Performance Analysis Of Himesis

Eugene Syriani Hans Vangheluwe

August 21^{th}, 2010

Abstract

In our quest of addressing model-driven engineering problems of industrial scale, we seek for an efficient data structure to optimally represent models. In AToMPM, models are represented as graphs. Furthermore, the tool is implemented in the Python language. We therefore compare two potential candidates, namely *IGraph* [1] and *NetworkX* [2]. A thorough performance analysis allows us to choose the most efficient one. Furthermore, we extend it to efficiently manipulate models, more specifically to perform model transformation. We describe the different algorithms that ensure this task and compare its performance with a standard graph transformation benchmark.

1 Introduction

Nowadays, model-based development is emerging in industry. However, the models industry deals with are very large and tend to scale with up to 10^5 to 10^6 elements. Model-based tools, such as AToMPM (the new version of AToM^3 [3]), must allow the modeller to work with such industrial-scale models. It should be able to handle common tasks such as loading, saving, representing visually, and transforming models.

Since graphs are often used to model the state of a system, models are represented as graphs in AToMPM. The data structure used internally is called Himesis, representing typed, attributed, directed graphs. In order to make the kernel of AToMPM as efficient as possible (although implemented in the Python object-oriented language), Himesis must allow one to optimally manipulate graphs.

In Section 2, we first compare the performance of two promising Python libraries achieving these tasks. Section 3 examines the performance of the most efficient one, focusing on the tasks accomplished in model-based tools. One critical task in particular is to find sub-graphs in a given graph, following some constraint conditions (meta-model type, multiplicities, OCL constraints, etc). Since graph transformation heavily relies on the matching algorithm, Section 4 describes the algorithms implemented in Himesis for (1) computing sub-graph isomorphisms and (2) pattern matching as used in model transformation. The performance of Himesis is analysed in Section 5 in the context of a standard graph transformation benchmark. Finally we conclude in Section 6.

2 Making the Right Choice

After seeking for existing libraries for existing libraries that efficiently manipulate graphs, our investigation resulted with two potential candidates: *IGraph* [1] and *NetworkX* [2].

2.1 IGraph & NetworkX

Both IGraph [4] and NetworkX [5] are open source software packages for creating and manipulating graphs. IGraph is implemented in ANSI C, although it offers a Python API. NetworkX is entirely implemented in Python. They

both allow creating directed multi-graphs, *i.e.,* where edges have a source to target orientation and there can be more than one edge between any two (not necessarily distinct) nodes. Attributes can be assigned to nodes, edges, or to the graph itself[1]. The values can be of any type, including graphs, thus supporting hierarchical graphs [6]. Although both libraries exhibit very similar features, they differ in the way data is stored internally.

In IGraph, nodes are not explicitly stored. Instead, the internal structure only keeps track of the total number of nodes in the graph. Nodes and edges are each identified by a non-negative integer ID. Node and edge ID numbering is always continuous which may require re-numbering when deletion occurs. Consequently, the attribute values of a node are not stored in the node itself. Instead, an overall additional vector is assigned globally to the graph. The drawback is that if an attribute is only meaningful for a small subset of nodes, the required memory space will be assigned for all nodes, as if they all had this attribute defined on themselves. Attributes are nevertheless conveniently accessible by lookup/reference tables.

In NetworkX, a graph is stored by its adjacency list implemented in a Python dictionary of dictionaries. The outer dictionary is keyed by nodes to values that are themselves dictionaries. The latter are keyed by neighbouring nodes to values that are edge attributes associated with that edge. Nodes can take the form of any hashable Python object. For non-hashable objects, NetworkX allows to represent the node as a unique identifier and assign the data as a node attribute. This is the same way IGraph allows for arbitrary objects be stored in a node. However with NetworkX, the burden is on the developer to guarantee the uniqueness of the identifiers.

2.2 IGraph vs. NetworkX

Given these differences and most of all that IGraph's kernel is implemented in a language much more efficient then NetworkX is, we will examine how each library performs for model manipulation tasks. For our concern, these tasks are: creation, deletion, and modification of nodes and edges, as well as the traversal of all the elements in the graph (the well known CRUD operations).

2.2.1 Experimental Conditions

Figure 1 shows a heat graph, represented as a table, assessing for which case one library is more optimal than the other. For each operation we vary two parameters. The first one is the number of times n an operation is applied. In this comparison, we generate an initial Erdõs-Rényi random graph $G(50, 0.5)$. It is a graph with 50 nodes such that an edge is created between any two nodes with probability $p = 0.5$, the randomness being sampled from a uniform distribution function. The probability chosen generates a dense graph, given that directed multiple edges and self loops are allowed. Note that the same initial graph G is used for both libraries, ensuring non-biased experiments. The second parameter is how data is stored in the graph. For this experiment, we evaluate the case when no data is stored in the graph (depicted by the "No Attributes" label in Figure 1) and when nodes hold attribute values (depicted by the "Attributed" label in Figure 1)

The table in Figure 1 represents the results of the experiments along three dimensions: whether data is stored, the operation under study, and the number of times the operation is performed. In this table, each inner cell is defined by $((d, op, n), r)$ where:

- $op \in \{AN, AE, UN, TN, DE, DN\}$ is the operation of interest: *Add nodes*, *Add edges*, *Update nodes*, *Traverse*, *Delete edges*, and *Delete nodes*, respectively. In this experiment, the operations were applied in this order to evaluate their performance independently.
- $d \in \{NA, AT\}$ indicates whether data is stored: *No attributes* or *Attributed*, the latter stores data at the node level using the library's node attribute mechanism. For the attributed case, the size of the data stored at each node is 4,118 bytes, which is considered as a light-weight attribute in Python.
- $n \in \mathbb{N}$ is the number of times the operation has been applied.

[1] We only considered attributed nodes for the experiments of this section.

Data	Operation	10	20	50	70	100	200	500	700	1,000	2,000	5,000	7,000	10,000	20,000	50,000	70,000	100,000	120,000	150,000	170,000	200,000
No attributes	Add nodes	8.9	14.0	35.0	37.1	73.7	90.2	286.7	242.1	296.5	657.3	1467.2	1741.4	1002.5	1425.1	761.7	1161.6	1046.4	1170.6	982.4	1270.2	1347.9
No attributes	Add edges	0.9	1.8	4.1	5.2	6.7	10.6	15.2	15.6	16.2	18.6	20.0	20.8	20.3	20.7	28.9	27.0	27.1	28.3	28.6	29.7	30.2
No attributes	Update nodes	2.5	4.5	11.0	14.0	19.6	33.0	68.4	67.1	77.6	118.5	209.6	202.1	153.5	161.2	137.3	137.7	121.7	132.2	135.3	136.9	146.1
No attributes	Traverse	1.0	1.0	1.5	1.8	3.3	2.5	1.7	1.6	1.7	2.0	1.6	2.0	1.8	1.6	2.6	2.5	2.3	2.4	2.4	2.5	2.7
No attributes	Delete edges	0.2	0.4	1.0	1.3	1.9	3.6	7.9	10.3	13.5	23.5	42.2	52.5	58.5	56.1	56.4	49.3	46.5	45.8	43.7	43.8	44.9
No attributes	Delete nodes	2.2	4.2	9.5	12.1	15.0	25.8	40.3	45.3	50.7	63.3	75.8	79.3	83.2	74.2	68.5	59.9	53.3	54.0	54.3	51.6	51.5
Attributed	Add nodes	6.0	3.5	2.0	1.7	1.5	1.2	1.1	1.1	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Attributed	Add edges	1.0	1.8	3.7	4.8	6.2	9.6	14.3	14.6	16.0	18.1	20.7	20.5	21.0	19.5	33.1	31.3	42.8	40.4	59.3	55.5	52.8
Attributed	Update nodes	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Attributed	Traverse	0.9	0.9	1.3	2.0	3.4	2.7	1.7	1.7	1.6	2.2	1.7	2.1	1.9	1.9	3.3	4.1	3.6	4.3	4.0	3.9	4.1
Attributed	Delete edges	0.3	0.4	0.9	1.4	1.8	3.4	7.8	10.2	13.0	23.1	42.7	50.4	57.9	54.0	55.2	47.0	43.8	43.9	43.2	42.8	41.2
Attributed	Delete nodes	1.8	3.1	5.8	7.1	7.6	8.7	7.3	6.9	6.5	5.7	5.0	4.7	4.5	4.1	4.2	4.3	4.3	4.2	4.2	4.2	4.2

Figure 1: Comparing IGraph with NetworkX for CRUD operations. The darker the colour, the better IGraph performs and vice versa.

- $r \in [0, +\infty[$ is the ratio of the computation time between the two libraries[2]. When $0 \leq r < 1$, NetworkX is faster and when $r > 1$, IGraph is faster. The boundary case of $r = 1$ simply depicts that they were as fast for performing the operation op.

Each cell can therefore be uniquely identified by the tuple (op, d, n). To alleviate the notation, replacing any element of this tuple by ∞ refers to the corresponding sub-table. For example, $(NA, \infty, 10)$ represents the results of the experiment where all the operations are applied 10 times on a graph with no attributes. Note that every (d, ∞, n) combination is a distinct experiment with a new graph G as input. The system running these experiments is composed of 32 nodes with 7,200 RPM Intel Core 2 Duo process of 2.66 GHz, 8 GB of memory with a 667 MHz DDR2, and two times 4MB of L2 cache. The connection is a 4 gigabit port switch with layer 2 switching.

