AN FUML EXTENSION SIMPLIFYING EXECUTABLE UML MODELS IMPLEMENTED FOR A C++ EXECUTION ENGINE

Francesco Bedini
Ralph Maschotta
Alexander Wichmann
Armin Zimmermann

Systems and Software Engineering Group
Technische Universität Ilmenau
Ehrenbergstraße 29
Ilmenau, TH, Germany
francesco.bedini@tu-ilmenau.de

ABSTRACT

The fUML allows creating models by using a subset of UML diagram elements. This approach has the benefit of keeping the execution engine simple, but it is sometimes limiting, as it does not permit to specify certain advanced constructs such as loops in a concise way. This paper shows and discusses some of these constructs and introduces additional component specifications or modifications to existing ones to reduce the number of elements needed to realize a model. The methodology consists in enriching an existing execution engine for C++ by specifying additional UML items that do not belong to the fUML subset. The execution engine allows the execution of diagrams that model complex structures with less overhead leading to models which are easier to realize and maintain. Moreover, the model-driven execution engine allows supporting those additional elements without impairing the reached level of conformance.

Keywords: fuml, uml, execution engine.

1 INTRODUCTION

The fUML, defined by the Object Modeling Group (OMG), permits the creation of executable UML models for a restricted subset of UML activity and class diagram elements by providing for them a precise execution management (OMG 2016). The execution is based on a predefined execution model, which is computationally complete for the subset, and a Java execution engine that is specified and available as a reference implementation. This approach can be used to simulate UML-specified models and has the benefit of keeping the execution engine small and simple. However, it is limiting from the point of view of a modeler (Ellner et al. 2011), as it does not allow several aspects of the used UML diagrams. If such constructs are needed, they may be emulated by complex submodels as a workaround, but this leads to less compact and straightforward models which are then harder to read and understand.

This paper reviews some "forbidden" constructs and proposes ways to specify them in a simpler manner compared to workarounds, while still keeping the resulting fUML models fully executable. This is done by introducing additional constraints, extensions to existing components, and the use of additional UML elements to reduce the total number of components needed.

SpringSim-Mod4Sim 2017, April 23-26, Virginia Beach, VA, USA

This work has been motivated by the model-based specification of executable models for optimization heuristics, when necessary control structures were hard to specify with the set of available elements (Wichmann et al. 2016). The methodology consists in enriching an existing execution engine for C++ and a model generation process realized with the Eclipse component Acceleo by specifying additional UML elements that do not belong to the fUML subset.

Most of the proposed elements have been implemented in the execution engine, which now allows the execution of diagrams that model complex structures with less overhead. This allows having models which are easier to realize and maintain. Moreover, it is leading to a reduced amount of generated components and better code structure. Hence, the computational effort as well as the size of the executable elements decreases, thus accelerating a subsequent model simulation. As the C++ execution engine is itself realized in a model-driven manner, it supports the additional elements in a modular way without impairing the achieved fUML conformance level.

The aim of this paper is to define a complementary simplified notation for certain fUML elements and to extend an existing C++ execution engine to support additional items that were excluded from the fUML specification.

This paper is structured as follows. Section 2 shows the constructs that are currently difficult to realize in fUML in a simple manner and proposes a possible way of realizing them in an easier manner. Section 3 briefly shows how the elements have been added to the execution engine and the results of their validation. Finally, Section 4 discusses the results of this paper and suggests future improvements.

2 FLAWS OF FUML AND POSSIBLE SOLUTIONS

The fUML was originally specified and made first available in February 2011 by selecting a subset of UML elements that led to a computationally complete visual language able to describe the execution of UML's activity diagrams in a precise formal manner. However, sometimes this subset is quite limited and makes it difficult to implement certain common structures, such as loops, in a concise way.

FUML, as an executable subset of UML, was meant to prevent over-specification inside UML diagrams (Rumpe 2014), preventing the creation of extremely detailed UML diagrams to show or express the executability of the system.

