A Graph Solver for the Automated Generation of Consistent Domain-Specific Models

Oszkár Semeráth[1,2], András Szabolcs Nagy[1,2] and Dániel Varró[3,1,2]
[1] MTA-BME Lendület Cyber-Physical Systems Research Group, Hungary
[2] Budapest University of Technology and Economics, Department of Measurement and Information Systems, Hungary
[3] McGill University, Canada
semerath@mit.bme.hu,nagya@mit.bme.hu,varro@mit.bme.hu

ABSTRACT

Many testing and benchmarking scenarios in software and systems engineering depend on the systematic generation of graph models. For instance, tool qualification necessitated by safety standards would require a large set of consistent (well-formed or malformed) instance models specific to a domain. However, automatically generating consistent graph models which comply with a metamodel and satisfy all well-formedness constraints of industrial domains is a significant challenge. Existing solutions which map graph models into first-order logic specification to use back-end logic solvers (like Alloy or Z3) have severe scalability issues. In the paper, we propose a graph solver framework for the automated generation of consistent domain-specific instance models which operates directly over graphs by combining advanced techniques such as refinement of partial models, shape analysis, incremental graph query evaluation, and rule-based design space exploration to provide a more efficient guidance. Our initial performance evaluation carried out in four domains demonstrates that our approach is able to generate models which are 1-2 orders of magnitude larger (with 500 to 6000 objects!) compared to mapping-based approaches natively using Alloy.

ACM Reference Format:
Oszkár Semeráth[1,2], András Szabolcs Nagy[1,2] and Dániel Varró[3,1,2]. 2018. A Graph Solver for the Automated Generation of Consistent Domain-Specific Models. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 12 pages.
https://doi.org/10.1145/3180155.3180186

ACKNOWLEDGMENTS

This paper is partially supported by MTA-BME Lendület Cyber-Physical Systems Research Group, the NSERC RGPIN-04573-16 project and the UNKP-17-3-III New National Excellence Program of the Ministry of Human Capacities. We are grateful for Alexandra Sólyom and Gábor Szárnyas for preliminary experiments in using design space exploration for model generation and all contributors to past model generators of query and transformation benchmarks.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05.
https://doi.org/10.1145/3180155.3180186

1 INTRODUCTION

Motivation. The automated generation of graph models has recently become a key approach in various application areas of software and systems engineering. Representing objects and pointers as graphs in object-oriented programs, automatically generated models may serve as complex test stubs [1, 2]. Auto-generated graphs help the testing and benchmarking of graph databases [3] since obtaining real graphs from business use cases is often difficult to due to protection of intellectual property rights. Automated synthesis of prototypical test contexts [4] aims to systematically derive previously unanticipated contexts in the form of graph models for the assurance of smart cyber-physical systems (CPS).

Similarly, in many design and verification tools used for engineering complex CPSs, system models are also represented internally as graphs, and model generators may be used for validating, testing or benchmarking such design tools [5–8]. As a main practical motivation for this scenario, while tool qualification of design and verification tools are necessitated by safety standards (like DO-178C [9], or ISO 26262 [10]), tool qualification is extremely costly due to the lack of effective best practices for validating the tools themselves. Considering tool qualification as a long-term objective, our approach will be illustrated in the context of an industrial domain-specific modeling tool (Yakindu Statecharts), but it could be adapted to many other practical scenarios above.

In [11], four desirable properties are stated as key challenges for graph model generators used in such scenarios. A set of auto-generated graphs should ideally be (1) *consistent*, i.e. each graph should satisfy all well-formedness (WF) constraints, (2) *diverse*, i.e. each pairs of graphs should be structurally distant from each other, (3) *scalable*, i.e. the size of models are exponentially growing, and (4) *realistic*, i.e. auto-generated graphs cannot be distinguished from real models created by engineers by using advanced graph metrics [3, 12, 13]. For instance, characteristics (1,2) are essential in a functional testing scenario, while properties (3,4) are crucial for benchmarking and stress testing.

However, real models created by engineers are dominantly consistent (see [11] for a statistical analysis) according to the correct-by-construction principle (e.g. commits are disallowed when test cases fail or constraints are violated). Similarly, auto-generated graphs violating a single WF constraint but satisfying all the others are frequently necessitated for testing purposes. Moreover, random generation of graphs is unable to guarantee consistency, i.e. many WF constraints will be violated. This way, the *synthesis of consistent graphs is a prerequisite of both realistic and diverse graph models* .

Problem Statement. This paper aims to automatically generate well-formed graph models of a specification defined by (1) a

metamodel (graph schema), (2) a set of well-formedness (WF) constraints expressed in first-order graph logic with transitive closure and optionally (3) an initial model fragment. Existing approaches like [14–20] map the instance generation problem of consistent graph models into logic solvers such as Alloy [21, 22], SMT-solvers [23], SAT-solvers or constraint solvers when the efficiency of graph model generation depends on the scalability and performance of back-end logic solvers, which primarily excel in *finding inconsistencies in complex specifications*. However, the generation models as a side-effect of the proof construction is much less efficient.

In fact, these solvers guarantee neither scalability [24] nor diversity [25] when they need to generate well-formed graph instances of a specification – regardless of how smart the mapping is from a high-level graph model to the underlying logic solver. From a practical perspective, while the specification of complex industrial modeling tools may contain hundreds of classes (in their metamodel) and WF constraints, no existing model generation technique could derive a consistent graph that contains at least one object from each class.

Contribution. We propose a novel automatic generation technique to derive consistent domain-specific graph models for specifications by exploiting and innovatively combining a multitude of advanced graph-based and core SAT-solving techniques.

(1) We formulate model generation as a *refinement of partial models* [26, 27] where initial abstract model fragments are gradually refined and concretized during exploration.
(2) We provide *partial model refinement rules* as *decision* and *unit propagation* steps by following core SAT-solving techniques.
(3) We use incremental graph query evaluation of the VIATRA engine [28] to efficiently *evaluate violations of constraints over partial models* during model generation [27].
(4) We integrate *shape analysis as state encoding* [29–31] for graphs to efficiently detect if two partial models should be treated as equivalent during exploration.
(5) We exploit rule-based design space exploration [32] to *drive the generation process directly over graph shapes* using an objective function approximating the distance from a solution.
(6) We *evaluate the scalability of our approach* using 6 tests sets of four domains (including industrial DSLs) and compare its performance with the well-known Alloy Analyzer [21].

Added Value. To our best knowledge, our framework is one of the first attempts to automatically generate consistent models by operating natively over (typed and attributed) graphs. Moreover, according to our scalability experiments, it is capable of *generating consistent graph models of 1-2 orders of magnitude larger* (with 500-6000 nodes) compared to models derived by Alloy and the generated model suite is also more diverse [24]. As such, our graph solver can serve as a back-end where Alloy was used previously for model generation purposes in testing and benchmarking scenarios.

2 MODELING PRELIMINARIES

Our model generation technique will be illustrated by automatically generating test inputs for Yakindu Statecharts Tools [33], which is an industrial integrated modeling framework for developing reactive, event-driven systems. First, we give a brief introduction to the formal definition of partial models using a three-valued logic, which will drive the automated generation of consistent models.

2.1 Metamodels

A domain-specific (modeling) language (DSL) is typically defined by a *metamodel* and a set of *well-formedness constraints*. A metamodel defines the main concepts and relations in a domain, and specifies the basic graph structure of the models, and WF constraints further restrict valid models of the language by defining additional design rules. In this paper, the Eclipse Modeling Framework (EMF) [34] is used for domain modeling, which is a de facto industrial standard.

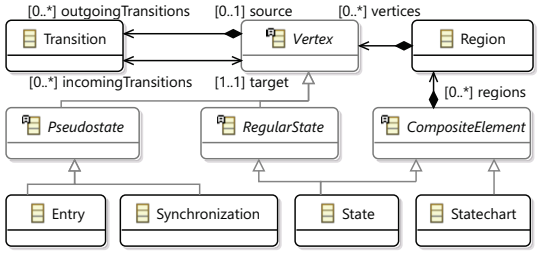


Figure 1: Metamodel of Yakindu statecharts

Example 2.1. A metamodel extracted from Yakindu is illustrated in Figure 1. A Statechart consists of Regions, which in turn contain Vertexes and Transitions. An abstract state Vertex is further refined into RegularStates (like State) and PseudoStates like Entry and Synchronization states. The source and target states of a transition are identified by the source and target references.

To reason about the consistency of DSLs, a formal algebraic specification is frequently created [19, 27, 35–38]. We briefly revisit the notation of [27] to provide a precise background for our model generation approach (but for space consideration, we omit the detailed handling of attributes, which could be introduced accordingly).

