees that certain (over- and under-approximation) properties hold for all potential refinements or concretizations of the partial model. For all logic and numeric predicates φ and ψ, if $P \sqsubseteq Q$ then $[\![\varphi]\!]^P \sqsubseteq_L [\![\varphi]\!]^Q$ and $(\!|\psi|\!)^P \sqsubseteq_L (\!|\psi|\!)^Q$, consequently:

- **Logic under-approximation**: If $[\![\varphi]\!]^P = 1$ in a partial model P then $[\![\varphi]\!]^Q = 1$ in any partial model Q where $P \sqsubseteq Q$.
- **Numeric under-approximation**: If $(\!|\psi|\!)^P = 1$ in a partial model P then $(\!|\psi|\!)^Q = 1$ in any partial model Q where $P \sqsubseteq Q$.
- **Logic over-approximation**: If $[\![\varphi]\!]^Q = 0$ in a partial model Q then $[\![\varphi]\!]^P \leq 1/2$ in a partial model P where $P \sqsubseteq Q$.
- **Numeric under-approximation**: If $(\!|\psi|\!)^Q = 0$ in a partial model Q then $(\!|\psi|\!)^P \leq 1/2$ in a partial model P where $P \sqsubseteq Q$.

Semantics of Logic Predicates

$$[\![\mathsf{P}(v_1, \ldots, v_n) \Leftrightarrow \varphi]\!]^P_Z := \mathcal{I}_P(\mathsf{P})(Z(v_1), \ldots, Z(v_n)) \Leftrightarrow [\![\varphi]\!]^P_Z$$
$$(\!|\mathsf{A}(v_1, \ldots, v_n) \Leftrightarrow \psi|\!)^P_Z := \mathcal{I}_P(\mathsf{A})(Z(v_1), \ldots, Z(v_n)) \Leftrightarrow (\!|\psi|\!)^P_Z$$
$$\text{where } X \Leftrightarrow Y := \begin{cases} 0 & \text{if } (x = 1, y = 0) \text{ or } (x = 0, y = 1) \\ 1 & \text{if } (x = 1, y = 1) \text{ or } (x = 0, y = 0) \\ 1/2 & \text{otherwise} \end{cases}$$

Semantics of Logic Expressions (φ)

$$[\![\mathsf{T}(v)]\!]^P_Z := \mathcal{I}_P(\mathsf{T})(Z(v))$$
$$[\![\mathsf{R}(v_1, v_2)]\!]^P_Z := \mathcal{I}_P(\mathsf{R})(Z(v_1), Z(v_2))$$
$$[\![\mathsf{P}(v_1, \ldots, v_n)]\!]^P_Z := \mathcal{I}_P(\mathsf{P})(Z(v_1), \ldots, Z(v_n))$$
$$[\![\mathsf{A}(v_1, \ldots, v_n)]\!]^P_Z := \mathcal{I}_P(\mathsf{A})(Z(v_1), \ldots, Z(v_n))$$
$$[\![v_1 \sim v_2]\!]^P_Z := \mathcal{I}_P(\sim)(Z(v_1), Z(v_2))$$
$$[\![\varepsilon(v)]\!]^P_Z := \mathcal{I}_P(\varepsilon)(Z(v))$$
$$[\![\neg\varphi]\!]^P_Z := 1 - [\![\varphi]\!]^P_Z$$
$$[\![\varphi_1 \land \varphi_2]\!]^P_Z := min([\![\varphi_1]\!]^P_Z, [\![\varphi_2]\!]^P_Z)$$
$$[\![\varphi_1 \lor \varphi_2]\!]^P_Z := max([\![\varphi_1]\!]^P_Z, [\![\varphi_2]\!]^P_Z)$$
$$[\![\exists v : \varphi]\!]^P_Z := max\{[\![\varepsilon(v) \land \varphi]\!]^P_{Z, v \mapsto o} : o \in O_P\}$$
$$[\![\forall v : \varphi]\!]^P_Z := min\{[\![\neg\varepsilon(v) \lor \varphi]\!]^P_{Z, v \mapsto o} : o \in O_P\}$$

Semantics of Numerical Expressions (ψ)

$$(\!|\psi_1 \langle cmp \rangle \psi_2|\!)^P_Z := \begin{cases} 1/2, & \text{if } (\!|\psi_1|\!)^P_Z \text{ or } (\!|\psi_2|\!)^P_Z \text{ is } ? \\ (\!|\psi_1|\!)^P_Z \langle cmp \rangle (\!|\psi_2|\!)^P_Z, & \text{otherwise} \end{cases}$$
$$(\!|\psi_1 \langle op \rangle \psi_2|\!)^P_Z := \begin{cases} ?, & \text{if } (\!|\psi_1|\!)^P_Z \text{ or } (\!|\psi_2|\!)^P_Z \text{ is } ? \\ (\!|\psi_1|\!)^P_Z \langle op \rangle (\!|\psi_2|\!)^P_Z, & \text{otherwise} \end{cases}$$
$$(\!|v|\!)^P_Z := \mathcal{V}_P(Z(v))$$
$$(\!|\,literal\,|\!)^P_Z := literal \text{ (e.g. concrete numbers)}$$

Figure 3: Inductive semantics of graph predicates

Using these properties, model generation becomes a monotonous derivation sequence of partial models which starts from the most abstract partial model where all predicate constraints are evaluated to $1/2$. The partial model is gradually refined, thus more and more predicate values are evaluated to either 1 or 0. The under-approximation lemmas ensure that when an error predicate is evaluated to 1 it will remain 1, thus exploration branch can be terminated without loss of completeness [69]. The over-approximation lemmas assure that if a partial model can be refined to a concrete model where error predicate is 0, then it will not be dropped.

3 MODEL GENERATION WITH REFINEMENT

3.1 Functional overview

Our model generation approach takes the following inputs:

(1) the signature of a domain $\langle \Sigma, \alpha \rangle$ with structural logic symbols $\mathsf{P}_1, \ldots, \mathsf{P}_p$ and numerical attribute symbols $\mathsf{A}_1, \ldots, \mathsf{A}_a$,
(2) a logic theory consisting of the negation of the error predicates and the compatibility of the predicate symbols with their definition (i.e. the axioms): $\mathcal{T} = \{\neg\mathsf{E}_1, \ldots, \neg\mathsf{E}_e, (\mathsf{P}_1 \Leftrightarrow \varphi^{\mathsf{P}_1}), \ldots, (\mathsf{P}_p \Leftrightarrow \varphi^{\mathsf{P}_p}), (\mathsf{A}_1 \Leftrightarrow \psi^{\mathsf{A}_1}), \ldots, (\mathsf{A}_a \Leftrightarrow \psi^{\mathsf{A}_a})\}$
(3) some search parameters (e.g., the required size, or the required number of models).

