Himesis Representation of ArkM3 Structures

Simon Van Mierlo

June 30, 2013

1 Introduction

AToMPM, A Tool for Multi-Paradigm Modelling is a (meta)modelling tool currently under development. The tool allows the explicit modelling of modelling language, through the construction of a metamodel. A model created in a language is said to conform to the metamodel of that language.
Models are related to each other by model transformations. A transformation receives a model in the domain language as input and produces a model in the target language as output. A transformation is modelled by a number of rules and their scheduling. Each rule rewrites a part of the input model, resulting in a part of the output model.
Transformations are a special kind of model, conforming to the transformation metamodel. As such, they can be transformed by so-called Higher-Order Transformations (HOTs). These transformation models, however, contain condition code, describing input model constraints for each rule, and action code, describing a set of actions to be executed after rewriting the input model. From this, it follows that action and condition code should conform to a metamodel as well, and not be coded in, for instance, a string.
ArkM3 is the new metamodel kernel of AToMPM. In ArkM3, every element is explicitly modelled and conforms to its metamodel. In particular, action and condition code is metamodelled explicitly. ArkM3 structures are physically represented as Python objects. A textual notation (i.e. visual concrete syntax) has been developed, including a parser, which creates Python structures from a textual definition, and a pretty printer, which does exactly the opposite. An interpreter and compiler have been developed, attaching semantics to the models of actions and constraints.
In AToMPM, transformations are implemented as graph rewriting algorithms. To enable transformation of ArkM3 structures, a second physical level is introduced in this report. Each ArkM3 structure is represented as a graph or a set of nodes in a graph. The graph structure of our choice is Himesis, which is an abstraction layer on top of the graph library *igraph*. For each ArkM3 element, a suitable mapping onto Himesis nodes and/or graphs has to be made. This mapping is then used to construct the physical representation in Himesis of each element when that element is constructed. This allows the ArkM3 structure to be manipulated on two different levels: the ArkM3 level (for instance, executing an action which changes the value of a model property in ArkM3) and the Himesis level, by executing a transformation.
This report is structured as follows. In Section 2, the layered structure of ArkM3 is further explained. In Section 3, the design choices of the mapping implementation is detailed.

Lastly, in Section 4, the actual mappings of ArkM3 elements onto Himesis structures are presented.

2 ArkM3 Layered Structure

In this section, the layered structure of ArkM3 is introduced, using an example. In Section 2.1, the example is explained. In Section 2.2, the example is represented in concrete visual syntax, both visually and textually. Lastly, in Section 2.3, the physical representation of the ArkM3 structures is given, both in ArkM3 and Himesis.

2.1 Example

The example features one top-level package called *atompm*, which contains a sub-package *test*. This package contains a class called *A*, which in turn contains an action with one statement. Although the example is small, it is representative, as it contains most major ArkM3 structures. In the next section, the concrete syntax of the example is given.

2.2 Concrete Syntax: Textual and Visual

The concrete textual syntax of the example is shown in Listing 1. This textual notation is parsed, which results in the object diagram shown in Figure 1. The object diagram is an instantiation of the ArkM3 metamodel presented in [1]. As can be seen, a number of SetValues and SequenceValues are introduced to implement one-to-many relations between objects.

```
package atompm
        package test
                class A:
                        action a:
                                return 1 + 5 < 4
```

Listing 1: Textual Concrete Syntax

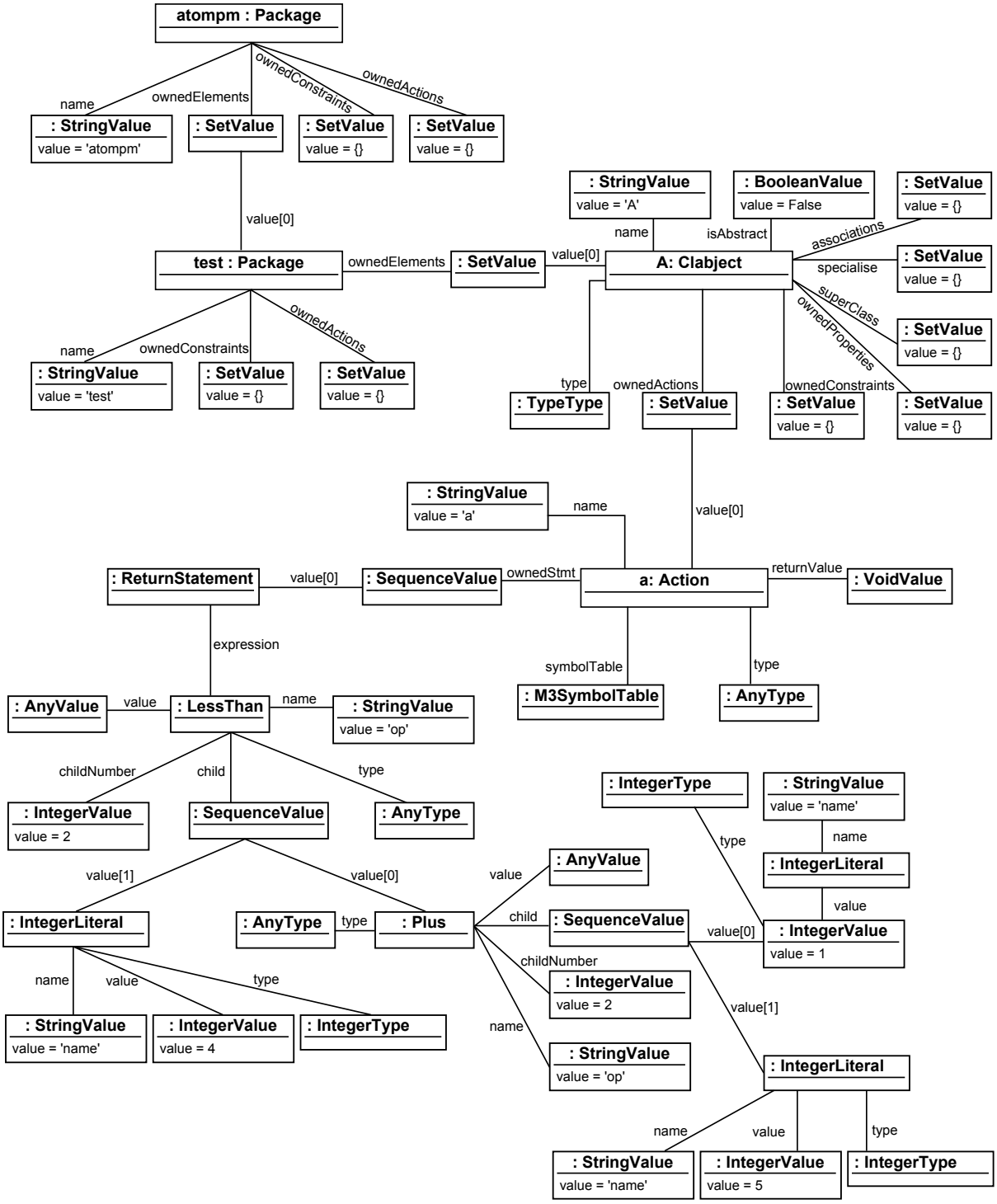


Figure 1: Visual Concrete Syntax: Object Diagram

2.3 Physical Realization

ArkM3 structures are physically realized on two levels. First, there is the Python level, where the ArkM3 classes to create the necessary structures are implemented. Second, each ArkM3 structure is physically realized as a (number of) Himesis node(s) and graphs. Himesis allows the creation of attributed graphs containing attributed nodes which are connected by directed, attributed edges. It also supports hierarchical graphs, by allowing a node in a Himesis graph to contain another Himesis graph. In Section 3.1, the graph kernel Himesis is further explained. In the next two subsections, both the physical realization in Python and in Himesis of the example are explained.

2.3.1 ArkM3

In Listing 2, the physical realization of the example in Python is shown. It shows the Python code which is used to construct the ArkM3 structure. The resulting object graph is visually represented in the object diagram of Figure 1.

```
expression = LessThan(
                Plus(
                        IntegerLiteral(
                                value = IntegerValue(1)
                        ),
                        IntegerLiteral(
                                value = IntegerValue(5)
                        )
                ),
                IntegerLiteral(
                        value = IntegerValue(4)
                )
        )
statement = ReturnStatement(expression)
action_a = Action(StringValue('a'))
action_a.add_statement(statement)
class_A = Clabject(StringValue('A'))
class_A.get_ownedActions().add(action_a)
pkg_test = Package('test')
pkg_test.add_ownedElement(class_A)
pkg_atompm = Package('atompm')
pkg_atompm.add_ownedElement(pkg_test)
```