2.2.2 Analysis of the Comparison

At a first glance, Figure 1 reflects much more dark cells then light ones, indicating that IGraph performs better than NetworkX overall. For a graph with no attributes (when $d = NA$), since IGraph stores nodes very efficiently as explained previously, it clearly outperforms NetworkX with respect to the creation of elements: 1.3×10^3 times faster for the creation of 2×10^5 nodes and 30 times faster for the creation of the same amount of edges. NetworkX is up to 4 times faster for deleting small amounts of edges (less than 50), while IGraph is up to 45 times faster for larger values of n. As for the deletion of nodes, IGraph is up to 80 times faster for mid-sized graphs (10^4 edges) and around 50 times faster for larger graphs. This ratio is significantly smaller than for the creation of nodes because of the re-numbering required to ensure a continuous numbering of nodes in IGraph. Traversing all nodes in the graph is twice as fast on average in IGraph for any size of the graph. The update operation ($op = UN$) in this case is simply the sum of adding and removing the same amount of nodes n, since no attributes are stored in the nodes. On average this operation is 100 times faster for IGraph.

Now that we know IGraph is significantly more efficient than NetworkX for non-attributed graphs, we will examine if it is still the case when data is added to the graph. When $d = AT$, the creation of nodes is about 3 times faster in IGraph than in NetworkX for $n \leq 100$. Creating a larger number of nodes is as fast in both libraries as the initialization of attributes becomes an overhead on the actual creation of a node. This is confirmed with the fact that the update operation is also as fast in both libraries. The ratio for edge creation is the same as for the non-attributed case for $n \leq 2 \times 10^4$. This is predictable as the presence node attribute does not influence edge creation. However, when creating more edges, IGraph is slightly even more efficient: the ratio is 1.5 times higher than for the non-attributed case. Nevertheless, edge deletion for the attributed case is as performant as its non-attributed homologue. Node deletion becomes only 4 times faster with IGraph when attributes are present. NetworkX is slower traversing the graph with attributed nodes (up to 4 times slower for graphs with 2×10^4 nodes). Table 1 summarizes the overall comparison.

[2] All numeric results of the experiments presented in this paper have an error margin of $\pm 1.000 \times 10^{-1}$ unit.

Operation	No Attributes		Attributed	
Add nodes	719.9	583.3	1.5	1.2
Add edges	17.9	9.8	23.2	18.0
Update nodes	99.5	65.5	1.0	0.0
Traverse	2.0	0.6	2.5	1.2
Delete edges	28.7	22.8	27.8	22.1
Delete nodes	46.4	25.5	5.2	1.7

Table 1: Average (first column) and standard deviation (second column) over all values of n of the efficiency factor of IGraph over NetworkX.

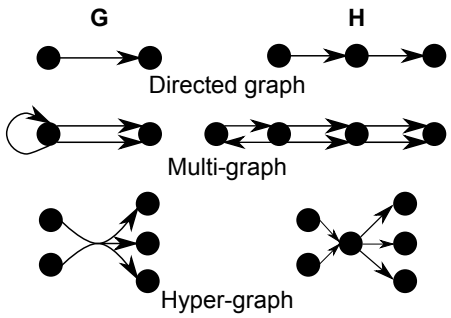


Figure 2: Different types of graphs G and their representation as Himesis graphs H.

3 Optimal Representation of Models

Given the results of the previous experiment, IGraph is overall significantly more efficient in time complexity. It is also more efficient in space complexity since the machines running the NetworkX library ran out of memory at $n = 3 \times 10^5$, while no thrashing was observed when IGraph was dealing with graph sizes of up to 10^6 elements. In this section, we investigate for the optimal representation of data in the graph. We analyse the performance and relative cost of the CRUD operations.

3.1 Realizing Domain-Specific Models as Directed Simple Graphs

In domain-specific languages, models represent an abstraction of a real-life system. These models are often considered as entity/relation systems: *entities* represent the concepts and data of the model and *relations* describe how these concepts are related. Moreover, a relation may itself hold data. When such models are realized as directed graphs, representing entities as nodes and relations as edges seems obvious at a first glance. Moreover, IGraph supports attribute assignment on both elements. In specific cases, a relation may itself be related to another relation or entity[3]. But then the graph representation would require to consider such relations as nodes. Therefore, to uniformly represent entities and relations of a model, we propose that they be represented as nodes. In this case, an edge represents the link between an entity and a relation, a relation and an entity, or a relation and a relation. Hence attributes need only to be stored on nodes. Another advantage of this uniform representation of models is when a model has a multi-graph or hyper-graph topology. Figure 2 describes how the relations are represented in each case. However, our representation does not consider multi-hyper-graphs, which are not a common topology of domain-specific models.

Now let us examine the cost of this uniform representation of a model M. Let $G = (V(G), E(G))$ denote a directed graph representing the entities of M as nodes $V(G)$ and its relations as edges $E(G)$. Let H denote the graph representing

[3]For example, in UML class diagrams [7], an association class can relate two classes and also be part of an inheritance relationship with another association class.

the entities and relations of M as nodes $V(H)$ and the links as edges $E(H)$. (Examples of G and H are depicted in Figure 2.) We then have that $|V(H)| = |V(G)| + |E(G)|$ and $|E(H)| = 2 \times |E(G)|$. Therefore there is only a constant difference between the size of H and G. This is also the case when G is a multi-graph. When G is a hyper-graph, $|V(H)|$ is as before but now $|E(H)| = |Src(E(G))| + |Tar(E(G))|$, where *Src* and *Tar* represent respectively the source nodes and target nodes of each edge.

3.2 Performance Evaluation of CRUD operations in IGraph

To investigate for the optimal representation of data, we need to modify the domain of d in the condition tuple (d, op, n) such that: $d \in \{NA, LA, HA, LO, HO\}$, respectively *No Attribute, Light Attribute, Heavy attribute, Light object,* and *Heavy object*. They span two dimensions: data representation and the size of the data. The "attribute" label indicates that, for each node, data is stored as a separate node attribute. The "object" label indicates that all the data is wrapped in a single object and only that object is stored as a node attribute. In our experiments, a light attribute is 139 bytes, whereas a heavy attribute is 4,330 bytes. The first corresponds to the size of two integers and three characters in Python. The second corresponds to the size of two integers, two 50-character-long strings and another string of 4,094 characters long.

Figure 3 represents the time performance of IGraph for each value of d. When no data is stored in the graph, *i.e.*, $d = NA$ (Figure 3(a)), node creation is the least costly operation with less than 10 milliseconds for adding 10^6 nodes. Node deletion is also very efficient with 100 milliseconds for the same amount of nodes. As mentioned before, for the case where $d = NA$, the update node operation is evaluated as first deleting then adding a new node to replace it. This is why it is about the same speed as the delete node operation. Edge deletion seems to perform faster than edge creation for larger graphs with respectively 100 milliseconds versus 1 second. Node traversal is undoubtedly the most costly operation. IGraph can traverse 2×10^4 nodes within a minute, but it takes almost 6.5 days to traverse 10^6 nodes in breadth-first search.

The plots for $d = LA$ and $d = LO$ are very similar to each other. This indicates that the relative performance of the operations is the same when light data is stored in the graph (Figure 3(b) and 3(d) respectively). Edge operations become the fastest. Nevertheless, from $n = 2 \times 10^4$ on, node deletion performs better than edge creation but worst than edge deletion, both by a factor of 2. Node creation is now slower by a factor 10^3 in the case of $d = LA$ and by a factor of 5×10^2 in the case of $d = LO$. Moreover, node update is about the same speed as node creation, which confirms that the setting of attribute values is an overhead for node creation. Traversal is still the most costly operation.

Finally, the plots for $d = HA$ and $d = HO$ are also very similar. The exceptions are that node deletion is now more expensive than edge creation with 1.7 seconds for $n = 10^6$. Also, node creation (and thus the update operation) are more costly than the traversal operation up to $n = 2 \times 10^4$ nodes.

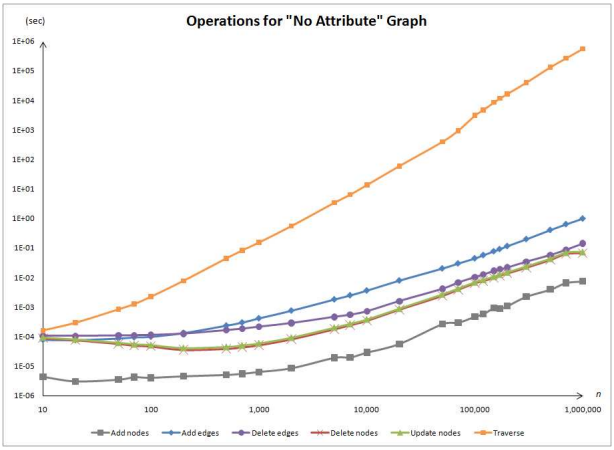
To better illustrate the described results, the table in Figure 4 presents the average performance time of each operation for different sizes of the graph. The graphs are grouped in three categories. Small graphs (less than 10^3 nodes) are typically used for small examples or debugging purposes. Medium graphs (between 10^3 and 10^5 nodes) are considered as large graphs for academics but average size for industrial projects. Large graphs (more than 10^5 nodes) are typically used in large industrial applications such as mobile networking. Furthermore, the table in Figure 5 summarizes the impact of choosing the "attribute" or the "object" representation for data in the graph. It clearly shows that the "object" approach is is more efficient than the "attribute" approach.

Add Nodes. From Figure 6(a), node creation is linear with a slope of 10^{-8}, 10^{-5}, and 10^{-3} respectively for the case where there are no attributes, for light attributes, and for heavy attributes.