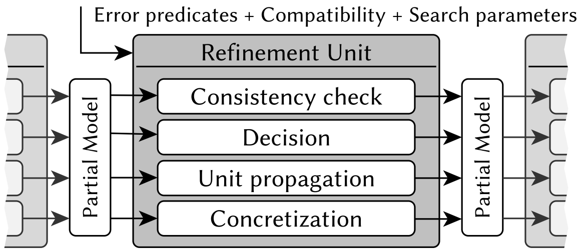


Figure 4: Schematic overview of a refinement unit

The output of the generator is a sequence of models $M_1, \ldots, M_m$, where each M_i is (1) a regular concrete model of $\langle \Sigma, \alpha \rangle$, which is (2) consistent with $\mathcal{T}$, i.e. no error predicates have a match $[\![\neg \mathsf{E}_j]\!]^{M_i} = 1$ (for any i, j), and all predicates P_j and A_j are compatible their definition $[\![\mathsf{P}_j \Leftrightarrow \psi^{\mathsf{P}_j}]\!]^{M_i} = 1$ and $(\!|\mathsf{A}_j \Leftrightarrow \psi^{\mathsf{A}_j}|\!)^{M_i} = 1$ (for any i, j), and it (3) adheres to search parameters (e.g., $|O_{M_i}| = size$).

The model generator framework combines individual *refinement units* to solve structural and numerical problems. Each refinement unit analyzes a partial model (which is an intermediate state of the model generation), and it collaborates with other units by refining it. This is in conceptual analogy with the interaction of background theories in SMT-solvers [38, 39]. A refinement unit provides four main functionalities, as illustrated in Figure 4:

- **Consistency check:** The refinement unit evaluates whether a partial model may satisfy the target theory (thus it can be potentially completed to a consistent model), or it surely violates it (thus no refinement is ever consistent).
- **Decision**: The unit makes an atomic decision by a single refinement in the partial model (e.g. adding an edge by setting a $1/2$ value to 1) which is consistent with the target theory. This new information makes the model more concrete, thus reducing the number of potential solutions.
- **Unit propagation:** After a decision, the unit executes further refinements necessitated by the consequences of previous refinements wrt. the target theory without introducing new information or excluding potential solutions. This step automatically does necessary refinements on the partial partial model without making any decisions.
- **Concretization:** Finally, the unit attempts to complete the partial model in a single refinement step by setting all uncertain $1/2$ edges to 0, and checks if the concrete model is consistent with the target theory or not.

In this paper, we combine two of such refinements units: we reuse a graph solver [56] as *structural refinement unit* to efficiently generate the structural part of models (i.e. to reason about $[\![\varphi]\!]_Z^P$), and we propose a novel *numerical refinement unit* that uses efficient SMT-solvers in the background to solve the numerical part of models to reason about $(\!|\psi|\!)_Z^P$. The refinement units interact with each other bidirectionally via the refinement of partial models: the structural refinement unit refine truth values on attribute predicates (based on the structural part of the error predicates), which need to be respected by the numerical refinement unit. Symmetrically, the numerical refinement unit can refine attribute predicates (based on the numerical part of the error predicates), which need to be respected by the structural refinement unit in turn.

3.2 State space exploration by refinements

Our model generation framework derives models by exploring the search space of partial models along refinements carried out by refinement units. As such, the size of the partial models is continuously growing up to a designated size, while the exploration process tries to intelligently minimize the search space. The detailed steps of this exploration process are illustrated in Figure 5.

0. Initialization: First, we initialize our search space with an initial partial model. This is derived either from an existing initial model provided by an engineer (thus each solution will contain this seed model as a submodel), or it can be the *most general partial model* $P_0 = \langle O_{P_0}, \mathcal{I}_{P_0}, \mathcal{V}_{P_0} \rangle$ where $O_{P_0} = \{new\}$ has a single element, $\mathcal{V}_{P_0}$ is $1/2$ for every symbol, and $\mathcal{V}_{P_0}(new) = ?$.

1. Decision: Next, we select an unexplored decision candidate proposed by a refinement unit, and execute it to refine the partial model by adding new nodes and edges, or by populating a data object with a concrete value. In our setup, this decision step is executed mainly by the structural refinement unit which has more impact on model generation. If no decision candidates are left unexplored, the search concludes with an UNSAT result.

2. Unit propagation: After a decision, the framework executes unit propagation in all refinement units until a fixpoint is reached in order to propagate all consequences of the decision.

3. State coding: The search can reach isomorphic partial models along multiple trajectories. To prevent the repeated exploration of the same state, a state code is calculated and stored for a new partial model by using shape-based graph isomorphism checking [42, 43]. If exploration detects that a partial model has already been explored, it drops the partial model and continues search from another state. Otherwise the framework calculates the state code of the newly explored partial model and continues with its evaluation.

4. Consistency check: Next, each refinement unit checks whether the partial model contains any inconsistencies that cannot be repaired. Structural refinement unit evaluates the (logic) under-approximation of the error predicates (see Section 2.4), which can detect irreparable structural errors. The numerical refinement unit carries out a satisfiability check of the numerical constraint determined by a call to the numerical solver.

5. Concretization: Then the framework tries to concretize the partial model to a fully-defined solution candidate by resolving all uncertainties, and checks its compliance with the target theory and model size. If no violations are found and the model is of given size, then the instance model is stored as a solution. Note that this directly ensures the correctness of all solutions. If this concretization fails, it indicates that something is missing from the model, so the refinement process continues.

6. Approximate distance & Add to state space: When a partial model is refined, our framework estimates its distance from a solution [33]. This heuristic is based on the number of missing objects and the number of violations in its concretization. Then, the new partial model is added to the search space of unexplored decisions where the exploration continues at **1. Decision**.

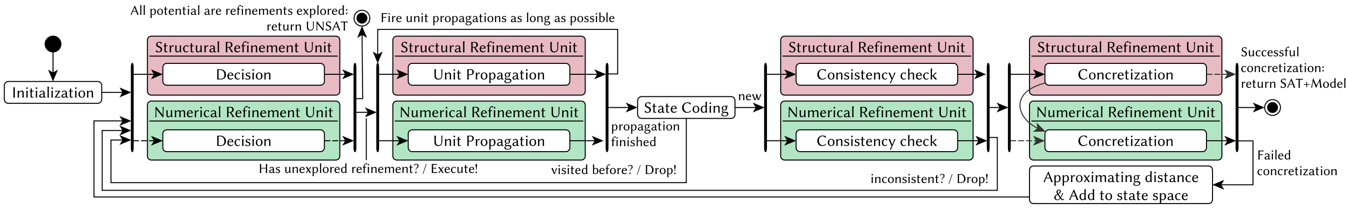


Figure 5: Overview of exploration strategy for model generation

Further heuristics: For selecting the next unexplored decision to refine, we use a combined exploration strategy with best-first search heuristic, backtracking, backjumping and random restarts with an advanced design space exploration framework [21, 56].

Example 3.1. Figure 6 illustrates a model generation run to derive a family tree. Search is initialized with a `FamilyTree` *f1* as root and two (abstract) objects to represent new objects and new integers.

State 1 highlights the execution of a decision refinement that splits the new object and the new integer, creating a new `Member` $m1$ with its undefined `age` attribute.