This goal has been partly reached, but fUML diagrams are currently not as scalable as many components are needed to realize simple actions. Another problem for fUML modelers is the lack of a custom-made fUML modeling environment that allows creating models that only comply with the fUML specification and allowing to specify the element behaviors easily. These flaws make fUML diagrams good in theory, but difficult to be used and applied in an industrial environment or real-life scenarios, where other kinds of proprietary models are thus applied more often (Baker, Loh, and Weil 2005).

For example, it would be practical to have UML's *Data Store* nodes defined in the fUML's specification, to have an element that allows to specify an infinite loop on the same object by storing and sending an object token. All redundant object flows and merge nodes needed to implement an equivalent model could then be avoided, resulting in a simpler and clearer model.

The existing literature shows the fUML's limits and shortcomings. For example, other extensions to the standardized fUML virtual machine have already been proposed in (Mayerhofer et al. 2012). Similarly, other authors find the graphical realization of fUML diagrams for certain scenarios to be tedious (Lazăr, Motogna, and Pârv 2010). The following subsections cover some of these flaws and propose solutions.

```
for(int j = 0; j < tokens.size(); j++) {
    Token token = tokens.getValue(j);
    Value value = ((ObjectToken)token).value;
    if (value != null) {
        parameterValue.values.addValue(value);
    }
}
```

Figure 1: The Java code snippet shows how the fUML specification defines the collection of the objects token from the activity parameter nodes at the end of the execution of an activity.

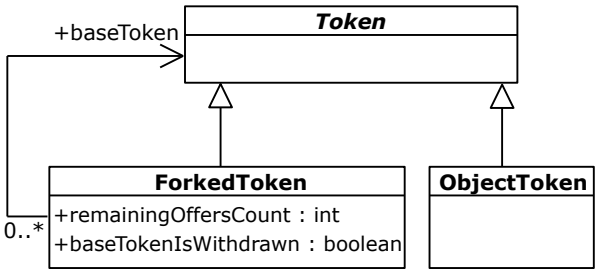


Figure 2: A UML class diagram showing the relation between a ForkedToken and an ObjectToken. As it can be seen they are not directly related and therefore a cast of a ForkedToken to an ObjectToken is not possible.

2.1 Forked Tokens as Activity Outputs

During the process of developing the C++ execution engine, it has been noticed that the fUML specification does not allow forked tokens to be consumed by activity parameter output nodes. As the code listing from the fUML specification shown in Figure 1 (OMG 2016) shows, the value of the token is retrieved by casting the token to an ObjectToken and accessing its value property afterwards.

Unfortunately, as shown in the class diagram in Figure 2, `ForkedTokens` and `ObjectTokens` cannot be cast one to the other, as they are not in an inheritance relationship. They are both Tokens, but it is not possible to retrieve an object instance from a *ForkedToken* in the same way as it is retrieved from an *ObjectToken*.

To access the instance that is pointed to by the *ForkedToken*, the code shown in Figure 3 should be used. It is necessary to check if the given token is a *ObjectToken*, and then directly getting its value, or whether it is a *ForkedToken*, in which case the value has to be retrieved through the *baseToken* attribute.

Our execution engine now allows to correctly terminate activities by providing forked tokens to activity output parameter nodes.

```
for(int j = 0; j < tokens.size(); j++) {
    Token token = tokens.getValue(j);
    Value value = null;
    if (token instanceof ObjectToken) {
        value = ((ObjectToken)token).value;
    }
    else if (token instanceof ForkedToken){
        value = ((ForkedToken)token).baseToken.value;
    }
    if (value != null) {
        parameterValue.values.addValue(value);
    }
}
```

Figure 3: The Java code snippet extends the fUML specification by supporting forked tokens as valid tokens by activity output parameter nodes.

2.2 Modeling Loops

This section and its parts describe and discuss in detail another problem that we have encountered during the realization of loops in fUML, which has been solved by a special handling of target pins of *CallOperationActions*.

