A C++ Implementation of UML Subsets and Unions for MDE

Francesco Bedini, Ralph Maschotta, Alexander Wichmann, and Armin Zimmermann
Software and Systems Engineering Group, Technische Universität Ilmenau, Ilmenau, Germany
{*francesco.bedini, ralph.maschotta, alexander.wichmann, armin.zimmermann*}*@tu-ilmenau.de*

Keywords: Subset, Union, UML, Ecore, C++, Variadic Template

Abstract: This paper shows and discusses the realization of advanced data structures used in the UML specification (namely subsets, unions, and subset-unions) for a C++ execution engine. Those data structures have been realized thanks to the use of variadic templates, which were first introduced in C++11. Thanks to those templates which allow to take as parameters a non-fixed number of elements in an elegant manner, it has been possible to automatically generate from the Ecore and UML ecore models type-safe data structures which avoid elements being duplicated or the generation of additional lists during run-time. A performance analysis is presented to show how our implementation behaves compared to the other possible approaches.

1 INTRODUCTION

To describe a complex structure such as that of the UML metamodel, advanced abstract data structures are required. The Object Modeling Group (OMG) makes a massive use of subsets and unions to describe the inheritance of attributes between classes.

Those annotations allow to easily and quickly define in a model collections of unique elements, thanks to the intrinsic properties of a set. The association end properties hugely simplify the life of a modeler, as he or she does not need to make sure whether inherited elements are already present in a collection more than once, for example when related subsets reference at the end the same union.

On the other hand, the realization is not straightforward and requires a careful analysis or a compromise between wasting memory and speed to comply with the UML definition of subsets and unions.

This paper proposes the realization of such complex data structure by using recently introduced C++ constructs such as variadic templates (Gregor and Järvi, 2007) and tuples.

In this paper, we extend our Execution Engine already described in (Jäger et al., 2016; Bedini et al., 2017) by defining and implementing a set of data structures instead of using collecting operations to create the requested sets on the fly. Our approach minimizes the full duplication of elements in different lists and is entirely type-safe, as each item stored in our collection preserves its type. Thanks to its robust recursive implementation, elements which are inserted in a subset are assured to be inserted in all the corresponding unions and subsets.

As an implementation of the often used UML subsets and unions is, of course, not natively supported by any programming language or library, an implementation is necessary.

The rest of this paper is structured as follows. Section 2 describes the state of the art, Section 3 describes the proposed method, whereas Section 4 describes realization details. In Section 5, a validation example is shown, while Section 6 shows a performance benchmark evaluating the time and memory required for different test cases. Finally, Section 7 summarizes the paper and proposes improvements as future works.

2 STATE OF THE ART

The UML specification makes a significant use of abstract data structures to model class attributes which are derived from related classes. Those relations include associations, inheritances, aggregations, and compositions.

Those dependencies are explicitly indicated on the association ends with properties enclosed in curly brackets such as {subsets <*end*>} or {union}, as it can be seen in Fig. 1. The parameter <*end*> is the name of the referred union which will contain the elements stored in this and other subsets.

The few existing implementations of the UML metamodel, in particular, the one included in the IDE Eclipse, show us different implementation ap-

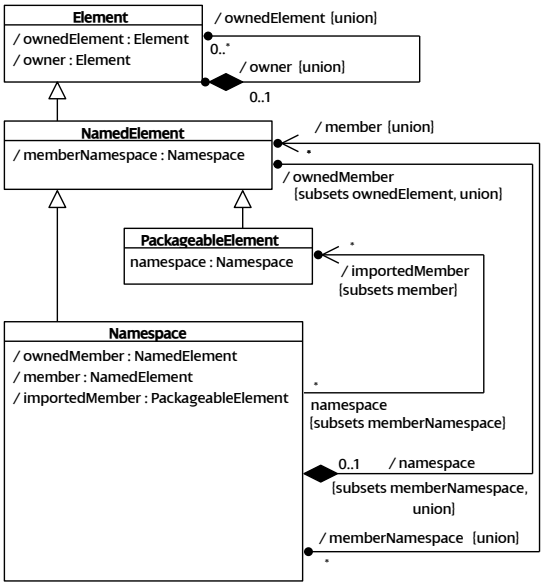

Figure 1: UML class diagram of a portion of the UML metamodel showing the dependencies between four classes as an example.

proaches, which as it will be shown are not efficient nor dynamic, either for their execution time or because of wasted storage due to the duplication of elements or the management of additional caches.