In **State 2.1**, a loop `parent` edge is added as a decision. When investigating error predicate `parentTooYoung`(*child*, *parent*), the search reveals that all conditions of the error predicate are surely satisfied on objects $m1$ and $a1$ except for attribute predicate $\texttt{check}_{p \le c+12}$. Therefore, the structural refinement unit can refine the partial model by setting $\mathcal{I}_{S2.1}(\texttt{check}_{p \le c+12})(a1, a1)$ to $\texttt{0}$ without excluding any valid refinements, which implies that $(\!| p \le c + 12 |\!)^{S2.1}_{p \mapsto a1, c \mapsto a1} = \texttt{0}$. The numerical refinement unit (with the help of an underlying SMT solver) can detect that no value $\mathcal{V}_{S2.1}(a1)$ can be bound to object $a1$ such that $\mathcal{V}_{S2.1}(a1) \le \mathcal{V}_{S2.1}(a1) + 12$ is false, therefore the model cannot be finished to a consistent model thus it can be safely dropped.

In **State 2.2**, a new `Member` $m2$ is added to the `FamilyTree`, the framework attempts to concretize the model by resolving all uncertainty in **State 3.1**. First, the structural refinement unit concretizes in the structural part of the model, all $\texttt{1/2}$ values are set to $\texttt{0}$ (e.g. all the potential `parent` edges disappear). Then, sample valid values are generated for the attributes by the numerical refinement unit. When the concretization is checked, error pattern `twoMembersHaveNoParent`$(m1, m2)$ indicates that there are some missing `parent` edges, so the framework drops the concretization but continues to explore State 2.2.

Eventually, after adding a `parent` edge in **State 3.2**, the framework is able to concretize a model in **State 4.2** that satisfies the target theory, thus concluding the search with a consistent model.

3.3 Structural refinements by a graph solver

Our structural refinement unit uses a graph solver [56].

The structural *consistency* of a partial model can be verified by checking the compatibility of all predicates P as $[\![\mathsf{P} \Leftrightarrow \varphi^{\mathsf{P}}]\!]^P_Z$. If a predicate is incompatible with its definition, or an error predicate is satisfied, the partial model is inconsistent (see Section 2.4).

Our framework operates on a graph representation of partial models (without a mapping to a logic solver), thus structural predicates are evaluated directly on this graph representation. The query rewriting technique [57] enables to efficiently evaluate the 3-valued semantics of logic predicates $[\![\varphi]\!]^P_Z$ by a high-performance incremental model query engine [67, 68], which caches and maintains the truth values of logic predicates during exploration.

Structural refinements are implemented by graph transformations [56, 69]. *Decisions* are simple transformation rules that rewrite a single $\texttt{1/2}$ value to a $\texttt{1}$ in the partial model, or an equivalence predicate $\sim$ to $\texttt{0}$ to split an object to two (like *m1* is separated from *new object* in **State 1**). On the other hand, *concretization* rewrites all $\texttt{1/2}$ values to $\texttt{0}$, and self-equivalences to $\texttt{1}$.

The compatibility of predicate symbols is checked by structural *unit propagation rules*, which are derived from error predicates to refine a partial model when needed to avoid a match of an error predicate. We rely on two kinds of unit propagation rules:

- We derive unit propagation rules from the structural constraints imposed by the metamodel to enforce type hierarchy, multiplicities, inverse references, and containment hierarchy [56]. For example, when a new `Member` is created, a new `Int` is also created with an `age` predicate between them.
- From each error predicate $\mathsf{E}(v_1, \ldots, v_n)$, unit propagation rules are derived to check when a $\texttt{1}$ (or $\texttt{0}$) value would satisfy the error predicate $[\![\mathsf{E}(v_1, \ldots, v_n)]\!]^P = \texttt{1}$. In such cases, the value is refined to the opposite $\texttt{0}$ (or $\texttt{1}$). Such unit propagation rules may add numerical implications of error predicates.

3.4 Numerical refinements by SMT-solvers

The numerical refinement unit is responsible for maintaining the compatibility of numerical constraints and attribute predicates, checking consistency of numerical constraints, and deriving concrete numeric values in the model.

Numerical refinement is based on a purely numerical problem created from a partial model. Let P be a partial model with attribute predicates $\mathsf{A}_1, \ldots, \mathsf{A}_a$ defined by $\psi^{\mathsf{A}_1}, \ldots, \psi^{\mathsf{A}_a}$. Let O^{Num}_P denote the set of data objects where $[\![\texttt{Int}(v) \vee \texttt{Real}(v)]\!]^P_{v \mapsto o} \ge \texttt{1/2}$. For each data object $o \in O^{Num}_P$, we create a numeric variable $\mathsf{V}(o)$ denoting its potential value. If $[\![\texttt{Int}(v)]\!]^P_{v \mapsto o} \ge \texttt{1/2}$, then the type of this variable is integer, while if $[\![\texttt{Real}(v)]\!]^P_{v \mapsto o} \ge \texttt{1/2}$ then it is real.

The numerical problem derived from the partial model is defined over those variables. First, if the value of a data object $o \in O^{Num}_P$ is already known in the partial model (i.e. $\mathcal{V}_P(o) \ne ?$), then we assert its value as a numerical equation: $\mathcal{V}_P(o) = \mathsf{V}(o)$. Next, for each attribute A_i, we assert its definition ψ^{A_i} for all data objects:

- If $(\!| \mathsf{A}_i(v_1, \ldots, v_n) |\!)^P_{\bar{v} \mapsto \bar{o}} = \texttt{1}$, we assert $\psi^{\mathsf{A}_i}(\mathsf{V}(o_1), \ldots, \mathsf{V}(o_n))$
- If $(\!| \mathsf{A}_i(v_1, \ldots, v_n) |\!)^P_{\bar{v} \mapsto \bar{o}} = \texttt{0}$, we assert $\neg\psi^{\mathsf{A}_i}(\mathsf{V}(o_1), \ldots, \mathsf{V}(o_n))$

