A Model-driven Implementation of PSCS Specification for C++

Maximilian Hammer, Ralph Maschotta, Alexander Wichmann, Tino Jungebloud, Francesco Bedini and Armin Zimmermann

Systems and Software Engineering Group, Computer Science and Automation Department, Technische Universität Ilmenau, Ilmenau, Germany
https://www.tu-ilmenau.de/sse/

Keywords: Model Driven Software Development, Composite Structures, Executable UML, Code Generation, PSCS, fUML, UML, C++.

Abstract: OMG's PSCS specification extends the execution model of fUML by precise runtime semantics for UML composite structures. With composite structures being a concept for describing structural properties of a model, the majority of execution semantics specified by PSCS concern analysis and processing of static information about the model's fine-grained structure at runtime. Using Model-To-Text-Transformation to generate source code, which serves as an input for PSCS's actual execution environment, the runtime level of model execution can be relieved by outsourcing analysis and processing of static information to the level of code generation. By inserting this step of preprocessing, the performance of the actual model execution at runtime can be improved. This paper introduces an implementation of the PSCS specification for C++ based on code generation using Model-to-Text-Transformation. Moreover, it presents a set of test models validating the correct functionality of the implementation as well as a performance benchmark. The PSCS implementation presented by this paper was developed as a part of the *MDE4CPP*[*] project.

1 INTRODUCTION

With Model-driven Architecture (MDA), the Object Management Group (OMG) provides a standardized approach to Model-driven engineering (MDE) techniques. MDE, and respectively Model-driven Software Development (MSDS) focusing specifically on software development processes, suggest using modeling languages to describe, represent and analyze complex (software-)systems on higher levels of abstraction and use them to generate artifacts (e.g., source code) automatically throughout the whole development process. By that, model-driven approaches aim to increase flexibility, reusability as well as efficiency of development processes (Stahl and Völter, 2005).

In the domain of systems and software engineering, OMG's Unified Modeling Language (UML) has established as a de facto standard for modeling languages (Hutchinson et al., 2011). One of the main objectives of MDA is the ability to execute UML models. That means being able to create executable applications directly from conceptual models, either by complete transformation or by simulating the models using an execution environment (OMG, 2014). With UML, however, mainly being designed to be widely applicable for all sorts of systems and software, and aiming to be a tool for describing conceptual models rather than being some sort of compilable programming language, UML lacks precise semantical specification of its modeling concepts and metamodel elements (Bedini et al., 2017). This circumstance intuitively contradicts the idea of being able to execute UML models like compilable source code. In 2011 OMG released the initial version of the fUML specification (Semantics of a Foundational Subset for Executable UML Models), which specifies precise execution semantics for a minimal subset of UML activities and classes (OMG, 2011). fUML makes it possible to realize executable activity diagrams being one of the most widely used UML concepts for modeling behavior in the industry (Hutchinson et al., 2011). In 2015 the OMG extended fUML by PSCS (Precise Semantics of UML Composite Structures), which specifies runtime semantics for UML composite structures (OMG, 2015).

Composite structures are used to model the structure of systems as complex part-whole relationships.

[*]See https://www.tu-ilmenau.de/sse/forschung/projekte/mde4cpp/

Hammer, M., Maschotta, R., Wichmann, A., Jungebloud, T., Bedini, F. and Zimmermann, A.
A Model-driven Implementation of PSCS Specification for C++.
DOI: 10.5220/0010267801000109
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 100-109
ISBN: 978-989-758-487-9

Such composite structures (metaclass *StructuredClassifier*) can be described as a topology of parts (metaclass *Property* whose *AggregationKind* is composite) that are linked through connectors (metaclass *Connector*) to form the internals of their owning element (e.g., a *Class* or a *Component*) in terms of a network. A classifier may also define interaction points (metaclass *Port*) to model communication interfaces between itself (or its internal parts) and its environment. Ports are dedicated parts that encapsulate the behavior of their owning classifier (metaclass *EncapsulatedClassifier*) and specify its provided and required interfaces (OMG, 2017).

The PSCS specification defines runtime semantics for executable UML models that include *StructuredClassifiers*, as well as *EncapsulatedClassifiers*. Like fUML, PSCS provides a metamodel that describes an execution environment to realize those runtime semantics defined in the specification. In terms of structural semantics, PSCS specifies the lifecycle management of composite structures at runtime (i.e., creation and destruction of instances of composite structures). In terms of behavioral semantics, PSCS defines how instances of composite structures communicate with each other. More precisely, the behavioral semantics of PSCS specify how communication, either synchronously using operation calls or asynchronously using signals is forwarded through a network of interconnected runtime objects (OMG, 2019). While the latter concerns dynamic aspects of a model which have to be evaluated at runtime, the structural semantics of PSCS mainly focus on the analysis and evaluation of static information of a model. That is, for example, evaluating how composite structures and their internal topologies have to be instantiated, depending on the definition of their structural properties in the model (e.g., their parts and ports, their connectors, involved multiplicities, etc.). The fact that such information is static for a model and does not change during execution gives us the ability to outsource its analysis and evaluation to a step of preprocessing before the actual execution happens and thus decrease the amount of data processing required at runtime.