Listing 2: Physical Realization: Python

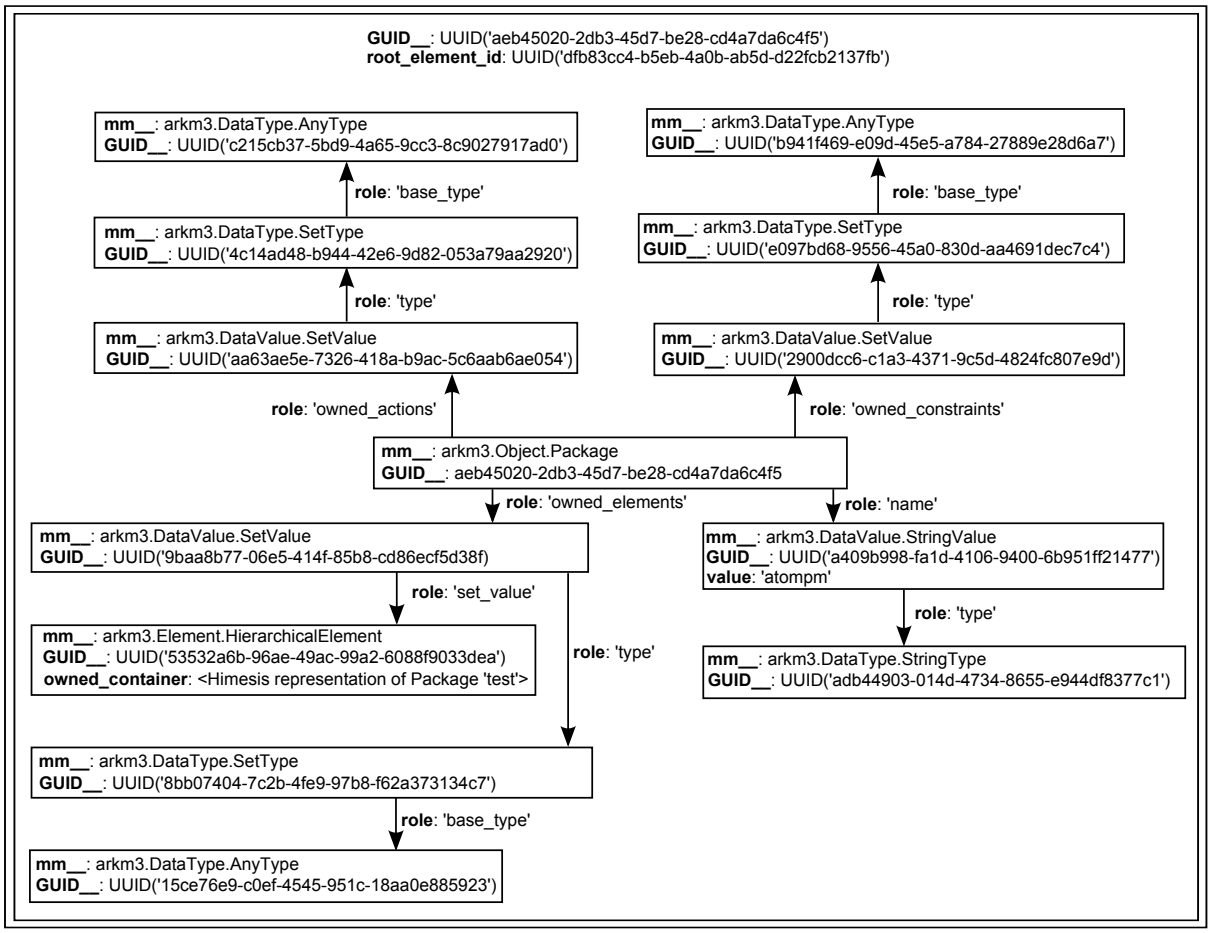


Figure 2: Physical Realization: Himesis

2.3.2 Himesis

In Figure 2, a part of the example's physical realization in Himesis is shown. Due to space constraints, only the top-level package is shown. A package is represented by a Himesis graph, denoted by the large double-sided square. Inside of the graph, its contained nodes are represented by squares.

The graph has two attributes: a unique identifier and a root element id, which points to the root node of this graph (in this example, the node of the 'atompm' package).

Every ArkM3 element is mapped onto exactly one Himesis node, possibly connected to a set of other Himesis nodes in the same graph. This results in the Package being mapped onto a graph, containing one root node which is connected to a set of other nodes.

A Himesis node contains a number of attributes. Each Himesis node representing an ArkM3 element has at least two attributes. The first is called **mm__**, which is the fully expanded class name of the corresponding ArkM3 element. The second is a unique identifier.

An ArkM3 element has a number of child elements. For instance, a Package has a child *StringValue*, representing its name. In the corresponding Himesis graph, a child is represented as an edge between the source node (corresponding to the Package) and the target node (corresponding to the StringValue). Every edge has a 'role' attribute.

This attribute represents the role which the target node fulfils in the source node. For instance, the role of the Package's StringValue is 'name'.
The 'atompm' package has exactly one owned element: the 'test' package. In Himesis, this is realized by using hierarchy. A *HierarchicalElement* is introduced, which is mapped onto a node containing an attribute 'owned_container'. The value of that attribute is another Himesis graph, in this case the Himesis representation of the 'test' package.
Note that there is a one-to-one mapping of each element in Figure 1 to an element in Figure 2. A detailed explanation of all ArkM3-to-Himesis mappings can be found in Section 4. There, a visual syntax is introduced to represent the mappings, which is more clear and generalized than the one used in this example.

3 Design

This section presents a number of design choices made during the development of the ArkM3-to-Himesis mappings. Firstly, in Section 3.1, Himesis, the graph structure onto which each ArkM3 element is mapped, is introduced. Next, Section 3.2 lists the design considerations and choices which were made during the development of the ArkM3-to-Himesis mappings. Lastly, Section 3.3 presents a performance analysis of a number of possible ArkM3-to-Himesis mappings. The results of this analysis ultimately lead to the mappings presented in 4.

3.1 Himesis: an *igraph* Abstraction Layer

Himesis, introduced in [2], is a kernel for graph-based model representation and manipulation. It is implemented using the *igraph* library, as described in [3]. This has a few consequences when considering what a good ArkM3-to-Himesis mapping is. For instance, Himesis allows for hierarchical graphs, which means that a node in one graph can contain another Himesis graph. It also allows for nodes and edges to have arbitrary Python structures as attributes. There are a number of decisions to make: how much information is represented as attributes of nodes and edges? How many nodes and edges do we want a typical graph to contain? Do we regularly want to make use of hierarchy? In this and later subsections, a number of rules are constructed that guide the ArkM3-to-Himesis mappings.
The choice of Himesis and *igraph* has a few advantages and drawbacks, as described in [3]. The ones that are important for constructing the ArkM3-to-Himesis mappings are repeated here.

- In *igraph*, nodes are not explicitly stored. The internal structure only keeps track of the total number of nodes. As a result, the attribute values of a node are not stored in the node itself, but in a vector which is globally assigned to the graph. This means that the required memory space for each attribute is assigned for all nodes. This leads to our first conclusion, which is to avoid excessive use of attributes. What should definitely be avoided is the use of attributes which are dependent on a run-time property of the system (for instance, the identifier of a node), as this would lead to an explosion in the number of attributes as the size of the graph increases.

- Both nodes and edges can have attributes. In constructing possible ArkM3-to-Himesis mappings, this fact should be taken into account. It may be that having particular attributes on edges instead of nodes is more efficient. This is examined in Section 3.3.

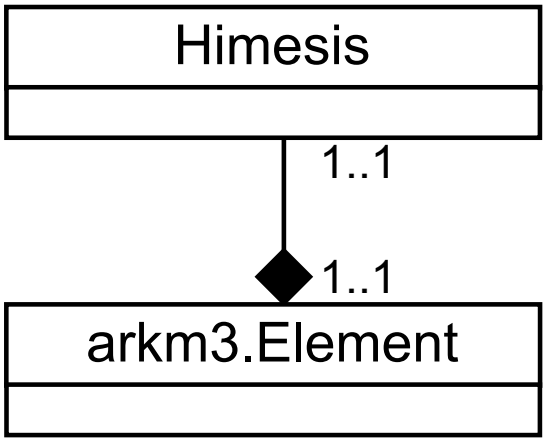
- As was said previously, Himesis allows for hierarchical graphs. Whether or not hierarchy should be regularly used is explored in the following subsections and Section 3.3.

3.2 Design Choices

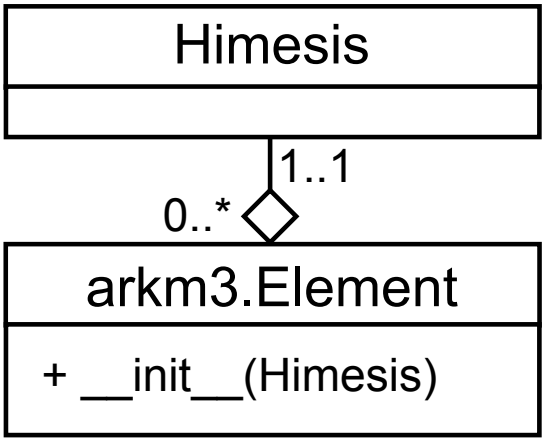
This section describes the design choices made when implementing the mapping of ArkM3 elements to Himesis graphs. Each ArkM3 structure is represented by a Himesis (sub)graph, which is used for executing transformations (using T-Core [3]) and (de)serialization. This section does not explain how each ArkM3 element is mapped onto Himesis, but rather how the design of the solution that implements it was constructed. The choices which are made here will influence what the final ArkM3-to-Himesis mappings will look like.
Section 3.2.1 explains the advantages and disadvantages of using different levels of hierarchy in the Himesis representation. Section 3.2.2 explains how ArkM3 elements are (de)serialized and Section 3.2.3 deals with transformations and how they are performed on ArkM3 structures.

3.2.1 Hierarchy

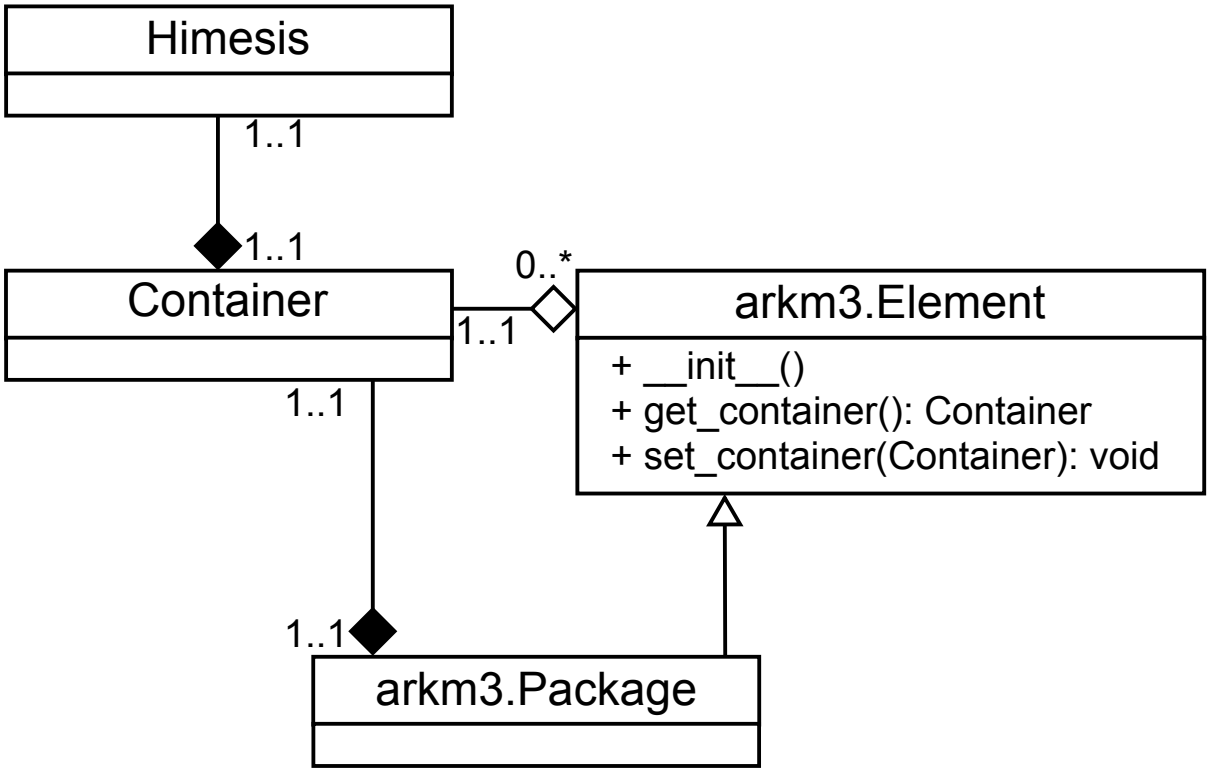
Starting from the fact that each ArkM3 element should be represented in Himesis, a first idea would be to represent each ArkM3 element as a separate Himesis graph, as presented in Figure 3a. This has a few advantages. For one, it is easy to implement, as each ArkM3 element now 'has-a' Himesis graph. By letting each Element own an instance of a Himesis graph, each element can manage its own graph representation. It also means each ArkM3 element can be the subject of transformation, as we can transform all Himesis graphs. However, this may not be exactly what we want. Does it really make sense to only transform an *IntValue* without taking into accounts its surrounding context (an action, a constraint, a property, ...)?
A number of other disadvantages exist with this approach. As each element is represented as a Himesis graph, we need to make excessive use of hierarchy. A Himesis graph consists of a number of nodes and a number of edges, connecting those nodes. The nodes can have attributes, and these attributes can be arbitrary structures, even other Himesis graphs. It is, however, not possible to connect two graphs with each other. This means that whenever an ArkM3 element is associated with other ArkM3 elements (which is often the case: an *Action* contains a *SequenceValue* of *Statement*, which can contain an *Expression*, etc.), we need to represent this as an attribute of the node. As was explained in [3] (and confirmed by the performance analysis in Section 3.3), we should try to avoid the use of a large number of attributes. When we look at how Himesis graphs are serialized, a second disadvantage becomes apparent. When hierarchy is encountered, the serialization process creates a new file for each Himesis graph. The name of the file corresponds to the name of the attribute, which means that when two *MappingValues* are serialized, and they both have the attribute 'a', which has as value a Himesis graph, these graphs would



(a) All Elements are mapped onto a Himesis graph.