For example, a Java version is already implemented in the Eclipse EMF. Subsets cannot be changed after their creation, and are lazily created and cached once. This implementation limits the tool's capability of supporting real-time changes to the model, without the need of destroying and creating the list every time.

Fig. 1 shows an UML class diagram representing a small extract of the UML metamodel. It shows the relationship between four UML classes. From this diagram, the following *subsets* (S) and *unions* (U) can be inferred:

Element::ownedElement (U) composed by Namespace::ownedMember (S)

Namespace::member (U) composed by Namespace::importedMember (S)

NamedElement::memberNamespace (U) composed by NamedElement::namespace (U) and NamedElement::memberNamespace (S)

Although this example might seem trivial, in reality, a union could be made of plenty of subsets, even from very different levels of the inheritance hierarchy.

As the UML specification action do not strictly formalize what a subset is in the Meta-Object Facility (MOF) infrastructure (Alanen and Porres, 2008), for this paper, we use the following definition:

Subsets: are unordered collections of unique elements, sub-setting one or more unions.

Unions: every element which is referenced by at least one subset, is a union, and it contains all the elements contained by its subsets, at most once.

The possibility to implement those kinds of association properties consists of two possible approaches. The first one consists of storing the attributes once in the base class containing the union and then using methods in the lower elements which are subsets referencing that element to walk through the inheritance tree and collect the corresponding elements in a new container. Possibly this container may be cached for faster subsequent retrievals.

The second one would consist of duplicating the elements or a reference to those items in every class, making the insertion operation more demanding but speeding up the *get* operation as the lists are already built and available at the right place. In both cases, those data structures need to be able of storing elements of different types.

There are different possibilities to store elements of various kinds inside a collection in C++. The most basic and dangerous one would be to store the object pointers in a collection of void pointer types (`void*`). This technique can lead to undefined behaviors as the compiler would not throw any exception when the pointer is statically cast to a wrong type.

A second use would be to use the *"any"* library included in the well known open source portable *Boost* library set (Dawes et al., 1998). Using this library, it would be possible to define a collection of type `boost::any`, which is a proper value type. To use the value or pointer stored in it, it is necessary to retrieve it using the method `boost::any_cast<Type>(element)` : `Type` (Karlsson, 2005). The advantage of using *Boost* rather than void pointers lays in the fact that the cast would throw an exception if the types are not consistent, preventing the standard C++ undefined behavior.

Another possibility, which is the one which has been chosen for our realization, is introduced in the following section.

3 METHOD

The evolution of C++ allows a more elegant and type-safe way of realizing this kind of components, which is by using *variadic templates* and the concept of *tuples*, both introduced in the C++11 specification (ISO/IEC., 2011). Variadic templates are function templates which contain at least a *parameter*

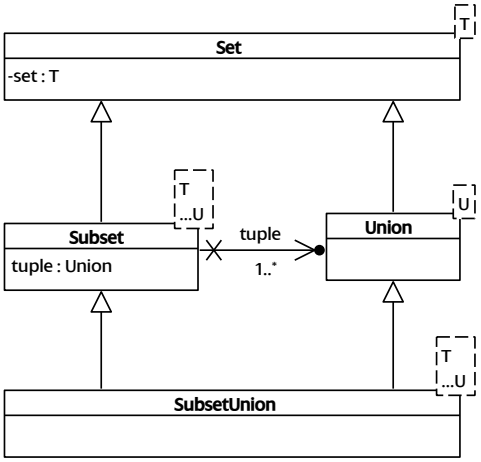

Figure 2: UML class diagram of the proposed containers.

pack, which is a template parameter that accepts zero or more template arguments (Gregor, 2006). An ellipsis following a parameter name defines it as a parameter pack. Tuples are data structures which allow to collect items of different type and access them based on their defined order.

The order of the elements inside the template consists of taking the current collection type as the first parameter, and then the related unions' types. The order is assured to be consistent in the whole generated model as the Acceleo code generator takes care of defining it in a query, which always returns the same output given the same input parameters.