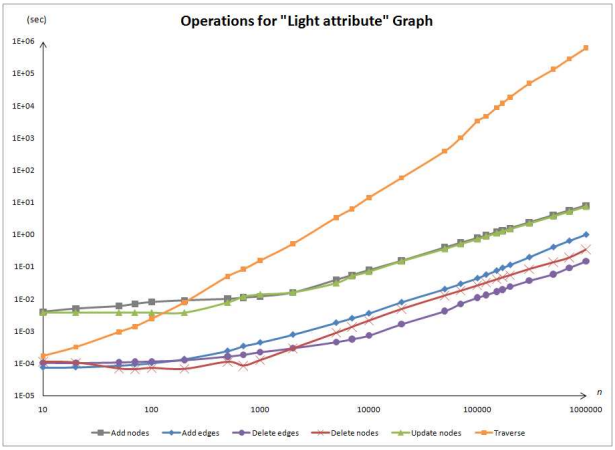
Add/Delete Edges. From Figure 6(b), edge creation is independent from the data representation and size. It is in fact quadratic in terms of n. It is the same case for edge deletion as shown in Figure 6(c).

Delete Nodes. From Figure 6(d), node deletion is also quadratic in terms of n. Here we see that the "attribute" representation is slightly more optimal for small to medium-sized graphs by 30%.

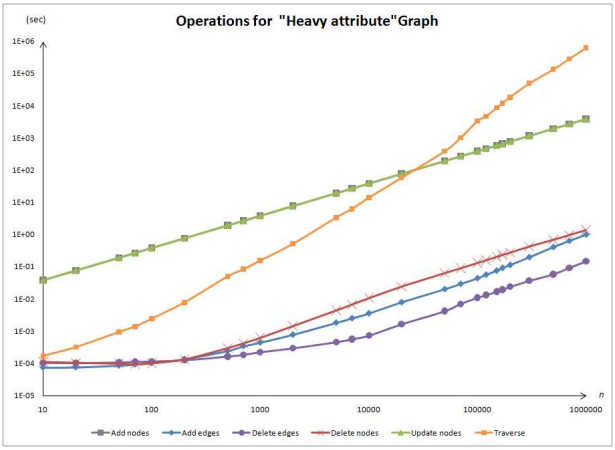
Update Nodes. In Figure 6(e), updating light data represented in the "attribute" approach is 30 times slower than the "object" approach. As for heavy-weight data, either approach is as slow by a factor 10^3.



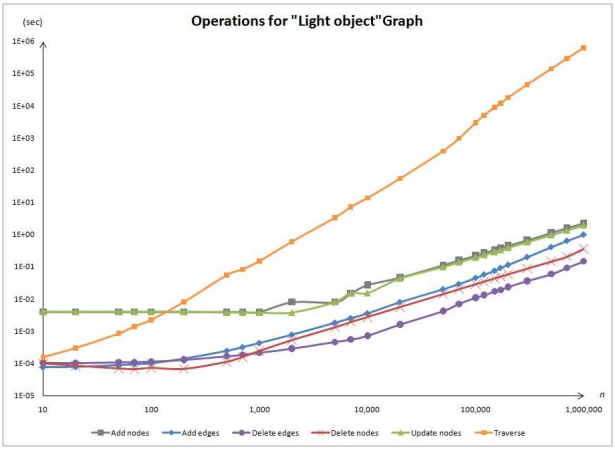
(a)



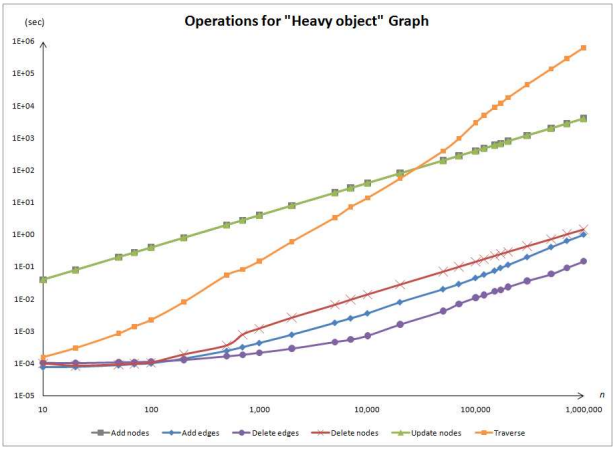
(b)



(c)



(d)



(e)

Figure 3: The effect of data representation. The graphs are plotted in log-log scale.

Operation	No attribute			Light attribute			Light object			Heavy attribute			Heavy object		
	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$	Small $[0,10^3[$	Medium $[10^3,10^5[$	Large $[10^5,10^6]$
Add nodes	5.E-06	2.E-04	5.E-03	2.E-03	2.E-01	3.E+00	7.E-04	7.E-02	8.E-01	6.E-03	2.E-01	3.E+00	5.E-03	7.E-02	8.E-01
Add edges	2.E-04	2.E-02	3.E-01	2.E-04	2.E-02	5.E-01	2.E-04	2.E-02	5.E-01	2.E-04	2.E-02	3.E-01	3.E-04	3.E-02	3.E-01
Update nodes	1.E-04	2.E-02	2.E-01	2.E-03	2.E-01	2.E+00	5.E-04	5.E-02	7.E-01	2.E-03	2.E-01	3.E+00	5.E-04	6.E-02	7.E-01
Traverse	2.E-02	4.E+02	2.E+03	5.E-02	6.E+02	3.E+05	5.E-02	5.E+02	3.E+05	3.E-02	4.E+02	2.E+03	9.E-03	6.E+02	2.E+03
Delete edges	2.E-04	4.E-03	3.E-02	2.E-04	5.E-03	8.E-02	2.E-04	5.E-03	8.E-02	2.E-04	5.E-03	3.E-02	2.E-04	5.E-03	3.E-02
Delete nodes	4.E-05	3.E-03	4.E-02	3.E-05	2.E-03	2.E-02	3.E-05	1.E-03	1.E-02	1.E-04	8.E-03	1.E-01	1.E-04	5.E-03	6.E-02

Figure 4: Average speeds for executing CRUD operations.

Operation	LO/LA	HO/HA	HA/LA	HO/LO
Add nodes	3.E-01	5.E-01	2.E+00	3.E+00
Add edges	1.E+00	1.E+00	9.E-01	1.E+00
Update nodes	3.E-01	3.E-01	1.E+00	1.E+00
Traverse	1.E+00	1.E+00	4.E-01	4.E-01
Delete edges	1.E+00	1.E+00	8.E-01	9.E-01
Delete nodes	7.E-01	7.E-01	4.E+00	4.E+00

Figure 5: Factor effect of using IGraph's node-level attribute mechanism for all node attributes versus wrapping all attributes in one object stored using IGraph's node-level attribute mechanism.

Traverse. Finally, from Figure 6(f), traversal of the graph is independent from the data representation and size. The plots are quadratic reflecting the traversal's complexity.

3.3 Optimal Representation of Data of Models in IGraph

The previous experiment considered graphs in general. In the following experiment, we investigate for an optimal representation of attributes of AToMPM models. Elements of these models can hold an arbitrary number of data (attributes). A typical element of an AToMPM model includes the following data: a universally unique id, two integers, two booleans, two 50-character long strings, an additional 10-character long string encoding the type of this element, a 1000-character long string representing an action or constraint on the element (typical for elements of transformation models), and a list of seven 10-character long strings enumerating all the sub-types of the type of this element Thus the total size of this typical element is 1,382 bytes, which is an average size according to the previous experiment.

We now consider three different alternatives for representing data in the nodes of IGraph graph:

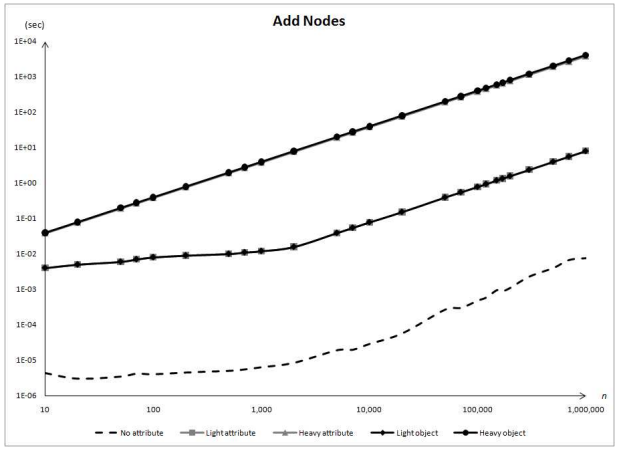
- Node attribute mechanism used for each of the above attributes (this is the *AT* approach used previously).
- A python object encapsulating all the attributes, stored as one node attribute (this is the *OT* approach used previously).
- A hash table holding all the attributes, stored as one node attribute (this will be referred to as HT).

In order to determine which of AT, OT, or HT is the optimal representation to use in Himesis, we evaluate their performance on CRUD operations applied to nodes only, since Section 3.2 has confirmed that data stored in nodes has no impact on the performance of edge operations.

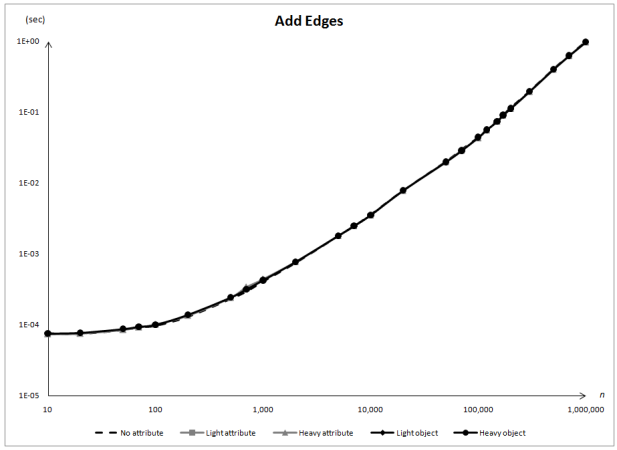
Create Nodes. Figure 7(a) shows the scales for creating[4] nodes for each representation: 9×10^{-3} for AT, 5×10^{-3} for OT, and 3×10^{-3} for HT.