(b) All Elements are mapped onto a set of Himesis nodes.



(c) Addition of the Container class.

Figure 3: Different types of ArkM3-to-Himesis mappings.

be serialized to the same file, which is of course not possible. A solution here would be to modify the serialization process, or introduce unique names for each attribute. However, the names of the attributes would then depend on run-time properties of the system, which we already decided against.
A second approach regards each ArkM3 element as a (set of) node(s) in a Himesis graph, presented in Figure 3b. This would mean that when an ArkM3 element is created, these nodes are created as well. However, Himesis nodes can only be created inside of a Himesis graph, not independently from it. So, when creating an ArkM3 element, a Himesis graph should be passed to the constructor of that element, such that it knows where to create its Himesis nodes.
An advantage of this approach is that excessive use of hierarchy is avoided. We now somewhere have a 'root' Himesis graph, where elements create their nodes and connect them with each other. Only these 'root' Himesis graphs can be serialized and transformed. However, then the question arises what these 'root' graphs are. It should be those ArkM3 elements that can be the subject of a transformation, such as packages.
A question with this approach is whether all elements should always have a representation in Himesis. For instance, we could first create an *Action* and later add it to a *Package*. As an action in itself is not a Himesis graph, it first has to add its nodes to another Himesis graph than the package (as the package is not created yet). When it is added to the package, it has to move its nodes to the Himesis graph representing that package. This is double work: do we really want a 'default' graph where each element not added to a package can 'park' its nodes? A solution is to allow ArkM3 elements to not have a representation in Himesis 'for a while' and only create the representation when they are added to an ArkM3 element that is represented by a Himesis graph. This makes sense: only a small set of ArkM3 elements can be the subject of a transformation, and we only want to serialize a small set of ArkM3 elements. So, we don't need the Himesis representation for the other ArkM3 elements unless they are part of an element which is in that small set.
We end up with the design in Figure 3c. A class called *Container* is added as an indirection layer between ArkM3 and Himesis. This class exposes some methods to make the setting and getting of attributes, creating of edges, etc. easier for ArkM3 elements. For now, only an ArkM3 *Package* creates a container on construction. The rest of the e lements can add their Himesis representation to a container, but the point at which this happens is decided by the container. For instance, an action could decide that a statement has to add its representation when the 'add_statement' method is called on itself. The result of all this is that for the user, the interfaces of the ArkM3 elements looks the same as before. In the background, the Himesis representation is available for those cases where it is needed. Of course, a number of methods managing the ArkM3 representation are added to the interface of Element (such as *get_container* and *set_container*). However, these should not be used outside of the Element class and its subclasses. In other programming languages, these methods would be *protected*, but Python does not allow for such scoping rules. As such, users of the class should be aware that they never should call these methods.
An advantage of this approach is that an element is free to choose its representation inside of the Himesis graph. The most optimal can be chosen (see Section 3.3). A second advantage is that only elements which can be transformed are represented by Himesis

graphs. This means that no graph has to be built at runtime, when the transformation is to be executed. The graph is readily available, as the element itself has the graph. This representation also allows for the caching of values. Instead of always looking at the Himesis graph for the correct value, the ArkM3 structure can keep its own data structures and perform a look-up faster that way.
As we now have a way to create the Himesis representation of ArkM3 elements, we can look at other functionalities which have to be implemented: (de)serialization of ArkM3 elements and transformations.

3.2.2 Serialization

There are two options for serialization of ArkM3 structures: either a completely new serialization process is created, tailored to ArkM3. Or, we reuse the already existing serialization process for Himesis graphs.
When using the already existing serialization process for Himesis graphs, no extra work has to be performed when serializing an ArkM3 element. Only elements that have a container, such as Package, can be serialized, as only they are represented by a graph in Himesis.
However, when we deserialize this Himesis graph, we may be interested in rebuilding the ArkM3 structure represented by that Himesis graph. So, extra work has to be performed. One solution is to add two extra (optional) parameters to the constructor of each ArkM3 element, specifying the container and identifier of a specific node in that container, which is the physical representation of the ArkM3 element. The element 'knows' what its representation looks like, so starting from that node, it can rebuild its internal structure. A factory is created which creates the ArkM3 element starting from the deserialized Himesis graph. The Himesis graph needs to save what the 'root' of the graph is (for instance, a *Package*), and the factory then uses this information to build the 'root' ArkM3 element. This element subsequently creates its children, by traversing the Himesis structure (for instance, a Package instantiates a SetValue for its owned Elements, this SetValue instantiates its child elements, and so on). A prerequisite for this to work is that a Himesis node contains sufficient meta-information to construct the ArkM3 element with which it corresponds.
A disadvantage of this approach is that each Element requires two new parameters in its constructor and all other parameters have to be made optional. This could be confusing for the user, as we may want to make some parameters obligatory.
Creating a completely new (de)serialization process requires extra work at the point of serialization, instead of at the point of deserialization. Instead of saving the Himesis structure, we save the ArkM3 structure. This could be implemented as a visitor, which visits the top-level element (a container), then traverses down the tree and makes sure all elements are added to the correct container.
From these two, we chose to use the existing Himesis serialization process, as it allows a Himesis structure to be deserialized independently of ArkM3. This means that without knowledge of ArkM3, a serialized Himesis graph can be deserialized, rewritten, and serialized again.

3.2.3 Transformations

Transformations are performed on Himesis structures. As was said in previous sections, some ArkM3 structures have containers, which are Himesis graphs. These containers can be transformed. However, the changes on the Himesis level should be reflected on the ArkM3 level. As the ArkM3 structures cache their values instead of always querying the Himesis structure (as this would lead to very inefficient code), there should be a way to invalidate this cache.
One way to achieve this would be making the Himesis structure representing the ArkM3 structure and the transformation engine ArkM3-aware: the Himesis nodes keep a reference to the ArkM3 structure they represent (this reference should not be serialized, for obvious reasons) and the transformation engine would then edit the ArkM3 structures in its rewrite phase.
Another, much easier, solution would be to introduce a 'dirty' attribute on each Himesis node. The ArkM3 structure should then check whether this flag is set each time it is accessed. If the flag is set, it should rebuild its internal structure. It remains to be investigated how efficient this approach is. However, it is clear that transformations are not performed very frequently, and that rebuilding part of the ArkM3 structure should not introduce a significant performance hit.

3.3 Performance Analysis

This secion describes the results of analysing the performance of different of ArkM3-to-Himesis mappings. Two ArkM3 structures were included in the test: SequenceValue and MappingValue. These are representative cases, as they require iteration and should be optimized for various frequent CRUD operations.

3.3.1 A Note on String Comparisons

The performance of String comparisons was tested in Python, to see whether performance could be gained by replacing all Strings by integers in the Himesis structures. This was done on two levels: first, we investigated the difference between String and integer keys in Python dictionaries. Then, String and integer comparisons were compared by searching for a certain value in a list.
For the first test, we found that Python's dictionary implementation is optimized for String keys. We conclude that using String keys instead of integer keys improves the overall performance.
For the second test, we compared the performance of comparing four different types of values: unique identifiers (UUIDs), integers, strings, and strings that were mapped onto integers (so to find such a string, we looked up the integer it was mapped onto, and looked up that value in the list). The list of strings consisted of strings with a fixed (random) 18-character prefix, a fixed (random) 18-character postfix and a number in the middle. When searching for a string, the tester was asked to input a value from the list to search for. The list of UUIDs consisted of randomly generated UUIDs, and the list of integers consisted of non-zero, increasing values. The results are summarized in Figure 4.

As we can see from the figure, all comparison functions have a linear complexity, which is to be expected. However, comparing UUIDs is significantly slower than comparing other

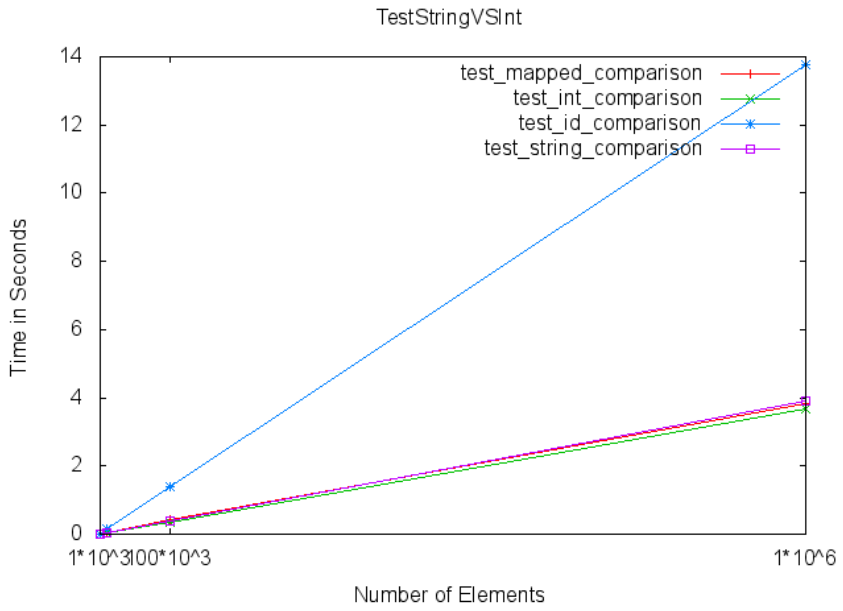


Figure 4: Performance graph of comparing values of different data types.

data types. Comparing strings and integers have (almost) equal performance. As such, we can conclude that, performance-wise, it does not make a difference whether we work with strings or integers when creating the ArkM3-to-Himesis mappings.