Of course, since PSCS (as well as fUML) is designed-platform independent and makes no assumptions about the environment it is implemented in, all of PSCS's functionality is encapsulated in its metamodel classes which form a pure virtual machine for executing models (OMG, 2019). The aim of the PSCS implementation presented by this paper is to make use of auto-generated, model-specific source code to form the basis of the model execution as an input for PSCS's actual execution environment. By that, the required functionality for executing PSCS models concerning analysis and processing of static, structural information about the model (which is known at generation time and does not change as long as the underlying model does not change) can be outsourced from the execution environment itself to the process of code generation. The described approach should reduce computation overhead during execution because the evaluation of structural model information (which is a significant part of PSCS's runtime functionality) is done only once during generation instead of multiple times at runtime during the actual model execution.

Section 2 shows the realization workflow of the PSCS implementation that is introduced by this paper and selected design challenges and how they were solved. Section 3 presents how the implementation's conformance to the original specification and its correct functionality were validated as well as a performance evaluation. The results of the validation process and possible future work are ultimately discussed in section 4.

2 METHODOLOGY

This section describes the design workflow and the components that were implemented to realize a model-driven implementation of the PSCS specification. One of the major challenges was the porting of PSCS's concept of links, that connect runtime objects to form a topology, to the level of source code generation as well as the resulting requirements for memory management. Details on how those challenges were solved will also be explained in this section.

2.1 Workflow

The realization of the PSCS specification presented by this paper is based on the *MDE4CPP* project (Systems and Software Engineering Group, 2016), which develops an open-source framework for MDSD using C++. Being one of the most important third party components of the MDE4CPP framework, the *Eclipse Modeling Framework*[1] (EMF) is used as the foundational toolset for creating and analyzing models as well as developing the source code generation facilities required for the implementation of PSCS presented by this paper.

MDE4CPP provides model-driven implementations for the meta models of UML as well as fUML (which are required for implementing PSCS). Eclipse's Ecore model is used as the meta meta

[1] see https://www.eclipse.org/modeling/emf/

model, because it is well integrated into EMF. Moreover, MDE4CPP provides generators for Ecore models as well as UML/fUML models (Jäger et al., 2016), which are implemented using the open-source code generator tool Acceleo (Eclipse Foundation, 2018).

As described before, not all functionalities of the PSCS meta model can be substituted by static code generation. In the presented implementation, functionalities that concern dynamic aspects of a model at runtime shall remain in the meta model. For realizing the PSCS meta model as the first step, a machine-readable representation in the form of an XMI model provided by OMG was used, which was manually converted into an Ecore model. Because Ecore's structure is realized mostly equivalent to EMOF[2] (Steinberg et al., 2009), it only provides capabilities to describe a models structure but not its behavior. The model's behavior was implemented using annotations that carry the functionality of the model classes in the form of C++ source code. Based on the implemented PSCS Ecore model, a C++ library of the model is generated using MDE4CPP's Ecore4CPP generator (Systems and Software Engineering Group, 2016).

In order to outsource parts of PSCS's functionality to code generation, those semantical aspects that are suitable to do so have to be identified. After an evaluation of the structural semantics of PSCS, two main aspects were chosen to be implemented generation-based:

1. Instantiation of composite structures, which includes:
   - Recursive instantiation of parts and ports
   - Instantiation and extraction of links acting as connections between runtime objects, respectively creating and retrieving runtime topologies
   - Object creation based on default values, which are modeled using instance specifications
2. Destruction of objects in the context of composite structures, which includes:
   - Recursive destruction of part and port instances
   - Destruction of corresponding links, respectively cleanup of runtime topologies after object destruction

The corresponding functionalities were implemented as extension modules for the existing generators UML4CPP (Jäger et al., 2016) and fUML4CPP (Bedini et al., 2017), which produce model-specific source code for the generated model libraries that substitutes the functionalities which were outsourced from the meta model. Other functionalities of PSCS remain in the meta model, respectively the model library generated from the corresponding PSCS Ecore model, which is linked with the generated model-specific source code during compilation to produce executable applications. Figure 1 depicts this process.

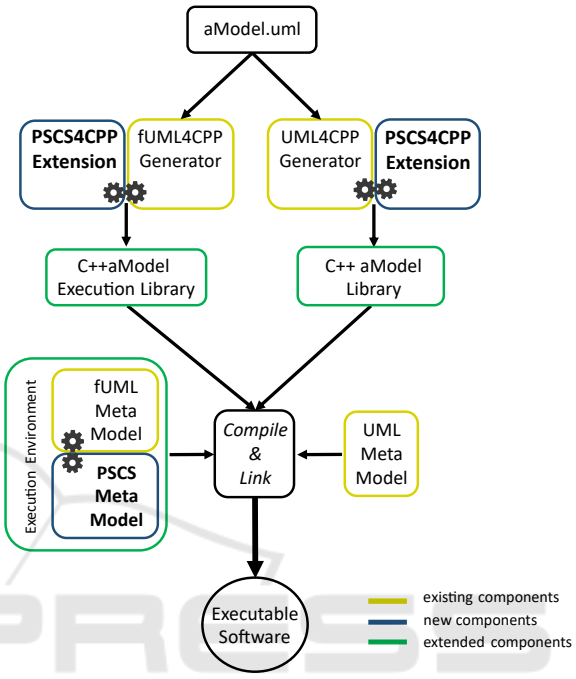

Figure 1: Resulting workflow for executable PSCS models and integrated components.