Update Nodes. Figure 7(b) shows the scales for updating nodes for each representation: 9×10^{-3} for AT, 5×10^{-3} for OT, and 3×10^{-3} for HT. Not surprisingly, this is the same order as for adding nodes since, according to Section 3.2, the addition of nodes takes significantly less time than initializing its attributes (about a 10^3 times faster).

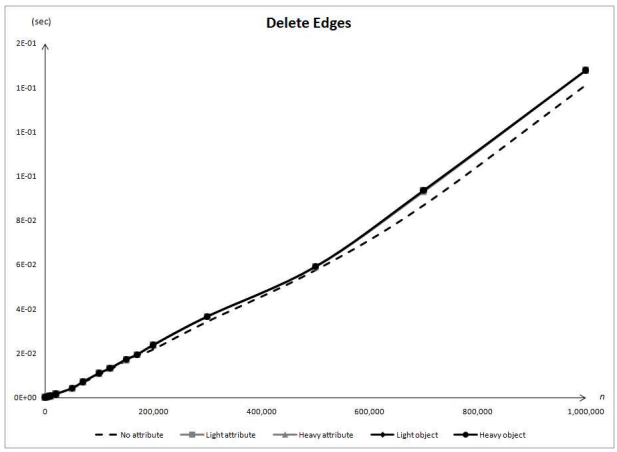
[4] Addition of nodes and initialization of its attributes.



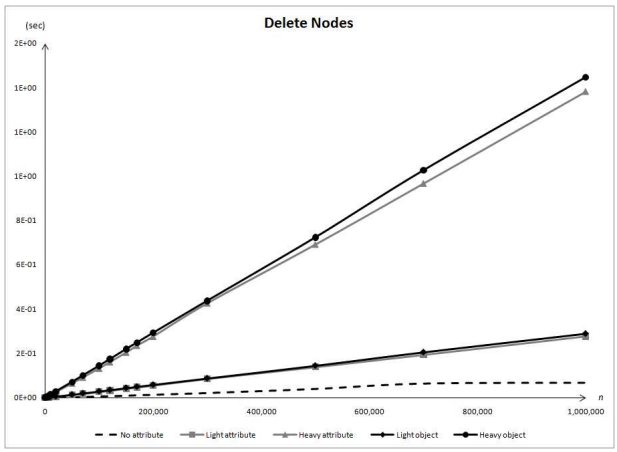
(a)



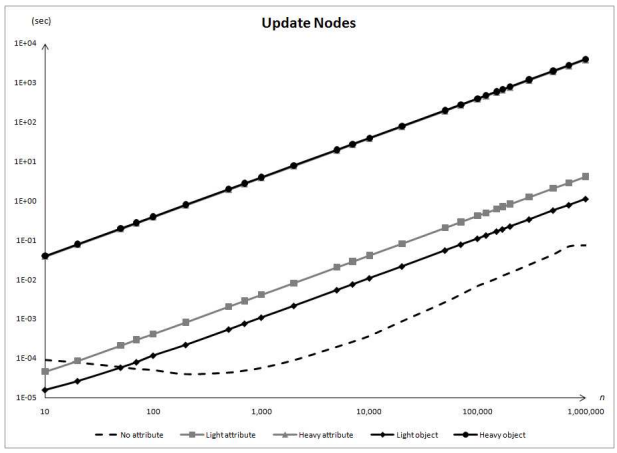
(b)



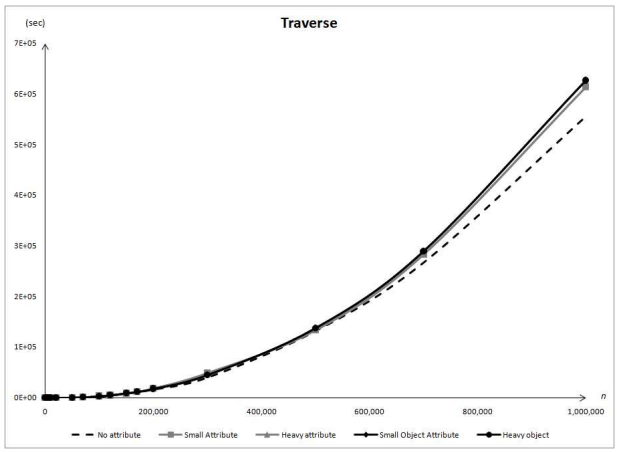
(c)



(d)



(e)



(f)

Figure 6: CRUD operations on nodes for each representation of data. The plots are in log-log scale.

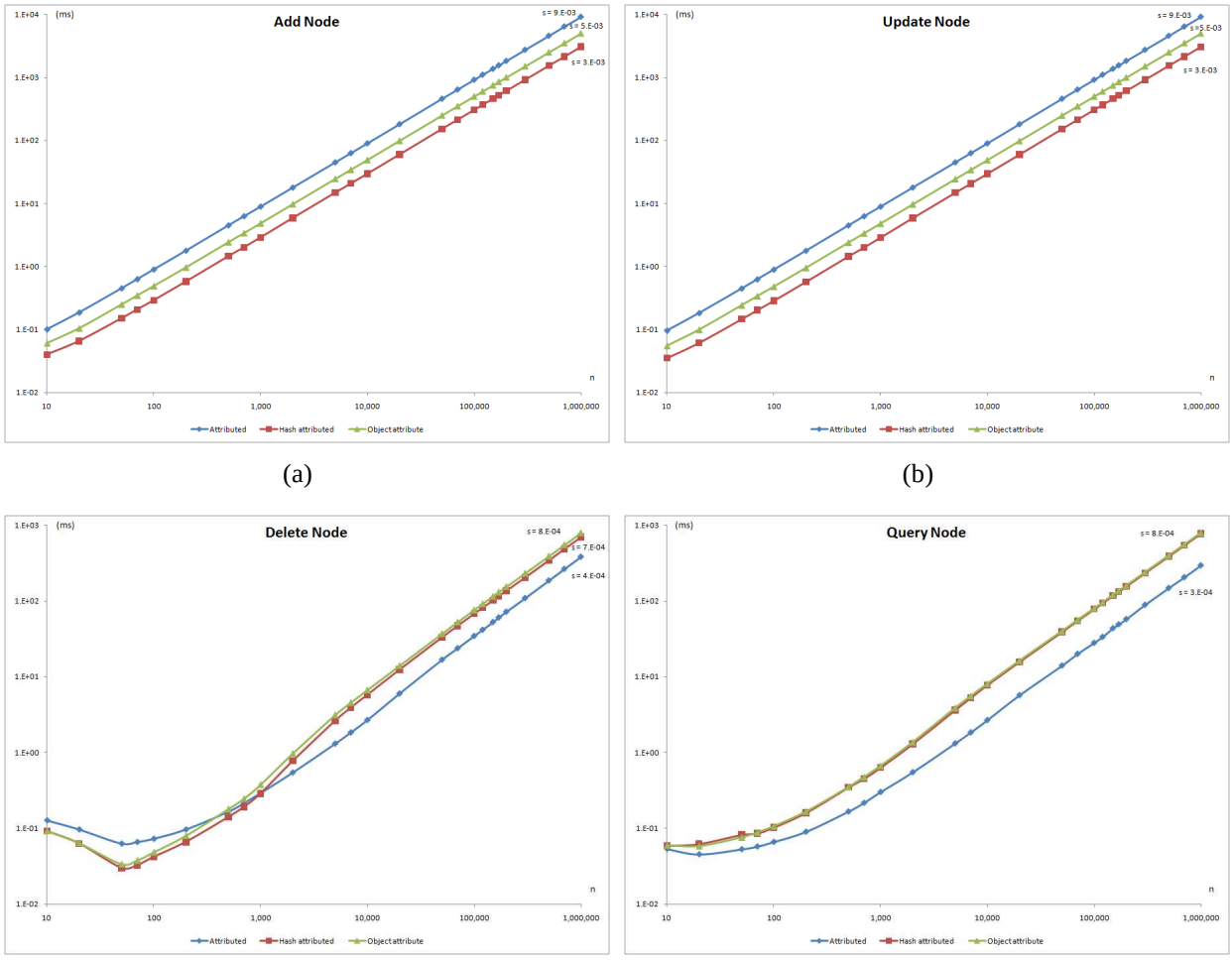


Figure 7: CRUD operations on nodes for each representation of data. The plots are in log-log scale.

Delete Nodes. Figure 7(c) shows the scales for deleting nodes for each representation: 4×10^{-4} for AT, 8×10^{-4} for OT, and 7×10^{-4} for HT.

Query Nodes. To query nodes, we have investigated for the optimal way of retrieving the data from nodes: using the mechanism built in IGraph for querying nodes (the `select` method) or programmatically retrieving attribute values. The results were very conclusive: using the IGraph query mechanism for HT and OT is 1.6 times faster, and 3.1 times faster if using the IGraph query mechanism for AT. Thus we only consider the IGraph mechanism for querying nodes. Figure 7(d) shows the scales for querying nodes for each representation: 3×10^{-4} for AT and 8×10^{-4} for both OT and HT.