3.3.2 SequenceValue

The first ArkM3 structure to be tested was the SequenceValue. A SequenceValue has a number of children, which are accessed by an index. The currently supported operators are append and remove. The append operation adds an element to the end of the list, while the remove operation removes the first value from the list that matches a given value.

Two mappings to Himesis were tested, as shown in Figure 5. The first has a root node for the SequenceValue. The contents of the list are represented by nodes which are connected to this root node. Each child node keeps track of its index in the list. It is easy to deduce that an append operation has complexity **O(1)** (we keep track of the length of the list) and the remove operation **O(n)** in the worst case (all subsequent neighbors have to have their 'idx' attribute updated).

The second mapping also has a root node, but uses a more traditional representation for a linked list. The SequenceValue keeps track of the head of the list, and each node has a link to the next. Again, the append operation has complexity **O(1)**. The remove operation, however, has complexity **O(1)** as it is possible to retreive the previous and next neighbor of the deleted node in the Arkm3 structurs, and connect those, in constant time.

Three tests were run. The first measured the time to initialize the SequenceValue with a number of elements. These timings are shown in Figure 6. In the figures, 'node_idx' refers to the left mapping of Figure 5 and 'edge_attr' to the right one. As can be seen, these timings do not differ significantly between mappings. Do note that in both cases it

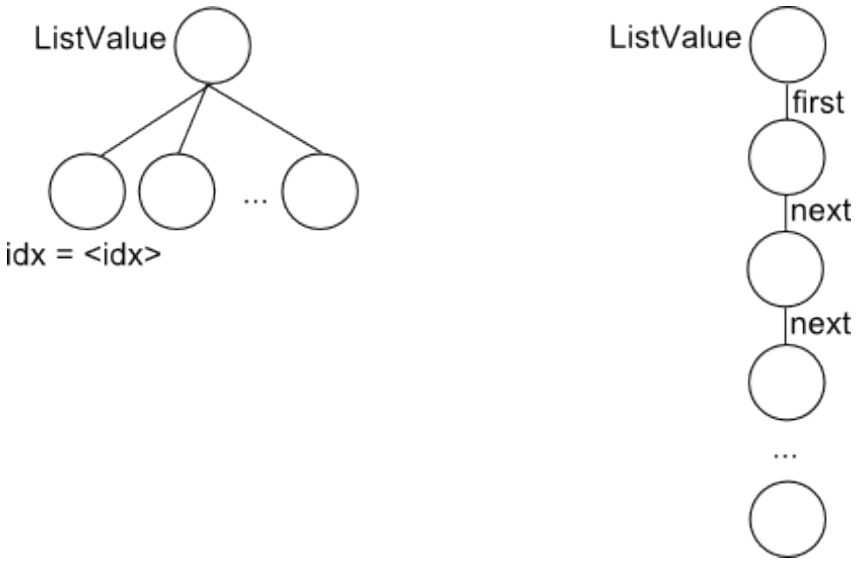


Figure 5: The two mappings which were tested.

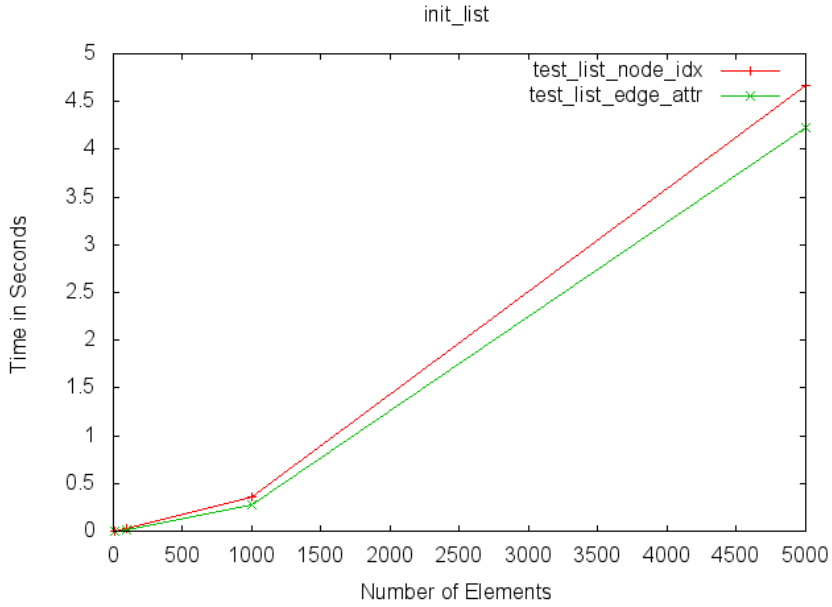


Figure 6: The timings for the initialize test.

takes a considerable amount of time for a list to initialize. Future work will investigate where the largest computational cost is.
The second test measured the time it takes to create a SequenceValue instance starting from the Himesis structure. This operation will be performed when deserializing a structure, as only Himesis structures are serialized. Figure 7 shows the results. As we can see, when we use the approach with 'next' edges, it takes more than two minutes to read in a graph of moderate size, while this is not the case for the other approach.
When an element is removed, the approach with the 'next' edges is again faster. (see Figure 8). This is to be expected, and can be explained by the complexities of the operations above.

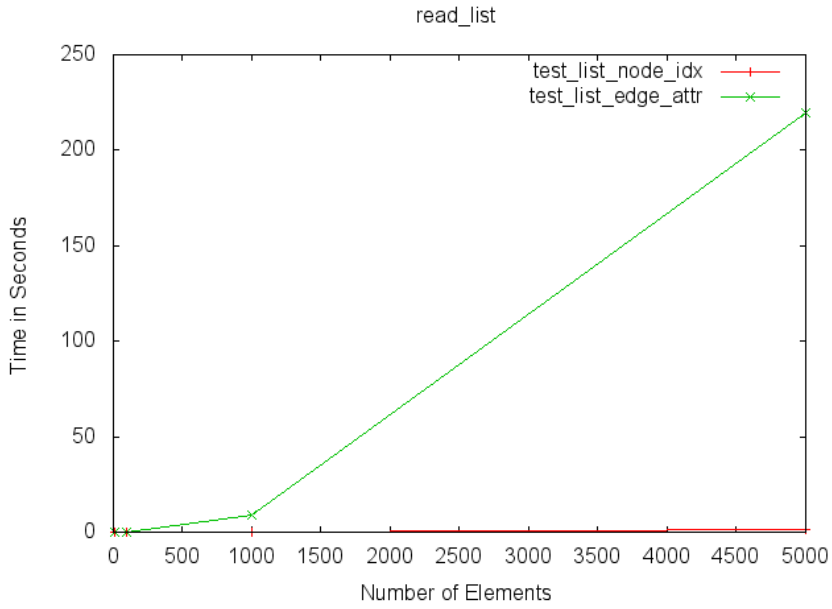


Figure 7: The timings for the read test.

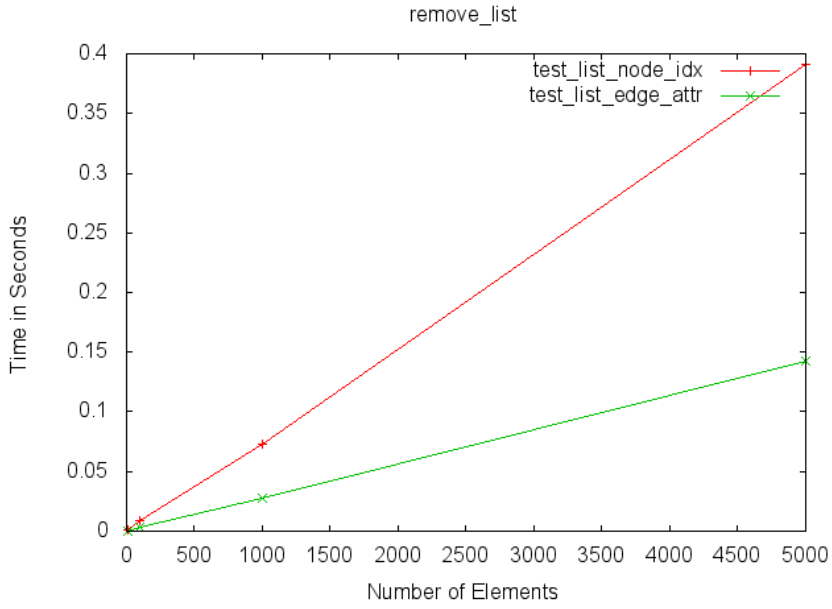


Figure 8: The timings for the remove test.

3.3.3 MappingValue

The second ArkM3 structure to be tested was the MappingValue, which basically has the same functionality as the Python dictionary. In Figure 9, four mappings which were tested are shown. The top left one is similar to the first mapping for lists: the values of the map are connected to the root node, and have as attribute the mapping key (this mapping is called 'node_key' in the result graphs). In the top right one, the key attribute is moved to the edge which connects the value to the root nodes (this mapping is called

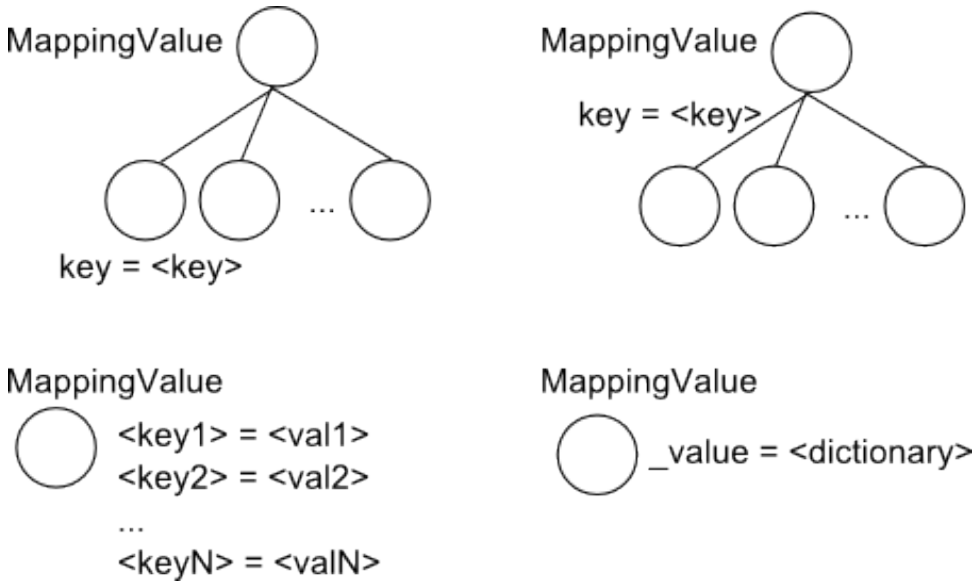


Figure 9: The three mappings which were tested.