Formally, a metamodel defines a vocabulary $\Sigma = \{C_1, \ldots, C_n, exist, R_1, \ldots, R_m, \sim\}$ with unary predicate symbols C_i $(1 \leq i \leq n)$ defined for each *EClass*, and binary predicate symbols R_j $(1 \leq j \leq m)$ for each *EReference*. To represent abstract (partial) models, a unary *exist* predicate is introduced to denote the existence of an object in a given model, while $\sim$ denotes an equivalence relation over objects.

2.2 Partial Models

Partial models (PM) were introduced in [35, 38] to represent uncertain (possible) elements in instance models where one partial model represents a range of possible instance models. In this paper, 3-valued logic [39] is used to explicitly represent unspecified or unknown properties of models with a third $^1/_2$ truth value (besides 1 and 0 which stand for *true* and *false*) in accordance with [11, 27, 31]. A *partial model* is represented as a 3-valued logic structure $P = \langle Obj_P, \mathcal{I}_P \rangle$ of Σ, where Obj_P is the finite set of individuals in the model (i.e. the objects), and $\mathcal{I}_P$ provides a 3-valued interpretation for all constants in *Id* and predicate symbols in Σ.

Type Predicates. $\mathcal{I}_P$ is a 3-valued interpretation of each class symbol C_i in Σ: $\mathcal{I}_P(C_i) : Obj_P \rightarrow \{1, 0, ^1/_2\}$: 1, 0 and $^1/_2$ means that it is true, false or unspecified if an object is an instance of a class C_i.

Reference Predicates. $\mathcal{I}_P$ gives a 3-valued interpretation to each reference symbol R_j in Σ: $\mathcal{I}_P(R_j) : Obj_P \times Obj_P \rightarrow \{1, 0, ^1/_2\}$, where

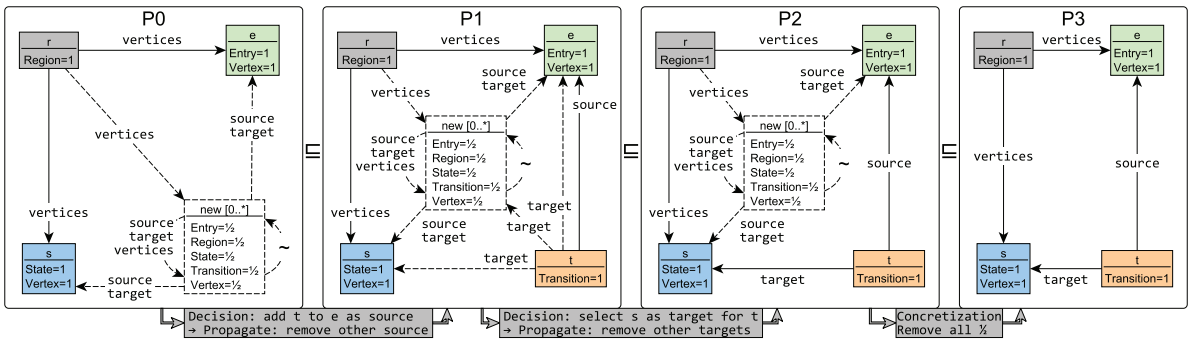


Figure 2: Sample partial models with uncertain elements and their refinement

1, 0 and ½ means that it is true, false or unspecified if there is a reference R_j between two objects.

Existence Predicate. $\mathcal{I}_P$ gives a 3-valued interpretation for the *existence predicate* $\mathcal{I}_P(exist) : Obj_P \rightarrow \{1, 0, ½\}$, where 1 values represent an object that must be, ½ value represents objects that may be included in a model.

Equivalence Predicate. $\mathcal{I}_P$ gives a 3-valued interpretation to the $\sim$ relation between the objects $\mathcal{I}_P(\sim) : Obj_P \times Obj_P \rightarrow \{1, 0, ½\}$. An uncertain ½ value relation between two objects means that those object may be equal and can potentially be merged. On the other hand, uncertain equivalence of a single object (with itself) implies that it may represent multiple separate objects.

Example 2.2. Four partial models are illustrated in Figure 2. As a notation guide, (1) the truth value of a *type predicate* is denoted by labels on nodes, where missing labels are treated as 0 values, while (2) *reference predicate* values 1 and ½ are represented by edges with solid and dashed lines (respectively), while missing edges between two objects represent 0 values for a predicate, (3) *existence predicate* values 1 and ½ are represented by nodes with solid and dashed borders, respectively, while objects with 0 existence values are simply not depicted. Finally, (4) uncertain ½ equivalences are marked by dashed line with an $\sim$ symbol. Otherwise, each node represents a single, unique object (i.e. for all object o: $[\![o \sim o]\!] = 1$ and for all different objects o_1 and o_2: $[\![o_1 \sim o_2]\!] = 0$).

In P_0 (on the left side of Figure 2), object r is of type Region but not of type State: $[\![\text{Region}(r)]\!]^{P_0} = 1$ and $[\![\text{State}(r)]\!]^{P_0} = 0$. In case of object *new*, all type predicate are ½, which means that the object may represent any type of objects. In P_0 there is a certain vertices reference between r and s and r and e, and a possible reference between r and *new* or as a self-loop of *new*. Nodes r,e and s represent objects that must exist, and *new* represent possible objects which may exist or they might be removed later from the model. Node *new* may also represent multiple objects (note the self-loop edge $\sim$), which can later be refined into multiple distinct model elements.

2.3 Refinement and Concretization of PMs

During model generation, the level of uncertainty will gradually be reduced by *refinements* deriving partial models that represent more concrete instance models. In a refinement step, properties with ½ values can be refined to either 0 or 1 which imposes an information ordering relation $X \sqsubseteq Y$ where either $X = ½$ and Y is refined to 1 or 0, or values of X and Y remain equal: $X \sqsubseteq Y := (X = ½) \vee (X = Y)$. A refinement is defined as a function $ref : Obj_P \rightarrow 2^{Obj_Q}$ which maps each object of a partial model P to a set of objects in the refined partial model Q. A refinement respects the information ordering of type, reference, equivalence and existence predicates for each $p_1, p_2 \in Obj_P$ and $q_1 \in ref(p_1)$, $q_2 \in ref(p_2)$:

- for each class C_i: $[\![C_i(p_1)]\!]^P \sqsubseteq [\![C_i(q_1)]\!]^Q$.
- for each reference R_j: $[\![R_j(p_1, p_2)]\!]^P \sqsubseteq [\![R_j(q_1, q_2)]\!]^Q$
- $[\![p_1 \sim p_2]\!]^P \sqsubseteq [\![q_1 \sim q_2]\!]^Q$
- $[\![exist(p_1)]\!]^P \sqsubseteq [\![exist(q_1)]\!]^Q$, and if $[\![exist(p)]\!]^P = 1$ then $ref(p)$ is not empty

Refinement from partial model P to Q is denoted by $P \sqsubseteq Q$.

If a 3-valued partial model P only contains 1 and 0 values, and there are no $\sim$ relations between different objects (i.e. all equivalent nodes are merged), then P represents a traditional *instance model*. A *concretization* defines a trivial refinement to a 2-valued fully defined concrete model by rewriting all ½ type, reference and exist predicates values to 0, and rewriting all ½ equivalence predicate to 0 between different objects, and to 1 on the same object.

Example 2.3. Figure 2 illustrates two refinement steps from P_0 to P_2. In P_1 object *new* is split into two different objects: *new* and t of P_1, where $[\![new \sim new]\!]^{P_0} = ½$ is refined to $[\![new \sim t]\!]^{P_1} = 0$, $[\![t \sim t]\!]^{P_1} = 1$ and $[\![new \sim new]\!]^{P_1} = ½$. Additionally, $[\![exist(new)]\!]^{P_0} = ½$ is refined to $[\![exist(t)]\!]^{P_1} = 1$, and $[\![\text{Transition}(new)]\!]^{P_0} = ½$ is refined to $[\![\text{Transition}(t)]\!]^{P_1} = 1$, thus creating a new Transition object t, while all other ½ type predicates are refined to 0. Finally, source predicates are refined to 1 with e as target, and to 0 with all other objects as target.

In step P_1 to P_2, possible target predicates are refined to 1 with s as target object, and to 0 with e and *new* as target objects. Note that refinement step $P_1 \sqsubseteq P_2$ will illustrate the decision rule while $P_2 \sqsubseteq P_3$ will illustrate the unit propagation rule later in section 3.1. P_3 is a concretization of P_2, which is also a refinement $P_2 \sqsubseteq P_3$.