We would like to minimize the time each of the CRUD operations takes in a rule. Here, we assume that a rule consists of a left-hand side (LHS) pre-condition pattern graph and a right-hand side (RHS) post-condition pattern graph. From the observations above, we can write the following formulas representing the time cost of a rule application for each representation of data:

$$\begin{aligned} AT: &\quad 90a + 90u + 4d + 3q \\ OT: &\quad 50a + 50u + 8d + 8q \\ HT: &\quad 30a + 30u + 7d + 8q \end{aligned}$$

, where a, u, d, and q are the number of times the add, update, delete, and query operations[5] on nodes happens in a rule, respectively. Therefore, choosing the optimal representation depends on the solution of the following inequalities:

$$\text{Choose } HT \text{ over } OT \quad \Leftrightarrow \quad 20(a+u) + d > 0 \tag{1}$$

$$\text{Choose } AT \text{ over } OT \quad \Leftrightarrow \quad q > 8(a+u) - 0.8d \tag{2}$$

$$\text{Choose } AT \text{ over } HT \quad \Leftrightarrow \quad q > 12(a+u) - 0.6d \tag{3}$$

Equation 1 is always true since, by definition, a rule applies at least one of the add, update, or delete operations. Hence OT will not be considered anymore and equation (2) can be discarded. The left-hand side of (3) represents the operation performed in the matching phase of the rule (querying nodes). The right-hand side of (3) represents the operation performed in the rewriting phase of the rule (add, update, delete nodes). Recall that the matching phase queries all nodes of the pre-condition pattern as well as all nodes of the source graph G (in the worst case). Hence $q \in O(|V(LHS)| + |V(G)|)$. Following a similar reasoning, we have that $a \in O(|V(RHS - LHS)|)$ $u \in O(|V(LHS \cap RHS)|)$ $d \in O(|V(LHS - RHS)|)$. On the one hand, in the extreme case where the LHS is empty, we therefore have that $|V(G)| > 12.6\,|V(RHS)|$. On the other hand if the RHS is empty, then $|V(G)| > -13.6\,|V(LHS)|$, which is always true if both the LHS and G are not empty. Therefore, a sufficient condition for choosing the AT approach is if there are 13 times more nodes in the source graph than in the RHS. This is very likely to hold given that relation-like model elements are also represented as nodes in Himesis. Moreover, favouring the AT approach reduces attribute access time for other model manipulations as well. The plot in Figure 8 classifies the performance of each graph operation performed on a Himesis graph implemented with the AT approach.

4 Match & Rewrite Operations in Himesis

Model transformation plays a crucial role in model-driven development. A transformation is commonly expressed as a set of *transformation rules*. A rule consists of a pre-condition pattern and a post-condition pattern. The former depicts a pattern that should occur in the input model and the latter depicts how this occurrence shall be modified. When models are implemented as graphs, the pre-condition pattern specifies that an instance of this pattern must be a sub-graph of the input graph. Since pattern matching (and in particular, the sub-graph homomorphism problem) is NP-Complete [8], there are various exponential-time worst case solutions, for which the average-time complexity can be

[5]The update and query operations are performed on all the attributes of each node.

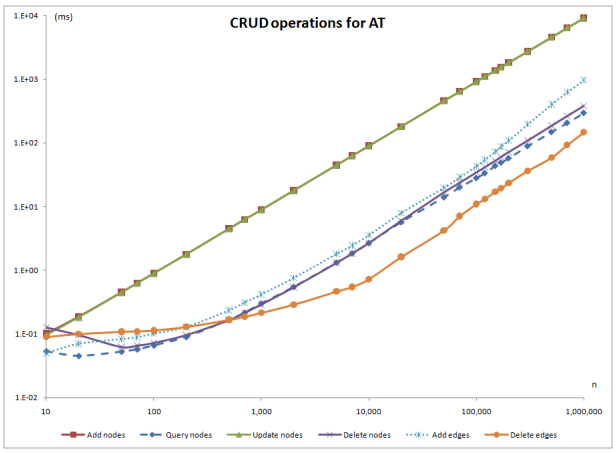


Figure 8: Performance of all operations on Himesis graphs.

reduced with the help of heuristics. These approaches can be divided in two major categories: *constraint satisfaction problems* and *search plans*.

On the one hand, search plan techniques [9, 10] define the traversal order for the nodes of the model to check whether the pattern can be matched. This is done by computing the cost tree of the different search paths and choosing the less costly one. Complex model-specific optimization steps can be carried out for generating efficient adaptive search plan [11]. On the other hand, graph pattern matching can be described as a constraint satisfaction problem [12], where the pre-condition elements are variables, the elements of model form the domain and typing, and the links and attribute values form the set of constraints. These techniques make use of backtracking algorithms [13] for finding a sub-graph of the input graph that is isomorphic[6] to the pre-condition graph. The algorithm explores the search space in a depth-first order. Well-known algorithms such as Ullmann [14] and VF2 [15] are some of the most efficient examples for solving the sub-graph isomorphism problem as a constraint satisfaction problem. In Himesis, we implement the pattern matching algorithm on top of these two algorithms.

4.1 An Efficient Sub-graph Isomorphism Algorithm

The matching algorithm of Himesis combines a variation of the VF2 algorithm together with the refinement strategy of Ullmann's algorithm, as outlined in Algorithm 1.

[6] In fact, it is homomorphic since the pattern graphs describe constraints on the attributes of the source graph.

Algorithm 1 `extend`(state)

```
1:  if mappingIsComplete(state) then
2:      storeMatch(state)
3:  end if
4:  for p, s in suggestMapping(state) do
5:      if areCompatible(p, s) then
6:          if areSyntacticallyFeasible(p, s) then
7:              if areSemanticallyFeasible(p, s) then
8:                  state.storeMapping(p, s)
9:                  extend(state)
10:                 state.undoMapping(patternNode, s)
11:             end if
12:         end if
13:     end if
14: end for
```

The procedure *extend* augments the state of the algorithm with all possible mappings from the pattern graph to the source graph. In the following, we call a *mapping* the one-to-one correspondence between a pattern node and a source node. We denote by a *match* the set of mappings in which all source nodes form a graph that is homomorphic to the pattern graph. Lines 4-14 recursively compute further mappings given the current state of the algorithm. The *state* stores the following information:

- M^P and M^S are the mapping sets holding the pattern nodes and the source nodes respectively in the current mappings,
- T^P_{out} and T^S_{out} hold the set of adjacent nodes to respectively M^P and M^S following outgoing edges, at any time;
- T^P_{in} and T^S_{in} hold the set of adjacent edges coming in respectively M^P and M^S following incoming edges, at any time;
- $T^P_{inout} = T^P_{out} \cap T^P_{in}$ and $T^S_{inout} = T^S_{out} \cap T^S_{in}$. The latter six sets are called the *terminal sets*.

Each step of the search computes a partial mapping of the nodes and verifies that it is does not violate the topology of the pattern graph. *suggestMapping* suggests a potential mapping of a source node s with a pattern node p (the pair (p,s) is also known as the candidate pair in [15]). The choice of the pair is done in the following order: first from $(T^P_{inout}, T^S_{inout})$, then from (T^P_{out}, T^S_{out}), then from (T^P_{in}, T^S_{in}), and finally from all other nodes.

Afterwards, *areCompatible* verifies if it is worth continuing this mapping. This is done by comparing the number of incident edges of s and p (this is known as the refinement step in [14]). The compatibility check verifies that:

$$|Out(p)| \leq |Out(s)| \wedge |In(p)| \leq |In(s)| \tag{4}$$

where $In(n)$ and $Out(n)$ respectively represent the set of incoming and outgoing adjacent edges of a node n. This is similar to the refinement step of Ullmann's algorithm.

Then comes the feasibility checks. *areSyntacticallyFeasible* ensures that the topology of the current mapping corresponds to a sub-graph of the pattern graph. This is done by looking at the number of incident edges when (p,s) is added to the current set of mappings (M^P and M^S).

Let $InOut(n) = In(n) + Out(n)$, for any node n,
let $Out_p = Out(p) \cap T^P_{out}$ and $Out_s = Out(s) \cap T^S_{out}$,
let $In_p = In(p) \cap T^P_{in}$ and $In_s = In(s) \cap T^S_{in}$,
let $All_p = M^P \cup T^P_{out} \cup T^P_{in}$ and $All_s = M^S \cup T^S_{out} \cup T^S_{in}$.

Then the following must be true to ensure syntactic feasibility of s and p:

$$\begin{aligned} |Out_p| &\leq |Out_s| \wedge \\ |In_p| &\leq |In_s| \wedge \\ |Out_p| + |In_p| + |InOut(p) - All_p| &\leq |Out_s| + |In_s| + |InOut(s) - All_s| \end{aligned} \tag{5}$$

The last test ensures that the semantics of s corresponds to the semantics of p. In our case, semantic information of the nodes is encoded in their attributes, but the deails of the function `areSemanticallyFeasible` will be elaborated later on. When s and p satisfy all of the above conditions, (p,s) is considered a valid mapping and is stored in the state (line 8). The algorithm then continues looking for remaining mappings. When all valid mappings have been computed (lines 1-3), the corresponding match is stored. The algorithm backtracks to the previous state when either a complete match is found or if the current partial match (set of mappings in M^P and M^S) does not allow for any further valid mapping. Note that a nice property of this algorithm is that any state in the search tree is visited exactly once.

Algorithm 2 allows to compute all matches between a pattern graph P and a source graph S. Furthermore, an initial set of mappings can be specified to prune the search tree constructed by the procedure `extend`. This initial mapping can also be seen as the initial context in which the matchings must be computed: it restricts specific pattern nodes to be mapped exactly to predefined source nodes.

