An Initial Performance Analysis of Graph Predicate Evaluation over Partial Models

Attila Ficsor, Oszkár Semeráth
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: attila.ficsor@edu.bme.hu, semerath@mit.bme.hu

***Abstract*—Graph-based modeling tools are widely used during the design, analysis and verification of complex critical systems. Those tools enables the automation of several design steps (e.g., by model transformation), and the early analysis of system designs (e.g. by test generation). The evaluation of complex graph predicates (or graph pattern matching) is a core technique in modeling and model transformation, and essential in scalable graph generation. This motivated the integration of industrial graph pattern matching tools directly to advanced data structures used in model checking and logic reasoning algorithms.**

In this paper we provide a report of a preliminary performance benchmark combining the incremental graph pattern matching algorithm of the Viatra framework with hash tries used for state space exploration on partial models.

***Index Terms*—predicate evaluator, graph generation**

I. Model generation and predicate evaluation

During the design and testing of critical systems, modeling tools are widely used, enabling the automation of several development and testing steps with graph-based models. Graph-based models are the primary development artifacts in those modeling environments on which advanced modeling frameworks are operating. To test those modeling applications, we need a diverse set of well-formed models as test input.

However, the synthesis of valid well-formed models is a challenging task. A common feature of scalable model synthesis algorithms is the continuous evaluation of graph predicates during an exploration process:

- The VIATRA Solver model generation approach [1] combines the incremental graph pattern matching [2] with rule-based design exploration framework [3].
- The SDG framework [4] combines standard OCL-based tooling [5] with genetic algorithms.

In this paper we compare the performance of a new prototype predicate evaluation technique using hash tries [6] for efficiently storing multiple versions of a graph models, and the incremental graph pattern matching algorithm [2]. We present the measurements on a case study, where we generate diverse and realistic scenarios for testing machine learning components used in advanced driver-assistance systems.

II. Challenges of modeling technologies

Until now, the only documented way to create patterns for VIATRA was in VIATRA Query Language [2]. To use this, we needed to work our way through a long list of steps setting up the integrated development environment (IDE). This included installing Eclipse with Eclipse Modeling Framework (EMF) and VIATRA, then creating a modeling project, where we could create a metamodel. From this we had to generate model code and editor code, which we had to use to start a Runtime Eclipse. In this instance of Eclipse we could create a Query Project, in it a VQL file, and in this file, we could write our pattern in VQL language. When we saved this file, some Java classes were generated, that we could use in our code.

This method used EMF objects to store data (i.e. the model), and there was limited options to use other data structures. In the existing code base in the VIATRA framework, there are two ways to create (partial) models used as the starting point of the generation. One is to create an EMF model either using the graphical user interface (GUI) editor, or using Java programs. We can then load this model, and VIATRA builds its own internal data structure from the model. The other way we can create a partial model is using a tabular method, where we can create a table which we can use to write our data into. Then based on this table, VIATRA creates its own internal data structure, similar to the previous method. Unfortunately, this is implemented for data types provided by EMF.

There are several challenges resulting from this:

- Setting up the IDE is not user-friendly, it has complicated software requirements and necessary settings, that are difficult to find.
- Portability is limited, since one version has Eclipse dependency, while the other version without this dependency is unstable.
- Originally, the pattern matching operates on data structures provided by EMF, which imposes performance limitations (e.g., scalability issues with ELists and inefficient state space exploration using EMF transactions).

To answer those challenges, we chose to integrate high performance data structures [6] to the core pattern matching mechanism of the VIATRA query framework.

- We provide a simple grammar to formulate type systems, the predicates (i.e., the queries) and instance models [7].
- The framework can be used without custom editors.
- Finally, [6] promises efficient and scalable data structures optimized for exploring huge search spaces necessitated by explicit model checking algorithms.

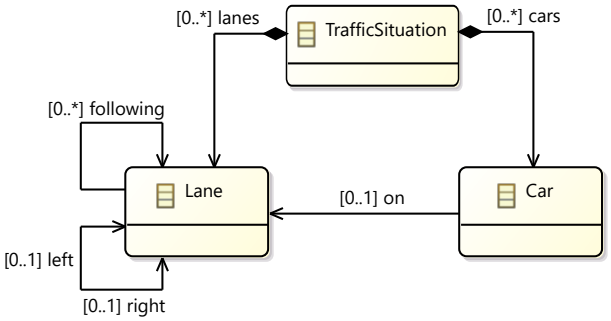


Fig. 1: EMF metamodel used in the measurements

In this paper, our main goal is to provide an initial performance comparison between the newly developed data structure and VIATRA. In the following, we present the domain we selected to execute our performance comparison. For this example we use a focused fragment of the Scenic traffic situation modeling language [8].