2.2 Generation-based Realization of Runtime Links

On the model level, parts and ports can be connected via connectors to form an internal network of a composite structure. In PSCS links (metaclass *CS_Link*) represent instances of connectors that interconnect specific runtime objects during model execution. That means that in the metamodel of PSCS, links are instances (*objects* in the term of programming languages) of a metaclass that explicitly carry information about which objects they connect. C++ itself does not provide any concept for generically connecting arbitrary objects with each other. To be able to implement PSCS's object instantiation and destruction semantics on the level of automated source code generation, a concept to represent links implicitly was developed. Consider the example classes of figure 2.

Because of association *A_left_right*, class *Left* has an attribute *my_right : Right* and class *Right* has n attribute *my_left : Left*. Assume that we want to model

[2]Essential MOF - an essential subset of OMG's Meta Object Facility

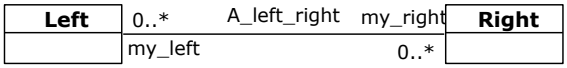

Figure 2: Class diagram showing two classes *Left* and *Right* associated bidirectionally via association *A_left_right*.

a simple composite structure *Comp* as depicted in figure 3.

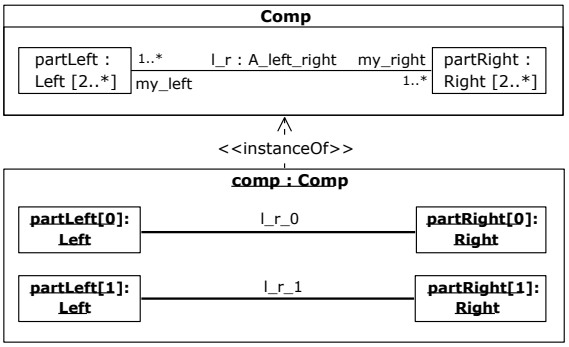

Figure 3: Composite structure diagram of class *Comp* with parts of classes *Left* and *Right* from fig. 2 that are connected via connector *l_r* typed by the association from fig. 2 (upper) as well as an instance *comp* of class *Comp* (lower).

When creating the instance *comp : Comp* we can represent its internal links implicitly by letting objects that should be connected refer to each other, using the end properties of the association that types the corresponding connector. To implicitly represent the link *l_r_0* of instance *comp* (see figure 3) for example, we must generate source code that creates a bidirectional reference between objects *comp::partLeft[0]* and *comp::partRight[0]*. To achieve that, we let *comp::partLeft[0]::my_right[0]* point to *comp::partRight[0]* as well as *comp::partRight[0]::my_left[0]* point to *comp::partLeft[0]*. That way, we can create topologies of connected objects without having to instantiate explicit link instances, but rather represent the information about connected objects implicitly depending on which objects refer to each other. This concept was used to implement a code generator that generates source code to instantiate different kinds of runtime topologies based on the model's definition by establishing bidirectional references as described above. The PSCS specification defines four kinds of topologies (so-called *connector patterns*) whose generator based instantiation is supported by the implementation presented by this paper:

- Empty Pattern: A topology without objects and hence without connections.
- Unconnected Pattern: A topology without connections between its participating objects.
- Array Pattern: A topology consisting of 1-to-1 connections between the corresponding parts/ports that form a sequential order of connected objects (figure 3 depicts an array pattern).
- Star Pattern: A topology in which each object of one part/port is connected to each object of the other part/port, forming a complete bipartite graph between the connected properties.

Additionally, a mechanism was implemented that extends the model-specific execution library produced by the fUML4CPP generator (see section 2.1). The extension covers a generator-based adapter functionality between the generator-based components and the metamodel level of a model execution. This adapter functionality creates explicit link objects (used in the PSCS metamodel) based on implicit link information (used in the generator-based components) at runtime, if and only if the PSCS execution environment requires them (e.g., to evaluate potential targets during an invocation delegation). By implementing this mechanism generation-based and hence model-specific it can be ensured that only links that may be useful for a specific execution step are processed at runtime.

2.3 Memory Management

The MDE4CPP framework in which the PSCS implementation presented by this paper was realized uses shared pointers of the C++-11 standard for its memory management. On the one hand, such shared pointers guarantee that memory is not deallocated as long as it is referenced from somewhere. In other words, no object that is managed by shared pointers is deleted as long as there exists at least one shared pointer instance that references the object (Stroustrup, 2013). On the other hand, shared pointers guarantee that objects are deleted, only when they are not referenced (i.e., not needed) anymore. The concept for implicitly representing links using objects that hold mutual references to each other, which was introduced in the previous section, inevitably produces circular dependencies between connected objects. This causes problems trying to delete an object that is involved in such a connection. If an instance of a composite structure's part shall be deleted during model execution it would not truly be deleted (in terms of memory deallocation) as long as it is connected to other objects, meaning that other objects hold references to it. This leads to both memory leaks as well as semantically incorrect behavior during a model execution.