Algorithm 2 `computeMappings`(S, P, context)

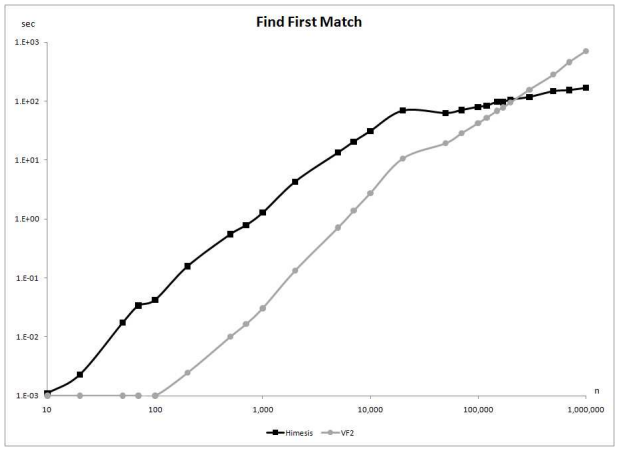
```
1: state ← initState(S, P)
2: for p, s in context do
3:     state.update(p, s)
4: end for
5: extend(state)
6: return state.getMatches()
```

Performance Evaluation of the Implementation

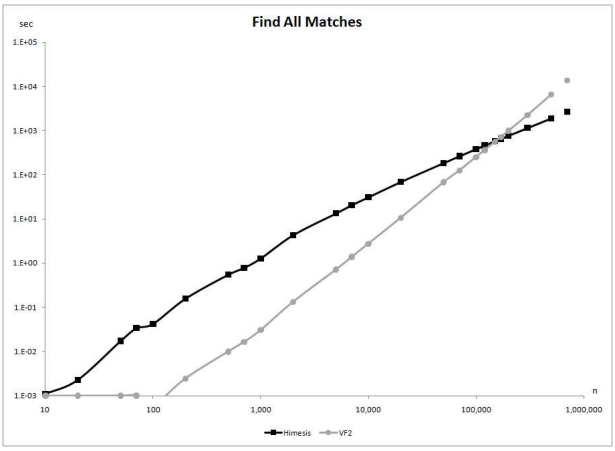
Let us first analyse the space complexity of the `extend` procedure. The state of the algorithm is encoded in the *state* variable. It holds the two partial mapping sets as well as all six terminal sets. Thus the number of nodes stored in the state is at most $5 \times |V(P)| + 3 \times |V(S)|$ which is linear in terms of the nodes of the source and pattern graphs. Moreover, since IGraph stores the nodes as integers, *state* is therefore quite compact. Additionally, the experiments below have shown that the algorithm performs better if the adjacency list (encoded as a hash table) is memoized as well. The size of this hash table is in the worst case $|V(P)|^2 + |V(S)|^2$ for fully connected, directed, simple graphs.

We now compare the time performance of the `extend` algorithm of Himesis with VF2's sub-graph isomorphism algorithm. We have chosen the IGraph implementation of VF2 as benchmark which is in direct correspondence with original implementation. Note that Himesis is implemented in Python whereas VF2 was implemented in C. According to http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=python&lang2=gcc, Python is in general slower than C by an average factor of 23, which is not integrated in the results presented here. In these experiments, we gathered the computation time with respect to the number of nodes n of the source graph. The source graph represents random valid class diagrams encoded as Himesis graphs. The average number of class diagram elements is shown in Figure 9(d) For each source graph we have run the algorithm on six pattern graphs which sizes range from 2 to 12 nodes. For both the source and pattern graphs, the number of edges is the same order as the number of nodes (which is typical in class diagrams). Each data point of the plots in Figure 9 represents the average time over the six pattern graphs.

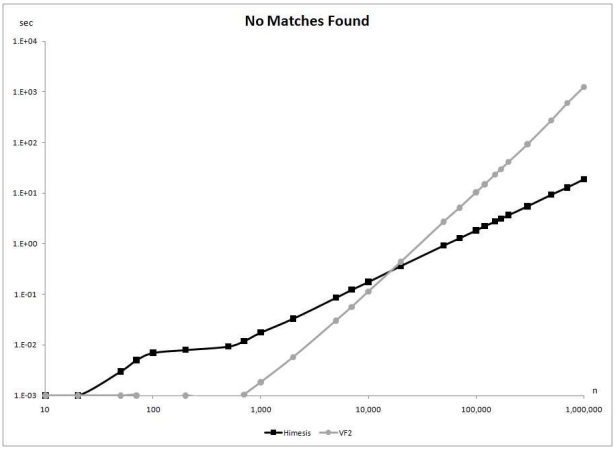
Figure 9(a) shows the performance of both algorithms for finding the first match only. For small graphs, VF2 is about 25 times faster than Himesis. For medium graphs, VF2 is twice as fast as Himesis. However at around 2.2×10^5 nodes, both perform as fast. At this point, Himesis takes over VF2 by a factor of 6 for large graphs.



(a)



(b)



(c)

Nodes		Edges	Class	Attribute	Associations	Matches	All	First	None
Small	$[0,10^3[$	399	51	45	51	729	0.016	0.04	0.034
Medium	$[10^3,10^5[$	33,727	4,133	4,249	4,196	83,766	0.067	0.125	0.348
Large	$[10^5,10^6]$	304,552	38,158	38,312	37,919	891,171	2.929	5.006	22.305

(d)

Figure 9: Average of sub-graph isomorphism matching over the six pattern graphs..

Figure 9(b) shows the performance of both algorithms for finding all matches. For small graphs, VF2 is about 60 times faster than Himesis. For medium graphs, VF2 is 5 times faster than Himesis. However at around 1.5×10^5 nodes, both perform as fast. At this point, Himesis takes over VF2 by a factor of 5 for large graphs.

Figure 9(c) shows the performance of both algorithms when no match exists. For small graphs, VF2 is about 24 times faster than Himesis. The medium graph category must be divided in two. For graphs with 10^3 and 10^4 nodes, VF2 is 3.6 times faster than Himesis. As for graphs with 10^4 and 10^5 nodes, Himesis takes over by a factor of 2.2. The break even point is around 1.7×10^4 nodes. At this point, Himesis takes over VF2 by 3 times for large graphs.

The table in Figure 9(d) summarizes these observations. Notice how Himesis outperforms VF2 significantly for large graphs.

4.2 Pattern Matching

The transformation kernel of AToMPM is *T-Core* [16]. In *T-Core*, the pre- and post-condition patterns of a rule are encoded as Himesis graphs. A pre-condition is composed of a positive condition graph (LHS) and optional negative condition graphs (NACs). Proposition (6) defines the semantics of a rule with n NACs: if an occurrence of the LHS is found in the source graph before the rule is applied and none of the NACs is found, then an occurrence of the RHS must be found in the source graph after the rule has been applied. A more formal definition based on category theory can be found in [17].

$$LHS \wedge \neg NAC_1 \wedge \neg NAC_2 \wedge \ldots \wedge \neg NAC_n \Rightarrow RHS \tag{6}$$

In Himesis, a node N of a pattern graph holds the following information:

- A universally unique identifier: such identifiers are ensured to be unique at all time.
- The type t of the model element N encodes: this represents the absolute path (across packages) of the name of the type element.
- A boolean flag *stm* specifying whether a source node mapped to N must of type t or a sub-type of t.
- The set *st* of all sub-type of t.
- The identifier of a binding pivot $\overleftarrow{x}$ (for pre-condition graphs): if specified, it predefines which source node that was assigned to the pivot x must be matched to N.
- The identifier of a pivot assignment $\overrightarrow{x}$: if specified, it indicates that the source node mapped to N will be assigned to the pivot x.
- A label global to the scope of the rule. Node labelling in the different pattern graphs of the rule is used as follows. In the LHS, a label allows one to distinguish between two nodes of the same type that must be mapped to different source nodes. A label present in both the LHS and the RHS or in both the LHS and a NAC corresponds to the same matched source node. A label present in a NAC but not in the LHS allows one to distinguish between two nodes of the same type that must be mapped to different source nodes.
- Each attribute of the meta-model element corresponding to t is subject to the RAM procedure [18]. In the LHS and the NAC, the node is assigned one constraint per attribute. The constraint can be of arbitrary complexity, but can only refer to source nodes bound to the corresponding pattern (LHS xor NAC). In the RHS, the node is assigned an action code per attribute. The action can be of arbitrary complexity, but can only refer to source nodes bound to the LHS pattern.

The size of the data stored in each pattern node is 1,342 bytes, without taking into consideration the meta-model attributes. Additional information is stored at the graph pattern level: the set of all meta-models involved in the pattern[7] as well as an additional constraint (for a LHS or a NAC) or action (for an RHS). The constraint and action follow the same conditions as for pattern node attributes.

[7] Because in AToMPM, rules can involve many meta-models as in *e.g.*, multi graph grammars [19].

Up to now, we have described an efficient solution for finding a sub-graph of the source graph isomorphic to the pattern graph. However, this is not sufficient for pattern matching as this only takes into account the topology of the pattern graph. Constraints and NACs must be taken into consideration as well. Therefore, Algorithm 3 specifies a procedure that extends the previous sub-graph isomorphism solution for pattern matching purposes. But we must first modify the `extend` procedure to handle constraints on meta-model attributes as well as node typing. The type of a pattern node p and a source node s must correspond. This requirement must be verified as early as possible to reduce the search space. We therefore modify the function `areCompatible` in Algorithm 1. More specifically, condition (4) must now take into considerations the types of the candidate pair (p,s) as specified in (7), such that the type of s is the same as the type of p or one of its sub-types. (4) can then be rewritten as:

$$|Out(p)| \leq |Out(s)| \wedge |In(p)| \leq |In(s)| \wedge ((s.t = p.t) \vee (p.stm \wedge s.t \in p.st)) \tag{7}$$

Additionally, the function `areSemanticallyFeasible` must ensure that the attributes held in s each satisfy the corresponding meta-model attribute constraints in p. Also, to help the algorithm find a match as soon as possible, we have parametrized the `suggestMapping` function with a priority mechanism to suggest a candidate pair. Our implementation allows to specify an arbitrary order of a terminal set. By default, `suggestMapping` will suggest an unmatched pattern node such that its type occurs the least often in the graph. This heuristic ordering can be modularly extended with further knowledge of the pattern graph and the source graph in our implementation.

