# Refinery: Graph Solver as a Service

## Refinement-based Generation and Analysis of Consistent Models

Kristóf Marussy*
Attila Ficsor*
Oszkár Semeráth*
marussy@mit.bme.hu
ficsor@mit.bme.hu
semerath@mit.bme.hu
Budapest University of Technology and Economics, Department of Measurement and Information Systems
Budapest, Hungary

Dániel Varró
daniel.varro@liu.se
Linköping University, Department of Computer and Information Science
Linköping, Sweden
McGill University, Department of Electrical and Computer Engineering
Montreal, Canada

## ABSTRACT

Various software and systems engineering scenarios rely on the systematic construction of consistent graph models. However, automatically generating a diverse set of consistent graph models for complex domain specifications is challenging. First, the graph generation problem must be specified with mathematical precision. Moreover, graph generation is a computationally complex task, which necessitates specialized logic solvers. *Refinery* is a novel open-source software framework to *automatically synthesize a diverse set of consistent domain-specific graph models*. The framework offers an expressive *high-level specification language* using partial models to succinctly formulate a wide range of graph generation challenges. Moreover, it provides a modern *cloud-based architecture* for a scalable *graph solver as a service*, which uses logic reasoning rules to efficiently synthesize a diverse set of solutions to graph generation problems by partial model refinement. Applications include system-level architecture synthesis, test generation for modeling tools or traffic scenario synthesis for autonomous vehicles.
**Video demonstration**: https://youtu.be/Qy_3udNsWsM
**Continuously deployed at**: https://refinery.services/

## KEYWORDS

Model generation, Partial modeling, Logic solver, Cloud service

**ACM Reference Format:**
Kristóf Marussy, Attila Ficsor, Oszkár Semeráth, and Dániel Varró. 2024. Refinery: Graph Solver as a Service: Refinement-based Generation and Analysis of Consistent Models. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639478.3640045

*The first three authors contributed equally to this research.


*ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal*

ACM ISBN 979-8-4007-0502-1/24/04.
https://doi.org/10.1145/3639478.3640045

## 1 INTRODUCTION

*Motivation and challenge.* Model-based systems engineering is a popular approach in the industry for the design of critical software-intensive cyber-physical systems [31] supported by a variety of modeling tools (like Artop, Capella, Yakindu, OpenModelica and many closed-source alternatives). These tools help reveal design flaws early, thus reducing costs and improving product quality. In this context, there is an increasing need for a *diverse set of synthetic graph models* to represent test cases and benchmarks for modeling tools or candidate designs in systems engineering. However, such synthetic graph models need to be *consistent* to comply with the underlying domain-specific standards (e.g. AUTOSAR, SysML) captured in the form of metamodels and well-formedness constraints.

However, the automated synthesis of a diverse set of consistent domain-specific graph models is very challenging. (A) *Random model generators* can derive large and diverse graphs, but the derived graphs are not consistent. (B) *Search-based model generators* [17, 24] can derive large and consistent graphs using evolutionary algorithms but without any guarantees for completeness or diversity. (C) *Solver-based model generators* [6–8, 32] map the graph generation problem to a SAT- or SMT-problem in the background to detect inconsistencies of specifications, but they fail to derive a diverse set of graphs with more than a few hundred of graph nodes. Conceptually, a graph solver can also be regarded as an SMT-solver for the domain of complex graph (or relational) data, but the main focus is to derive a diverse set of consistent models (if they exist).

*Objective and scope.* The Refinery framework supports the efficient generation of consistent and diverse domain-specific graph models. It offers (1) a *high-level specification language* to capture the domain and control the range of graphs requested by end users, (2) a semantically well-founded *graph generation approach* based on *refinement of partial models* using 4-valued logic, and (3) a modern *cloud-based architecture* that provides a partial modeling editor, a partial model reasoner and a graph solver for engineers made available in a web browser or programmatically as a Java library.

First, the end user needs to provide a *domain specification*, which consists of a metamodel, an (optional) initial partial model, a set of predicates and constraints and a *scope definition* to restrict the size of the generated models. Then *server-side automated graph generation* back-end can be initiated by a push of a button (or

programmatically). The diverse set of auto-generated consistent graph models can be visualized or serialized in a textual format.