In cases where circular dependencies between objects are needed, it is suggested to use weak pointers for the back-reference (e.g., when implementing

a composite pattern where both the parent and child class refer to each other) (Stroustrup, 2013). In our case there is no hierarchy between the connected objects, hence there is no identifiable 'back direction'. Using weak pointers for one of the bidirectional references would also have solved the issue for one 'direction' only. That is why the usage of weak pointers would not have been a sufficient solution for the implementation presented by this paper. To solve the issue sufficiently, a mechanism that generates model-specific deletion routines for each model class was developed. When the deletion routine of an object is invoked, all references of connected objects that refer to the object that is to be deleted, are destroyed. After this process, there is only one shared pointer left that manages our object. This last reference can now safely be deleted, which ultimately results in the deallocation of memory that holds the object, hence true deletion.

3 VALIDATION

This section first describes how the correct functionality as well as conformance of the presented PSCS implementation to OMG's specification were validated using a set of test models. An example model is described in section 3.1. Furthermore, section 3.2 presents a performance evaluation, including execution time and memory footprint of the PSCS implementation presented by this paper compared to a C++ reference implementation of PSCS as well as a Java reference implementation of fUML.

3.1 Functional Validation

For validating correct functionality of the presented PSCS implementation as well as its conformance to the original specification, a set of test models combined with corresponding unit tests was implemented. The test models are based on the PSCS test suite that is provided by OMG and described in the specification document itself. The PSCS specification states that passing all test cases defined in its test suite is sufficient for proving conformance of a tool that implements the specification (OMG, 2019). It should be mentioned that asynchronous communication semantics of PSCS are currently excluded from the presented implementation (and therefore from the validation process) as the processing of signals is not yet supported in MDE4CPP.

The realized set of test models consists of four different test suites, each addressing certain units of functionality specified by PSCS:

1. Instantiation of topologies of runtime objects based on composite structures.
2. Destruction of runtime objects which exist in the context of composite structures (i.e., instances of composite structures themselves or part/port objects of such instances).
3. Synchronous communication through a network of runtime objects connected through links via delegation of operation calls.
4. Synchronous communication via delegation of operation calls (as in point 3) using the *onPort* attribute of metaclass *InvocationAction*.

All test cases that were implemented to show the presented PSCS implementation's correct functionality could successfully be validated. The following section describes an example model of test suite 4 that combines all functionalities mentioned above. The original test model's description can be found in (OMG, 2019) on pages 94 and 95.

Example. The example model that is explained below addresses synchronous communication between instances of composite structures. More precisely, when the model is executed, an operation call is invoked in an initial caller object. The call is then forwarded through a network of port objects connected through links to a target object, where it is ultimately executed.

Figure 4 shows a class diagram of the example model. Every class realizes interface *I*, which defines the operation *assignP*. Only class *B* has an attribute *p* of type *int*. Hence, class *B* is modeled as the only class that truly implements the operation *assignP*, which means that only class *B* will define a method for the operation.

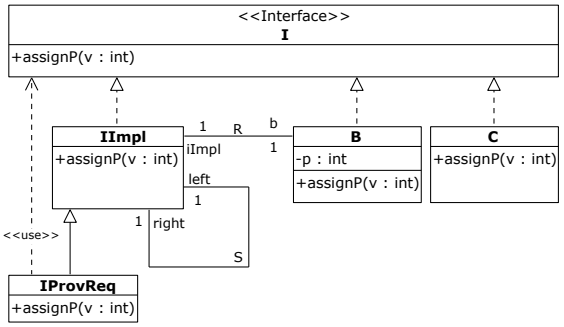

Figure 4: Class diagram for test model.

Figure 5 depicts class *C* from figure 4 as well as three newly-introduced classes A, *E* and *D* as composite structure diagrams. Classes *A*, *C* and *E* have ports of type *IImpl* (which stands for I_Implementation) or *IProvReq* (which stands for I_Provided_Required)

from figure 4. The types of a port specify its provided and required interfaces. In our example, class *IImpl* provides interface *I*, because it has an interface realization relationship (metaclass *InterfaceRealization*) with *I*. Class *IProvReq* both provides and requires *I* because it is inherited from *IImpl* (which determines provision) and at the same time, it uses (metaclass *Usage*) interface *I* (which determines request). When an operation call arrives at a port object, its provided and required interfaces determine how the call is further delegated from that port.