2.4 Defining constraints over PMs

In many industrial modeling tools, domain-specific WF constraints are captured either by standard OCL constraints [40] or by graph patterns (GP) [28, 41]. A graph pattern captures complex structural conditions over an instance model. In order to have a unified and

$$
\begin{aligned}
[\![\mathsf{C}(v)]\!]^P_Z &:= \mathcal{I}_P(\mathsf{C})(Z(v)) \\
[\![\mathsf{R}(v_1,v_2)]\!]^P_Z &:= \mathcal{I}_P(\mathsf{R})(Z(v_1),Z(v_2)) \\
[\![exist(v)]\!]^P_Z &:= \mathcal{I}_P(exist)(Z(v)) \\
[\![v_1 \sim v_2]\!]^P_Z &:= \mathcal{I}_P(\sim)(Z(v_1),Z(v_2)) \\
[\![\varphi_1 \wedge \varphi_2]\!]^P_Z &:= min([\![\varphi_1]\!]^P_Z, [\![\varphi_2]\!]^P_Z) \\
[\![\varphi_1 \vee \varphi_2]\!]^P_Z &:= max([\![\varphi_1]\!]^P_Z, [\![\varphi_2]\!]^P_Z) \\
[\![\neg\varphi]\!]^P_Z &:= 1 - [\![\varphi]\!]^P_Z \\
[\![\exists v : \varphi]\!]^P_Z &:= max\{[\![exist(x) \wedge \varphi]\!]^P_{Z,v\mapsto x} : x \in Obj_P\} \\
[\![\forall v : \varphi]\!]^P_Z &:= min\{[\![\neg exist(x) \vee \varphi]\!]^P_{Z,v\mapsto x} : x \in Obj_P\} \\
[\![\varphi^+(v_1,v_2)]\!]^P_Z &:= [\![\varphi(v_1,v_2) \vee (\exists m : \varphi(v_1,m) \wedge \varphi^+(m,v_2))]\!]^P_Z
\end{aligned}
$$

Figure 3: Semantics of graph logic expressions

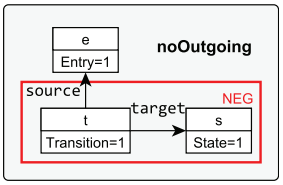
semantically precise handling of evaluating graph patterns constraints for regular and partial models, we use a first-order graph logic formalism with transitive closure that covers the key features of several concrete graph pattern languages. This semantics was introduced in [27] being influenced by [31].

Syntax. A graph pattern is a first order logic predicate with transitive closure $\varphi(v_1,\ldots,v_n)$ over (object) variables. A graph predicate φ can be inductively constructed by using object variable symbols $(v_1, v_2, \ldots)$, atomic predicates ($\mathsf{C}(v)$, $\mathsf{R}(v_1,v_2)$, $v_1 \sim v_2$), standard logic connectives ($\neg$, $\vee$, $\wedge$), logic quantifiers ($\exists$ and $\forall$), and transitive closure over binary predicates denoted as $\varphi^+(v_1,v_2)$.

Semantics. A predicate $\varphi(v_1,\ldots,v_n)$ can be evaluated on partial model P along a variable binding Z, which is a mapping $Z : \{v_1,\ldots,v_n\} \to Obj_P$ from variables to objects in M. The truth value of φ can be evaluated over a partial model P and Z (denoted by $[\![\varphi(v_1,\ldots,v_n)]\!]^P_Z$) in accordance with the semantic rules defined in Figure 3. Note that *min* and *max* takes the numeric minimum and maximum values of 0, 1/2 and 1, and the rules follow 3-valued interpretation of standard logic formulae as defined in [27, 31]. A variable binding Z where the predicate φ is evaluated to 1 over P is called a *pattern match*, formally $[\![\varphi]\!]^P_Z = 1$.

Graph predicates are frequently used for defining complex structural WF constraints and validation rules [28]. In such a case, *a predicate match denotes a constraint violation*, thus the corresponding graph formula needs to capture the erroneous cases, and a match detect a violation of a WF constraint. Therefore, a set of WF predicates $\{\varphi_1^{WF},\ldots,\varphi_n^{WF}\}$ defines a theorem of valid models $\mathcal{T}$, where $\mathcal{T} = \{\neg\varphi_1^{WF},\ldots,\neg\varphi_n^{WF}\}$. WF predicates of $\mathcal{T}$ are derived from two sources: the metamodel defines the basic structure, and additional validation constraints of a domain can be defined by using OCL [40] or graph patterns [28].

There are structural constraints imposed by the underlying graph representation. In case of EMF metamodels, such constraints include (1) *Type Hierarchy (TH)* which expresses that a more specific (child) class has every structural feature of the more general (parent) class, (2) *Type Compliance (TC)* that requires that for any relation $\mathsf{R}(o,t)$, its source and target objects o and t need to have compliant types, (3) *Abstract (ABS)*: If a class is defined as abstract, it is not allowed to have direct instances, (4) *Multiplicity (MUL)* of structural features can be limited with upper and lower bound in the form of "lower..upper" and (5) *Inverse (INV)*, which states that two parallel references of opposite direction always occur in pairs. Finally



$\mathsf{noOutgoing}(e) :=$
$\mathsf{Entry}(e) \wedge \neg\exists t_1, t_2, e_1, s :$
$\mathsf{source}(t_1, e_1) \wedge \mathsf{target}(t_2, s)$
$\wedge e \sim e_1 \wedge t_1 \sim t_2$

Figure 4: Sample statechart WF constraint as graph query

instance models in EMF are expected to be arranged into a (6) *Containment (CON)* hierarchy, which is a directed tree along relations marked in the metamodel as containment (e.g. regions or vertices). Since a formalization of these structural restrictions as WF constraints is provided in [19], the graph predicate language of Figure 3 can uniformly be used for both kinds of structural constraints.

Example 2.4. The Yakindu documentation states several constraints for statecharts that can be formalized as graph predicates [24]. For instance, constraint $\mathsf{noOutgoing}(e)$ in Figure 4 (depicted as a graph pattern and a graph predicate) detects an entry state e without an outgoing transition. The explicit use of equality constraints $e \sim e_1$ and $t_1 \sim t_2$ is responsible for performing a natural join operation over the edges as predicates. As a result, the same formula can be evaluated with both 2-valued and 3-valued interpretation.

2.5 Approximating constraints over PMs

While WF constraints can be directly evaluated on concrete instance models, checking the correctness of a partial model is a challenging task, because one partial model may represent multiple concretizations. A constraint φ can be evaluated on a partial model $\mathcal{P}$ using the 3-valued logic and open-world semantics by a constraint rewriting technique [27] which over- and under-approximates the results:

- **Under-Approximation**: If $[\![\varphi]\!]^P = 1$ in a partial model P, then $[\![\varphi]\!]^Q = 1$ in any partial model Q where $P \sqsubseteq Q$.
- **Over-Approximation** If $[\![\varphi]\!]^Q = 1$ in a partial model Q, then $[\![\varphi]\!]^P \geq 1/2$ in a partial model P where $P \sqsubseteq Q$.

Using these properties, we define a monotonous derivation sequence of valid partial models which (1) starts from the most abstract partial model where all constraints are evaluated to 1/2, which partial model (2) is gradually refined into more and more concrete partial models (with less number of predicates evaluating to 1/2). Refinement steps are continued until a concretized graph model of a designated scope eventually satisfies all WF constraints with 2-valued interpretation. During these refinement steps, special care needs to be taken to handle two situations:

- **False negatives** are partial models P which do not violate an under-approximated (must) constraint, i.e. $[\![\varphi]\!]^P \leq 1/2$, but they can never be concretized into a valid instance model with $[\![\varphi]\!]^Q = 0$ where $P \sqsubseteq Q$. These sequences are dead ends, and ideally, they should be detected as early as possible.
- **False positives** are partial models P which violate an over-approximated (may) constraint, i.e. $[\![\varphi]\!]^P \geq 1/2$, but they can be further refined into partial model $[\![\varphi]\!]^Q = 0$ where $P \sqsubseteq Q$. These sequences might be postponed during model generation, but they may eventually lead to a concrete model that satisfies all WF constraints, thus they should be kept.

Our graph generation approach will derive instance models along refinements. As such, partial models will gradually become more and more concrete after each refinement step which implies that checking WF constraints on partial models also becomes more precise. The practical benefit compared to consecutive calls to back-end solvers [24] is that the complex model finding problem can be divided into a sequence of small decisions while WF constraints can be checked (approximately) on intermediate solutions.

3 AUTOMATED GRAPH GENERATION

In this paper, we propose a general and automated graph model generation approach which takes a domain specified by (I_1) a vocabulary Σ defined by a metamodel, and (I_2) a theorem $\mathcal{T}$ defined by a set of well-formedness constraints $\{\neg\varphi_1^{WF}, \ldots, \neg\varphi_n^{WF}\}$, and ($I_3$) a search scope (i.e. minimal and maximal number of nodes in a solution graph) and (O) generates a consistent (valid) concrete graph model $G \models \mathcal{T}$ as output.