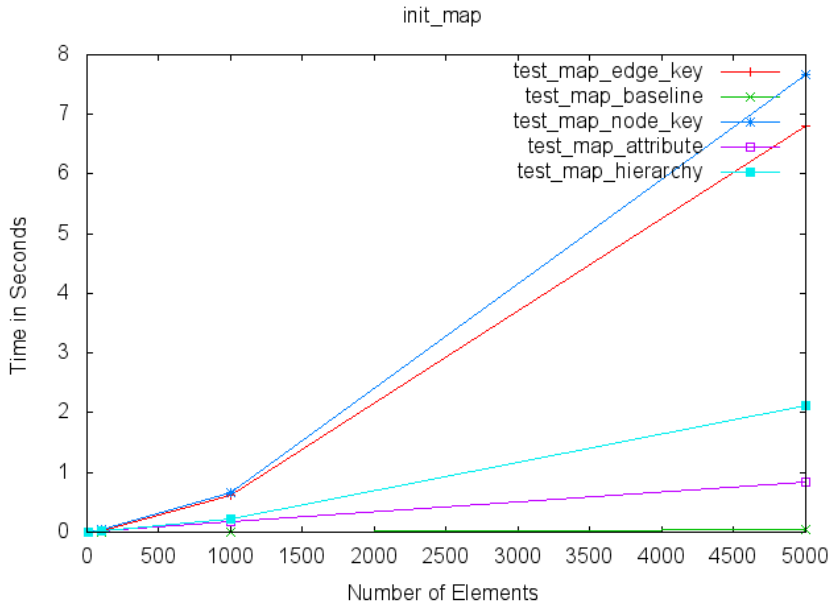


Figure 10: The timings for the initialize test.

'edge_key'). On the bottom left, all key-value pairs are represented as attributes of the root node (this mapping is called 'hierarchy'). The attribute name corresponds to the key, the value corresponds to the mapping value (which is a Himesis node). The bottom right mapping also has one node for the MappingValue, and only one attribute. This attribute has the name '_value' and the value of this attribute is the Python dictionary representing the value of this MappingValue. This mapping is called 'attribute'.
Four tests were run. The first test, whose results are shown in Figure 10, initializes the map with a certain amount of values (this operation has complexity **O(n)** for all mappings). The second reads the Himesis structure to build the corresponding ArkM3

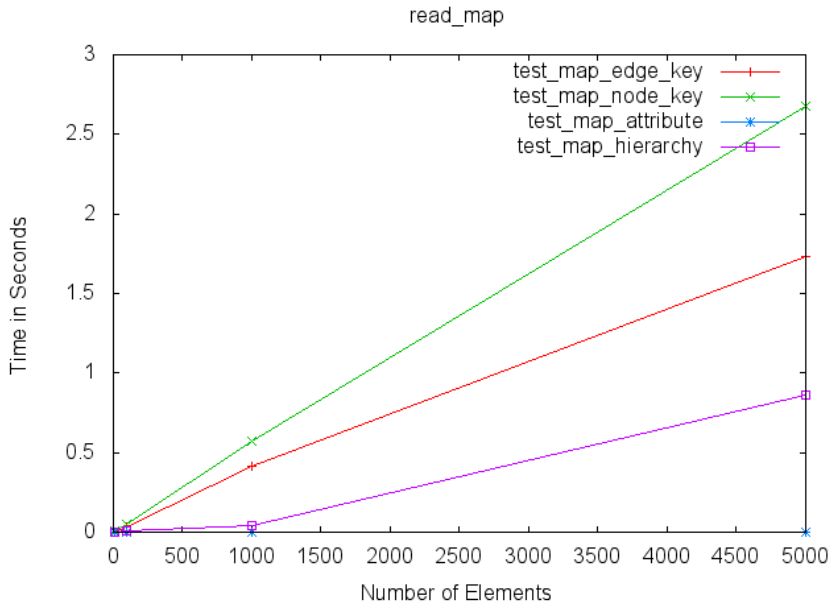


Figure 11: The timings for the read test.

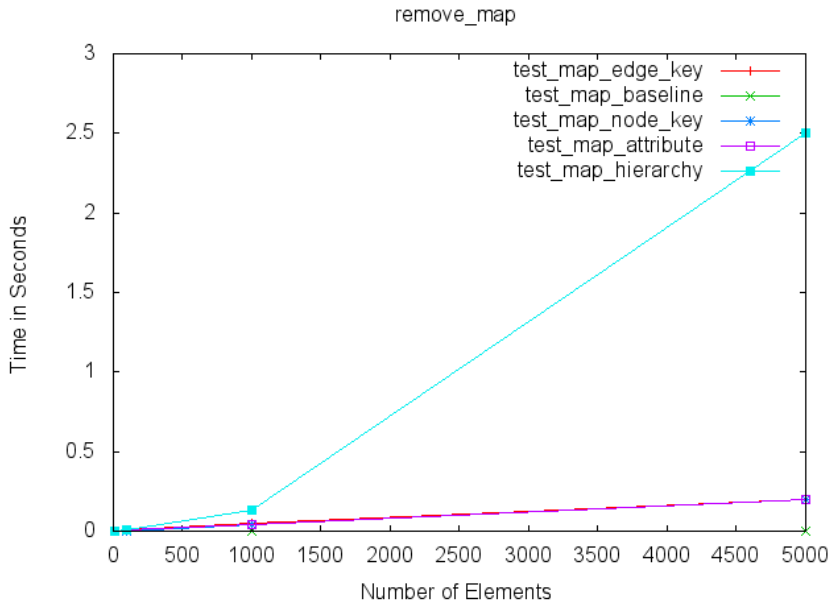


Figure 12: The timings for the remove test.

(MappingValue) structure - see Figure 11 for results (again, this operation has complexity **O(n)** for all mappings). The third removes a value from the map (this operation has complexity **O(1)** for all mappings) - see Figure 12 for results. The last, shown in Figure 13, sets the value of a key in the map to a certain value (this operation has complexity **O(1)** for all mappings). As we can see, the mapping which keeps the mapping as attributes of the root node is by far the slowest. The time it takes to remove or set a value grows exponentially with the size of the map. This is most likely caused by the fact that igraph stores all attributes on all nodes of the graph, which means the amount of

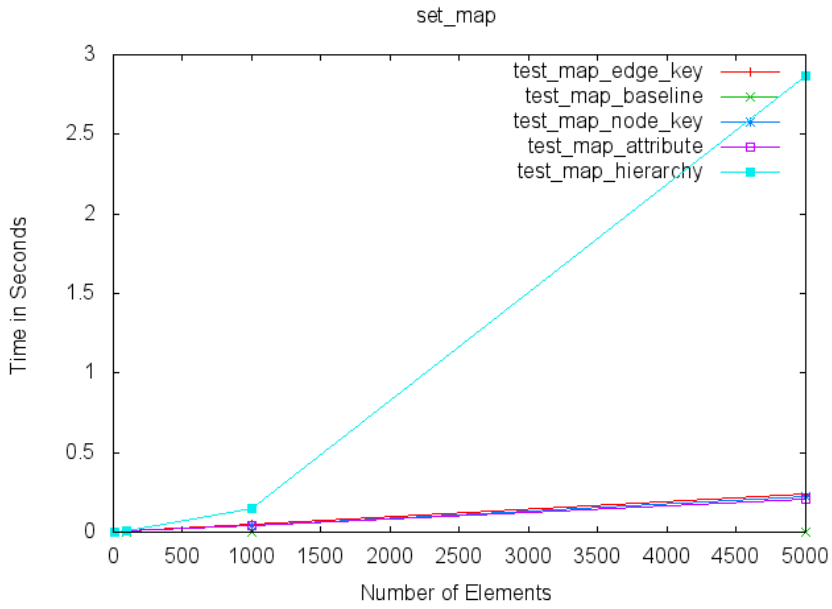


Figure 13: The timings for the set test.

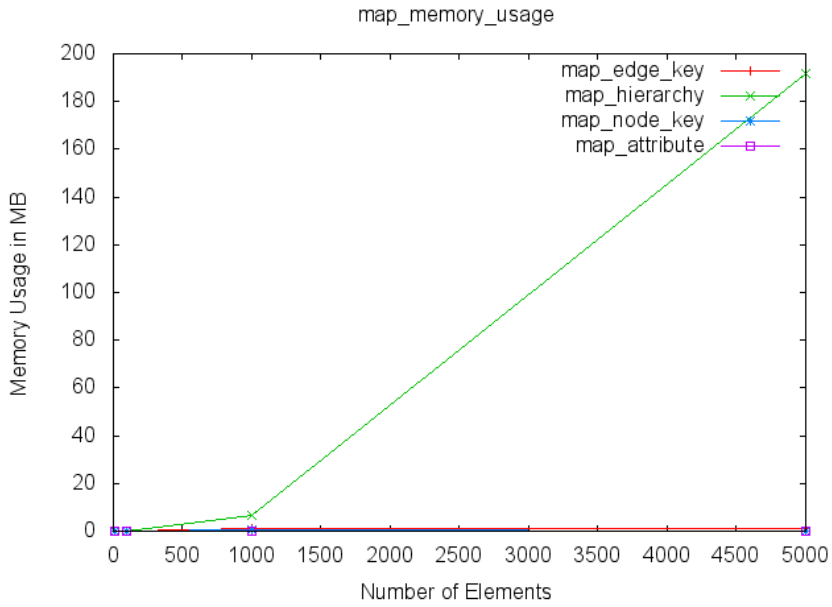


Figure 14: The memory usage of the MappingValues.

memory used grows quite rapidly when a node has lots of attributes. The memory usage of all mappings is visualized in Figure 14. From the other three mappings, the one storing the whole Python dictionary in one attribute is the fastest. It remains to be investigated whether this approach is usable when graph matching and rewriting is considered. The other two mappings are quite close in performance to each other.

4 Mapping ArkM3 to Himesis

In this section, the actual mappings of ArkM3 elements to Himesis structures is presented. Section 4.1 introduces the visual notation which is used in 4.2 to represent the ArkM3-to-Himesis mappings.

4.1 Notation

To represent the ArkM3-to-Himesis mappings, a combination of two visual syntaxes is used: class diagrams to represent the ArkM3 classes and relations and a visual syntax that represents Himesis graphs, which is detailed in Table 1. No graph instances are represented; rather, the diagrams have a meta-relation to Himesis instances. This relation corresponds to the relation between class and object diagrams.