Fig. 2 shows the structure of our realization of the containers proposed in this paper. Our base data type is a `Set`, containing an attribute `set` (here meant as a C++ set) of type T, which is the template type. We make use of the C++ templates, which are shown in angle brackets and separated by commas.

Each `Subset` then has its type and its set of elements, of type T, plus one or more unions, of type U. Each `Union` contains, in turn, a set, of type U.

There are associations which are at the same time both a subset and a union. For those cases, the class `SubsetUnion` should be used. It takes the same template parameters as a `Subset` but moreover can be referenced by other subsets, too.

To define the data structures in the metamodel, the following changes have been performed. The `EReference` elements contained in the UML Ecore metamodel need to be annotated with a defined annotation *"union"* or *"subset"* to be interpreted as such by our C++ generator. The subset annotation must, of course, include the referenced union, as the UML specification does with the *<end>* attribute inside the association ends.

With those annotations, the code generator identifies the unions and subsets. Their types are then derived from the model, and for each of them an attribute of type *Union*, *Subset* or *SubsetUnion* is created in the relevant classes. Moreover, getter methods are generated for settable references. Those methods, simply return a shared pointer to the class' reference which consists of a ready-to-use list.

4 REALIZATION

All our data structures allow adding, removing and finding elements contained in them. We use shared pointers to simplify the memory management.

The Listing 1 shows how the insertion of one element in a subset works. As we are using the C++ Sets as data structures, it is not needed to check that the inserted elements are unique. In case the element already exists in the collection, the element will not be added again (Musser et al., 2009).

```
virtual void add(shared_ptr<T> el)
{
 Set<T>::add(el);
 call_addEl_with_tuple(el, _tuple,
  index_sequence_for
  <shared_ptr<Union<U>>...>());
}

void call_addEl_with_tuple(
 shared_ptr<T> el,
 const tuple<shared_ptr
  <Union<U>>...>&tuple,
 index_sequence<Is ...>) {
  addElRecursive(el,
   get<Is>(tuple) ...);
}

template<class FirstU, class ... RestU>
void addElRecursive(shared_ptr<T> el,
 shared_ptr<Union<FirstU> > obj,
 shared_ptr<Union<RestU> > ... rest) {
  addElRecursive(el, rest ...);
  obj->add(el);
}

template<class FirstU, class ... RestU>
void addElRecursive(shared_ptr<T> el,
 shared_ptr<Union<FirstU> > obj) {
  obj->add(el);
}
```

Listing 1: *add* element method of a subset. All operations are private, except *add*, which is the public interface.

For brevity, namespaces in listings are omitted throughout this paper. The insertion of an element in a subset works as follows: the element is first inserted into the subset's own set, and then is recursively in-

serted in all the unions with the help of the auxiliary methods *call_addEl_with_tuple* which calls, in turn, the *addElRecursive* methods. There are two *addElRecursive* methods, as one is called with a variadic template that is at every iteration shrunk of one element, and one is called for a single item, namely the last union.

Our implementation moreover supports the insertion of multiple elements at once through an iterator or the inclusion of an entire Subset. The deletion of an item is realized in the same recursive manner.

5 EXAMPLE

Fig. 3 shows the C++ realization of a connection between the example UML metamodel shown in Fig. 1 and the defined Subsets and Unions of Fig. 2. The resulting created objects and connections are shown in the object diagram in Fig. 4. This corresponds to the C++ instructions shown in listing 2.

```
shared_ptr<Constraint> pe =
 umlFactory->
  createConstraint_in_Context(package);

shared_ptr<Class> n1 =
 umlFactory->
  createClass_in_Package(package);

shared_ptr<Class> n2 =
 umlFactory->
  createClass_in_Namespace(n1);

n1->getImportedMember()->add(pe);
```

Listing 2: Initialization code for the exposed example.

As `PackageableElement` is an abstract class, the subclass `Constraint` has been used instead.

The *umlFactory* allows to easily create elements and assign their container and set the back reference on their container consistently. First we create two Namespaces *n1*, *n2*, and the constraint *pe*, and then we import *pe* into *n1*.

The constraint *pe* has been created on purpose inside the package instead of inside the namespace to show that it will be included when the *Members* and *ImportedMembers* get retrieved, but not when the *OwnedElements* are requested.