The pattern matching algorithm in Himesis is described in Algorithm 3. The procedure `match` takes a source graph G and the LHS pattern graph as input. Pivot bindings may also be specified in the *context*. The procedure can be divided in three cases. In the following, we consider a match as *valid* if the source nodes in the mappings of the match satisfy the constraint of the pattern graph.

No NACs. When there are no NACs specified in the pre-condition pattern, then only lines 1,12-14 are applied. This simply calls the `computeMappings` procedure and returns the valid matches.

Unbound NACs. We denote a NAC as unbound if none of its nodes has a label present in the corresponding LHS. If the pre-condition has unbound NACs, it suffices to find one valid NAC match to prevent the pre-condition pattern from successfully finding any matches. Lines 3-14 describe this behaviour. First, G is matched on the NAC with the provided context. If no valid match is found, the procedure then tries to find matches for the LHS as in the previous case. Otherwise, no match is output.

Bound NACs. All other NACs are bound to the LHS. Since `computeMappings` is the most costly procedure, we want to avoid computing mappings twice, *i.e.,* the common part between the LHS and a NAC. Thus the idea is to first match the common part between the LHS and a NAC, then continue the matching along the NAC, and finally, if no valid NAC matches were found, continue from the match of the common part along the LHS.

A NAC having a common part with the LHS means that there is a sub-graph of the LHS that overlaps with the NAC. We denote this intersection as a pre-condition graph called *bridge*. In general, computing the bridge would require to find the maximum common sub-graph (MCS) between these two graphs. Solving the MCS isomorphism problem is NP-Complete. However, making use of the labels in the Himesis pattern graphs reduces the complexity to linear-time. Therefore the bridge can be constructed as follows: if a node has a label present in nodes of both the LHS and the NAC, then this node is part of the bridge. Also, every edge in the smallest graph between the LHS and the NAC whose source and target nodes are in the bridge is part of the bridge. However, recall that pattern nodes also hold a constraint for each meta-model attribute. Thus, each meta-model attribute of a bridge node is computed as the conjunction of the corresponding attribute constraint in the LHS and the corresponding attribute constraint in the NAC. Note that no constraint is added on the pattern graph of the bridge as in the LHS or NAC cases. It easy to show that the time complexity of constructing the bridge between the LHS and an NAC is $O(V+E)$, where $V = max(|V(LHS)|, |V(NAC)|)$ and $E = min(|E(LHS)|, |E(NAC)|)$[8].

In the `match` procedure, line 24 computes the bridge B with the largest number of nodes. Since a bridge can be statically computed, all bridges had already been precomputed and integrated in the corresponding NACs (at

[8] V should also be multiplied by the maximum number of meta-model attributes, which is small in practice.

Algorithm 3 `match`(G, LHS, context)

```
1:  validMatches ← ∅
2:  moreNACs ← False
3:  for NAC in LHS.getNACs() do
4:      bridge ← NAC.getBridge()
5:      if V(NAC.getBridge()) > 0 then
6:          moreNACs ← True
7:      else
8:          for nacMatch in computeMappings(G, NAC, context) do
9:              if NAC.checkConstraint(nacMatch) then
10:                 return ∅
11:             end if
12:         end for
13:     end if
14: end for
15: if not moreNACs then
16:     for lhsMatch in computeMappings(G, LHS, context) do
17:         if LHS.checkConstraint(lhsMatch) then
18:             validMatches ← validMatches ∪ {lhsMatch}
19:         end if
20:     end for
21:     return validMatches
22: end if
23: maxNAC ← LHS.getNACwithMaxBridge()
24: B ← maxNAC.getBridge()
25: for bMatch in computeMappings(G, B, context) do
26:     for maxNACMatching in computeMappings(G, maxNAC, bMatch ∪ context) do
27:         if not maxNAC.checkConstraint(maxNACMatching) then
28:             goto 20
29:         end if
30:     end for
31:     for lhsMatch in computeMappings(G, LHS, bMatch ∪ context) do
32:         if LHS.checkConstraint(lhsMatch) then
33:             for NAC in LHS.getNACs() do
34:                 if NAC ≠ maxNAC and V(NAC.getBridge()) > 0 then
35:                     for nacMatch in computeMappings(G, NAC, lhsMatch ∪ context) do
36:                         if not NAC.checkConstraint(nacMatch) then
37:                             validMatches ← validMatches ∪ {lhsMatch}
38:                         end if
39:                     end for
40:                 end if
41:             end for
42:         end if
43:     end for
44: end for
45: return validMatches
```

compile-time). On line 25, G is matched on B with the provided context. Then on lines 26-30, G is matched on the NAC corresponding to B. To prune the search space of this matching, the bridge mappings are provided as context together with the initial context. Those mappings are valid since the nodes in B are in the NAC as well. If a valid match for this NAC is found, then the current match of B is discarded and the next one is tried. When a match of B is found such that it does not induce a valid match, we match G on the LHS with again the bridge mappings provided as context together with the initial context. Each valid match of the LHS represents a potential valid match of the procedure. However, there may be additional bound NACs with a bridge having less nodes than B. In this case, lines 33-41 ensure that only the valid matches of the LHS that do not satisfy the remaining NACs are stored. Note that when applying the `computeMappings` procedure on G with the remaining NACs, the LHS mappings are provided as context together with any pivot node bound in the LHS that were given in the initial context. Finally on line 45, only the valid matches are output.

4.3 Rewriting the Matches

A rule is successfully applied when proposition (6) is satisfied. The pre-condition satisfaction is ensured by the pattern matching algorithm described previously. One way to satisfy the post-condition is to modify the matched nodes in the source graph adequately. To transform (or rewrite) the matches, a Himesis RHS pattern graph is provided with a compiled `execute` function. Given the LHS and the RHS pattern graphs, the rewriting of a match $M = \{(p,s)|p \in \text{LHS} \wedge s \in G\}$ can be statically determined. For each $(p,s) \in M$ we perform the following steps in that order:

1. If the label of p is present in both the LHS and the RHS, then p follows the *update operation*. Each attribute of s is set according to the action specified in the corresponding meta-model attribute of the RHS node that has the same label as p.
2. Let C represent the graph whose node labels are present in the RHS but not in K. Also edges of C are constructed in a similar way as for the bridge, *i.e.*, $E(C) = \{(n_i,n_j)|n_i,n_j \in V(C) \wedge (n_i,n_j) \in E(\text{RHS})\}$. Then the nodes and edges of C must follow the *create operation*. For each node (or edge) in $V(C)$ (or $E(C)$), a corresponding source node (edge) is created in the source graph. Furthermore, the attributes of the new nodes are initialised according to the action specified in the corresponding meta-model attribute of the respective node in C.
3. If the label of p is present in LHS but not in the RHS, then p must follow the *delete operation*: remove s from the source graph. Note that in IGraph, deleting a node automatically deletes its adjacent edges.
4. If p is assigned a pivot identifier $\vec{x}$, then $\vec{x}$ will be mapped to s.
5. Finally, after all nodes have been processed, we apply the action specified in the RHS on the source nodes that are in M as well as those created from C.

Since the rewriting phase is compiled, its run-time complexity is linear: $O(|V(\text{LHS})| + |E(\text{LHS})| + |V(\text{RHS})| + |E(\text{RHS})|)$ Note that according to graph transformation literature [20], Himesis' transformation procedure follows the Single-Pushout (SPO) approach in contrast with the Double-Pushout (DPO) approach. On the one hand, the identification issue of the gluing condition in DPO is avoided thanks to the labelling mechanism in place. That is because every node in each pattern graph is unique and thus may be mapped to exactly one node in each matching. On the other hand, we have explicitly chosen to sustain the dangling edges issue. That is if a matched source node must be deleted, all its adjacent edges will be deleted too. This has the advantage of reducing the number of rules in the transformation.

5 Related Work

The name *Himesis* was first introduced in [21]. In his masters thesis, Provost described an efficient framework for graph-subgraph isomorphism. The implementation of Algorithm 1 is based on his work. However, his approach does

not address pattern matching as used in model transformations. Also, there is no evaluation of the performance of each CRUD operation as done in this paper.

To compare our implementation of Himesis with other graph transformation approaches, we provide our results for a standard graph transformation benchmark: the *Distributed Mutual Exclusion Algorithm* benchmark presented by Varró in [22]. Although some measurements were reported in the original paper, Geiss et al. [10] provide a more complete spectrum of measurements with more tools. In the latter paper, the measurements were carried out on an AMD Athlon 3000+ with 1GB of RAM. To reuse these results, we multiplied[9] Geiss' figures by 0.684 to match the processor's speed specified in Section 2.

The tools used for this comparison are the following. GrGen.NET SP [10], FUJABA [23], and PROGRES [24] are transformation tools using search plan techniques for the matching phase. An approach from Varró [25] (hereafter referred to as VarroDB) to execute graph transformations directly in a relational database. GrGen.NET PSQL which, in contrast with GrGen.NET SP, also stores the graphs in a relational database. Finally, AGG [12] is the only tool that uses a CSP for the matching phase All experiments, were performed without any of the optimizations suggested by the benchmark as no measurement for these cases were available for the other tools. As Himesis provides a framework for manipulating graphs, we integrated it in *T-Core*, in combination with Python. More specifically, the *T-Core* matcher calls the procedure `match` form Algorithm 3 and the *T-Core* rewriter calls the `execute` method of the corresponding RHS graph to perform the rewriting.