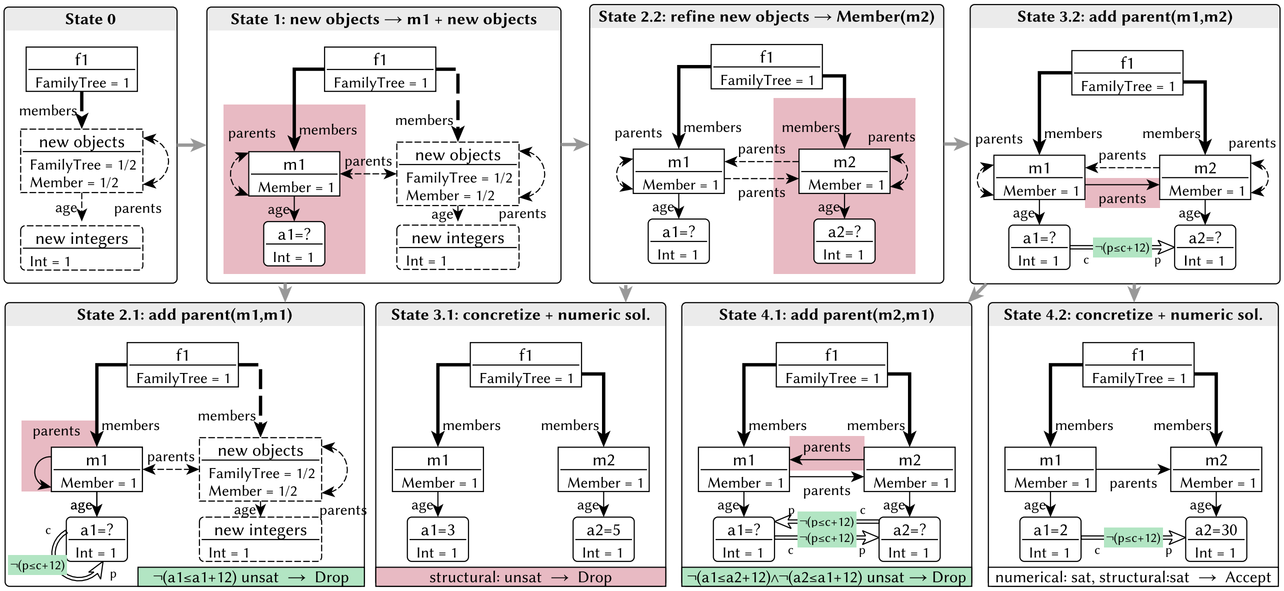


Figure 6: Sample exploration process example

- If $(\!|\mathsf{A}_i(v_1,\ldots,v_n)|\!)^P_{\bar{v}\mapsto\bar{o}} = 1/2$, then nothing is asserted.

The entire numerical problem Ψ_P is constructed as the conjunction of the respective numerical clauses, which can be solved by an SMT-solver like Z3 [12]. The SMT-solver calculates the satisfiability of the numerical problem as a *consistency check* for partial models. An unsatisfiable numerical problem Ψ_P would imply that the partial partial P model cannot be completed with consistent numerical values, thus it can be dropped.

If the problem is satisfiable, then each variable can be bound to a concrete number (value), and the value assigned to $\mathsf{V}(o)$ is recorded in the partial model as $\mathcal{V}_P(o)$. This step is used to complete partial models by bounding all unbounded data objects during *concretization*.

Additionally, fixing a potential value for a single data object can be used as a *decision*. However, in a typical model generation setting, the number of potential values that can be assigned to an object is huge, thus this step proved to be impractical.

The numerical consequences of the constructed Ψ_P can be used to refine a partial model during *unit propagation*. In our framework, three kinds of unit propagation operations are supported:

- When the values of objects $o_1, \ldots, o_n$ are all known in a partial model, then the truth value of $(\!|\psi^{\mathsf{A}_i}(v_1,\ldots,v_n)|\!)^P_{\bar{v}\mapsto\bar{o}}$ can be calculated in the model without calling an SMT solver.
- If an attribute predicate A_i has an unknown value $(\!|\mathsf{A}_i(v_1,\ldots,v_n)|\!)^P_{\bar{v}\mapsto\bar{o}} = 1/2$, and $\Psi_P \wedge \psi^{\mathsf{A}_i}(o_1,\ldots,o_n)$ is proved to be inconsistent by the SMT-solver, then $\mathcal{I}_Q(\mathsf{A}_i)(o_1,\ldots,o_n)$ must be set to 0 in the refined partial model Q. Similarly, if $\Psi_P \wedge \neg\psi^{\mathsf{A}_i}(o_1,\ldots,o_n)$ is inconsistent, the attribute can be refined to 1. In our case studies, this step was impractical thus this feature was not used.
- An unique solution for $\mathsf{V}(o)$ can be used to set $\mathcal{V}_P(o)$.

Example 3.2. We form a numerical problem based on the partial model of **State 4.1** in Figure 6. Since $\mathtt{check}_{p\leq c+12}$ is a condition in the error pattern $\mathtt{parentTooYoung}(child, parent)$, $\mathcal{I}_{S2.1}(\mathtt{check}_{p\leq c+12})(a1,a1)$ must be set to 0 in the model and the logical operator of $\mathtt{check}_{p\leq c+12}$ should be negated. Since $\mathtt{Member}$ m_1 is a parent of $\mathtt{Member}$ m_2, we have that $p \mapsto a_1$ and $c \mapsto a_2$. It follows that $a_1 \nleq a_2 + 12$. Similarly, since $\mathtt{Member}$ m_2 is a parent of $\mathtt{Member}$ m_1, we have that $a_2 \nleq a_1 + 12$. The numerical problem to solve here is $(a_1 \nleq a_2 + 12) \wedge (a_2 \nleq a_1 + 12)$. Feeding it into an SMT-solver, we will be informed that this problem is unsatisfiable.

3.5 Soundness and Completeness

With the combination of the structural and numerical refinement units, our proposed approach generates models with numerical attributes using partial model refinement. Our approach is *sound*: it generates consistent solutions only. This is guaranteed by the direct evaluation of the error predicates and compatibility predicates on the final stage of model refinement. Our approach is *structurally complete*: for a given scope (size), it is able to generate all models with different graph structures, which is ensured by the approximation lemmas in Section 2.4 and in [56, 69]. We intentionally avoid fulfilling *numerical completeness*, since even simple models could have potentially infinite number of attribute bindings.

4 EVALUATION

We conducted various measurements to address the following research questions:

RQ1: What are the costs and benefits of calling a SMT-solver continuously during exploration or calling it as postprocessing?

RQ2: How do the different exploration steps contribute to the execution time for generating the first model and incrementally generating subsequent models?

RQ3: What is the scalability of model generation when deriving large models with structural and attribute constraints?

RQ4: How structurally diverse are the synthesized models compared to model generation without attribute constraints?

4.1 Target Domains

We perform model generation campaigns in three complex case studies. The target domain artifacts as well as output models and measurement results are available on GitHub (https://github.com/viatra/VIATRA-Generator) and as a virtual machine (https://doi.org/10.5281/zenodo.3950552).

Fam: The *FamilyTree* domain is presented in Section 2 as our running example. We use the metamodel shown in Figure 1 which captures parenthood relations and the age of family tree members (with 2 classes, 3 references and 1 numeric attribute). Furthermore, 3 constraints are defined as graph predicates that place structural and numerical restrictions on family tree members. The initial model used for model generation contains a single `FamilyTree` node. While this domain looks simple, there is a subtle mutual dependency between structural and attribute constraints, which provides extra challenges for the interaction of different solvers.

Sat: The *Satellite* domain (introduced in [22]) represents *interferometry mission architectures* used for space mission planning at NASA. Such an architecture consists of collaborating satellites and radio communication between them, which are captured by a metamodel with 15 classes, 5 references and 2 numerical attributes. Additionally, 18 constraints are defined as graph predicates to capture restrictions on collaborating satellites. The initial model contains a single root node as the starting point for model generation.

Tax: The *Taxation* domain (used in [63, 64]) represents the personal income tax management application used by the Government of Luxembourg. We reused the original metamodel which contains 54 classes (including 15 Enum classes), 52 relations and 92 attributes, 44 of which are numerical. Additionally, we replicated the OCL constraints used in [64] as graph predicates.

In order to independently replicate the case study of [64] in a pure EMF context with strict containment hierarchy (instead of UML), we include a `Resource` class in the metamodel that contains instances of the `Household` class, which was the root class of the original *Taxation* metamodel. This allows the instantiation of multiple `Household` instances within the same model generation task. To enforce the same number of objects, we include an initial model containing a predefined number of `Household` instances and we prevent the generation of further instances of that class as in [64].