As the fUML specification does not support variables (OMG 2016), stating that object flows can be used to pass data between actions, it is necessary to use input/output pin pairs and object flows. Moreover, the fUML specification excludes the *setupParts* of the loop node, that therefore in fUML can only be composed of at least one *Test* and an optional *BodyPart* section.

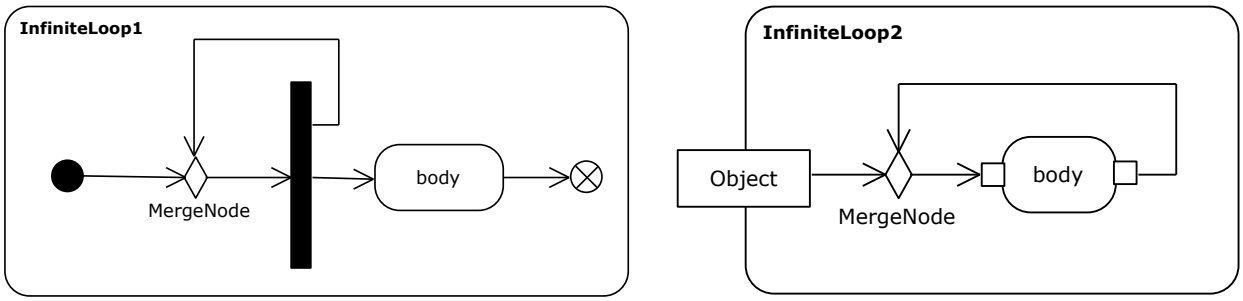
Loops are one of the basic elements needed in most programs and algorithms. While it is extremely simple to implement them in any programming language, or in UML by using loop nodes, fUML does not provide a satisfying full support of such an element.

Loop nodes are suitable for hierarchically structured loops but show difficulties in being used for small ones, which are more widely used by modelers. For this reason, in this paper we focus on the realization of loops without the use of *LoopNodes*. Possible ways of realizing such kind of loops using the current fUML specification are shown in Figure 4a and 4b.

Figure 4a shows an infinite loop realized with a merge and a fork node. In this case, there are no objects passed and the loop achieved by control flows only. Both fork node and merge node take care of providing a control token to the body action, that represents the body of the loop. A terminating condition has to be verified with a decision node between the merge and fork nodes to realize a finite loop.

Please note that this construct only works if the body part doe not accept other input parameters. If the fork node is fired before the input pin of the body action gets a token, it will result in an infinite loop. between the merge and fork node.

Figure 4b shows an infinite loop with object passing which is realized with a merge node and an action, which can either be a call behavior action with one input and one output pin or a call operation action with one target pin and one output pin returning the object itself. A terminating condition can be placed between the merge node and the input pin for a *while* loop for this case also, or between the output pin and the merge



(a) (b)

Figure 4: Possible realization of infinite loops in fUML. (a): an infinite loop realized with a merge and a fork node, (b): an infinite loop passing an object realized with an action.

node for a *do-while* loop. These two small examples show how to obtain an edge that always provides a token, which is exactly what a UML *DataStore* node would be suitable for.

2.2.1 Central Buffer Nodes and Data Store nodes

CentralBufferNodes are excluded from the fUML specification because *"they were judged to be unnecessary for the computational completeness of fUML"* (OMG 2016). A data store node is defined in the UML specifications as a "*central buffer node for non-transient information. A data store keeps all tokens that enter it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any tokens in the object node containing that object*" (OMG 2015). *DataStoreNodes* are activities' *ObjectNodes* and are not supported in fUML as they are a subclass of the excluded *CentralBufferNode*.

For this reason, data store nodes have been realized in our fUML C++ execution engine by following the behavior specified in the UML specifications. They can be now used to realize an input or target pin that holds its token instead of consuming it immediately at the first activation, as shown in Figure 5.