4.2 Mappings

In this section, all ArkM3 packages are are modelled visually in UML class diagrams. The notation introduced in the previous section is then used to represent the mapping of each class to a (set of) Himesis node(s). A few notes:

- A one-to-one association between classes in the UML class diagram is represented either by a UML association with multiplicities 1..1 on both sides, or as an attribute in the source of the association. Both representations have equal semantics.
- In cases where a large inheritance hierarchy exists (for instance, there are a large amount of classes inheriting from the *Operator* class), but where all subclasses have the same instance variables as their superclass, the representation of the subclasses in Himesis is omitted. In such cases, the node to which the subclasses are mapped is equal to the node to which the superclass is mapped, with the exception that the **mm__** attribute of the node correctly references the subclass.
- One-to-many relations in the class diagram are implemented either as SetValues or SequenceValues in ArkM3. Subsequently, it are these SetValues and SequenceValues that are mapped onto Himesis nodes.
- In practice, the mapping of SequenceValue and MappingValue is constructed differently than any of the mappings described in in Section 3.3. It was decided to only have one attribute on each edge, 'role'. This allows for fast lookups of the values of a list, or a dictionary. Then, the node corresponding to the ArkM3 structure contains an attribute which allows for the efficient rebuilding of the ArkM3 structure starting from the Himesis graph. For instance, the node which corresponds to a SequenceValue has an 'index_list' attribute, which is an ordered list of the identifiers of its child elements.
- When constructing the final ArkM3-to-Himesis mapping, both the design choices of Section 3.2 and the performance analysis results of Section 3.3 were taken into account. By design, each ArkM3 element is mapped onto one Himesis node, possibly connected to a number of other Himesis nodes. This results in a one-to-one mapping of ArkM3 elements to Himesis nodes.

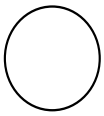
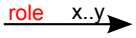
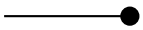
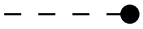
Graphical Notation	Semantics
	Represents a Himesis node, which is always contained in a Himesis graph. A Himesis node can have a number of attributes.
	Represents a Himesis graph, which contains a number of Himesis nodes. A Himesis graph can have a number of attributes.
attr-name: attr-type (comment)	Represents an attribute of a Himesis node or graph. An attribute has a name and a type, as well as an optional comment. The comment clarifies what the attribute is used for. While it is true that each Himesis node contains all attributes of all nodes, the notation used here signifies that an attribute is *meaningful* for that particular nodes. In other words, for all nodes that do not contain this attribute, the value of that attribute in Himesis is undefined.
	Represents an edge between two Himesis nodes. Each Himesis edge has a *role* attribute, which is a string value, representing the role of the target node in the source node. The *role* annotation of an edge represents such a key. The edge is also annotated by a multiplicity, which indicates the minimum and maximum number of nodes that can be connected to the source node with this particular role.
	Represents inheritance between two Himesis nodes or two Himesis graphs. The source inherits the (meaningful) attributes and connected nodes of the target. This is necessary as we are drawing 'meta'-diagrams of graphs.
	Represents the mapping of an ArkM3 structure to a Himesis structure, which is either a graph or a node whose graphical representation is made bold. When an ArkM3 structure is mapped onto a graph, the graph contains at least one root node. That node is represented bold as well.
	When the Himesis representation of an ArkM3 structure is needed, but defined in a different diagram, this notation is used. It relates an ArkM3 structure to its Himesis representation (node or graph), but the actual detailed representation is given elsewhere, using the previous notation.

Table 1: ArkM3-to-Himesis: Mapping Notation

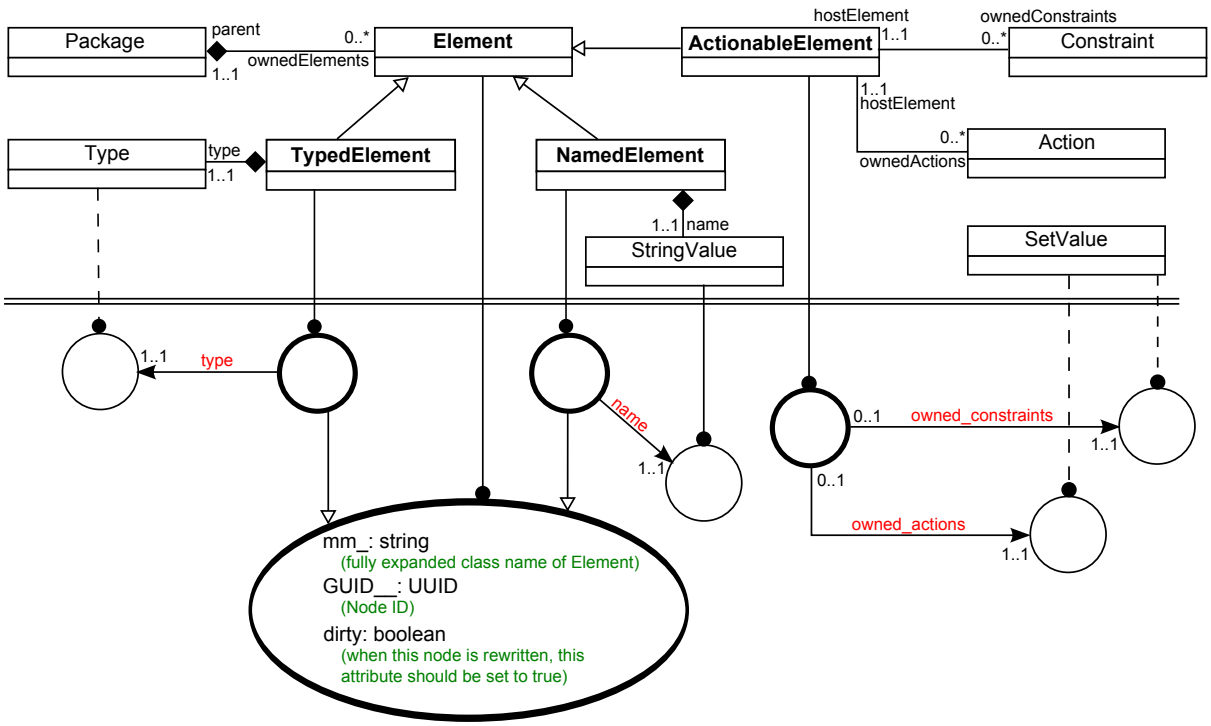


Figure 15: Mapping to Himesis: Element Package

References

[1] X. Dong, "Ark, the metamodelling kernel for domain specific modelling," Master's thesis, McGill University, 2011.

[2] M. Provost, "Himesis: A hierarchical subgraph matching kernel for model driven development.," Master's thesis, McGill University, 2005.

[3] E. Syriani, *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, McGill University, 2011.