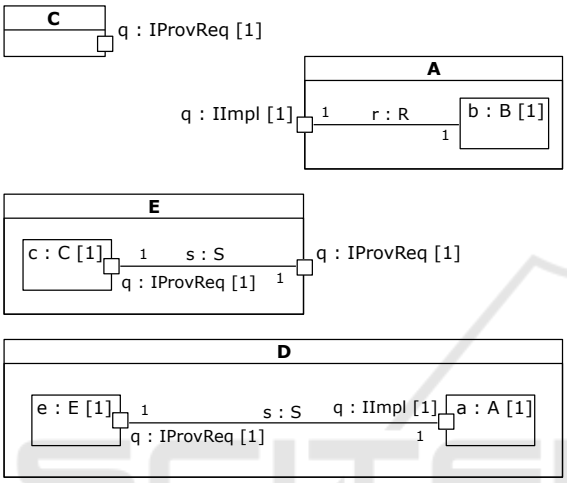

Figure 5: Composite structure diagram for test model.

Besides its structural aspects, the example model also contains a test behavior called *actTestCallDelegation* which is shown in figure 6 as an activity diagram. First, an instance *d : D* is created. Further, an operation call for *assignP* is invoked in *d::e::c* with port *C::q* being set for the *onPort* attribute (not depicted) of action *assignP() on Port q* inside of activity *actSetP*. Instance *d* is returned by activity *actTestCallDelegation* to evaluate certain postconditions after its execution.

If the PSCS implementation presented by this paper works correctly, the invoked operation call should first be delegated from port *d::e::c::q* to port *d::e::q* over a link that was created from connector *E::s* (see figure 5). Then the call should be dispatched out of *d::e* and forwarded to port *d::a::q* over a link that was created from connector *D::s*. Finally the operation call must be dispatched inside *d::a* and delegated to *d::a::b* over a link that was created from connector *A::r* where it is ultimately executed. If the invocation was delegated correctly, then after activity *actTestCallDelegation* has finished, property *d::a::b::p* must be set to the value *v* that was provided into the activity. In this case, *v* is chosen as 4.

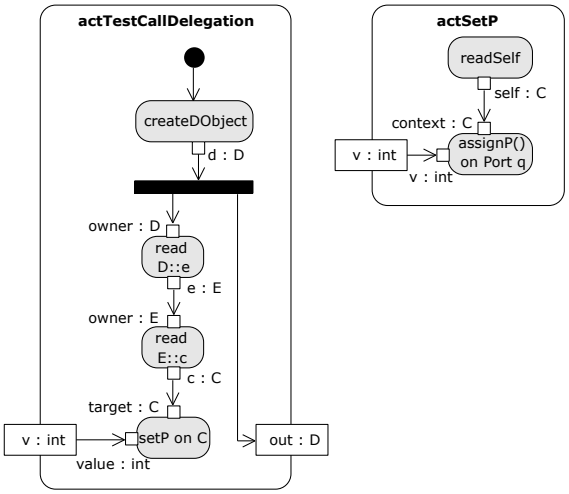

Figure 6: Activity diagram for test model.

The output of the example model's unit test routine is shown below:

```
Test model : Feature on both Required and
Provided Interface
-- Running test case: Feature on both
Required and Provided Interface --

d->a->b->p = 4
Operation call forwarded out of c through
c::q, out of e through e::q into a through
a::q to a::b : true

Test case successful : true
-- End of test case --
```

3.2 Performance Evaluation

This section presents a performance evaluation of the presented PSCS implementation. Because there is no third-party open-source implementation of PSCS available by now, a C++ reference implementation was developed in the context of the *MDE4CPP* project. This reference implementation was used for comparative evaluation. Furthermore, to be able to compare our results to those of a third-party model execution environment, the open-source Java fUML reference implementation by *Model Driven Solutions*[3] was used to reproduce certain PSCS-specific semantics for this benchmark.

Test Models. For performance evaluation, a test case that addresses instantiation and destruction of composite structures was chosen. Figure 7 shows the classes used in the realized test models. The internals of class *A* represent an array pattern (see section 2.2).

[3] see https://github.com/ModelDriven/fUML-Reference-Implementation

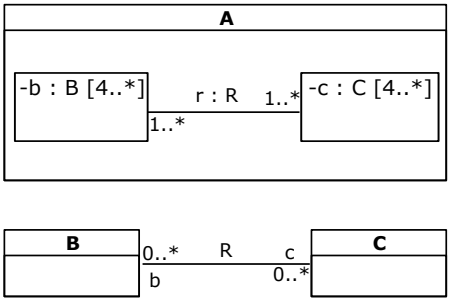

Figure 7: Classes used in the performance evaluation test models. Class *A* is depicted using a composite structure diagram.

When the test models are executed, instances of class *A* shall be created and destroyed iteratively in a loop. Concerning PSCS, this can be modeled by using the corresponding actions to create and destroy objects (metaclasses *CreateObjectAction* and *DestroyObjectAction*). The instantiation itself is then handled by the implemented PSCS semantics at runtime.

Because such semantics are not part of fUML, they have to be reproduced using activities in the fUML test model. The test model executed by *Model Driven Solution*'s Java fUML reference implementation includes corresponding activities. Figure 8 depicts the activity *actCreateParts*, which instantiates parts *A::b* and *A::c*. It does so by creating four instances of classes *B* and *C* each and adding them to the feature values of the corresponding parts using *AddStructuralFeatureValueActions*.
The activity *actCreateArrayPattern*, which is depicted in figure 9, is then used to instantiate the connections between those instances as defined by the array pattern. The realization of links as described in section 2.2 is reproduced by activity *actCreateArrayPattern* by adding each part instance to the corresponding feature value of its linked instance. The main loop activity of the fUML test model is shown in figure 10.