III. Representing models with EMF and VIATRA

First, we present the domain of measurement using standard modeling technologies.

First, review the metamodel. A metamodel describes the main concepts and relations, of a model, and defines its main structure. In this paper, we used a simple metamodel shown in Figure 1 using EMF. In this metamodel, a Traffic situation consists of Lanes and Cars. A Lane can be connected to another Lane via the following reference, and its relation with the other lanes are represented with the left and right references. Cars are placed onto a single lane. Instance models in our measurements are in accordance with this metamodel

In our measurements we are using VIATRA as a comparison. We implemented seven graph patterns in VQL to query

- incoming empty lane segments without preceding lanes to spawn new cars into the scenario;
- outgoing lane segments without following lanes to despawn cars from the scenario;
- cars on the same or adjacent lanes for listing potentially dangerous situations;
- and potential trajectories for lane following and changing maneuvers [8].

The implementation of the last pattern is illustrated below.

```
pattern moveCar(from: Lane, to: Lane, car: Car) {
    Lane.following(from,to);
    Car.on(car,from); }
or{ Lane.left(from,to);
    Car.on(car,from); }
or{ Lane.right(from,to);
    Car.on(car,from); }
```

IV. 4-valued partial models

Next, we illustrate the same problem with 4-valued partial models. Partial modeling is a technique to explicitly represent uncertainty in models by abstracting a collection of possible models into a single partially specified model. In this paper, we use 4-valued logic to represent uncertainty, where traditional

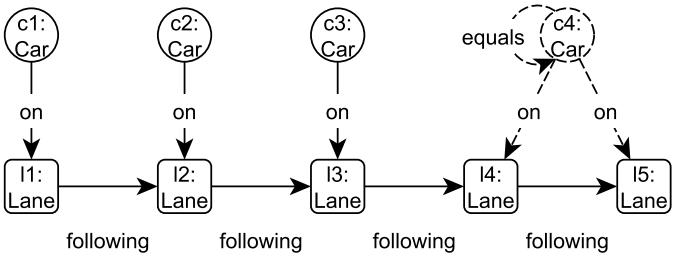


Fig. 2: Partial model with unknown relation values

Car	
c1	true
c2	true
c3	true
c4	true

Following		
l1	l2	true
l2	l3	true
l3	l4	true
l4	l5	true

exists	
l1	true
···	true
c3	true
c4	unknown

Lane	
l1	true
l2	true
l3	true
l4	true
l5	true

On		
c1	l1	true
c2	l2	true
c3	l3	true
c4	l4	unknown
c4	l5	unknown

equals		
l1	l1	true
···		true
c3	c3	true
c4	c4	unknown

TABLE I: Relational representation of an example model

logic values **true** and **false** are extended with a logic value **unknown** to represent uncertain or incomplete data where both true and false are possible in the represented concrete models, and with **error** to represent inconsistencies.

Figure 2 shows a partial model with five lanes and four cars. Lanes 1-5 are following each other, and cars 1-3 are on lanes 1-3. We know that these objects and relations exist, so they are marked with solid arrows. The last car, c4 can be on lanes l1 or l2. In this example, it is unknown whether it is on either of them, these relations are marked with dashed arrows. We extend this notation of uncertainty to existence and equivalence as well, which enables abstract nodes that can represent multiple or no nodes. In our example, c4 is denoted with a dashed loop edge with the label "equals", which means that c4 can represent multiple nodes. By default, other objects are different from each other, and equal with only themselves. Moreover, c4 is denoted with dashed line, which means that its existence is uncertain: it can be included to, or excluded from the model.

The same model is shown in Table I, in similar tables as the ones used in our predicate evaluation algorithm. The Car and Lane tables show the types of the objects, while the Following and On tables contain the relations between the objects. A **true** value means the relation exists in the model, while an **unknown** value means the relation may exist in the model. The most numerous **false** values are omitted in both Figure 2 and Table I. Error values are not present in the model, since that would mean there is an inconsistency.