We run this code and print out the elements retrieved by different operations executed on the instance *n1*:

```
n1->getOwnedElement (1 element)
-n2
n1->getOwnedMember (1 element)
-n2
n1->getMember (2 elements)
-n2
-pe
n1->getImportedMember (1 element)
-pe
```

which is coherent with what was defined in Fig. 1.

5.1 Subsets and Unions applied to the Ecore Metamodel

Although subsets and unions are defined and used in the UML specification, we have additionally applied them to the Ecore metamodel to generate the getter and setter operation automatically from the model, which allowed us to remove the corresponding operation definition from the Ecore model. This change reduces the size of the Ecore metamodel and simplifies its maintainability in the future (Kleppe et al., 2003).

Once all the subsets and union have been denoted as such in the metamodel, our generator has been able to automatically produce compilable code correctly representing all the containment relationship between all elements.

Fig. 5 shows an extract of the Ecore metamodel. `EReferences` and `EAttributes` are both `EStructuralFeatures`, which can be retrieved from a `EClass` by calling the method `getEStructuralFeatures`. This method must return all the `EAttributes` and all the `EReferences` which are owned by the `EClass`.

Our code generator generates the getter operations shown in Listing 3, which simply returns the associated attribute of the same type.

```
shared_ptr<Union<EStructuralFeature>>
 getEStructuralFeatures();

shared_ptr<Subset<EAttribute,
 EStructuralFeature>> getEAttributes();

shared_ptr<Subset<EReference,
 EStructuralFeature>> getEReferences();
```

Listing 3: Automatically generated Subsets and Unions getter source code.

It is important to notice that each getter method automatically returns the correct type, without any additional cast needed to use the retrieved elements.

6 PERFORMANCE EVALUATION

This section shows the results of the measurements performed on 3 different implementations, two of which in C++ and one in Java. The first one is depicted as “SubsetUnion” in all following plots, and it is the one described throughout this paper. The second and

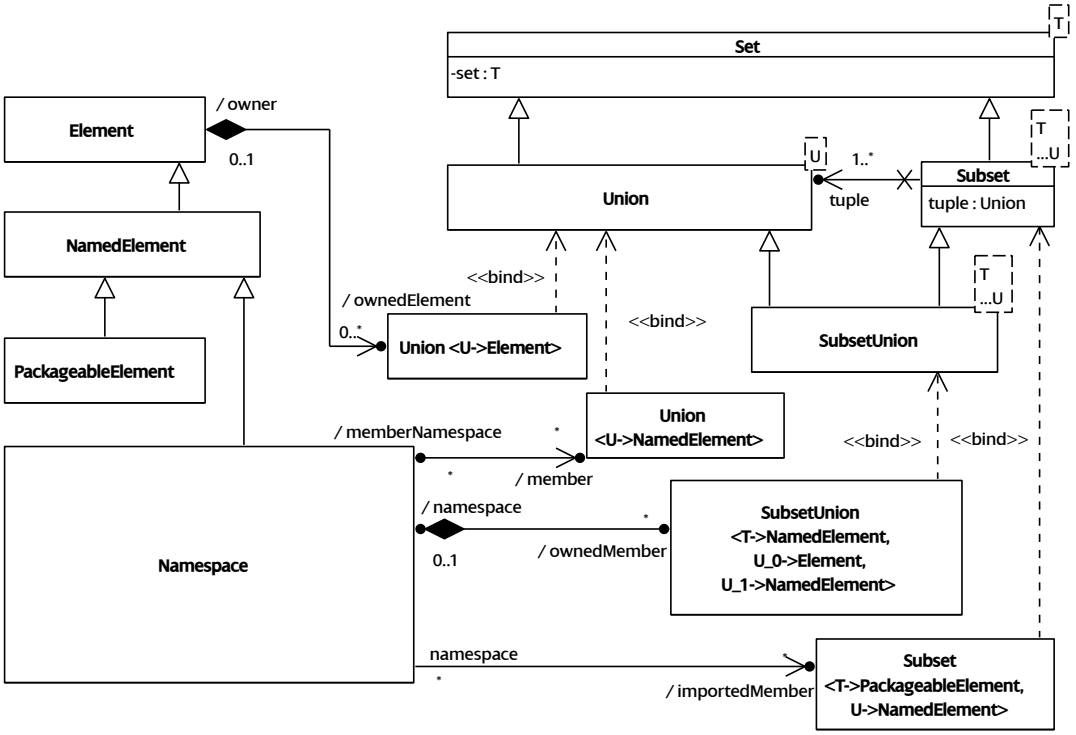

Figure 3: UML class diagram of the C++ realization of the data structures, using the template T and the variadic templates U.