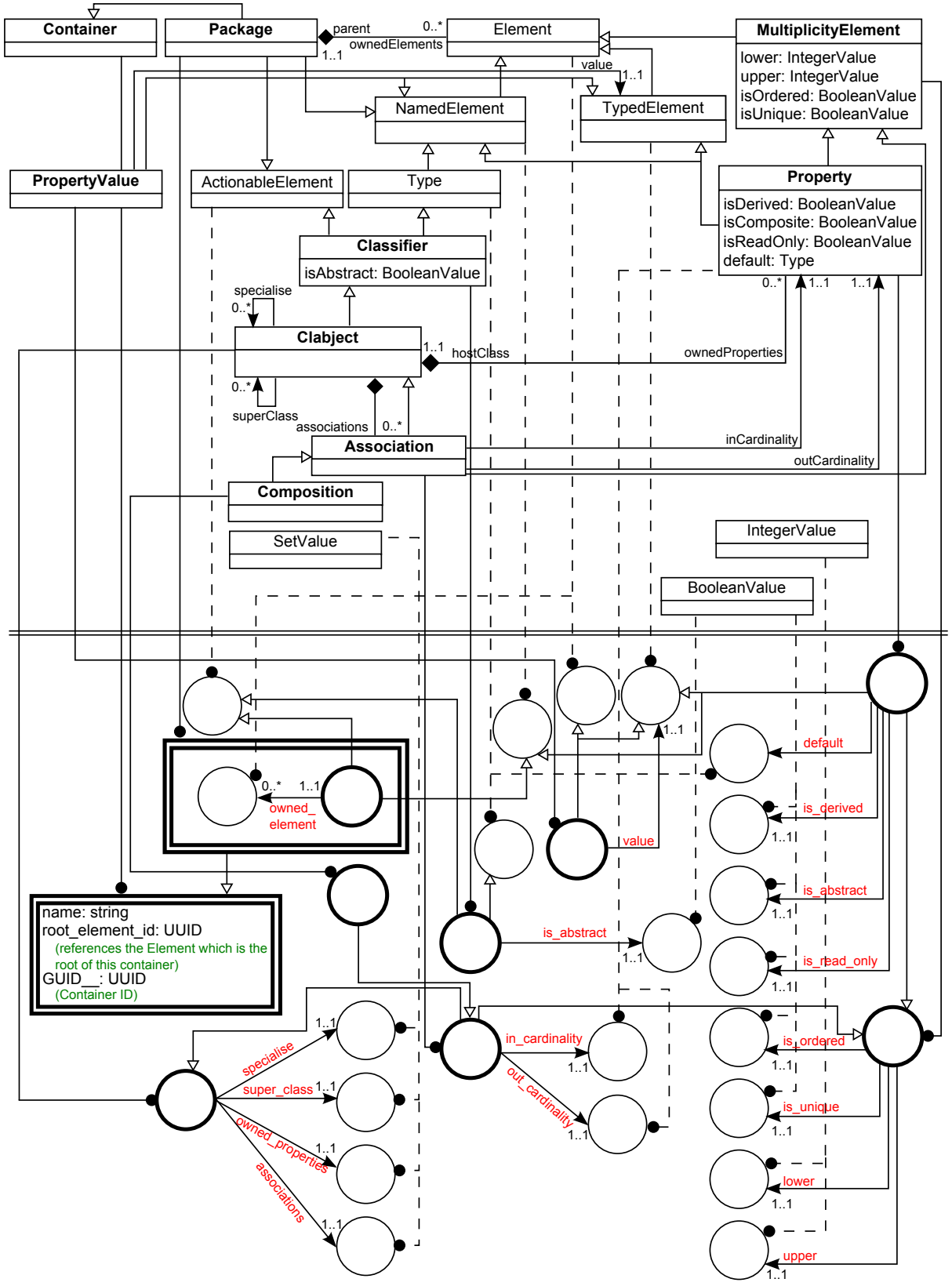


Figure 16: Mapping to Himesis: Object Package

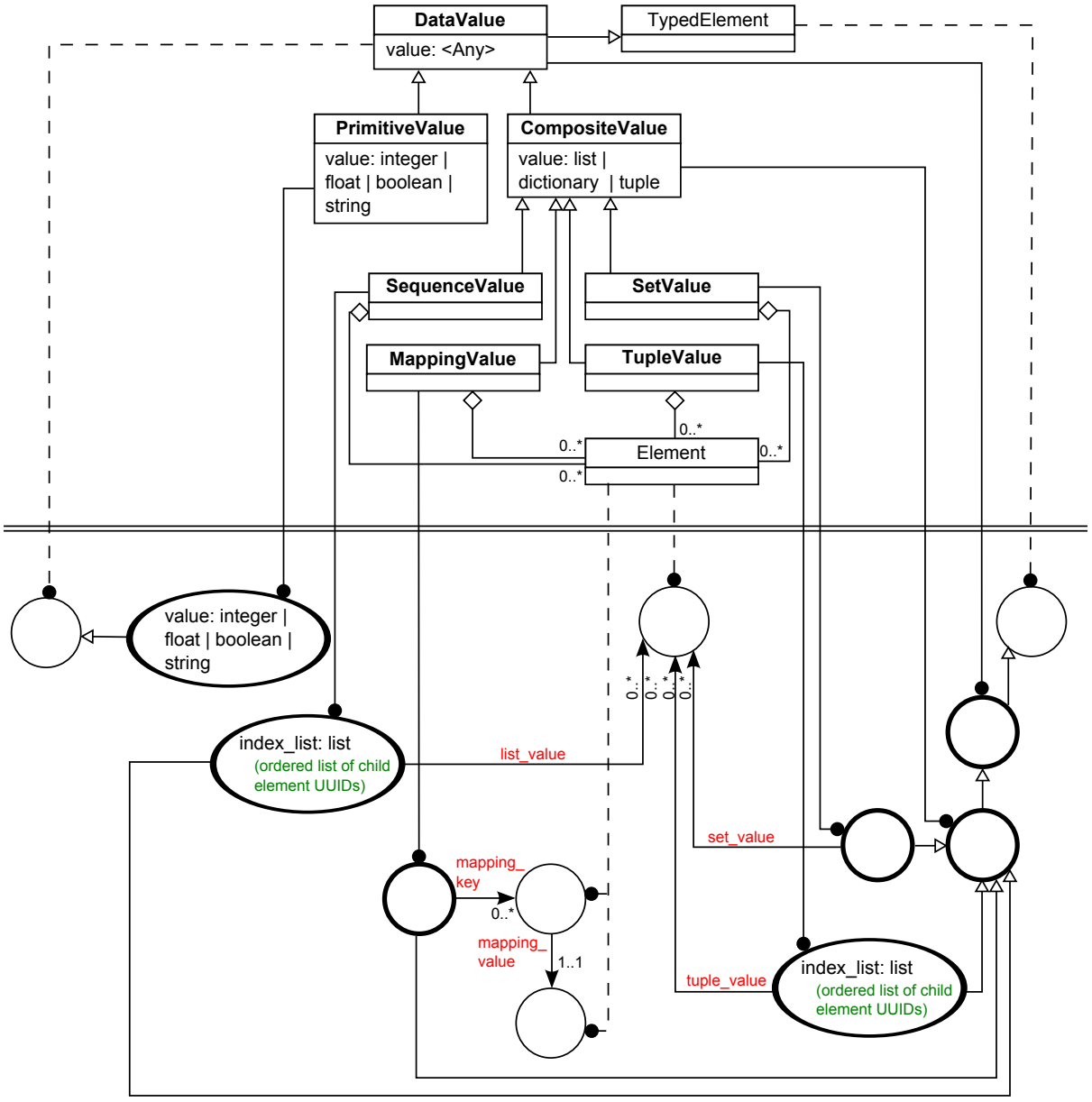


Figure 17: Mapping to Himesis: Data Value Package

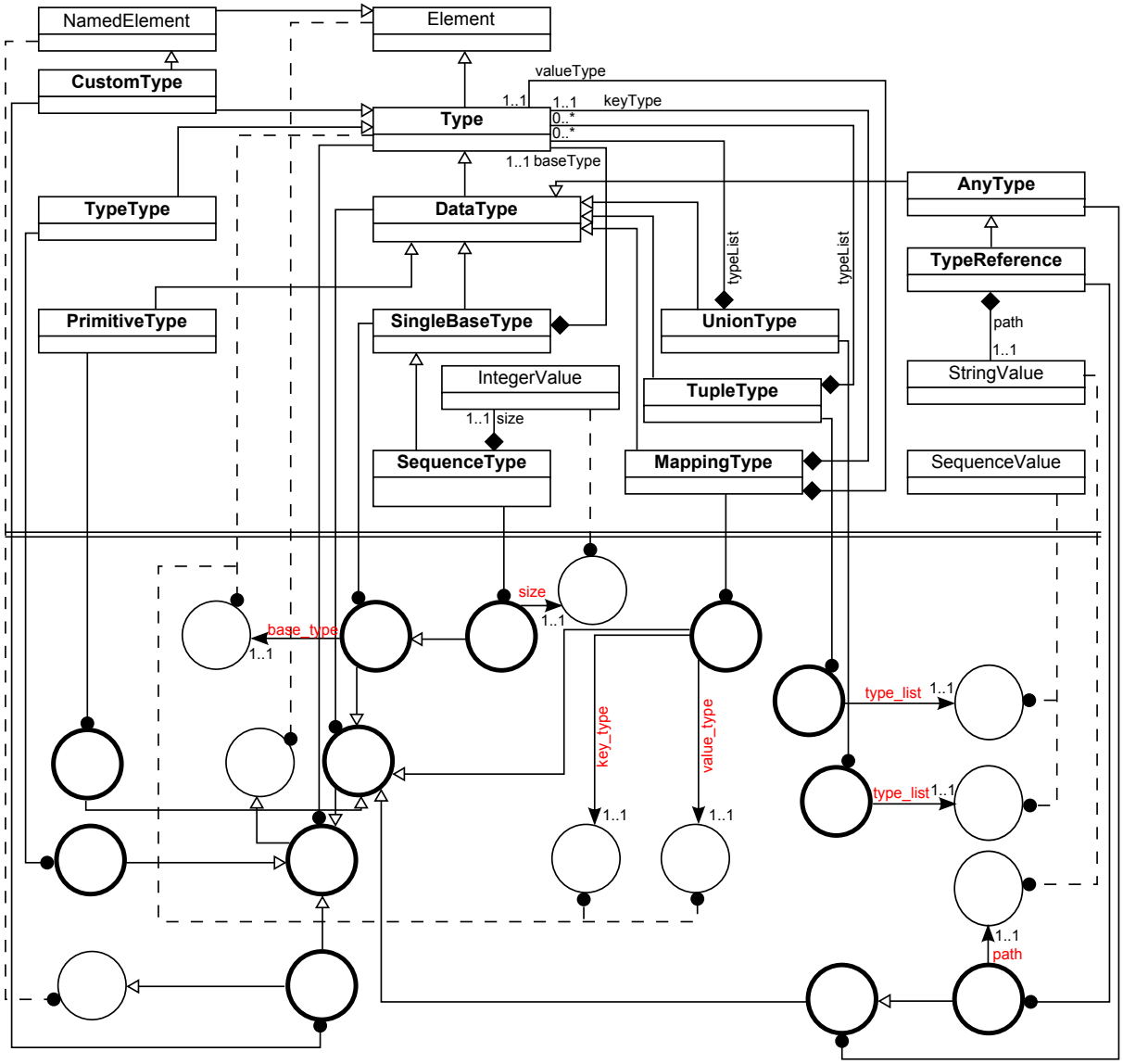


Figure 18: Mapping to Himesis: Data Type Package

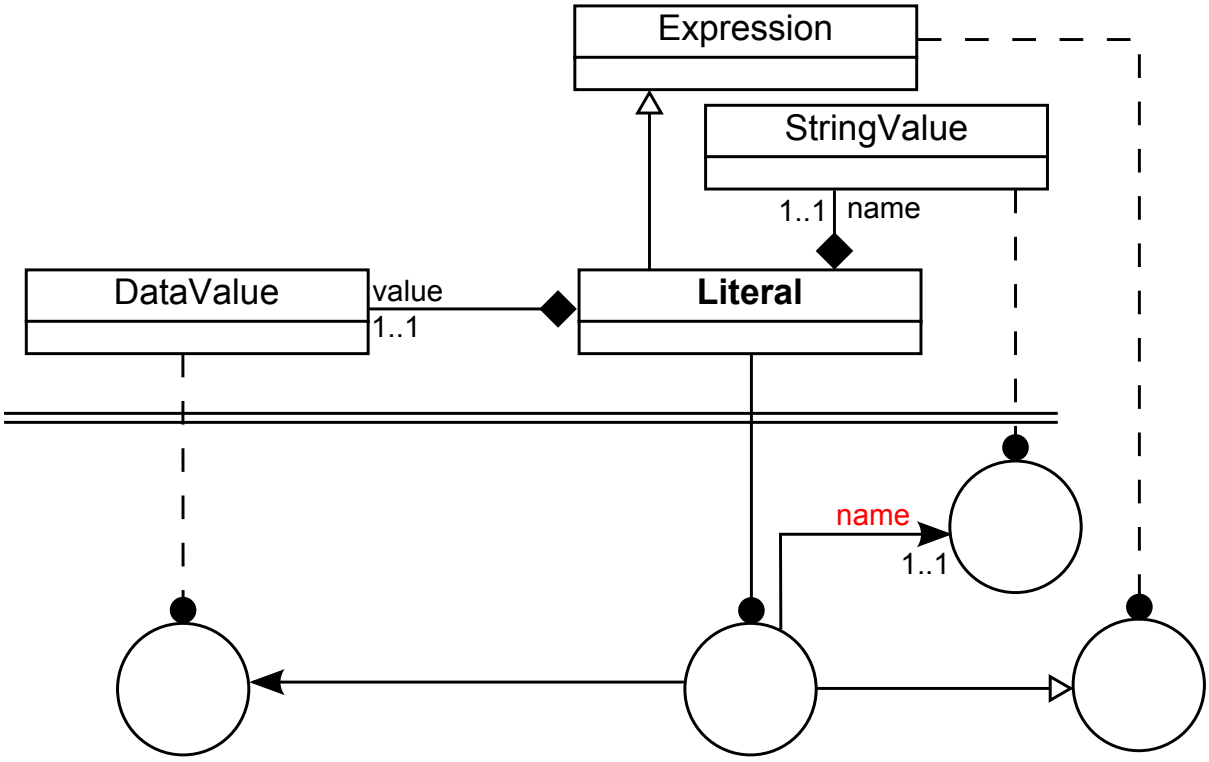