General setup: To account for warm-up effects and memory handling of the Java 8 VM, an initial model generation task is performed before the actual measurements and the garbage collector is called explicitly between runs. We performed the measurements on an enterprise server[1].

4.2 RQ1: Integration with an SMT-solver

Measurement setup: We compare three model generation approaches that call an SMT-solver in fundamentally different ways:

- *postSMT* (used as a baseline) does not make any SMT solver calls during model generation only as a postprocessing step.

[1] 12×2.2 GHz CPU, 64 GiB RAM, CentOS 6, Java 1.8, 12 GiB Heap

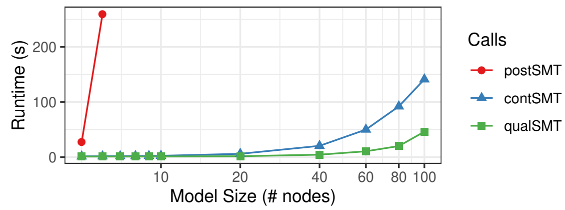


Figure 7: Calls to SMT-solvers wrt. size of generated models

- *contSMT* calls the SMT solver at every model generation step to repeatedly evaluate numerical constraints.
- *qualSMT* qualitatively approximates a numerical constraint with a manually added structural constraint that enforces an acyclic graph structure for families. Then model generation first addresses the structural constraints while the qualitative abstraction is resolved to concrete numerical attribute values in a postprocessing step.

Note that the SMT-solver is not used to derive models satisfying the structural constraints, only the numerical constraints as poor scalability was reported in [4, 54] for structural constraints.

For **RQ1**, we perform measurements exclusively in the **Fam** domain, which contains a complex dependency between structural and numerical constraints, thus it is expected to serve as a stress test for SMT-solver calls. We aim to generate models of different size: from 5 to 10 objects with an increment of 1, and from 20 to 100 objects with an increment of 20. The range for numeric values were not bounded a priory. Ten runs are executed for each approach and model size with a timeout of 5 minutes, and the median of runtimes is taken.

Analysis of results: Figure 7 compares the execution times for the three approaches. Unsurprisingly, *postSMT* could only generate models of size 5 and 6 (with a failure rate of over 90% for larger models). These figures imply if there is *mutual dependency between both numerical and structural constraints* in a domain, then the handling of numerical constraints cannot be postponed to a postprocessing phase while focusing exclusively on structural constraints first during model generation as the SMT-solver cannot correct the incompatibilities introduced by an inconsistent structure.

Additionally, we notice that *qualSMT* is faster than *contSMT* by a factor of 3 for larger models. This is partly attributed to the use of VIATRA as a back-end solver for the *qualSMT* approach, which has been shown in [4, 56] to perform better than Z3 (used in *postSMT*) for generating model structures. Moreover, future model generation strategies may skip calls to the SMT-solver in certain steps. While the extra structural constraint that enforces an acyclic graph structure for families was added by human intuition, providing such qualitative abstractions of numerical constraints in an automated way is also a promising direction of future research.

RQ1: *Mutual dependencies between structural and numerical constraints necessitate repeated calls to an SMT-solver during model generation, which cannot be postponed to postprocessing step. Qualitative structural abstractions of numerical constraints may accelerate model generation by introducing an approximate causality.*

4.3 RQ2: Cost of exploration phases

Measurement setup: We perform measurements in all three domains to compare runtimes and their distribution between the different phases of model generation. For each domain, we generate models with an increasing minimum model size of 20, 40, 60, 80 and 100. Again, the range for numeric values were not bounded. We exclude larger model sizes to ensure high success rates and to enable cross-domain comparison of execution phases. For the **Tax** domain, the initial model contains one instance of the `Household` class for every 20 generated nodes (which is the typical size of a household in the models generated in [64]) to balance the difficulty of model generation regardless of the target model size.

We execute 10 runs per target model size and take the median runtime values. For each model generator run, we aim to produce the first 10 models within a timeout of 1 hour.

Analysis of results: The decomposition of runtime measurements for all three domains are shown in Figure 8a - Figure 8c. Each phase of model generation is represented by a different color. The initialization phase (0.9 seconds for **Fam**, 3.5 seconds for **Sat**, 150 seconds for **Tax**) is a one-time penalty which is proportional to the complexity of the metamodel and the WF constraints.

In the **Fam** domain, the runtime is dominated by SMT solver calls. This is attributed to the fact that this domain needs to enforce a global structural constraint (acyclicity as discussed in Section 4.2) by solving numerical constraints while the numerical constraints of other domains are dominantly local (e.g. to fill attribute values). However, extra cost of generating subsequent models is low. In the **Sat** case study, generating the first model takes less than 30 seconds (dominated by the time required for state encoding), but the cost of incrementally generating the next model is relatively larger. For the target model sizes, execution times in the **Tax** case study are still dominated by the initialization phase due to the large metamodel and numerous constraints of the domain, while the actual model generation and constraint solving phases were rapid.

RQ2: *Different phases of model generation can be dominating for modeling problems with different characteristics. For domains with global numeric constraints such as **Fam**, runtime is dominated by SMT-solver calls. For structure-dominant challenges, such as **Sat**, runtime is dominated by state encoding, and the incremental time required to generate additional models is larger. For domains with a large and complex metamodel such as **Tax**, the initialization phase can be substantial, but the sheer complexity of the domain does not directly influence the actual model generation.*

4.4 RQ3: Scalability of model generation

Measurement setup: We perform measurements in all three domains with increasing model sizes starting from 100 objects with a step size of 50/100 objects and timeout of 1 hour. A single model is generated in each run. A campaign of 10 runs is executed for each measurement point and the median of *successful* execution times is taken (i.e. that provide a finite model as result within the given time). We terminate the scalability measurement for a case if any of the 10 runs at a particular size fails to output a finite model. For the **Tax** domain, we provide `Household` instances as part of the initial model following the 1-to-20 ratio discussed in Section 4.3.

Analysis of results: Measurement results for **RQ3** are shown in Figure 8d-Figure 8f. Interestingly, the proposed approach scaled best for the largest metamodel of the **Tax** case deriving models with 1,100 objects within an hour. Furthermore, we were able to generate models with 1,200 objects within the same time limit with a success rate of 80%. Model generation with 100% success rate scaled up to 300 objects for the **Fam** and **Sat** domains. However, root cause of scalability limits was very different (the SMT-solver in **Fam** and graph solver in **Sat**). These results also a posteriori validate our choice of including **Fam** as a case study, which turned out to be the most complex one for assessing the use of SMT-solvers.

RQ3: *Our approach was able to generate consistent models with 300 objects for all three case studies within an hour. For the **Tax** case, scalability is comparable to figures reported in [64] with well over 1000 objects.*

4.5 RQ4: Diversity

Measurement setup: To evaluate the structural diversity of the generated models, we used a neighbourhood-based [44] internal diversity metric [55, 58], which correlates with mutation score in mutation testing scenarios. This metric calculates the proportion of different local neighborhoods of nodes included in a graph model. We checked the structural diversity of models only; the measurement and generation of diverse attribute values with the underlying solver requires further research and beyond the scope of the paper.