A main use case of Refinery is to synthesize graphs as test cases in applications such as the testing of modeling tools or system-level testing of autonomous vehicles [2]. Refinery also helps experts to semi-automatically provide a variety of consistent design candidates with complex graph structures as part of design space exploration [14], partial modeling [5], or feature modeling [12].

*Envisioned users.* The Refinery framework primarily aims to target *systems and software engineers* to derive complex test suites for industrial modeling tools (like Artop, Capella, Yakindu, OpenModelica and many closed source alternatives). An ongoing initiative is to provide test models for the new SysML standard [13]. However, the modern web interface enables the use of the framework by other *domain experts*, e.g. safety experts developing test scenarios in autonomous vehicles, or Blockchain experts (see section 4). During the development of the framework, we have received regular feedback from researchers at Budapest University Technology and Economics (Hungary), McGill University (Canada) and Linköping University (Sweden) and engineers at IncQuery Labs.

## 2 CONCEPTUAL OVERVIEW AND USAGE

Our framework offers a high-level specification language for model generation using refinement of 4-valued partial models.

### 2.1 Specification Language

The framework provides a concise yet precise specification language for generating graphs based on the syntax for partial models proposed in [10]. The language allows to control the range of generated models using four kinds of language elements:

$$\langle problem \rangle := (\langle domain \rangle | \langle assertion \rangle | \langle predicate \rangle | \langle scope \rangle)*$$

A *domain specification* in Refinery captures the key concepts and relations of the domain using an essential subset of XCore [27], a popular textual metamodeling language integrated with Eclipse Modeling Framework [26]. In a domain specification, the user can declare classes and associations as *relational symbols* (denoted by $\langle s \rangle$ in the grammar below), while a large set of logic constraints imposed by the structure of the metamodel is automatically translated to assertions and predicates (including the type hierarchy, multiplicities, and containment hierarchy, as illustrated in [10]).

$$\langle domain \rangle := (\texttt{abstract})?\texttt{class}\langle s \rangle(\texttt{extends}\langle s \rangle(,\langle s \rangle)*)?\{$$
$$((\texttt{contains}?)\langle s \rangle[\langle min \rangle..\langle max \rangle]?\langle s \rangle(\texttt{opposite}\langle s \rangle)?) * \}$$

For example, graphs representing file structures may provide classes such as `FileSystem`, `File`, `Directory`, and Symbolic Links (`SymLink`). A `FileSystem` contains a `File` as a `root`; each `Directory` contains multiple `File`s, and a `SymLink` can refer to other `File`s.

```
class FileSystem { contains File[1] root }
class File.
class Dir extends File{contains File[0..*]element}
class SymLink extends File { File[1] target }
```

*Assertions* define facts in an instance model (similarly to Prolog or Datalog). Each assertion assigns a *truth-value* to a relational term (which represents a graph node or edge).

$$\langle assertion \rangle := \langle s \rangle(\langle term \rangle(,\langle term \rangle) * ):\langle truth\text{-}value \rangle .$$

A non-annotated $\langle truth\text{-}value \rangle$ denotes `true` value assignment, and the `!` symbol denotes `false` value assignment. For example, one can prescribe that a directory called `resources` exists in a graph, which contains an `img` file and a symbolic `link` pointing to the image, while one can state that the image file cannot be a `Dir`:

```
Dir(resources). element(resources,img).
element(resources,link).target(link,img).!Dir(img).
```

*Logic predicates* provide custom model views while *constraints* (error patterns) allow to further restrict the range of valid graphs. A logic predicate in Refinery declares a new *n*-ary relational symbol (with header variables $\langle v \rangle$), and defines a constraint formulated as a disjunction of multiple bodies (separated by the ";" character), which are composed as a conjunction of literals (separated by the "," character). A literal refers to the truth-value of a symbol with variables: by default, the literal refers to "*" denotes transitive closure. The keyword `error` denotes error patterns: in a valid model, such predicates must be false for each node [4].

$$\langle predicate \rangle := (\texttt{error})?\texttt{pred}\langle s \rangle(\langle v \rangle(,\langle v \rangle) * )\texttt{<->}\langle body \rangle(;\langle body \rangle).$$
$$\langle body \rangle := \langle literal \rangle(,\langle literal \rangle)*$$
$$\langle literal \rangle := (!)?\langle s \rangle(*)?(\langle v \rangle(,\langle v \rangle) * )$$