On the model level, the PSCS implementation presented by this paper uses a simple *CreateObjectAction* for instantiation. The creation of part and port objects as well as links between them is then handled by the constructor of the auto-generated model classes during execution. No further actions or declarations of any kind are necessary.

The PSCS reference implementation that was developed for comparative evaluation additionally requires the call of an empty constructor operation using a *CallOperationAction*. In this case an empty operation is an operation with no defined method. Constructor operation means, that the UML stereotype *<<Create>>* has to be applied to it. Conforming to the PSCS specification, such a special operation

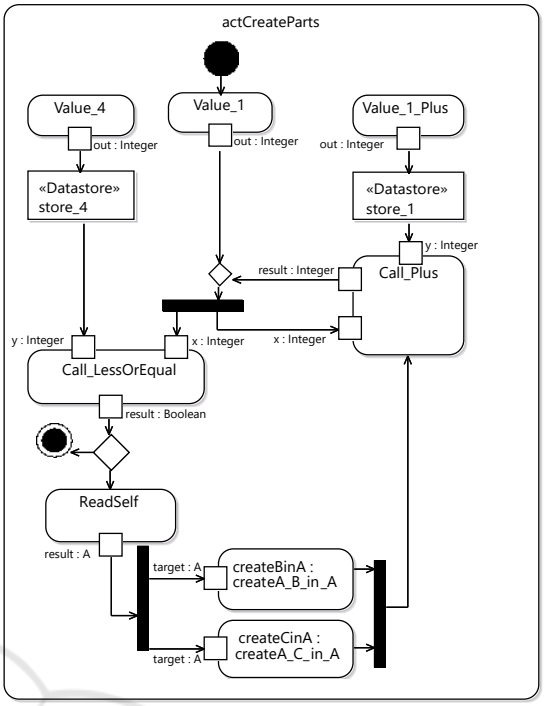

Figure 8: Activity *actCreateParts* creates four instances each for parts *A::b* and *A::c*.

serves as an indication for the execution engine, that when it is called, an instance of its owning class has to be created based on the defined instantiation semantics. Figure 11 shows the main loop acitivty of the test model for the PSCS reference implementation.

Creation and Deletion Benchmark. To compare execution times between the PSCS implementation presented by this paper and the fUML and PSCS reference implementations mentioned above, the test models described in section 3.2 were executed with different numbers of loop iterations: 1000, 2500, 5000, 10000, 50000 and 100000. The number of loop iterations is equal to the number of instances of composite structure *A* from figure 7 that are created and destroyed during model execution. The resulting execution times shown in figure 12 are arithmetical mean values calculated from 10 model executions each. The x-axis represents the number of iterations used for the model executions. Each region is split by the different execution environments that were used. The y-axis represents the average execution times in milliseconds.

The model execution of the PSCS implementation presented by this paper runs faster than the executions of both reference implementation test models. This result was expected as the presented PSCS implementation outsources functionalities for checking and

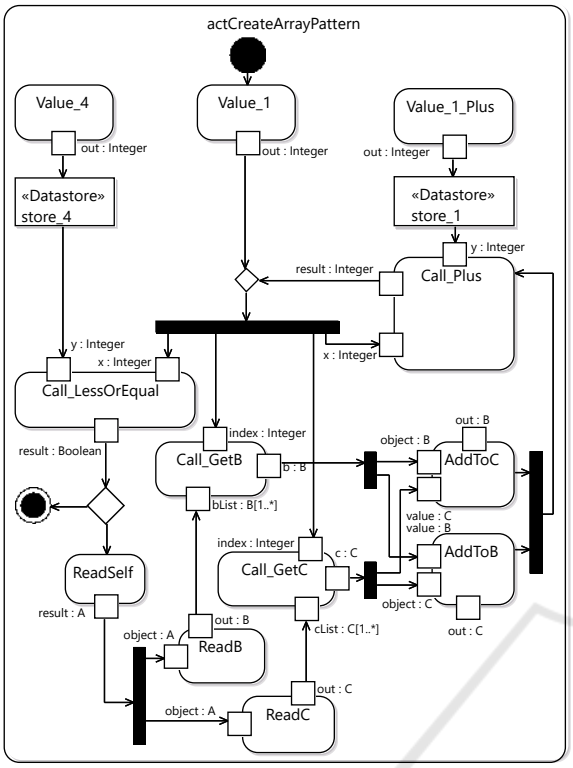

Figure 9: Activity *actCreateArrayPattern* reproduces the links between the instances of parts *A::b* and *A::c*. It does so by adding each instance of *A::b* to the feature value of feature *C::b* of the corresponding instance of *A::c* and vice versa.