We used a neighborhood range=3, which classifies two objects to be identical, if they cannot be distinguished with at most 3 navigations (hops). To measure structural diversity, the values of data objects are not taken into account. We measured the diversity of 10-10 models for all three case studies with 100 objects (**Fam**+N, **Sat**+N, **Tax**+N). As a comparison, we generated 10-10 models without bounding the attributes values or respecting the attribute constraints (**Fam**-N, **Sat**-N, **Tax**-N) by the graph solver [56].

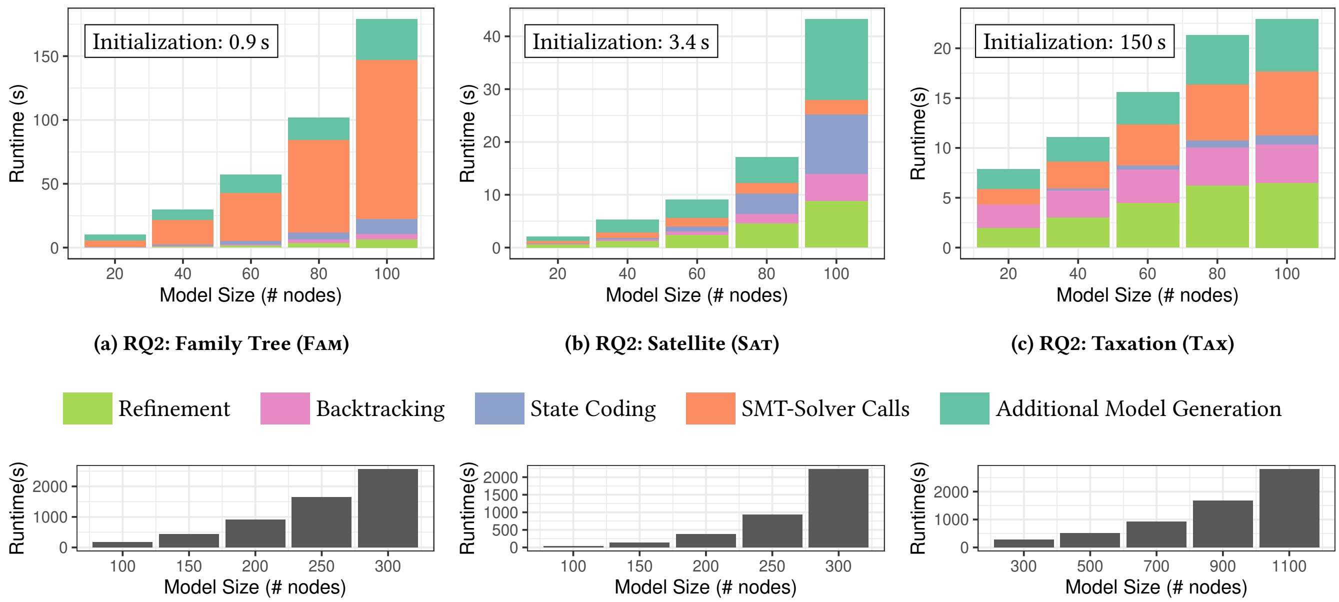
Analysis of Results: The distribution of internal diversity is illustrated in Figure 9. Note that **Fam**+N,**Fam**-N, **Sat**+N and **Sat**-N showed similarly high internal diversity 80%, while diversity values were somewhat lower for **Tax**+N and **Tax**-N.

RQ4: *Our approach provides similar structural diversity when generating consistent models with structural and numerical constraints compared to the diversity provided by a graph solver [56].*

4.6 Threats to validity

Construct validity. We replicated the **Tax** case study [64] in a new technological context, which involved (1) to create an Ecore metamodel from an equivalent UML diagram and (2) to manually transform the OCL constraints into equivalent VQL graph patterns. The Ecore metamodel was kindly provided to us by the authors of [64], while we validated each replicated OCL constraint by performing manual equivalence checks. We used similar number of Household objects as in [64] and investigated the output models by graph visualization tools to ensure that similar model generation outputs are obtained, but we refrain from direct numerical comparison of execution times due to those technological differences.

Internal Validity. To strengthen internal validity, our experiments include a warm-up run executed prior to the actual measurements to decrease the fluctuation of runtime results caused by the



(a) RQ2: Family Tree (Fam)

(b) RQ2: Satellite (Sat)

(c) RQ2: Taxation (Tax)

(d) RQ3: Family Tree (Fam)

(e) RQ3: Satellite (Sat)

(f) RQ3: Taxation (Tax)

Figure 8: Runtimes of different exploration steps when generating models of increasing size

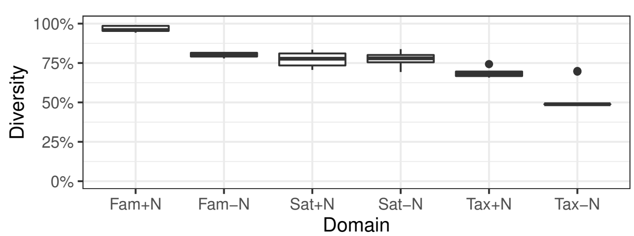


Figure 9: Internal Diversity distributions

Java VM instead of the natural fluctuation of solver runtimes. As the exploration strategy relies on some randomness, our scalability measurements only report cases with over 90% success rate.

External Validity. We mitigate threats to external validity by including a diverse set of case studies which involve calls to both a structural and a numerical solver. However, our approach is as scalable as the underlying attribute solver. While we were able to generate large consistent models in all three cases, the numeric constraints in our case studies are limited to basic arithmetic operations (excluding e.g. trigonometric operations and quadratic equations). We focused on numeric attributes as they are the most frequent data types. Handling models containing different kinds of attributes (e.g., string or bitvectors) can be a challenge in terms of performance (although the underlying Z3 solver promises efficient background theorems for both [12, 71]). Additionally, the numeric values derived by the SMT-solver may not be diverse.

5 RELATED WORK

We provide an overview of graph generation approaches that derive consistent graphs. We also discuss some key numerical abstractions and decision procedures.

Logic solver approaches. These approaches translate graphs and WF constraints into a logic formulae and use underlying solvers to generate graphs that satisfy them. Back-end technologies used for this purpose include SMT solver such as Z3 [26, 52, 70], SAT-based model finders (like Alloy [25]) [3, 5, 9, 23, 30, 35, 54, 59, 60, 62], CSP-solvers [8, 10, 11, 18], theorem provers [4], first-order logic [6], constructive query containment [41], higher-order logic [20] and an incremental query engine [56].

For most of these approaches, scalability is limited to small models/counter-examples. These approaches are either a priori bounded (where the search space needs to be restricted explicitly) or they have decidability issues. Furthermore, handling of numeric constraints is not available for some of these approaches, particularly ones based on SAT-solvers and first-order logic formulations.