For the Short Transformation Sequence experiment (STS), Figure 10(a) shows that *T-Core* performs averagely compared to the other tools. It however performs on average 5.6 times better than *AGG*, which is the only other tool whose matching phase is also implemented as a CSP. For the As Long As Possible experiment (ALAP), Figure 10(b) shows that, once more, *T-Core* performs averagely compared to the other tools. It however performs on average 9.2 times better than *AGG*. For the Long Transformation Sequence experiment (LTS), the only results available are for N=1,000 (N processes with one resource). Figure 10(c) shows that *T-Core* performs quite well compared to the other tools. It now performs on average over 100 times better than *AGG* and about as fast as GrGen.NET using ProgresSQL. The table in Figure 10(d) summarizes the results.

6 Conclusion

This paper contributes in providing an efficient framework for AToMPM. Himesis, the framework we developed based on IGraph, allows one to efficiently manipulate models encoded as graphs. The primitive CRUD operations are very fast even for models with up to 10^6 elements. Moreover, we have described the implementation of the pattern matching algorithm used to perform model transformation. The comparison of performance with other existing tools and approaches show that Himesis is indeed an efficient framework.

One reason for the average performance results for graph transformation tasks may be that Himesis is entirely implemented in Python. Future plans are to implement the core algorithms in a faster target language, such as C.

References

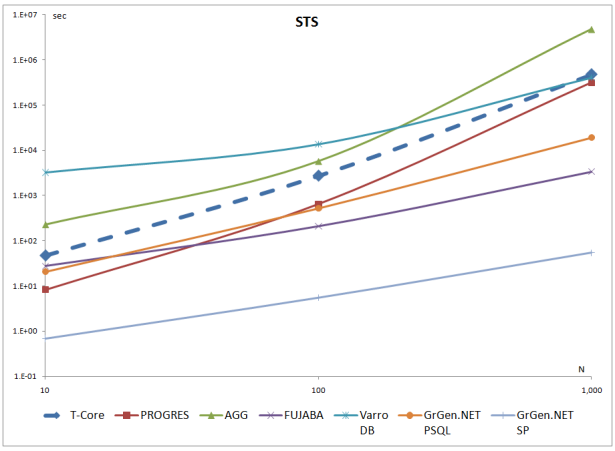
[1] Csárdi, G. and Nepusz, T. (2006) The igraph software package for complex network research. *InterJournal Complex Systems*, **1695**.

[2] Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008) Exploring network structure, dynamics, and function using NetworkX. In Varoquaux, G., Vaught, T., and Millman, J. (eds.), *SciPy'08*, Pasadena (USA), August, pp. 11–15.

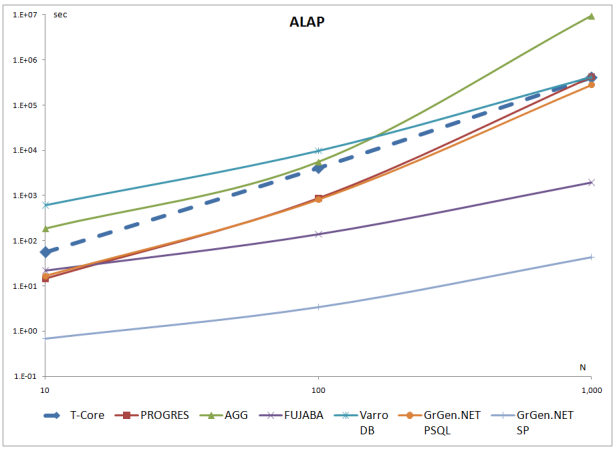
[3] de Lara, J. and Vangheluwe, H. (2002) AToM^3: A tool for multi-formalism and meta-modelling. In Kutsche, R.-D. and Weber, H. (eds.), *FASE'02*, Grenoble (France), April, LNCS, **2306**, pp. 174–188. Springer-Verlag.

[4] igraph.sourceforge.net (2009). Igraph Library v0.5.3.

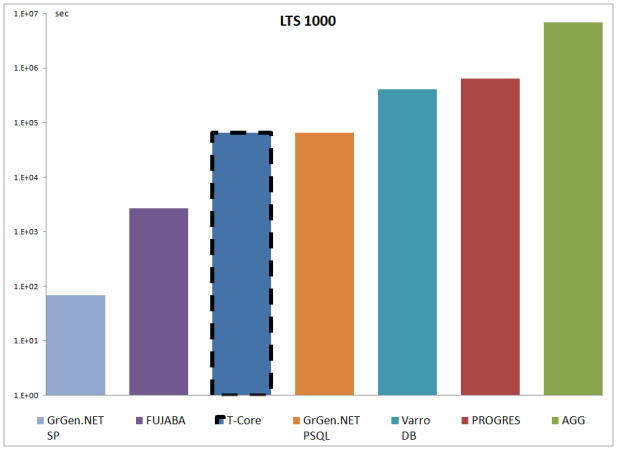
[9] This factor is obtain from the SPEC organization at http://www.spec.org/cpu2000/results/cpu2000.html.



(a)



(b)



(c)

Case	N	T-Core	PROGRES	AGG	FUJABA	Varro DB	GrGen.NET PSQL	GrGen.NET SP
STS	10	48	8	226	27	3,213	21	1
STS	100	2,745	647	5,677	209	13,560	520	5
STS	1,000	474,325	313,956	4,706,604	3,370	405,954	18,957	54
ALAP	10	56	14	185	22	611	16	1
ALAP	100	4,087	867	5,490	139	9,636	807	3
ALAP	1,000	407,493	417,650	9,339,336	1,930	408,211	277,704	44
LTS	1,000	65,929	644,396	6,840,000	2,651	405,749	65,996	68

(d)

Figure 10: Performance comparison for the Distributed Mutual Exclusion Algorithm benchmark with no optimization.

[5] networkx.lanl.gov (2010). NetworkX v1.1.

[6] Drewes, F., Hoffmann, B., and Plump, D. (2002) Hierarchical graph transformation. *JCSS*, **64**, 249–283.

[7] Object Management Group (2009) *Unified Modeling Language Superstructure*.

[8] Mehlhorn, K. (1984) *Graph Algorithms and NP-Completeness*, Monographs in Theoretical Computer Science. An EATCS Series, **2**. Springer.

[9] Zündorf, A. (1994) Graph pattern matching in PROGRES. In Ehrig, H., Engels, G., and Rozenberg, G. (eds.), *Graph Grammars and Their Application to Computer Science*, Williamsburg,USA, November, LNCS, **1073**, pp. 454–468. Springer-Verlag.

[10] Geiß, R., Batz, G. V., Grund, D., Hack, S., and Szalkowski, A. (2006) GrGen: A fast SPO-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., and Rozenberg, G. (eds.), *ICGT'06*, Heidelberg (Germany), September, LNCS, **4178**, pp. 383–397. Springer-Verlag.

[11] Varró, G., Varró, D., and Friedl, K. (2005) Adaptive graph pattern matching for model transformations using model-sensitive search plans. In Karsai, G. and Taentzer, G. (eds.), *GraMoT'05*, Tallinn (Estonia), September, ENTCS, **152**, pp. 191–205. Elsevier.

[12] Rudolf, M. (1998) Utilizing constraint satisfaction techniques for efficient graph pattern matching. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (eds.), *TAGT'98, Selected Papers*, Paderborn (Germany), November, LNCS, **1764**, pp. 381–394. Springer.

[13] Krissinel, E. B. and Henrick, K. (2004) Common subgraph isomorphism detection by backtracking search. *SPE*, **34**, 591–607.

[14] Ullmann, J. R. (1976) An algorithm for subgraph isomorphism. *Journal of the ACM*, **23**, 31–42.

[15] Cordella, L., Foggia, P., Sansone, C., and Vento, M. (2004) A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI*, **26**, 1367–1372.

[16] Syriani, E. and Vangheluwe, H. (2010) De-/re-constructing model transformation languages. *ECEASST*, **29**.

[17] Ehrig, H., Prange, U., and Taentzer, G. (2004) Fundamental theory for typed attributed graph transformation. In Ehrig, H., Engels, G., Parisi-Presicce, F., and Rozenberg, G. (eds.), *ICGT'04*, Rome (Italy), September, LNCS, **3256**, pp. 161–177. Springer-Verlag.

[18] Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., and Wimmer, M. (2010) Explicit transformation modeling. In Ghosh, S. (ed.), *MODELS 2009 Workshops*, LNCS, **6002**, pp. 240–255. Springer.

[19] Königs, A. and Schürr, A. (2006) MDI: A rule-based multi-document and tool integration approach. *SoSym*, **5**, 349–368.

[20] Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (1997) *Handbook of graph grammars and computing by graph transformation, Volume 1: Foundations*. World Scientific Publishing Co., Inc.

[21] Provost, M. (2005) Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis. McGill University Montréal (Canada).

[22] Varró, G., Schürr, A., and Varró, D. (2005) Benchmarking for graph transformation. *VL/HCC'05*, Dallas (USA), September, pp. 79–88. IEEE Press.

[23] Fischer, T., Niere, J., Turunski, L., and Zündorf, A. (2000) Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (eds.), *Theory and Application of Graph Transformations*, Paderborn (Germany), November, LNCS, **1764**, pp. 296–309. Springer-Verlag.

[24] Zündorf, A. and Schürr, A. (1992) Nondeterministic control structures for graph rewriting systems. In Mayr, E. W. (ed.), *Graph-Theoretic Concepts in Computer Science*, Fischbachau,Germany, May, LNCS, **570**, pp. 48–62. Springer-Verlag.

[25] Varró, G., Friedl, K., and Varró, D. (2006) Implementing a graph transformation engine in relational databases. *SoSym*, **5**, 313–341.