The model generation framework gradually refines partial models by rule-based design space exploration (DSE) [32] into a well-formed instance model which complies to the metamodel and all WF constraints are satisfied, if such a concrete model exists within the given search scope. During exploration, our framework simultaneously operates on (concrete) instance models and WF constraints as well as (abstract) partial models and approximated WF constraints introduced in section 2 by applying refinement rules. *Refinement rules* are defined as graph transformation rules [27, 42] manipulating directly over partial models with 3-valued interpretations by concertizing a single atomic uncertain ½ value in each step.

Refinement rules are grouped into two categories: (1) *Decision rules* are derived from Σ to reduce the number of valid concretizations of a partial model (i.e. new information is added) while (2) *Unit propagation rules* are derived from $\mathcal{T}$ to propagate the consequences of previous decisions in order to simplify a solution candidate without excluding potential solutions.

Model generation is initiated from an *initial partial model* provided as input by an engineer, or from the *most abstract partial model* where all predicates are unknown, i.e. (1) there is a single (abstract) object $Obj_P = \{new\}$; (2) $exist(new) = ½$ and $new \sim new = ½$ thus this object may represent multiple possible objects of the concrete models; (3) for all class predicate C_i: $C_i(new) = ½$; and (4) for all reference predicates R_j: $R_j(new, new) = ½$.

3.1 Decision Rules

Decision rules (see Figure 5) define various refinements to concretize information in partial models to construct possible solutions. They are derived from the vocabulary Σ of the metamodel, where each predicate symbol C_i and R_j represents a *Class* and *Reference*. In general, decision rules are responsible for (1) introducing new objects by splitting the abstract *new* object or (2) rewriting an ½ value to 1 as detailed by (the scheme of) four decision rule classes.

- Rule addRoot(C) selects a non-abstract class C (see the precondition in the left hand side) if no other roots have been created (denoted by NEG) to ensure that the model has a single root (as required by EMF). Its effect (prescribed by the right hand side) is to split the initial *new* object by creating a new *root* as an instance of C, which acts as a root element for the containment hierarchy and all self-loop references on *new* are extended to both objects.
- Rule addChild(C_P,R_C,C_C) selects an existing *parent* object of type C_P, the *new* object with a non-abstract type C_C, and a containment reference R_C from *parent* to *new*. Upon execution, it splits *new* into a new object *child* of type C_C connected to *parent* via R_C, thus unfolding a new object along the containment hierarchy. During the unfolding step, all outgoing R_i, incoming R_j and loop R_k references of *new* are extended (copied) to *child*.
- Finally, two rules addType(C) and addReference(R) refine uncertain ½ classes and references in the partial model. In the latter case, the rule requires that the types of the endpoints are already fixed appropriately.

Example 3.1. The refinement step $P_0 \sqsubseteq P_1$ in Figure 2 introduces a new object t by applying the *decision rule* addChild (of Figure 5), which changes ½ values of transition(t) and source(t, e) to 1.

3.2 Unit Propagation Rules

Unit propagation rules are responsible for refining unspecified elements in a partial model without excluding any valid solution to simplify the partial model propagating the consequences of previously applied decision rules. Unit propagation rules are applied repeatedly right after a decision rule is applied. They are derived from the structural constraints (1)-(6) introduced in section 2.4. In general, unit propagation rules rewrite ½ elements to 0 (or 1) if a 1 (or 0) value would contradict to a constraint. Figure 6 illustrates (the scheme of) unit propagation rules used in this paper.

- *Type hierarchy (TH)* is maintained by two rules: propTH$^+$(C,C_s) propagates a positive (1) type predicate of C to a supertype C_s; propTH$^-$(C′,C) rewrites a ½ type predicate value to 0 for type C if the object already has an incompatible type C′ where C and C′ do not have a common subclass (all classes are considered to be subclasses of themselves).
- *Type compliance (TC)* is checked by two rules: propTC^{From}(C,R) and propTC^{To}(R,C). Both rules remove possible references if the types of the reference end-points are incompatible.
- Rule propMUL^{Upper}(R) checks the *upper multiplicity (MUL)* of a reference, and removes all possible additional links if the upper limit is reached.
- The *Inverse (INV)* structural constraint is checked two rules: propINV$^+$(R,R_I) and propINV$^-$(R,R_I) which set a R predicate value to 1 or 0 if an inverse R_I value is already set.
- To ensure *Containment hierarchy (CON)*, first decision rules enforce that each non-root object has a parent. Then, additional possible incoming containment references are removed by unit propagation rule propCON^{2Parent}(R,R_C). Finally, propCON^{Loop}(R) removes possible reference predicates that would create a loop in the containment hierarchy.

Decision and unit propagation rules are in close analogy with the DLL62 algorithm of traditional SAT-solvers [43]: values of variables are graph elements in our case (instead of Boolean values), complex graph predicates are evaluated (instead of conjunctive normal formulae), and the state space of graphs needs to be continuously stored during exploration (instead of a search tree).

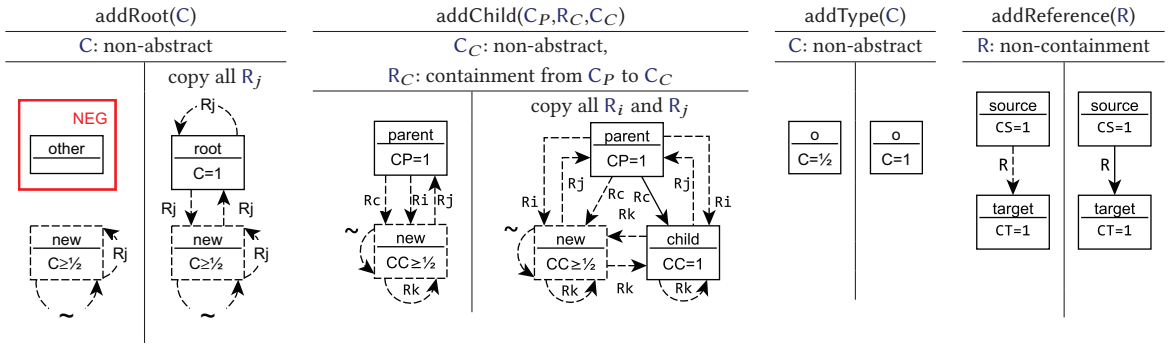


Figure 5: Decision rules for graph model generation

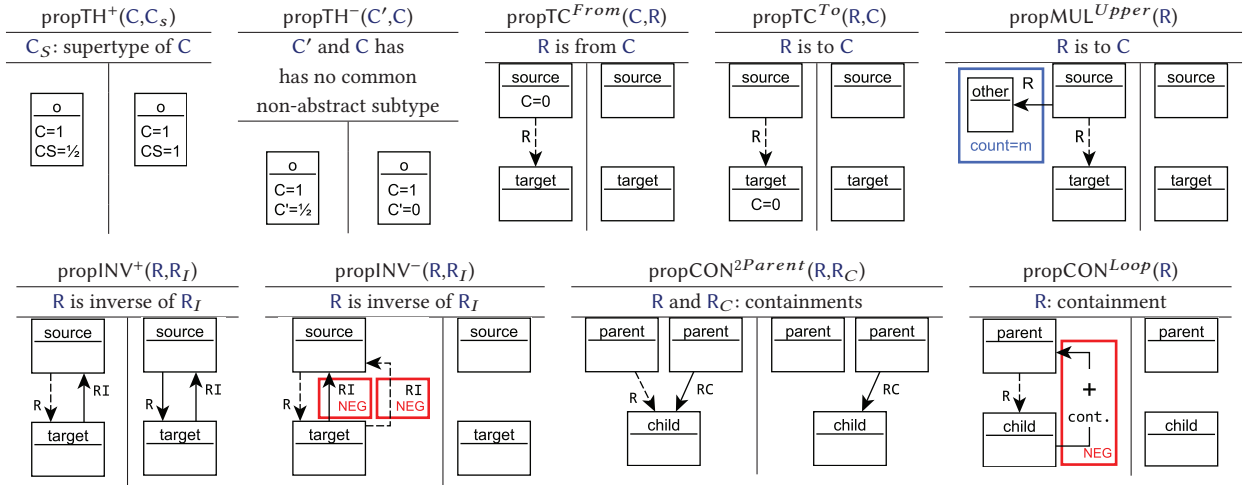


Figure 6: Unit propagation rules for graph model generation

Example 3.2. Refinement step $P_1 \sqsubseteq P_2$ is a result of a *decision rule* addChild to set the ½ value of target(t, s) to 1 followed by a *unit propagation rule* propMUL^{Upper}(target) which aims to prevent the partial model from violating an upper multiplicity constraint by setting ½ values of target(t, e) and target(t, new) to 0 as a direct consequence of the previous decision rule.