Figure 19: Mapping to Himesis: Literals

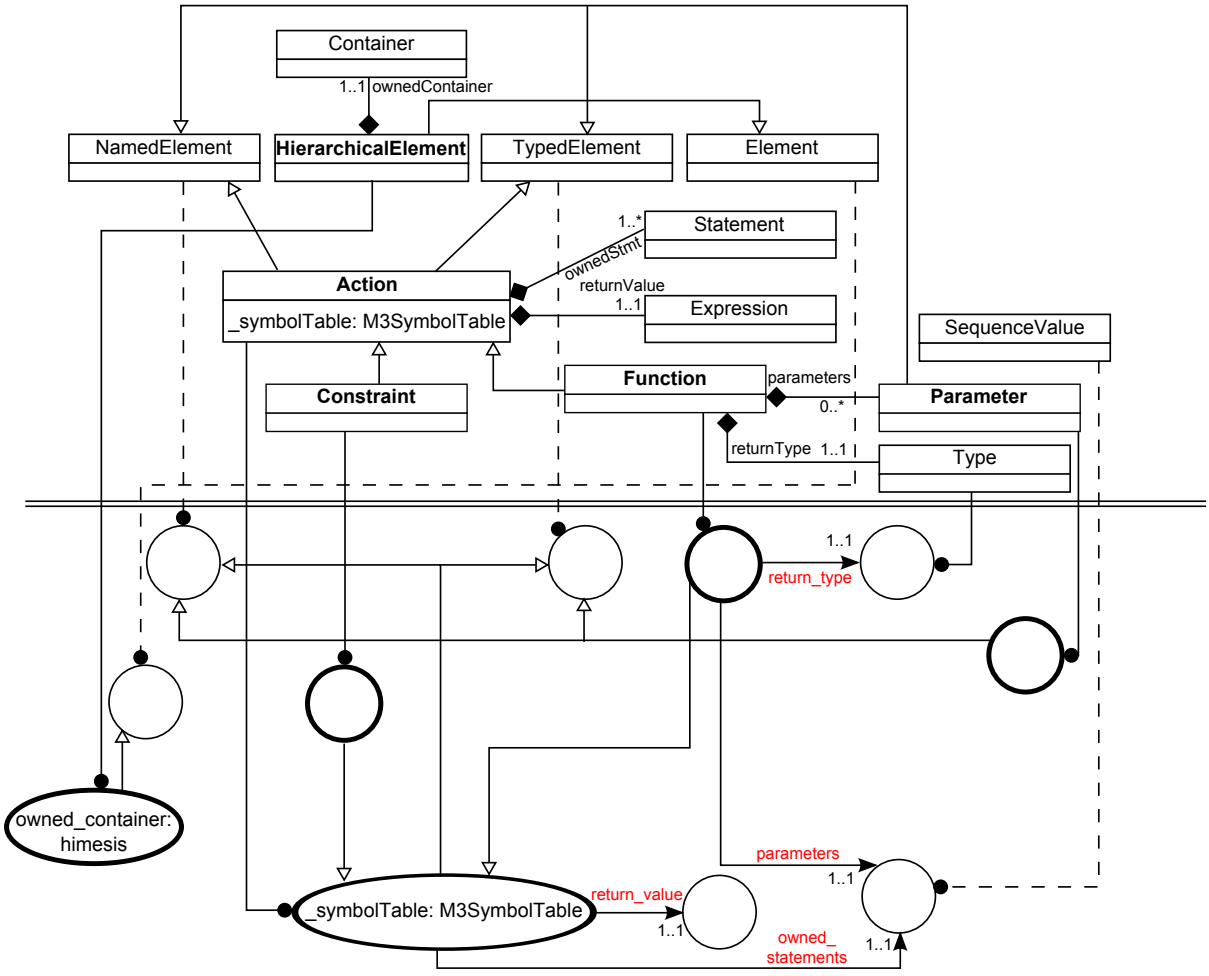


Figure 20: Mapping to Himesis: Action and Constraint

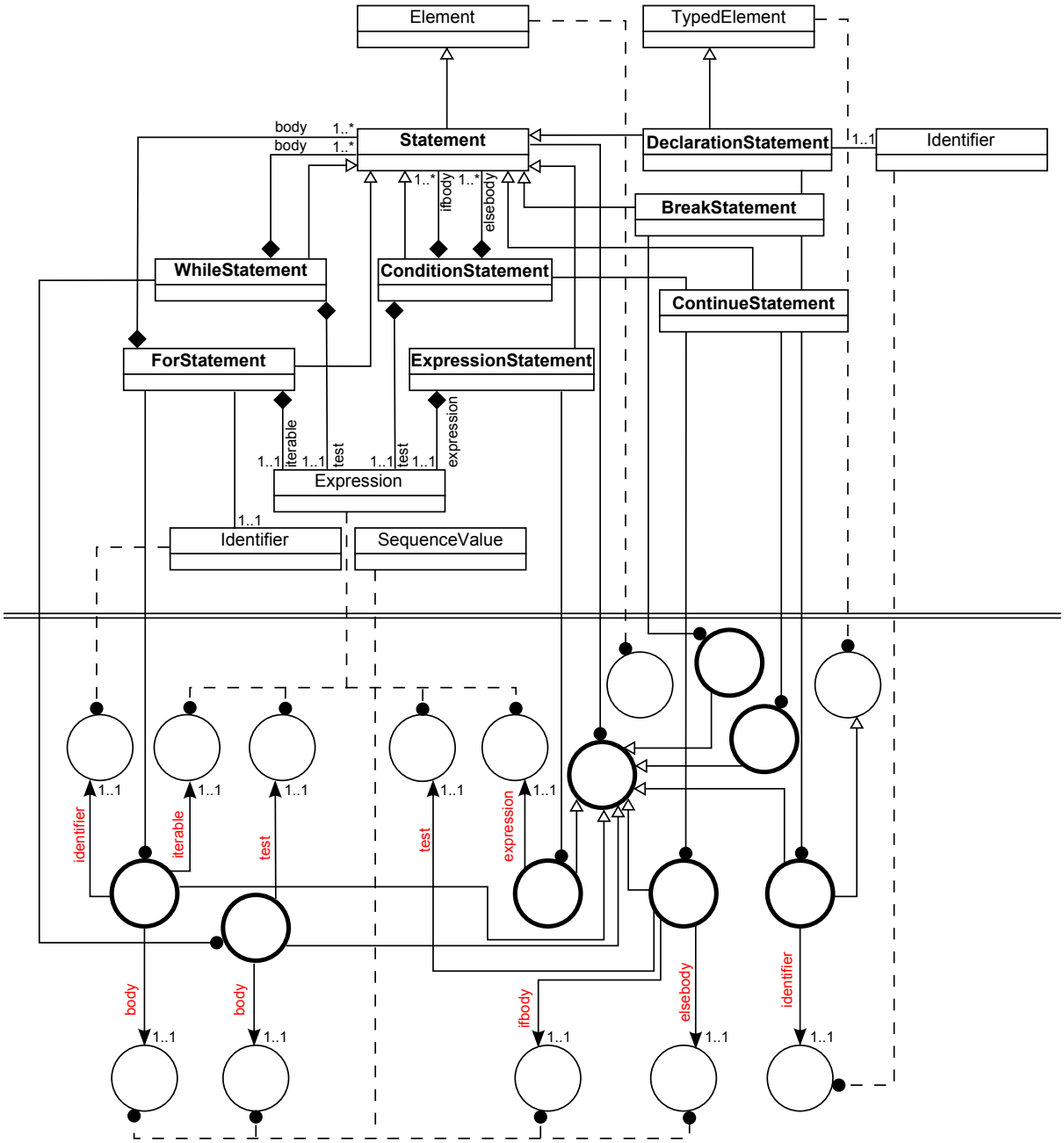


Figure 21: Mapping to Himesis: Statements

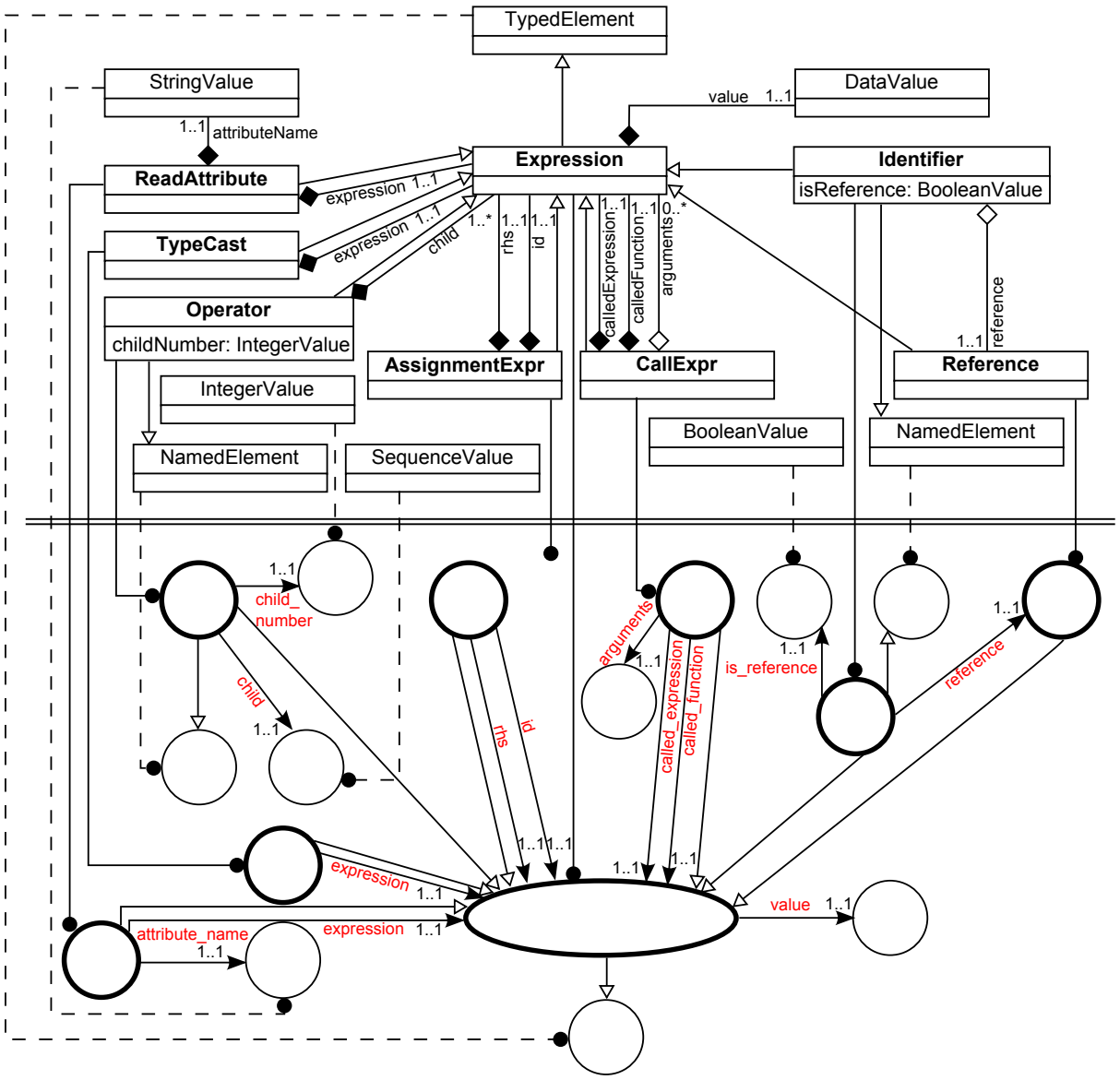


Figure 22: Mapping to Himesis: Expressions