Uncertain models. Partial models are similar to uncertain models, which offer a rich specification language [13, 48] amenable to analysis. They provide a more intuitive, user-friendly language compared to 3-valued interpretations, but without handling additional WF constraints. Potential concrete models compliant with an uncertain model can be synthesized by the Alloy Analyzer [50], or refined by graph transformation rules [49].

Iterative approaches. Iterative approaches generate models by multiple solver calls. An iterative approach is proposed specifically for allocation problems in [29] based on Formula. In [59] models are generated in by calling Alloy in multiple steps, where each step extends the instance model by a few elements. Finally, an iterative, counter-example guided synthesis is proposed for higher-order logic formulae in [36]. For these approaches, when scalability evaluation is included, it is limited to 50 nodes.

Symbolic model generation techniques. Certain techniques use abstract (or symbolic) graphs for analysis purposes. A tableau-based reasoning method is proposed for graph properties [1, 40, 51], 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).

Different approaches use abstract interpretation [44], or predicate abstraction [14, 19, 45] for partial modeling. 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.

Hybrid approaches. These approaches divide the model generation task into multiple sub-tasks and use a different underlying techniques to resolve each one. The PLEDGE model generation tool [64] provides such a scalable implementation by combining metaheuristic search for model structure generation with an SMT-solver based approach for attribute handling. The Evacon tool [24] implements a search-based evolutionary testing approach followed by symbolic execution to generate tests for object-oriented programs. Autograph [52] sequentially combines a tableau-based approach for model structure generation with an SMT-solver based approach for attribute handling. Such approaches combine multiple techniques in a sequential manner, which is a conceptual restriction for mutually dependent structural and numerical constraints. Moreover, none of these techniques assure completeness of model generation.

Another category of hybrid approaches involves assessing multiple components of the model generation task in parallel. This requires the implementation of a certain decision procedure such as DPLL(T) [15, 39] to iterate between underlying techniques, or combine them by sharing variables in their proofs [38]. Such decision procedures are presented alongside their associated properties (e.g. soundness and completeness) at an abstract level in [7, 39], which allows for formal reasoning about their implementations. However, those approaches handle graph-based models inefficiently [59, 69], thus the scalability of those techniques is limited.

Numerical abstractions. Handling numeric (integer or real) variables and constraints in model generation scenarios requires their abstract interpretation through numerical abstract domains [37, 61]. Numerical abstract domains may be used to summarize object attributes in value analysis of heap programs [14, 31, 34]. Summarized dimensions [19] 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 [2]. But these approaches do not generate graph models.

The uniqueness of our approach lies in combining numerical abstractions with partial models to guarantee soundness and completeness, while generating models with favorable scalability.

6 CONCLUSIONS

In this paper, we proposed an automated model generation approach to derive consistent models that satisfy structural and numerical constraints, which necessitates a bidirectional interaction between a graph solver and an SMT-solver. As a conceptual novelty, we proposed so-called refinement units that carry out consistency checking, decision, unit propagation and concretization steps in conceptual analogy with background theories used in SMT-solvers as part of an abstract DPLL procedure [39]. Therefore, refinement units can seamlessly incorporate different kinds of solvers (in a manner similar to [38]) for handling attribute constraints in the presence of a graph solver that handles partial models. We implemented our approach in the Viatra Solver framework [53]. The source code of our approach is publicly available (https://github.com/viatra/VIATRA-Generator).

We prepared a publicly available measurement environment (https://doi.org/10.5281/zenodo.3950552), and we carried out a detailed experimental evaluation of our approach in three complex case studies to assess scalability and diversity. We obtained favorable scalability results by consistently deriving models with over 250 objects in two cases within an hour, and models with over 1000 objects in the third case with same time limits. These model sizes are substantially larger than logic solver based model generation approaches (e.g. Alloy or Z3) could derive in the presence of structural constraints (see [4, 56, 64]). Moreover, our approach maintains other favorable quality attributes such as diversity and completeness investigated in depth in [55, 69].

ACKNOWLEDGEMENTS

We would like to thank Ghanem Soltana and Lionel C. Briand for their help in running the Taxation case study.

This paper is partially supported by MTA-BME Lendület Research Group on Cyber-Physical Systems, BME-Artificial Intelligence FIKP grant of EMMI (BME FIKP-MI/SC), the NSERC RGPIN-04573-16 project, the Fonds de recherche du Québec - Nature et technologies (FRQNT) B1X scholarship (file number: 272709), and Natural Sciences and Engineering Research Council of Canada (NSERC) PGSD3-546810-2020.

REFERENCES

[1] Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wasowski. 2016. Symbolic execution of high-level transformations. In *SLE 2016*. Springer, 207–220.

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2009. Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.* 11, 1 (2009), 53–67.

[3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2010. On challenges of model transformation from UML to Alloy. *Softw. Syst. Model.* 9, 1 (2010), 69–86.

[4] Aren A Babikian, Oszkár Semeráth, and Dániel Varró. 2020. Automated Generation of Consistent Graph Models with First-Order Logic Theorem Provers. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 441–461.

[5] Kacper Bak, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. Clafer: unifying class and feature modeling. *Softw. Syst. Model.* (2013), 1–35.

[6] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. 2002. Translating the Object Constraint Language into First-order Predicate Logic. In *Proc. VERIFY, Workshop at FLoC*.

[7] Martin Brain, Vijay D'Silva, Leopold Haller, Alberto Griggio, and Daniel Kroening. 2013. An Abstract Interpretation of DPLL(T). In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 455–475.

[8] Fabian Büttner and Jordi Cabot. 2012. Lightweight String Reasoning for OCL. In *ECMFA 2012 (LNCS, Vol. 7349)*, Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos (Eds.). Springer, 244–258.

[9] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. 2012. Verification of ATL Transformations Using Transformation Models and Model Finders. In *ICFEM*. Springer, 198–213.

[10] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2007. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE 2017*. ACM, 547–548.

[11] Jordi Cabot, Robert Clarisó, and Daniel Riera. 2014. On the Verification of UML/OCL Class Diagrams using Constraint Programming. *Journal of Systems and Software* (March 2014).

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

[13] Michalis Famelis, Rick Salay, and Marsha Chechik. 2012. Partial models: Towards modeling and reasoning with uncertainty. In *ICSE*. IEEE Computer Society, 573–583.

[14] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. 2012. TVAL+: TVLA and Value Analyses Together. In *SEFM 2012 (LNCS, Vol. 7504)*. Springer, 63–77.

[15] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification*, Rajeev Alur and Doron A. Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–188.

[16] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 1 (2007), 27 – 34.

[17] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. 2018. Achieving model quality through model validation, verification and exploration. *Comput. Lang. Syst. Struct.* 54 (2018), 474–511. https://doi.org/10.1016/j.cl.2017.10.001

[18] Carlos A. González, Fabian Büttner, Robert Clarisó, and Jordi Cabot. 2012. EMFtoCSP: a tool for the lightweight verification of EMF models. In *FormSERA 2012*. 44–50.

[19] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. 2004. Numeric Domains with Summarized Dimensions. In *TACAS 2004 (LNCS, Vol. 2988)*. Springer, 512–529.

[20] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. 2009. System Model-Based Definition of Modeling Language Semantics. In *FORTE (LNCS, Vol. 5522)*. Springer, 152–166.