As a preprocessing step, decision and propagation rules are derived. Moreover, *under-approximated (must) predicates* are synthesized from graph predicates in accordance with [27] to detect unresolvable WF constraints early in a partial solution.

3.3 Exploration

During exploration (see Figure 7), refinement rules are repeatedly applied driven by an objective function and a rule selection strategy. As such, the size of partial models is continuously growing up to the designated scope, while the number of uncertainties and constraint violations in these partial models are decreasing to ensure that the process converges to consistent instance models. Now we discuss the steps of the model generation process.

(1) After initializing the search with an initial partial model, *an unexplored decision rule is selected and applied* to derive a new (refined) partial model along a partial model refinement step (section 2.3). The role of these refinement rules is in direct analogy with the decision steps in SAT solvers.

(2) After executing a decision rule, our framework *executes all possible unit propagation rules on the partial solution* to propagate the consequence of the decision, thus further refining the partial model. This step is again in direct analogy with SAT solvers, but it is carried out by incremental, change-driven model transformation rules [44] to improve efficiency.

(3) To prevent traversing the same (graph) state twice, a *state code* is calculated and stored for the new partial model by using graph isomorphism checks over *graph shapes* [30]. Graph shapes abstract from node identities, but they efficiently identify if two graphs can be distinguished by the neighborhood (i.e. incoming and outgoing edges) of a node.

(4) Had the new state been already explored, the partial model is dropped and a new refinement rule is applied **(1)**.

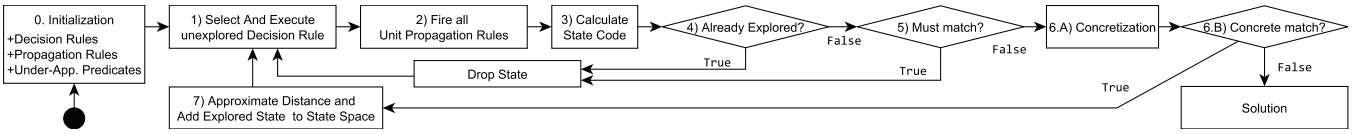


Figure 7: Exploration strategy for automated model generation

(5) If a new partial model is reached, then our framework *checks if it satisfies all under-approximated (must) constraints* by partial evaluation of these constraints [27] using an incremental graph query engine [28]. If an under-approximated constraint is violated by the partial model then an inconsistency is detected, thus the partial model can never be refined into a well-formed instance model so it can be dropped **(1)**.

(6) Next the partial model is *concretized into an instance model* by removing all uncertainties and all the original WF constraints are checked on this candidate model by the incremental graph query engine to detect inconsistencies. If no violations are found, then the instance model is stored as a solution, and the exploration may terminate (if designated model scope is reached) or continue to find other solutions **(1)**. Note that checking the original WF constraints on a concretized model guarantees the correctness of our solver.

(7) Finally, our framework approximates the *distance for a solution* by an objective function, adds the current partial model to the exploration path and continues refinement from with new unexplored decision refinement. For each partial model, this objective function is calculated as the sum of constraint violations and the number of missing objects wrt. a designated size.

For selecting the next match where a decision rule is to be applied, we use a combined exploration strategy with *best-first search* heuristic, *backtracking*, *backjumping* and *random restarts* in an advanced design space exploration framework [32]. The search selects the best candidate wrt. the objective function, then it randomly fires (with uniform distribution) an enabled decision rule, subsequently, it fires all possible unit propagation rules. If no further decision rules can be applied, then it backtracks to continue along the previous partial model candidate. At each step, the exploration may backjump to the best model candidate found so far during the exploration. Finally, the search is occasionally restarted from a randomly chosen intermediate model candidate. Such a random restart is a common technique in SAT-solvers to avoid local optima.

Our framework operates directly over graph models and the exploration itself is driven on such a high-level. Thus, it *combines the advantages of multiple advanced graph-based techniques with core SAT-solving techniques* to tackle the scalability problems of existing mapping-based approaches:

- The approximated and the original of WF constraints are efficiently evaluated over partial models (in **(5)** and **(6.B)**) using incremental graph query evaluation techniques [28].
- Refinement rules are divided into decision **(1)** and unit propagation rules **(2)** as facilitated by core SAT-solving algorithms.
- Isomorphic states are detected during exploration **(3)** by combining shape analysis techniques [30, 31].
- Our framework has full control over the graph generation process **(1,7)** via rule-based DSE techniques [32].

3.4 Soundness and Completeness

Starting from the abstract initial model, our decision rules guarantee the *completeness of graph model generation* [11], i.e. any concrete graph can possibly be derived within a bounded scope: Rule addRoot can derive any root objects, addChild is responsible for deriving new elements along the containment tree, addType can assign any types to an object while addReference can build any graphs by adding edges. However, efficient strategies are needed to initiate the execution of decision rules in proper order.

The *soundness of graph model generation* is guaranteed by checking all WF constraints on each generated candidate graph instance. As such we obtain at least the same guarantee as provided by Alloy, namely, if a consistent graph model exists within a given scope then our approach will derive it.

If no well-formed models are found within a given scope then Alloy provides no further details (stating that the problem *may* be inconsistent). In contrast, if the search is terminated in our approach with no valid refinements (i.e. all branches of the state space are closed) then the specification is *surely* inconsistent. This is primarily ensured by the fact that if an under-approximated predicate φ has a match in a partial solution it cannot be refined to a valid model. Furthermore, we guarantee that any well-formed model (of a larger scope) can only be derived by refining existing partial models at the edge of the search horizon, i.e. our approach can incrementally continue exploration for a larger scope.

3.5 Strengths and Limitations

Our approach operates on connected sparse graphs with edges as relations (i.e. no edge identities and no parallel edges of a type) as underlying data model, which is less expressive than full relational algebra in case of Alloy. As a current technical limitation, our graph generation approach is showcased for EMF metamodels and models, which are widely used in industrial modeling tools, but it could be easily adapted to other graph formalisms. The expressive power of graph predicates used for capturing WF constraints is equivalent to first order logic with transitive closure over binary predicates.

Our solver efficiently handles complex structural graph constraints defined in first order logic with transitive closure. However, it includes only enumerations as attribute values but excludes strings, integers, etc. Such attribute values could be handled in the future by calling external solvers (e.g. SMT-solvers) during the exploration or as a post-processing step.

While the decision procedure of our graph solver provides stronger completeness guarantees than Alloy within a bounded scope, it does not provide an unsatisfiable core (i.e. minimal contradictory set of formulae) to highlight contradiction between WF constraints, which is supported by many SAT and SMT-solvers.

4 EXPERIMENTAL EVALUATION

We carried out an experimental evaluation of generating consistent instance models to address the following research questions:

RQ1 How does our graph solver scale (in time and model size) when generating consistent models of increasing size?

RQ2 How does our approach scale (in time and model size) compared to the widely used model finder Alloy [21]?

RQ3 How do the different steps of the exploration influence performance of the graph solver?

Selected DSLs for evaluation. As model generation for DSLs still lacks systematically constructed performance benchmarks, we evaluated our approach in the context of 6 test sets of four different domains. First (1) a small File System (*FS*) example was taken from the Alloy documentation [45]. *Ecore*, the meta-metamodeling language of EMF [34], has been used as a case study by different approaches [14, 16, 19, 24, 46, 47] using Alloy as a background solver for model generation purposes. Our measurements also cover two DSLs that were developed in industrial projects, namely, (3) *Yakindu* [33] and (4) Functional Architecture Model (*FAM*) developed for avionics [48]. Due to their complexity, domains (3) and (4) are split into two cases: we generate models first with only general metamodel constraints (*w/o WF*), and then in the presence of extra WF constraints (*with WF*). In addition to their direct practical relevance, these DSLs have already been used in the context of model generation in numerous papers [19, 24, 49] in the past.

Benchmarking setup. To measure scalability, we set up a timeout of 3 minutes for each model generation run with increasing model size. For each measurement point, model generation was executed 30 times and the median of the runs were taken. To account for warm-up effects and memory handling of the Java 8 virtual machine, we added an extra 20 runs before the actual measurements and called the garbage collector explicitly between runs. As a baseline of comparison, Alloy Analyzer V4.2 (the latest stable version available at the Alloy download site) was used with two underlying SAT solver libraries: Sat4J (default in Alloy) and MiniSAT (recommended by Alloy). All measurements were executed on an average desktop computer[1] with 12 GB heap size.