4-valued partial models enable the representation of classes and references with unknown existence using abstract nodes and edges like c4 in Figure 2. As a syntactic sugar, [7] introduces classes and references (illustrated below) which are translated to abstract nodes and edges internally.

```
class Lane {                    class Car {
```

```
  Lane[0..*] following                Lane[0..1] on
  Lane[0..1] left opposite right    }
  Lane[0..1] right opposite left
}
```

The predicates implementing the graph patterns in the new specification language are shown below. We chose to use direct predicates (**direct pred**), so we could specify to match for **true** and **unknown** values for potential values of car trajectories, giving us a 2-valued result. Using a predicate (**pred**) without specifying the **true** and **unknown** values would give a 4-valued result [9]. These predicates are semantically equivalent to the ones implemented in VIATRA.

```
direct pred moveCar(from,to,car) <->
   following(from,to)=true|unknown,on(car,from)=true|unknown
 ; left(from,to)=true|unknown,on(car,from)=true|unknown
 ; right(from,to)=true|unknown,on(car,from)=true|unknown.
```

V. Evaluation

A. Research questions

We evaluated the performance of the query engine by formulating various research questions and answering them by measuring execution times. These are the research questions we aim to answer:

RQ2 How does the model building scale if we increase the model size?

RQ1 How does the pattern matching scale if we increase the model size?

B. Measurement setup

The measurement workflow is shown in Figure 3. The measurements have three parameters:

- x: The number of lanes following each other.
- y: The number of parallel lanes.
- n: The number of times the changes are applied and pattern matching is executed.

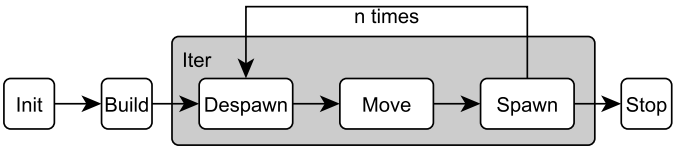


Fig. 3: Measurement setup

First, we initialize the predicates, an empty model and the query engine in the Init phase. Next, in the Build step we build up the model, which consists of $x * y$ lanes in a grid, and y cars placed randomly on these lanes. Figure 4 shows an example of a four by four grid of lanes with four cars. The forward direction is to the right, and the arrows show which lane each car is able to move to.

In the next step an iteration starts, where we first despawn all cars that are on a lane that has no following lane. In Figure 4 car C4 would be removed, since it can no longer go forward. Next we move all remaining cars from their current lane to its following lane or either of the lanes next to them. After this step the example might look like Figure 5. The third and final

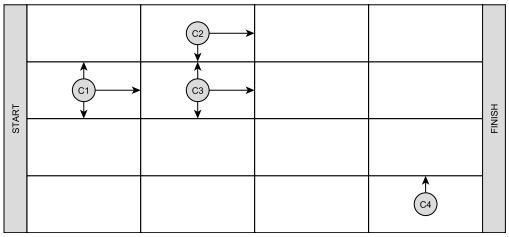


Fig. 4: Example model of a 4×4 grid of lanes with 4 cars

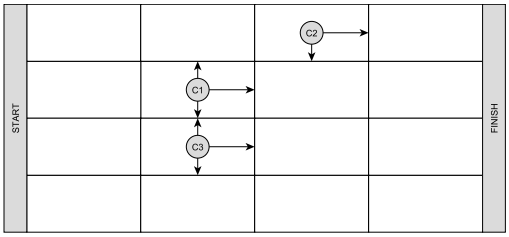


Fig. 5: An example model after despawning and moving

step in the iteration is spawning new cars to make sure there are y cars in the model. These new cars are placed randomly on the lanes. These three steps are repeated n times. After n iterations the measurement stops. The runtime of each step is measured separately.

C. Compared approaches

We ran the same measurement using four different approaches. First, we measured VIATRA in continuous evaluation mode (where each model change is processed immediately, denoted by VQL-continuous), then in coalescing mode (where model updates are processed after the full iteration, denoted by VQL-batch. VIATRA does not support 4-valued evaluation naively, those are matched on only **true** and **false** values, using EMF as data structure. Then we measured our approach once where we query both **true** and **unknown** values (where all four logic value is used), and once where we query exactly **true** values (denoted with Refinery and Refinery-Abstract). We run the simulation for 5000 iterations, while saving the runtime after every 1000 iterations. Before the measurement of both tools, we ran a similar, but smaller setup to account for the JVM warm up, and programmatically called the garbage collector after each run. Each measurement was repeated 25 times, and we used the median value of the results, to filter out the noise.

We executed the measurements for multiple model sizes. The sizes are illustrated in Table II For the measurements we used the following hardware, software versions, and settings: Java version: 17, maximum Java heap size: 8GB, VIATRA version: 2.6.0, OS: Windows 10, CPU: Intel Core i7-9750H.