For example, we can identify self-loops with a predicate that matches symbolic links targeting themself and forbid their occurrence with the `error` keyword. Similarly, we can ban empty directories with a predicate that matches nodes that are directories and have no elements. Finally, we can refer to some files as important if more different links point to them.

```
error pred selfLoop(s) <-> target+(s,s).
error pred emptyDir(d) <-> Dir(d), !element(d,_).
pred important(f)<->target(l1,f),target(l2,f),l1!=l2.
```

The *scope* controls the size of the generated models by defining the minimum and maximum number of instances (or predicate occurrences) of the scoped symbol with a `true` truth-value.

$$\langle scope \rangle := \texttt{scope}\langle s \rangle{=}\langle min \rangle..\langle max \rangle(,\langle s \rangle{=}\langle min \rangle..\langle max \rangle) * .$$

One may generate models with 25 to 30 nodes, two file systems, and at least one match for the predicate detecting important files:

```
scope node=0..30, FileSystem=2, important=1..*.
```

### 2.2 Generation with 4-Valued Partial Models

Model generation can be initiated by the user (*Generate* button) to derive a consistent model of the specification, if such a model exists. Simple inconsistencies can be highlighted in Refinery by error markers on the derived graph. To obtain a diverse set of graphs, each newly generated graph is structurally different from previous ones ensured by shape-based graph diversity metrics [22].

The Refinery framework uses 4-valued logic [3, 9] to explicitly represent incomplete, partial (*paracomplete*) models, and to tolerate errors and inconsistencies (*paraconsistency*) arising during the evaluation of computations over such models. 4-valued logic contains the usual `false` and `true` truth values, the `unknown` value introduced for uncertain (unspecified) properties, and the `error` value that signals inconsistencies. The subset {`false`, `true`, `unknown`} of logic values can express partial, but potentially consistent information (such as incomplete models). Conversely, the subset

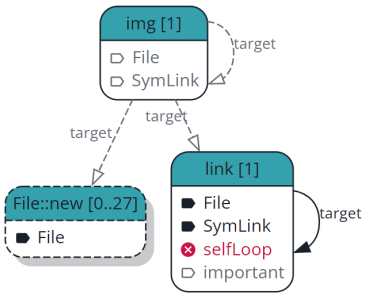


**Figure 1: Example of a partial model in Refinery**

{`false`, `true`, `error`} expresses possibly inconsistent, but complete information (such as over-constrainted or invalid models).

The Refinery framework collects all assertions and predicates of the problem specification, and semantically merges the information content into a 4-valued partial model. The framework continuously (and incrementally) checks and visualizes the problem specification, and immediately pinpoints if there is a set of inconsistencies (e.g. related to type errors, multiplicities).

During model generation, instance graphs are derived along refinements of partial models [2, 20]: at each generation step, (1) an uncertain element is selected and resolved by decision rules, and (2) the consequences of this decision are investigated by unit propagation rules (which also handles continuous type checking and multiplicity validation). As Refinery supports the incremental and partial evaluation of constraints [21], a (certain) match of an error predicate immediately triggers backtracking, while potential matches of error predicates provide search heuristics [18].

Figure 1 shows a partial model with inconsistent and incomplete values. Previously, node `img` was defined not to be a `Dir`, while `SymLink` was not explicitly excluded from its type, hence the node is marked as potential `SymLink` (white type label). If we now add the assertion `target(link,link)`, it causes an inconsistency flagged by an occurrence of the `error` predicate `selfLoop` (see the respective error marker in the partial model). Truth-values of other predicates are also automatically calculated and updated in the model: the node `link` is denoted as potentially `important`, while no other node in the model has the potential to be `important`.

## 3 ARCHITECTURE

The *architecture of the Refinery graph solver as a service* is illustrated in Figure 2. Refinery follows a modern *multi-tier software architecture*: graph generation problems are input via the *frontend* web application (or a Java library), while the *backend* comprises auto-scalable service deployed as *containers* behind a *load balancer*.

### 3.1 Frontend

*Web application.* A *Single-Page Application* (SPA) was created for editing and visualizing partial models and initiating model generation. To provide editor support, we opted to perform the bulk of the parsing and semantic analysis of the partial models in the backend in order to reuse the analyses already required by the model generation. In particular, *we display the logical consequences of the statements in the partial model* by executing *propagation* operations on the backend immediately after the user edits the partial model.