Experimental results. For **RQ1** we evaluate the execution time of our approach for the four domains by increasing the target model size from 50 to 500 objects (with a step size of 50 new objects), and measuring (in Figure 8a–Figure 8d) the total execution time. As a key observation, our approach is able to generate consistent models with 500 elements for all four domains within 10 seconds for FAM and FS, within 40 seconds for Ecore and 3 minutes for Yakindu. As a stress test, we also managed to generate even larger consistent models (1000 objects for Yakindu, 7000 objects for FAM, 4750 objects for FS and 2000 objects for Ecore, see Figure 8k) in 20 minutes (as a median of 10 measurements).

For **RQ2**, we compare model generation time of our Graph Solver with Alloy for small model sizes (from 5 to 50 objects, step size of 5 new objects) for the 6 test cases. As a baseline, we use a state-of-the-art EMF-to-Alloy mapping technique [16, 19, 24, 46, 47] and tool to obtain Alloy specifications for the Yakindu, FAM and Ecore domains, and the original Alloy specification is used for FS. According to the results (see Figure 8e–8j and Figure 8k), our approach scales much better as it generates models 1-2 orders of magnitude larger than Alloy could handle regardless of the back-end SAT solver which only had little impact on scalability. This is in line with previous measurements for Alloy in [19, 24]). Alloy dominantly ran out of memory when mapping input specification into a SAT problem which results in over 6 million variables and several million clauses when aiming to generate a model with 40-90 objects.

Note that the Alloy Analyzer is not primarily targeted to generate models but to check the consistency of a relational specification within a given scope and synthesize small counterexamples. In fact, Alloy had a smaller runtime for very small models, thus the warm-up cost of our graph solver is higher. However, our graph solver is able to generate much larger graph models even for all four domains with similar consistency guarantees as Alloy.

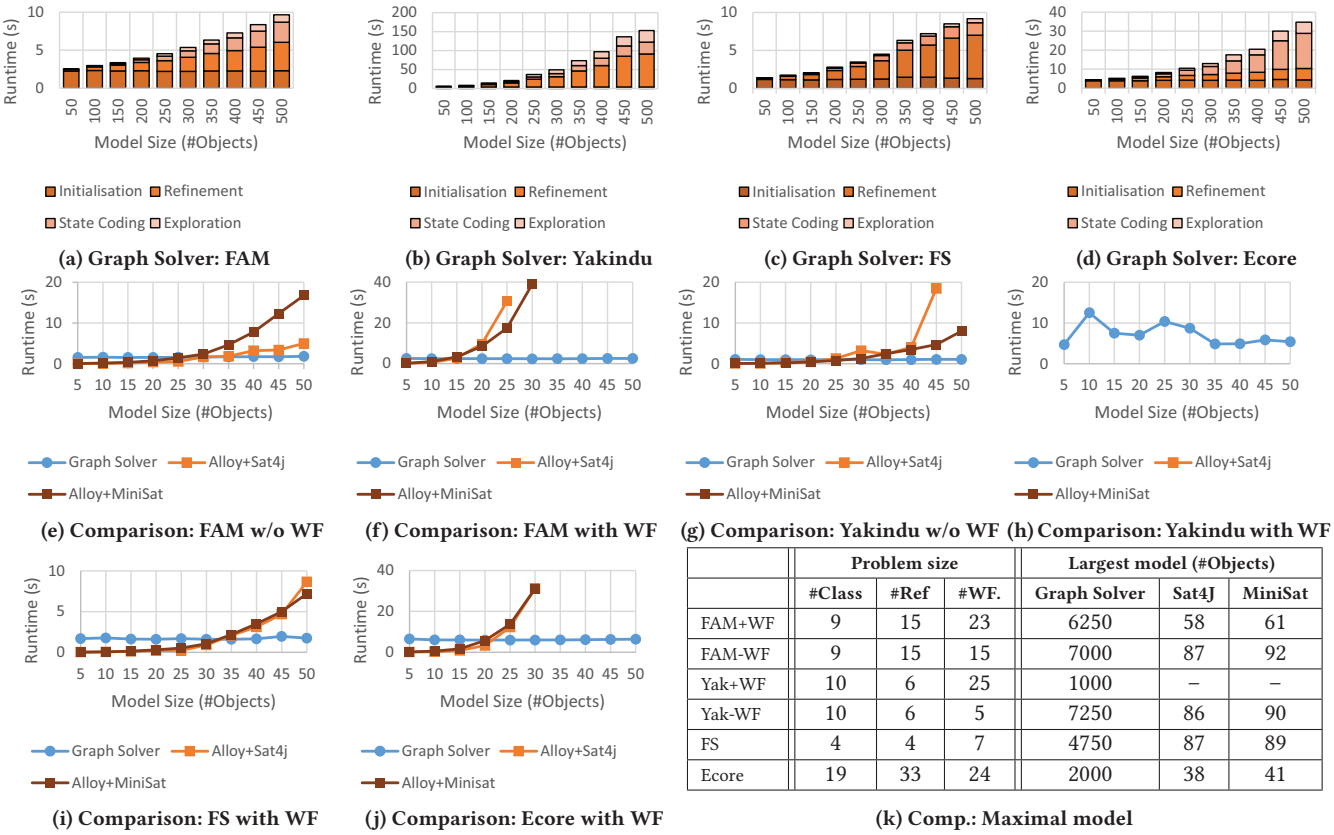
For **RQ3**, we also measured (in Figure 8a–Figure 8d) how much time is spent in the different phases of model generation by our graph solver (see Figure 7) such as initialization, partial model refinement, state encoding and exploration. The preprocessing phase (1.5 seconds for FS, 4 seconds for Ecore, 2 seconds for FAM and 4 seconds for Yakindu) is a one-time penalty which is is proportional to the complexity of the metamodel and the WF constraints, thus we expect it to be negligible for model generation in case of other domains. Refinement is the dominant phase in the Yakindu and FS cases, while state encoding is dominant for Ecore. These results show that future research should primarily improve on refinement by providing a better transformation engine or refinement rules.

Quality of generated models. The quality of the generated models can be investigated from different aspects. To ensure *correctness*, all WF constraints were checked on each generated graph instance by using an external tool, the VIATRA graph query engine [50]. To assess *diversity* when a sequence of models are generated by the proposed graph solver (within similar scope), each model is guaranteed to be non-isomorphic by the state codes (Step **(4)** in Figure 7) or by a distance metric [51] in case of repeated calls to the solver, which offers increased diversity compared to Alloy [51]. Systematically assessing the *realistic* nature of models is a more complex task [13] which necessitates to obtain a large set of real models authored by engineers. Our graph solver ensures that only enumeration values can be isolated nodes in a graph otherwise all graphs are connected by default, i.e. all regular nodes are arranged into a containment hierarchy. In order to assure connectedness, Alloy requires an extra constraint to capture this concept, thus by default, our solution appears to be more realistic. In addition, [11, 13] contain an in-depth investigation of realistic models for the Yakindu domain. All generated models are available at [52].

Threats to validity. In order to strengthen *internal validity*, our experiments include an extensive warm-up phase prior to the actual measurements to decrease the fluctuation of runtime results caused by the JVM (instead of the natural fluctuation of solver runtimes). We used default setups for running Alloy and our graph solver, i.e. no extra hints and performance optimizations were provided in the two approaches. Domain-specific fine tunings may improve scalability in some cases but it would simultaneously decrease the general-purpose nature of these solvers.

To address *external validity*, our measurements cover 6 test cases including 3 industrial domains (Ecore, Yakindu, FAM) with *complex*

[1] CPU: Intel Core-i5-m310M, MEM: 16GB, OS: Windows 10 Pro.



	Problem size			Largest model (#Objects)		
	#Class	#Ref	#WF.	Graph Solver	Sat4J	MiniSat
FAM+WF	9	15	23	6250	58	61
FAM-WF	9	15	15	7000	87	92
Yak+WF	10	6	25	1000	–	–
Yak-WF	10	6	5	7250	86	90
FS	4	4	7	4750	87	89
Ecore	19	33	24	2000	38	41

(k) Comp.: Maximal model

Figure 8: Runtime comparison with increasing model size and distribution of generation time

structural WF constraints, thus our experimental scalability results for our graph solver are likely generalizable to other domains of similar size and complexity within the limitations of section 3.5. In case of simple WF constraints, the difference between the performance characteristics of Alloy and our graph solver may be smaller. Since the performance of Alloy depends on the backend SAT-solver, our measurements already included two state-of-the-art solvers (SAT4J and MiniSAT). Thus the large scalability difference in the size of generated models can likely be attributed to our graph solver.