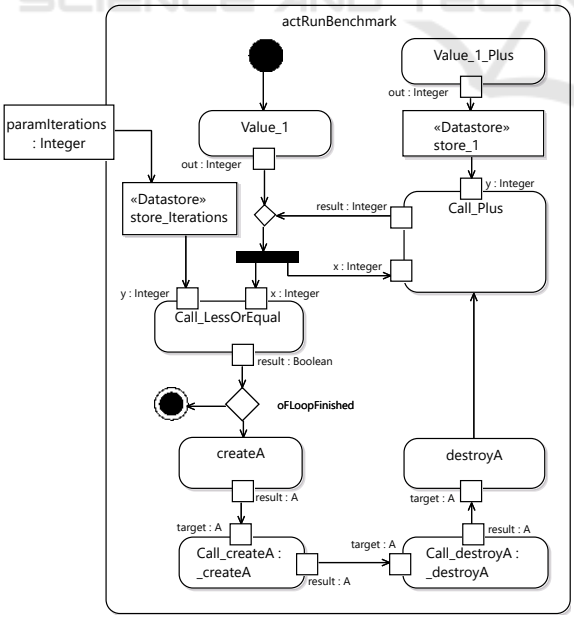

Figure 10: Activity *actRunBenchmark* iteratively creates, instatiates and destroys instances of composite structure *A* from fig. 7.

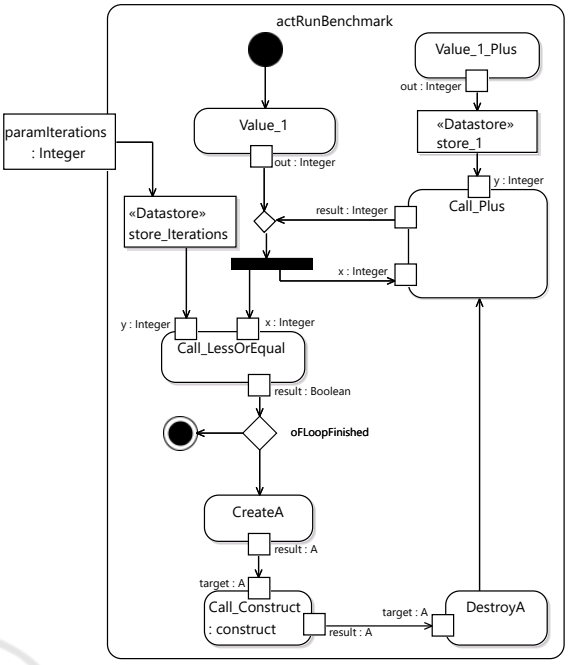

Figure 11: Activity *actRunBenchmark* of the test model executed by the PSCS C++ reference implementation. Action *Call_Construct* calls a special constructor operation to instantiate composite structure *A* as described in section 3.2.

evaluating model information to the level of code generation. For both reference implementations, those functionalities are executed by their respective execution environments. This produces computation overhead at runtime compared to the presented implementation. Execution times of the fUML test model executed by *Model Driven Solution*'s Java fUML reference implementation are higher compared to those of the PSCS reference implementation. On the one hand, this could be explained by general performance advantages of C++ over Java. On the other hand, a pure fUML model was used to reproduce PSCS-specific semantics. Because of that, the model is larger and contains much more behavior that has to be executed by the corresponding execution engine than the PSCS test models.

Memory Footprint. Memory usage during test model execution was measured in the same manner as the measurement of execution time described in section 3.2. Figure 13 depicts the arithmetical mean values of peak RAM usage from 10 model executions each. The Java fUML reference implementation uses significantly more RAM than both C++ PSCS implementations. The reason for this behavior might be the additional RAM usage caused by the Java virtual machine. The PSCS implementation presented by this paper requires the least amount of RAM. This can

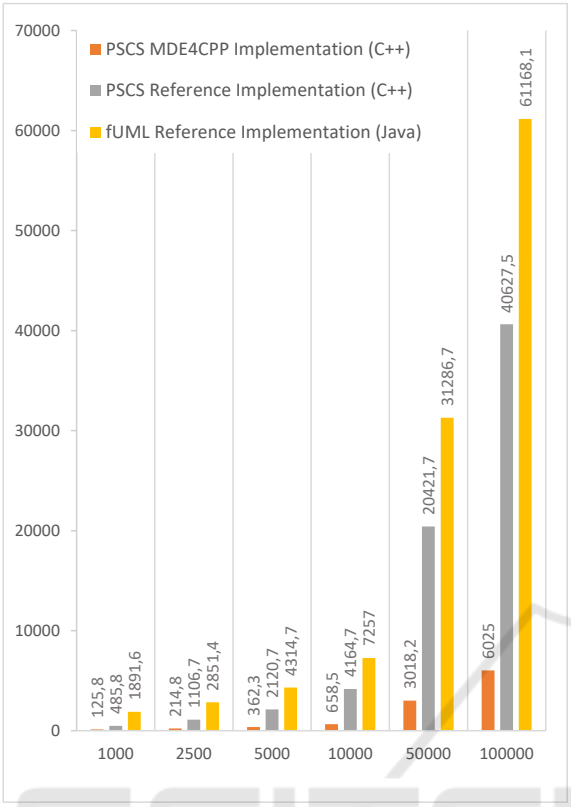

Figure 12: Average execution times per number of created and destroyed objects in milliseconds split by different execution environments.