D. Measurement results

Figure 6a shows the runtime building the model. The horizontal axis is the nodes and edges in the model. The vertical axis shows the time it took to complete building the model, in milliseconds.

size	lanes	cars	nodes+edges
50	50x50	50	10100
100	100x100	100	40200
150	150x150	150	90300
200	200x200	200	160400
250	250x250	250	250500
500	500x500	500	1001000
750	750x750	750	2251500
1000	1000x1000	1000	4002000
1250	1250x1250	1250	6252500

TABLE II: Model sizes

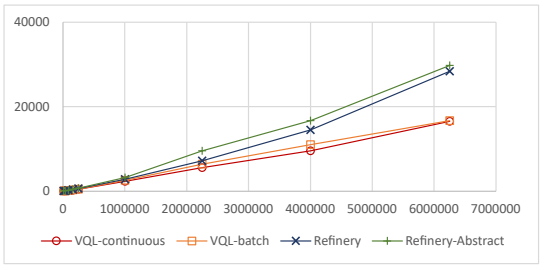
Figure 6b shows how the runtime changes if we increase the number of nodes and edges in the model used in the setup. The horizontal axis is the size of the model on which we ran the pattern matching, as detailed above. The vertical axis shows the time it took to complete 5000 iteration of modification on the model, in milliseconds.

E. Discussion of the results

On Figure 6a we can see that building the model is slower with our solution, than the two VIATRA configuration, because the current version of Refinery and Refinery-Abstract uses more relation (both left and right, exists and equals), and needs twice as many base indexing to support multiple logic values. However, these initial performance measurement showed potential performance improvements.

RQ1 With respect to the model size the original VIATRA scales better than Refinery and Refinery-Abstract.

As we can see in Figure 6b, all four measurements show a similar shape on the diagram. The fastest solution was our approach without abstraction (Refinery), and provided better performance then both VQL-continuous and VQL-batch are slower. This can happen as it uses more advanced data structures than EMF and skips model management steps irrelevant to running the core query evaluation engine (e.g., notification sending, order of elements in a list, resource management). The slowest solution was our approach with abstraction, where the result is calculated from the combination of predicates. This took almost twice as long to run, than without abstraction, since this had to check roughly twice as many rows.



(a) Runtime of model building

(b) Change in total runtime by model size

Fig. 6: Runtime measurements

RQ2 With respect to the model size, Refinery scales better than the original VIATRA. Compared to that, Refinery-Abstract needs almost twice as much time.

VI. Conclusion and Future Work

In this paper, we provided an initial performance benchmark using VIATRA with a novel data structure representing 4-valued partial models. Despite the richer expression power of our data structure, our solution produced favorable performance, better than the standard modeling technology (EMF).

In the future, we are planning to use this data structure in a design space exploration scenario used for model generation, replacing the backed engine of VIATRA Solver. Additionally, we are planning to evaluate the performance of the data structure using existing benchmarks (like [10]).

Acknowledgements: The first author was partially supported by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003) and by ÚNKP-21-4 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. The second author was partially supported by the NRDI Fund of Hungary, financed under the [2019-2.1.1-EUREKA-2019-00001] funding scheme.

References

[1] O. Semeráth, A. S. Nagy, and D. Varró, "A graph solver for the automated generation of consistent domain-specific models," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 969–980, 2018.

[2] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró, "A graph query language for EMF models," in *Theory and Practice of Model Transformations*, pp. 167–182, Springer Berlin Heidelberg, 2011.

[3] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth, "Multi-objective optimization in rule-based design space exploration," in *ACM/IEEE international conference on Automated software engineering*, pp. 289–300, 2014.

[4] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical constraint solving for generating system test data," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 1–48, 2020.

[5] M. Richters and M. Gogolla, "OCL: Syntax, semantics, and tools," in *Object Modeling with the OCL*, pp. 42–68, Springer, 2002.

[6] M. J. Steindorfer and J. J. Vinju, "Optimizing hash-array mapped tries for fast and lean immutable jvm collections," *SIGPLAN Not.*, vol. 50, p. 783–800, oct 2015.

[7] K. Marussy, O. Semeráth, A. A. Babikian, and D. Varró, "A specification language for consistent model generation based on partial models," *J. Object Technol.*, vol. 19, no. 3, pp. 3:1–22, 2020.

[8] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 63–78, 2019.

[9] O. Semeráth and D. Varró, "Graph constraint evaluation over partial models by constraint rewriting," in *International Conference on Theory and Practice of Model Transformations*, pp. 138–154, Springer, 2017.

[10] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró, "The train benchmark: cross-technology performance evaluation of continuous model queries," *Software & Systems Modeling*, vol. 17, no. 4, pp. 1365–1393, 2018.