Summary. Our graph solver provides a strong platform for generating consistent graph models which are 1-2 orders of magnitude larger (with similar or higher quality) than derived by mapping based approaches using Alloy with an underlying SAT-solver. Such a difference in scalability can only partly be dedicated to our conceptually different approach which combines several advanced graph techniques to improve performance instead of fine-tuning a mapping. However, it likely indicates fundamental shortcomings of existing mapping based approaches. Based on in-depth profiling we suspect that representing each potential edge between a pair of nodes as a separate Boolean variable blows up the state space for sparse graph with only linear number of edges. Moreover, SAT-solvers have major problems in evaluating complex predicates over larger graph models [11] where graph query evaluation was particularly efficient [28].

		Logic Solvers	Uncertain Models	Rule-Based	Iterative	Symbolic
Inputs	Partial Snapshot	+	++	-	+	-
	Local Constraints	+	-	+	+	+
	Global Constraints	+	-	-	+	+
Outputs	Metamodel	+	+	+	+	+
	Well-formed	+	-	-	+	+
	Scalable	-	-	++	+/-	-
	Decidability	-	+	+	-	+/-

Table 1: Comparison of related approaches

5 RELATED WORK

We compare our solution with existing model generation techniques with respect to the characteristics of *inputs* and *output results* in Table 1. As for *inputs*, the model generation can be (1) initiated from a *partial snapshot*. Additionally, an approach may support (2) *local* and (3) *global constraints* as WF constraints: a local constraint accesses only the attributes and the outgoing references of an object, while a global constraint specifies a complex structural pattern. Local constraints are frequently attached to objects (e.g. in UML class diagrams), while global constraints are widely used in DSLs. As *outputs*, the generated models may (i) be *metamodel-compliant* (ii) satisfy all *well-formedness* constraints of the language. We consider a technique (iv) *scalable* if there is no hard limit on the model size (as demonstrated in the respective papers). Finally, a model generation

approach may be (v) *decidable* which always terminates with a result. Our comparison excludes approaches like which do not guarantee metamodel- compliance of generated instance models.

Logic Solver Approaches. Several approaches map a model generation problem into a logic problem, which is solved by underlying SAT/SMT-solvers. Complete frameworks with standalone specification languages include Formula [35] (which uses Z3 SMT-solver [23]), Alloy [36] (which relies on SAT solvers like Sat4j[53]) and Clafer [20] (using backend reasoners like Alloy).

There are several approaches aiming to validate standardized engineering models enriched with OCL constraints [54] by relying upon different back-end logic-based approaches such as constraint logic programming [17, 37, 55], SAT-based model finders (like Alloy) [14, 16, 19, 24, 46, 47, 56], CSP solvers [49] first-order logic [57], constructive query containment [58], higher-order logic [59, 60], or rewriting logics [61]. Partial snapshots and WF constraints can be uniformly represented as constraints [19]. Growing models are supported in [15, 24] for a limited set of constraints.

Scalability of all these approaches are limited to small models / counter-examples. Furthermore, these approaches are either a priori bounded (where the search space needs to be restricted explicitly) or they have decidability issues. As our approach is independent from the actual mapping of constraints to logic formulae, it could potentially be integrated with most of the above techniques by complementing or replacing the back-end solvers.

Uncertain Models. Partial models are similar to uncertain models, which offer a rich specification language [38, 62] amenable to analysis. They a more 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 [63], or refined by graph transformation rules [26]. Each concrete model is derived in a single step, thus their approach is not iterative like ours. Scalability analysis is omitted from these papers, but refinement of uncertain models is always decidable, thus termination is guaranteed.

Approaches like [64] analyze possible matches 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 with [27]. As a key difference, our approach carries out model refinement while simultaneously evaluating graph query evaluation in an automated process.

Rule-based Instance Generators. A different class of model generators relies on rule-based synthesis driven by randomized, statistical or metamodel coverage information for testing purposes [65–67]. Some approaches support the calculation of effective metamodels [68], but partial snapshots are excluded from input specifications. Moreover, WF constraints are restricted to local constraints evaluated on individual objects while global constraints of a DSL are not supported. On the positive side, these approaches guarantee the diversity of models and scale well in practice [67, 69].

Iterative Approaches. Iterative approaches generate models by multiple solver calls. In [24] models are generated in by calling Alloy in multiple steps, where each step extends the instance model by a few elements. This approach scaled up to 50 object in 45s for generating valid Yakindu Statecharts. An iterative approach is proposed *specifically for allocation problems* in [70] based on Formula. Models are generated in two steps to increase diversity of results by first creating non-isomorphic submodels from an effective metamodel fragment followed by a problem-specific symmetry-breaking predicate [71] to ensures that no isomorphic models are generated twice while constraint checks are postponed to the very final stage. An iterative, counter-example guided synthesis is proposed for higher-order logic formulae in [22], but the size of derived models is fixed and smaller than 50 objects.

Symbolic Model Generation Technique Certain techniques use abstract (or symbolic) graphs for analysis purposes. A tableau-based reasoning method is proposed for graph properties [72–74], which automatically refine solutions based on well-formedness constraints, and handle state space in the form of a resolution tree. As a key difference, our approach refines possible solutions in the form of partial models, while [72, 73] resolves the graph constraints to a concrete solution. Therefore our approach is able to exploit efficient graph query engines to evaluate partial solutions, while those techniques are demonstrated on small (< 10 objects) graphs or with no scalability evaluation at all.

Additionally, different approaches use abstract interpretation [29, 30], or predicate abstraction [31] 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.

6 CONCLUSION AND FUTURE WORK

We presented a novel graph solver to generate consistent models of a designated size from a specification defined by a metamodel and a set of WF constraints. Unlike existing approaches which map the model generation problem to logic solvers (dominantly SAT or SMT-solvers), we address the model generation problem of consistent instances directly over graphs by combining advanced graph-based techniques with core SAT-solving rules. Our approach is fully automated and available as an open source tool [52].

Our experimental evaluation carried out over three industrial domains confirmed that our solver is able to synthesize consistent graph models with over 500-6000 objects with similar quality guarantees as provided by the popular relational model finder Alloy. The scalability of our solver is 1-2 orders of magnitude better than existing mapping based approaches using Alloy with a SAT-solver in the background. Such a difference in scalability likely indicates not only the benefits of our approach but also the inherent problems of mapping based model generation approaches deriving and solving a SAT problem. Thus our solver can serve as the output of mappings that previously used Alloy for model generation purposes. Altogether, our technique has the potential to be used in many testing scenarios including validation of large industrial DSLs, but its scalability is not yet sufficient for benchmarking purposes.

We have no precise claims on the diversity and realistic nature of our model generator. In the future, we aim to extend the framework to synthesize a set of graph models which are consistent, diverse and realistic at the same time. Numerous studies [24, 25, 51] have demonstrated that neither traditional SAT-solvers nor SMT-solvers provide sufficient diversity for their outcome when called repeatedly.

REFERENCES

[1] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid, "Korat: A tool for generating structurally complex test inputs," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 771–774. [Online]. Available: https://doi.org/10.1109/ICSE.2007.48

[2] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, 2001, p. 22. [Online]. Available: https://doi.org/10.1109/ASE.2001.989787

[3] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat, "gMark: Schema-driven generation of graphs and queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 4, pp. 856–869, 2017.

[4] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik, "A concept for testing robustness and safety of the context-aware behaviour of autonomous systems," in *KES-AMSTA*, ser. LNCS, vol. 7327. Springer, 2012, pp. 504–513.

[5] J. Härtel, L. Härtel, and R. Lämmel, "Test-data generation for Xtext," in *SLE*, 2014, pp. 342–351.

[6] V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser, "Towards an automation of the mutation analysis dedicated to model transformation," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 653–683, 2015.

[7] S. Ali, M. Z. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from OCL constraints with search techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 10, pp. 1376–1402, 2013.

[8] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró, "The Train Benchmark: cross-technology performance evaluation of continuous model queries," *Softw. Syst. Model.*, 2017.

[9] Special Committee 205 of RTCA, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," 2011.

[10] ISO, "Road vehicles – Functional safety," 2011.

[11] D. Varró, O. Semeráth, G. Szárnyas, and Ákos Horváth, "Towards the automated generation of consistent, diverse, scalable and realistic graph models," in *Graph Transformation, Specifications, and Nets (In Memory of Hartmut Ehrig)*. Springer LNCS 10800, 2018.

[12] W. van Leeuwen, A. Bonifati, G. H. L. Fletcher, and N. Yakovets, "Stability notions in synthetic graph generation: a preliminary study," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, 2017, pp. 486–489. [Online]. Available: https://doi.org/10.5441/002/edbt.2017.51

[13] G. Szárnyas, Z. Kővári, Á. Salánki, and D. Varró, "Towards the characterization of realistic models: Evaluation of multidisciplinary graph metrics," in *MODELS*, 2016.