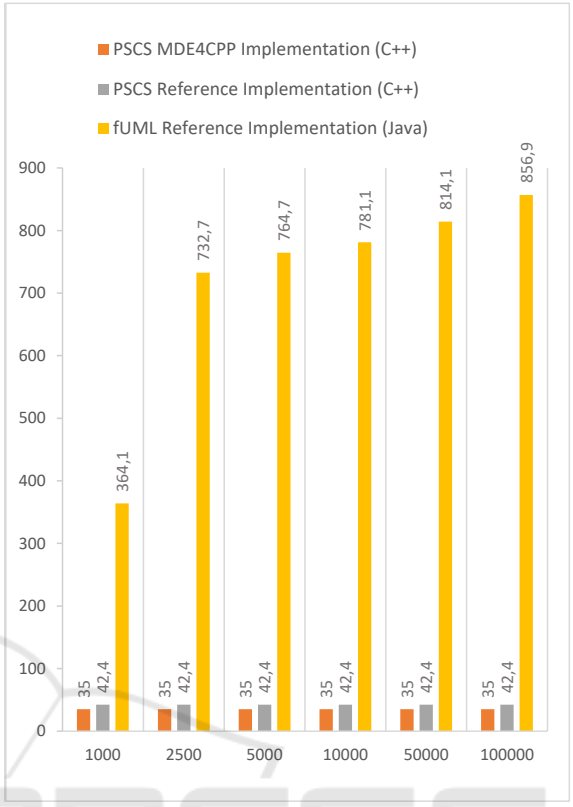

Figure 13: Average peak RAM usage per number of created and destroyed objects in MB split by different execution environments.

be explained by the fact that the instantiation and destruction semantics used by this implementation are directly translated to model-specific C++ code during generation. Thus, overhead produced by the execution engine of the PSCS reference implementation (which leads to a slightly higher RAM usage) is saved during the execution of the presented PSCS implementation. Moreover, the RAM usage of this implementation (as well as the compared PSCS reference implementation) is constant for each number of iterations that was tested. This indicates that no memory leaks exist.

4 CONCLUSION

This paper presented a model-driven implementation of OMG's PSCS specification in C++. The presented solution substitutes specific parts of the PSCS execution model with auto-generated, model-specific source code. More precisely, the implementation presented by this paper outsources PSCS's execution semantics for instantiating and destroying objects from the actual execution model to the generation level. By that, the performance of the actual model execution is improved. This is achieved because computational overhead at runtime produced by repeatedly evaluating and processing static (i.e., runtime-independent) structural model information is omitted and done pre-runtime during the process of code generation.

For functional validation, a set of test models and associated unit tests based on OMG's original PSCS test suite was realized. During the validation process, the presented implementation was tested against the set of test models described in section 3.1. The results showed that the presented PSCS implementation functions correctly and conforms to the PSCS specification.

The results of the performance evaluation presented in section 3.2 show that substituting runtime computation of the PSCS execution model with pre-runtime computation via code generation improves execution time significantly. The memory footprints of the evaluated test model executions prove that the usage of C++ combined with the memory management described in section 2.3 leads to much lower RAM usage compared to the Java fUML reference implementation.

Future Work. By now, PSCS semantics concerning asynchronous communication via signals is excluded from the presented implementation. To provide a full realization of PSCS, including all aspects of its runtime semantics, signal processing and asynchronous communication is yet to be realized.

Based on the results of this paper, which display the possibilities of exploiting automated source code generation to reduce runtime computation of model execution engines, we aim to develop model-driven execution engines for fUML, PSCS and also PSSM that rely more on code generation and also conform (functionality-wise) the underlying specifications.

REFERENCES

Bedini, F., Maschotta, R., Wichmann, A., Jäger, S., and Zimmermann, A. (2017). A model-driven fuml execution engine for c++. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2017, page 443–450, Setubal, PRT. SCITEPRESS - Science and Technology Publications, Lda.

Eclipse Foundation (2018). Acceleo, see https://wiki.eclipse.org/acceleo.

Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011). Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 471–480, New York, NY, USA. Association for Computing Machinery.

Jäger, S., Maschotta, R., Jungebloud, T., Wichmann, A., and Zimmermann, A. (2016). An emf-like uml generator for c++. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 309–316.

OMG (2011). Semantics of a Foundational Subset for Executable UML Models (fUML) Specification, Version 1.0. https://www.omg.org/spec/FUML/1.0/PDF. online.

OMG (2014). MDA Guide Revision 2.0. https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf. online.

OMG (2015). Precise Semantics of UML Composite Structure (PSCS) Specification, Version 1.0. https://www.omg.org/spec/PSCS/1.0/PDF. online.

OMG (2017). Unified Modeling Language, Specification, Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF. online.

OMG (2019). Precise Semantics of UML Composite Structure (PSCS) Specification, Version 1.2. https://www.omg.org/spec/PSCS/1.2/PDF. online.

Stahl, T. and Völter, M. (2005). *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 1st edition.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley, 4th edition.

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