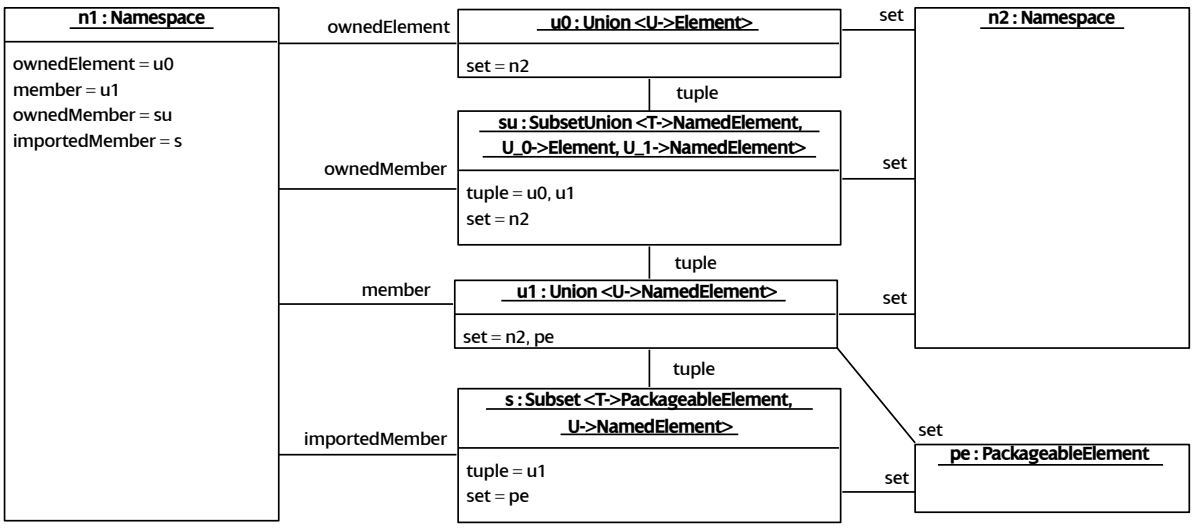

Figure 4: UML object diagram representing the objects created in the example and their relationship.

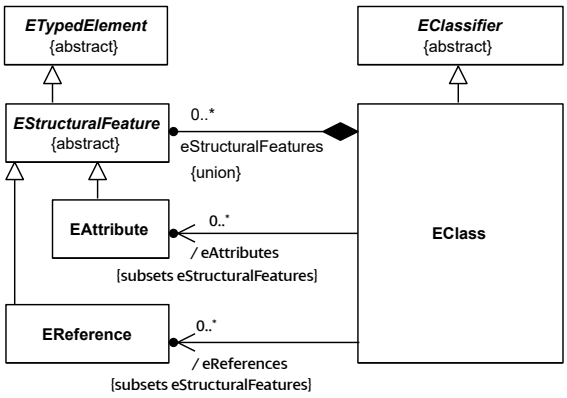

Figure 5: UML class diagram of a small portion of the Ecore metamodel, showing a union and two subsets.

third implementation have the same behavior, but they are realized in C++ (called "Manual") and in "Java".

Those two versions store their elements in the subsets only, and when a Union is requested, a list gets generated by going through the entire class hierarchy. This realization mimics the implementation that can be found in the Ecore realization in Eclipse.

For all the computations, a 10-level subset hierarchy has been used, as depicted in Fig. 6.

As the C++ version is currently not realized in a parallel manner, for keeping the comparison fair all tests have been executed on a 3,20GHz single-core 64bit machine with 16GB of RAM, as otherwise, Java would take automatically advantages of multi-core parallel execution.

6.1 Memory Occupation

Table 1 and the corresponding plot in Fig. 7 show the memory occupied during the creation of the ten levels of subsets and the fetching of all those levels in the three different implementations.