The SPA establishes a *WebSocket* connection and sends editing operations (text deltas) from the *textual partial model editor* (based on the *CodeMirror*[1] framework) to the backend. Features like *syntax error checking*, *content assist*, *find occurrences*, *semantic highlighting*, and *automatic formatting* are initiated via the WebSocket when the editor is idle or upon user request.

Additionally, when the partial model description is free of syntactic errors, the corresponding *partial model semantics*, including any discovered (semantic) *inconsistencies* are obtained back from the server. The user may apply further filtering (e.g., hide some nodes or relations) before visualizing the model as a graph using *Graphviz* and *D3*[2]. When the user initiates graph generation, the generated *solution* is also visualized. Subsequent generation requests return different solutions by choosing a different *random seed*.

*Client library. Refinery* is also available as a library in the Java programming language. Model generation tasks may be programmatically submitted using our textual partial modeling language. Alternatively, more direct interaction with the partial model management library and the model generator is available for more specialized use-cases, such as iterative model generation [23].

### 3.2 Backend

The backend consists of three main components: the (1) *partial model editor*, the (2) *partial model reasoner*, and the (3) *graph solver* for generating consistent graph models.

*Container-based* packaging provides easy deployment to cloud providers, such as AWS. Components (1)+(2) may be deployed as a single *Docker* container to serve as a backend of the web-based editor, while components (1)+(2)+(3) deployed together in a container enable model generation. In both cases, *Refinery* does not rely on any other server-side state. Thus, it can be automatically scaled behind an *load balancer* that can handle WebSocket connections.

A Docker image, containing (1)+(2)+(3) as a *monolith* is also available for local or on-premises deployments (e.g., for use-cases where the resource limits provided by the cloud service are insufficient).

*Partial model editor.* The editor contains a web server to handle incoming WebSocket connections and uses Eclipse Xtext [28] to parse partial models and provide *syntactic analysis* features. Then partial models are transformed into an internal semantic representation according to the Refinery *language semantics*.

In order to enable multiple concurrent users per server instance, we implemented an optimized, WebSocket-based protocol over the *Xtext Web* feature of the Xtext framework to maintain a *server-side copy* of the edited partial models. This allows to substantially lower resource utilization per user and latency on initial connection by serving multiple users with a single backend instance.

*Partial model reasoner.* The core of partial model reasoning in *Refinery* is an efficient *model management library* that enables the compact representation of multiple versions of partial models [25] as relational logic structures. This component can also be used as a standalone Java library to store and query partial models.

[1] `https://codemirror.net/`
[2] `https://graphviz.org/`, `https://d3js.org/`

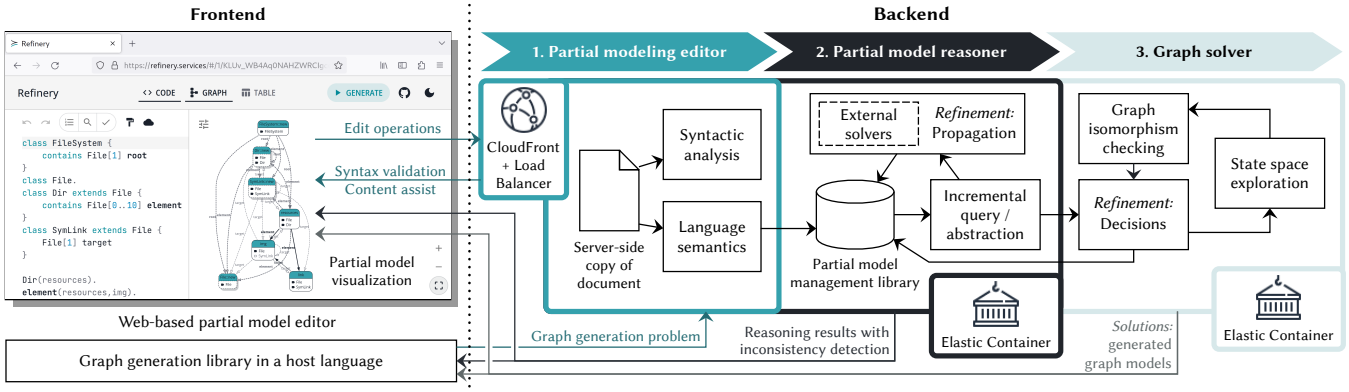


**Figure 2: Architectural overview of the Refinery graph solver as a service**

| | Problem size | | | Largest model |
|---|---|---|---|---|
| | #class | #ref | #error | #node |
| FileSys | 4 | 2 | 0 | 19750 |
| FAM | 9 | 13 | 4 | 15750 |
| Yakindu | 11 | 7 | 8 | 4250 |
| Ecore | 20 | 33 | 1 | 2000 |
| Social | 2 | 4 | 2 | 230 |