[21] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. 2015. A model-driven framework for guided design space exploration. *Automated Software Engineering* 22, 3 (2015), 399–436.

[22] Sebastian J. I. Herzig, Sanda Mandutianu, Hongman Kim, Sonia Hernandez, and Travis Imken. 2017. Model-transformation-based computational design synthesis for mission architecture optimization. In *IEEE Aerospace Conference*. IEEE.

[23] Frank Hilken, Martin Gogolla, Loli Burgueño, and Antonio Vallecillo. 2018. Testing models and model transformations using classifying terms. *Software and Systems Modeling* 17, 3 (2018), 885–912.

[24] Kobi Inkumsah and Tao Xie. 2008. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 297–306.

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

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

[27] Ethan K Jackson, Gabor Simko, and Janos Sztipanovits. 2013. Diversely enumerating system-level architectures. In *Proceedings of the 11th ACM Int. Conf. on Embedded Software*. IEEE Press, 11.

[28] Ethan K. Jackson and Janos Sztipanovits. 2006. Towards a Formal Foundation for Domain Specific Modeling Languages. In *EMSOFT* (Seoul, Korea). ACM, New York, NY, USA, 53–62.

[29] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. 2010. An Approach for Effective Design Space Exploration. In *Monterey Workshop*. LNCS, Vol. 6662. Springer, 33–54.

[30] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. 2011. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *TOOLS '11 (LNCS, Vol. 6705)*. 290–306.

[31] Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. 2007. Arithmetic Strengthening for Shape Analysis. In *SAS 2007 (LNCS, Vol. 4634)*. Springer, 419–436.

[32] István Majzik, Oszkár Semeráth, Csaba Hajdu, Kristóf Marussy, Zoltán Szatmári, Zoltán Micskei, András Vörös, Aren A Babikian, and Dániel Varró. 2019. Towards System-Level Testing with Coverage Guarantees for Autonomous Vehicles. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 89–94.

[33] Kristóf Marussy, Oszkár Semeráth, and Dániel Varró. 2020. Automated Generation of Consistent Graph Models with Multiplicity Reasoning. *Submitted to the IEEE for possible publication.* (2020). https://inf.mit.bme.hu/en/research/publications/automated-generation-consistent-graph-models-multiplicity-reasoning

[34] Bill McCloskey, Thopas Reps, and Mooly Sagiv. 2010. Statically Inferring Complex Heap, Array, and Numeric Invariants. In *SAS 2010 (LNCS, Vol. 6337)*. Springer, 71–99.

[35] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational Constraint Solving in SMT. In *CADE 2017 (LNCS, Vol. 10395)*. Springer, 148–165.

[36] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *ICSE 2015*. IEEE, 609–619.

[37] Antoine Miné. 2004. *Weakly Relational Numerical Abstract Domains*. Ph.D. Dissertation.

[38] Greg Nelson and Derek C Oppen. 1979. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 2 (1979), 245–257.

[39] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 6 (Nov. 2006), 937–977.

[40] Karl-Heinz Pennemann. 2008. Resolution-like theorem proving for high-level conditions. In *ICGT 2008 (LNCS, Vol. 5214)*. Springer, 289–304.

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

[42] Arend Rensink. 2004. Canonical Graph Shapes. In *ESOP*. Springer, 401–415.

[43] Arend Rensink. 2007. Isomorphism checking in GROOVE. *Electronic Communications of the EASST* 1 (2007).

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

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

[46] RTCA, Inc. 2011. DO-330 Sofware Tool Qualification Considerations.

[47] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 3 (2002), 217–298.

[48] Rick Salay and Marsha Chechik. 2015. A Generalized Formal Framework for Partial Modeling. In *Fundamental Approaches to Software Engineering*, Alexander Egyed and Ina Schaefer (Eds.). LNCS, Vol. 9033. Springer Berlin Heidelberg, 133–148.

[49] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. 2015. A Methodology for Verifying Refinements of Partial Models. *Journal of Object Technology* 14, 3 (2015), 3:1–31.

[50] Rick Salay, Michalis Famelis, and Marsha Chechik. 2012. Language Independent Refinement Using Partial Modeling. In *FASE*. Springer, 224–239.

[51] Sven Schneider, Leen Lambers, and Fernando Orejas. 2017. Symbolic model generation for graph properties. In *FASE 2017 (LNCS, Vol. 10202)*. Springer, 226–243.

[52] Sven Schneider, Leen Lambers, and Fernando Orejas. 2018. Automated reasoning for attributed graph properties. *STTT* 20, 6 (2018), 705–737.

[53] Oszkár Semeráth, Aren A. Babikian, Sebastian Pilarski, and Dániel Varró. 2019. VIATRA Solver: A Framework for the Automated Generation of Consistent Domain-Specific Models. In *ICSE 2019 Demonstrations*. IEEE, 43–46.

[54] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. 2017. Formal Validation of Domain-Specific Languages with Derived Features and Well-Formedness Constraints. *Softw. Syst. Model* 16, 2 (2017), 357–392.

[55] Oszkár Semeráth, Rebeka Farkas, Gábor Bergmann, and Dániel Varró. 2020. Diversity of Graph Models and Graph Generators in Mutation Testing. *Int. J. Softw. Tools Technol. Transf.* 22, 1 (2020), 57–78.

[56] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. 2018. A graph solver for the automated generation of consistent domain-specific models. In *ICSE*. ACM, 969–980.

[57] Oszkár Semeráth and Dániel Varró. 2017. Graph Constraint Evaluation over Partial Models by Constraint Rewriting. In *ICMT*. 138–154.

[58] Oszkár Semeráth and Dániel Varró. 2018. Iterative Generation of Diverse Models for Testing Specifications of DSL Tools. In *FASE*. Springer, 227–245.

[59] Oszkár Semeráth, András Vörös, and Dániel Varró. 2016. Iterative and Incremental Model Generation by Logic Solvers. In *FASE*. Springer, 87–103.

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

[61] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. A Practical Construction for Decomposing Numerical Abstract Domains. *Proc. ACM Program. Lang.* 2, POPL (2018). Article no. 2.

[62] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. 2010. Verifying UML/OCL models using Boolean satisfiability. In *DATE'10*. IEEE, 1341–1344.

[63] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2017. Synthetic data generation for statistical testing. In *ASE*. 872–882.

[64] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2020. Practical Constraint Solving for Generating System Test Data. *ACM Trans. Softw. Eng. Methodol.* 29, 2, Article 11 (April 2020), 48 pages.

[65] The Eclipse Project 2019. *Eclipse Modeling Framework*. The Eclipse Project. http://www.eclipse.org/emf.

[66] The Object Management Group 2014. *Object Constraint Language, v2.4*. The Object Management Group.

[67] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.* 98 (2015), 80–99.

[68] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling* 15, 3 (2016), 609–629.

[69] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. 2018. Towards the Automated Generation of Consistent, Diverse, Scalable and Realistic Graph Models. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig (LNCS, Vol. 10800)*. Springer, 285–312.

[70] Hao Wu, Rosemary Monahan, and James F. Power. 2013. Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. In *TASE*. 175–182.

[71] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 114–124.