It can be seen that the amount of memory occupied is always linear to the number of elements and that the Java implementation is the one requiring the

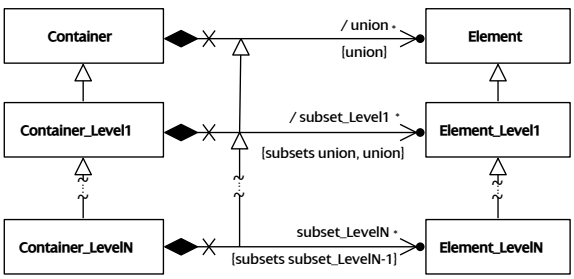

Figure 6: UML Class diagram showing the structure used for the benchmarks.

Table 1: Comparison between the occupied memory at the end of the benchmarks executions. The numbers on the first row show how many elements per layer are present.

Elements	10^3	10^4	10^5	10^6
SubsetUnion	2.1	13.5	127.1	1263.6
Manual	2.0	10.1	86.6	845.0
Java	13.5	61.9	211.1	1929.7

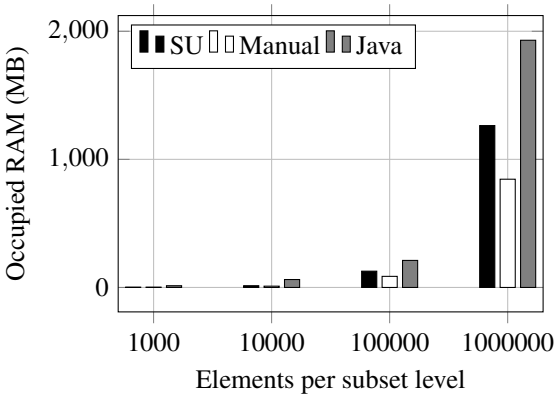

Figure 7: Comparison between the occupied memory at the end of the benchmarks executions.

highest amount of memory.

The difference between the SubsetUnion and Manual implementation shows how much memory is used by inserting a pointer to an element in all the unions pointed by all the subsets.

6.2 Execution Time

The plot shown in Fig. 8 shows the time needed to create 10^5 elements (solid lines) and the time required to retrieve the unions from each subset level (dashed lines) starting from 10 and proceeding up to level 1. At the end of the creation process, the union will contain 10^6 elements, as there are ten layers.

All the data shown in this section has been obtained by averaging the execution time of 11 runs, after taking out the longest and shortest execution time. The standard deviations are shown in the form of error bars, too.

As expected, inserting an element in the lower levels of the hierarchy requires more time for the SubsetUnion implementation, as they have to be inserted into the referenced unions too.

Although the benchmarks have been executed on a single core machine, the results obtained from the Java test do not seem to show any correlation between the subset level and the required time. Those regular peaks seem to suggest that the virtual machine might be occupied performing other operations, like the garbage collection checking the status of the allo-