**Table 1: Largest models generated in 60 seconds**

*Consistency checking* and *refinement* of partial models requires reasoning about the *type system* describing the specified and unspecified parts of graphs and the *graph constraints* that capture consistency rules. To this end, we integrated an *incremental graph query engine* based on VIATRA Query [4] in combination with advanced approximation techniques for *partial query evaluation* [21].

*Propagation* steps are partial model refinements that encode logical consequences of the type system and constraints. Refinery derives a *refined partial model* and analyzes its consistency every time the user edits the partial model. Numerical constraints are handled by *external solvers*, including GLOP[3] for linear constraints.

*Graph solver.* The solver takes the *initial partial model* provided by the user and generates a set of *consistent graphs* as output [21].

At each *decision* step, a new partial model is derived (as a new exploration state) by decreasing the number of uncertain nodes and edges in the partial model while simultaneously increasing its size.

*State space exploration* needs to repeatedly detect if a partial graph has already been reached (special *isomorphism detection*), and if a graph constraint is surely violated, when no consecutive refinements will ever lead to a consistent model. We rely on *graph shapes* [16] to detect isomorphic graphs and enforce diversity [22].

## 4 EVALUATION

*Scalability evaluation.* An initial evaluation of the Refinery framework is provided in Table 1 across five different domains used as case studies in previous research [2, 11, 20]. The number of classes, associations, and constraints in each domain is listed in Table 1 together with the largest model successfully generated as follows. In this initial experiment, each generation run had a *60-second timeout*. We incremented the model size by 250 until the generation for a given size timed out five times in a row. In the case of the Social Network domain, because of the limited size of the generated models, we increased the number of objects by 10. Finally, it is worth pointing out that model generation runs for each of these domains are available as part of the integration test suite of Refinery.

*Theoretical properties.* The Graph solver algorithm provides multiple formal guarantees [30], including *correctness* (a model is generated, then it satisfies the constraints) and *completeness* (if a model satisfies the constraints, it will be generated eventually).

*Uses cases in research and education.* The development of the Refinery framework has been supported by an Amazon Research Award, and it has been successfully used in several practical applications, which include (1) the automated synthesis of test scenes for autonomous vehicles, (2) generation of dependable blockchain architectures, and (3) automated synthesis of system architectures for early mission planning. The framework has also been used by MSc students as part of an advanced modeling lab offered at Budapest University of Technology and Economics. Moreover, a public tutorial of Refinery has been delivered at ASE 2023 by the authors.

*Related work.* There are other software tools that offer the automated synthesis of consistent graph models. Most notable examples include Alloy [8], Sterling [29], USE [6], Pledge [1, 24], UMLtoCSP [7], TAF [17] or VIATRA Solver [19]. However, we wish to point out that the size of models generated by Refinery compares very favorably to alternative model generation approaches (see e.g. [2, 11, 20] for detailed measurements of other tools). Compared to our previous work [19], Refinery provides a fundamentally novel, modern cloud-based architecture, a 4-valued partial modeling framework, and new decision procedures for type inference and propagation.

*Running example.* The example is available at [15].

*Acknowledgements.* This paper was partially supported by an Amazon Research Award, the European Union as part of the EDGE-Skills (101123471), the ÚNKP-23-4-II-BME-219 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[3] https://developers.google.com/optimization/lp/

# REFERENCES


[1] Abdessalem, R. B., Nejati, S., Briand, L. C., and Stifter, T. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), IEEE, pp. 1016–1026.

[2] Babikian, A. A., Semeráth, O., Li, A., Marussy, K., and Varró, D. Automated generation of consistent models using qualitative abstractions and exploration strategies. *Softw. Syst. Model. 21*, 5 (2022), 1763–1787.

[3] Belnap, Jr., N. D. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, vol. 2 of *EPIS*. Springer, 1977, pp. 5–37.

[4] Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., and Ökrös, A. Incremental evaluation of model queries over EMF models. In *MODELS 2010* (2010), vol. 6394 of *LNCS*, Springer, pp. 76–90.