[14] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Software and Systems Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[15] E. K. Jackson and J. Sztipanovits, "Constructive techniques for meta-and model-level reasoning," in *Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 405–419.

[16] M. Kuhlmann, L. Hamann, and M. Gogolla, "Extensive validation of OCL models by integrating SAT solving into use," in *TOOLS'11 - Objects, Models, Components and Patterns*, ser. LNCS, vol. 6705, 2011, pp. 290–306.

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

[18] F. Büttner, M. Egea, and J. Cabot, "On verifying ATL transformations using 'off-the-shelf' SMT solvers," in *Proc. of the 15th Int. Conf. on MODELS*, ser. LNCS, vol. 7590, 2012.

[19] O. Semeráth, A. Barta, A. Horváth, Z. Szatmári, and D. Varró, "Formal validation of domain-specific languages with derived features and well-formedness constraints," *Software and Systems Modeling*, vol. 16, no. 2, pp. 357–392, 2017.

[20] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski, "Clafer: unifying class and feature modeling," *Software & Systems Modeling*, vol. 15, pp. 811–845, 2016.

[21] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[22] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *37th IEEE/ACM Int. Conf. on Software Engineering, ICSE*, 2015, pp. 609–619.

[23] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[24] O. Semeráth, A. Vörös, and D. Varró, "Iterative and incremental model generation by logic solvers," in *19th International Conference on Fundamental Approaches to Software Engineering*, ser. LNCS, no. 9633. Springer-Verlag, 2016, pp. 87–103.

[25] E. K. Jackson, G. Simko, and J. Sztipanovits, "Diversely enumerating system-level architectures," in *Proceedings of the 11th ACM Int. Conf. on Embedded Software*. IEEE Press, 2013, p. 11.

[26] R. Salay, M. Chechik, M. Famelis, and J. Gorzny, "A methodology for verifying refinements of partial models," *Journal of Object Technology*, vol. 14, no. 3, pp. 3:1–31, 2015.

[27] O. Semeráth and D. Varró, "Graph constraint evaluation over partial models by constraint rewriting," in *Theory and Practice of Model Transformation - 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings*, 2017, pp. 138–154.

[28] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró, "EMF-IncQuery: An integrated development environment for live model queries," *Sci. Comput. Program.*, vol. 98, pp. 80–99, 2015.

[29] A. Rensink and D. Distefano, "Abstract graph transformation," *Electronic Notes in Theoretical Computer Science*, vol. 157, no. 1, pp. 39–59, 2006.

[30] A. Rensink, "Canonical graph shapes," in *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, 2004, pp. 401–415.

[31] T. W. Reps, M. Sagiv, and R. Wilhelm, "Static program analysis via 3-valued logic," in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 15–30.

[32] Á. Hegedüs, Á. Horváth, and D. Varró, "A model-driven framework for guided design space exploration," *Automated Software Engineering*, vol. 22, no. 3, pp. 399–436, 2015.

[33] Yakindu Statechart Tools, *Yakindu*, 2017, http://statecharts.org/.

[34] The Eclipse Project, *Eclipse Modeling Framework*, 2017, //www.eclipse.org/emf.

[35] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Reasoning about metamodeling with formal specifications and automatic proofs," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 653–667.

[36] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[37] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming," in *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. New York, NY, USA: ACM, 2007, pp. 547–548.

[38] M. Famelis, R. Salay, and M. Chechik, "Partial models: Towards modeling and reasoning with uncertainty," in *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 573–583.

[39] S. C. Kleene, N. De Bruijn, J. de Groot, and A. C. Zaanen, *Introduction to metamathematics*. van Nostrand New York, 1952, vol. 483.

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

[41] U. Nickel, J. Niere, and A. Zündorf, "The FUJABA environment," in *Proceedings of the 22nd Int. Conf. on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, 2000, pp. 742–745.

[42] H. Ehrig, G. Rozenberg, and H.-J. rg Kreowski, *Handbook of graph grammars and computing by graph transformation*. world Scientific, 1999, vol. 3.

[43] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[44] G. Bergmann, I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," *Software & Systems Modeling*, vol. 11, no. 3, pp. 431–461, 2012.

[45] "Alloy online tutorial," 2017. [Online]. Available: http://alloy.mit.edu/alloy/tutorials/online/

[46] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL transformations using transformation models and model finders," in *14th International Conf. on Formal Engineering Methods,ICFEM'12*. LNCS 7635, Springer, 2012, pp. 198–213.

[47] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using boolean satisfiability," in *Design, Automation and Test in Europe, (DATE'10)*. IEEE, 2010, pp. 1341–1344.

[48] Á. Hegedüs, Á. Horváth, I. Ráth, R. R. Starr, and D. Varró, "Query-driven soft traceability links for models," *Software & Systems Modeling*, vol. 15, no. 3, pp. 733–756, 2016.

[49] C. A. González, F. Büttner, R. Clarisó, and J. Cabot, "Emftocsp: a tool for the lightweight verification of EMF models," in *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, 2012, pp. 44–50. [Online]. Available: https://doi.org/10.1109/FormSERA.2012.6229788

[50] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Software and System Modeling*, vol. 15, no. 3, pp. 609–629, 2016. [Online]. Available: https://doi.org/10.1007/s10270-016-0530-4

[51] O. Semeráth and D. Varró, "Iterative generation of diverse models for testing specifications of dsl tools," in *21st International Conference on Fundamental Approaches to Software Engineering*, ser. LNCS. Springer-Verlag, 2018.

[52] Viatra Solver Project, 2018. [Online]. Available: https://github.com/viatra/VIATRA-Generator

[53] D. Le Berre and A. Parrain, "The sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010.

[54] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software and Systems Modeling*, vol. 4, pp. 386–398, 2005.

[55] F. Büttner and J. Cabot, "Lightweight string reasoning for OCL," in *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Lyngby, Denmark, July 2-5, 2012. Proceedings*, ser. LNCS, A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. S. Kolovos, Eds., vol. 7349. Springer, 2012, pp. 244–258.

[56] S. M. A. Shah, K. Anastasakis, and B. Bordbar, "From UML to Alloy and back again," in *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. ACM, 2009, pp. 1–10.

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

[58] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, "OCL-Lite: Finite reasoning on UML/OCL conceptual schemas," *Data Knowl. Eng.*, vol. 73, pp. 1–22, 2012.

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

[60] H. Grönniger, J. O. Ringert, and B. Rumpe, "System model-based definition of modeling language semantics," in *Formal Techniques for Distributed Systems*, ser. LNCS, vol. 5522. Springer, 2009, pp. 152–166.

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

[62] R. Salay and M. Chechik, "A generalized formal framework for partial modeling," in *Fundamental Approaches to Software Engineering*, ser. LNCS, A. Egyed and I. Schaefer, Eds. Springer Berlin Heidelberg, 2015, vol. 9033, pp. 133–148.

[63] R. Salay, M. Famelis, and M. Chechik, "Language independent refinement using partial modeling," in *Fundamental Approaches to Software Engineering*, ser. LNCS, J. de Lara and A. Zisman, Eds. Springer Berlin Heidelberg, 2012, vol. 7212, pp. 224–239.

[64] M. Famelis, R. Salay, A. Di Sandro, and M. Chechik, "Transformation of models containing uncertainty," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2013, pp. 673–689.

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

[66] F. Fleurey, J. Steel, and B. Baudry, "Validation in model-driven engineering: Testing model transformations," in *International Workshop on Model, Design and Validation*, Nov 2004, pp. 29–40.

[67] S. Ali, M. Z. Iqbal, M. Khalid, and A. Arcuri, "Improving the performance of OCL constraint solving with novel heuristics for logical operations: a search-based approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2459–2502, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9392-6

[68] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel, "Meta-model Pruning," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Denver, Colorado, USA, Oct 2009.

[69] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Synthetic data generation for statistical testing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 872–882. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115698

[70] E. Kang, E. Jackson, and W. Schulte, "An approach for effective design space exploration," in *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, ser. LNCS, R. Calinescu and E. Jackson, Eds. Springer Berlin Heidelberg, 2011, vol. 6662, pp. 33–54.

[71] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-breaking predicates for search problems," *KR*, vol. 96, pp. 148–159, 1996.

[72] S. Schneider, L. Lambers, and F. Orejas, "Symbolic model generation for graph properties," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2017, pp. 226–243.

[73] K.-H. Pennemann, "Resolution-like theorem proving for high-level conditions," in *International Conference on Graph Transformation*. Springer, 2008, pp. 289–304.

[74] A. S. Al-Sibahi, A. S. Dimovski, and A. Wasowski, "Symbolic execution of high-level transformations," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, 2016, pp. 207–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=2997382