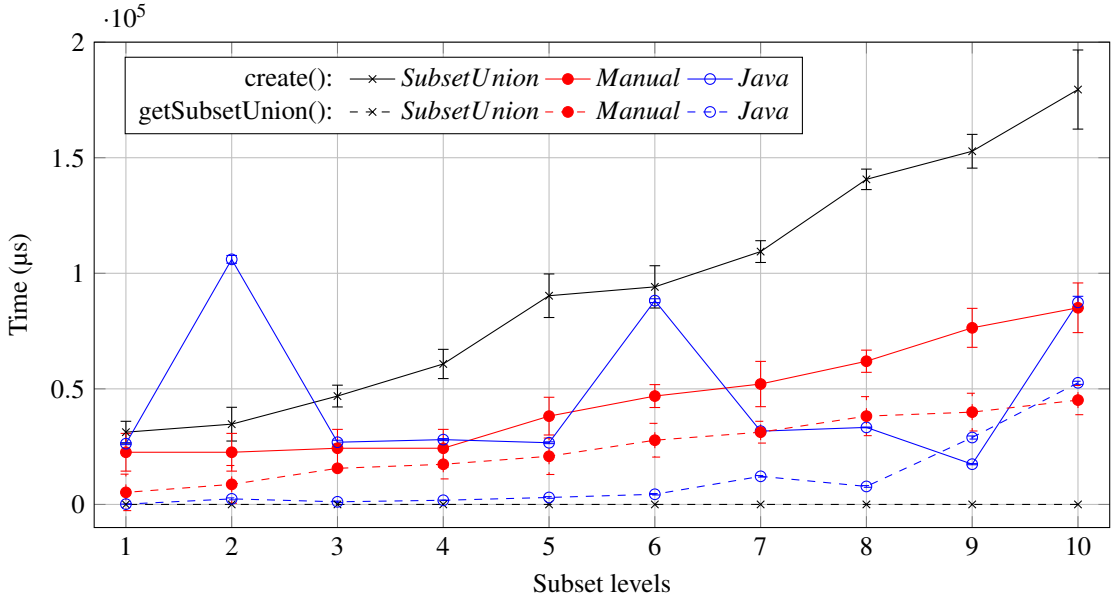

Figure 8: Execution time for the creation (solid lines) and the retrieval (dashed lines) of 10^5 elements on each subset level.

cated memory.

The results show that the construction of the union upon the creation time is more efficient when the retrieval operation of the union at the different level happens more than once.

The UML implementation contained in Eclipse tackles this problem by using a cache, which prevents the creation of the lists as long as the cache remains valid. This approach limits the ability to make changes to the model during run-time.

As the creation of the new lists would be executed at run-time and are demanding as they possibly require extra memory allocation and correctness checks, our type-safe implementation is a promising performance improvement, in particular in the cases where the *get* operations are frequent, as they have, in our case, a minimal complexity, consisting in just returning a pointer to the requested collection.

7 CONCLUSION

Thanks to the implementation of the Subset, Union and SubsetUnion data structures in C++ with mean of variadic templates, it has been possible to obtain an efficient, clear, and concise way to define them. This implementation technique avoids the on the fly construction of other lists and makes dynamic casting to the correct element type unnecessary.

From some preliminary tests, it could be seen that adding elements to the C++ standard library's sets is much slower than vectors when the number of items increases, as the uniqueness check has to be carried out upon every insertion.

As the generator has an overall view of the complete model or metamodel that is going to be generated, it is in the condition of assuring that elements which belong to classes with multiple inheritances get inserted only once.

In this way, some additional run-time checks can be saved, and the more efficient vectors can be used instead of sets without breaking any uniqueness constraint. Then two add methods can be offered, one which checks the uniqueness constraint, that should be used for run-time changes, and one that does not proofs uniqueness that can be used, for example, during the model instantiation.

ACKNOWLEDGEMENTS

This work has been supported by the *Federal Ministry of Economic Affairs and Energy* of Germany under grant FKZ:20K1306D.

The source code for the C++ code generator, the execution engine, the data structures introduced in this paper and the ecore models can be found at the MDE4CPP project page (Systems and Software Engineering Group, 2016) under the MIT license.

REFERENCES

Alanen, M. and Porres, I. (2008). A metamodeling language supporting subset and union properties. *Software & Systems Modeling*, 7(1):103–124.

Bedini, F., Maschotta, R., Wichmann, A., Jäger, S., and Zimmermann, A. (2017). A model-driven C++-fUML execution engine. In *5th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2017)*.

Dawes, B., Abrahams, D., and Rivera, R. (1998). Boost.org. online. Retrieved from http://www.boost.org/.

Gregor, D. (2006). A brief introduction to variadic templates.

Gregor, D. and Järvi, J. (2007). Variadic templates for C++. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 1101–1108, New York, NY, USA. ACM.

ISO/IEC. (2011). ISO international standard ISO/IEC 14882:2011(e) programming language C++. Retrieved from https://isocpp.org/std/the-standard.

Jäger, S., Maschotta, R., Jungebloud, T., Wichmann, A., and Zimmermann, A. (2016). An EMF-like UML Generator for C++. *4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2016)*.

Karlsson, B. (2005). *Beyond the C++ standard library: an introduction to boost*. Pearson Education.

Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.

Musser, D. R., Derge, G. J., and Saini, A. (2009). *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional.

Systems and Software Engineering Group (2016). Model Driven Engineering for C++ (MDE4CPP), see http://sse.tu-ilmenau.de/mde4cpp.