[5] Famelis, M., Salay, R., and Chechik, M. Partial models: Towards modeling and reasoning with uncertainty. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 573–583.

[6] Gogolla, M., Büttner, F., and Richters, M. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming 69*, 1-3 (2007), 27–34.

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

[8] Jackson, D. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 11*, 2 (2002), 256–290.

[9] Kamide, N., and Omori, H. An extended first-order Belnap-Dunn logic with classical negation. In *LORI 2017*, vol. 10455 of *LNCS*. Springer, 2017, pp. 79–93.

[10] Marussy, K., Semeráth, O., Babikian, A. A., and Varró, D. A specification language for consistent model generation based on partial models. *J. Object Technol. 19*, 3 (2020), 3:1–22.

[11] Marussy, K., Semeráth, O., and Varró, D. Automated generation of consistent graph models with multiplicity reasoning. *IEEE Transactions on Software Engineering* (2020).

[12] Mendonca, M., Wąsowski, A., and Czarnecki, K. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference* (2009), pp. 231–240.

[13] Object Management Group. *SysML v2: The Next-Generation System Modeling Language*. `https://www.omgsysml.org/SysML-2.htm`, `https://github.com/Systems-Modeling/SysML-v2-Release`.

[14] Palesi, M., and Givargis, T. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the tenth international symposium on Hardware/software codesign* (2002), pp. 67–72.

[15] Refinery authors. Filesystem case study, 2024. `https://github.com/graphs4value/refinery-tutorials`.

[16] Rensink, A. Isomorphism checking in GROOVE. *Elect. Comm. EASST 1* (2007).

[17] Robert, C., Guiochet, J., Waeselynck, H., and Sartori, L. V. TAF: a tool for diverse and constrained test case generation. In *21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021* (2021), IEEE, pp. 311–321.

[18] Semeráth, O., Babikian, A. A., Chen, B., Li, C., Marussy, K., Szárnyas, G., and Varró, D. Automated generation of consistent, diverse and structurally realistic graph models. *Software and Systems Modeling* (2021), 1–23.

[19] Semeráth, O., Babikian, A. A., Pilarski, S., and Varró, D. Viatra solver: a framework for the automated generation of consistent domain-specific models. In *41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019* (2019), IEEE / ACM, pp. 43–46.

[20] Semeráth, O., Nagy, A. S., and Varró, D. A graph solver for the automated generation of consistent domain-specific models. In *ICSE* (2018), ACM, pp. 969–980.

[21] Semeráth, O., and Varró, D. Graph constraint evaluation over partial models by constraint rewriting. In *Theory and Practice of Model Transformations* (2017), Springer, pp. 138–154.

[22] Semeráth, O., and Varró, D. Iterative generation of diverse models for testing specifications of DSL tools. In *FASE18* (2018), Springer, pp. 227–245.

[23] Semeráth, O., Vörös, A., and Varró, D. Iterative and incremental model generation by logic solvers. In *International Conference on Fundamental Approaches to Software Engineering* (2016), Springer, pp. 87–103.

[24] Soltana, G., Sabetzadeh, M., and Briand, L. C. Practical constraint solving for generating system test data. *ACM Transactions on Software Engineering and Methodology (TOSEM) 29*, 2 (2020), 1–48.

[25] Steindorfer, M. J., and Vinju, J. J. Optimizing hash-array mapped tries for fast and lean immutable JVM collections. *SIGPLAN Not. 50*, 10 (Oct. 2015), 783–800.

[26] The Eclipse Project. *EMF*. `https://eclipse.dev/modeling/EMF/`.

[27] The Eclipse Project. *XCore*. `https://wiki.eclipse.org/Xcore/`.

[28] The Eclipse Project. *Xtext*. `http://www.eclipse.org/Xtext/`.

[29] The Sterling Developers. Sterling. `https://sterling-js.github.io/demo/`.

[30] Varró, D., Semeráth, O., Szárnyas, G., and Horváth, Á. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig*. Springer, 2018, pp. 285–312.

[31] Whittle, J., Hutchinson, J. E., and Rouncefield, M. The state of practice in model-driven engineering. *IEEE Softw. 31*, 3 (2014), 79–85.

[32] Zheng, G., Bagheri, H., Rothermel, G., and Wang, J. Platinum: Reusing constraint solutions in bounded analysis of relational logic. In *FASE* (2020), pp. 29–52.