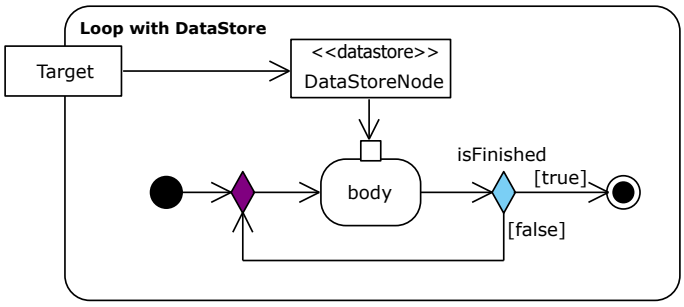


Figure 5: UML class diagram representing the data store node hierarchy.

Another possible way of realizing this persistent kind of tokens is explained in the next section.

2.2.2 Target Pins of Call Operation Actions

Figure 6 shows a simplified version of an optimization loop, similar to the one proposed in (Wichmann, Jäger, Jungebloud, Maschotta, and Zimmermann 2016). The loop realized with a chain of different call

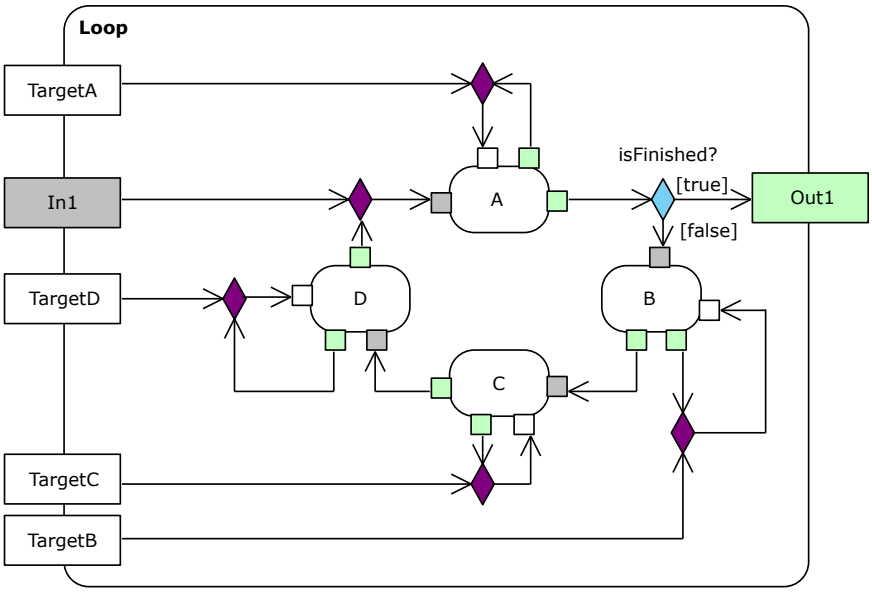


Figure 6: A loop realized with call operation actions as allowed by the current specification. The pins follow this scheme: white pins are target pins, gray pins are input pins and green pins are output pins. Merge nodes are shown in violet and decision nodes in cyan.

operation actions. It is currently necessary to provide an instance of the class on which the operations will be performed on the target pins of the call operation action elements. In this way, it is necessary to always provide an object flow that either comes from an activity input pin, a *ReadStructuralFeatureAction*, or an object generated by another action inside the activity. Otherwise, the loop would not execute more than once without these new tokens on the target pins.

In this way, it is necessary for the considered case to have four merge nodes and four output pins that are superfluous and make the diagram cumbersome to read and unpleasant to realize. A possible simplification can be obtained by defining a special behavior for target pins.

Persistent Tokens for Target Pins In Figure 7 a possible simplification is shown. Target pins can be defined as *persistent* pins. That means pins have a special behavior that allows their tokens to be preserved rather than consumed directly. The four small loops that were necessary to provide a new token to the call operation action's target pins can then be avoided.

Realizing the necessary connections is hardly feasible for larger activity diagrams, as their number grows quickly when different call operation actions use the same target object. In that case it is necessary to use a fork node to distribute the object tokens to all the call operation actions.

A special notation for Target Pins The suggested approach allows the target pins to remain disconnected under certain circumstances, and replaces this connection with a specifically structured name of the form *self.<attributeName>*. This is, of course, not a limitation to the expressiveness of the language, but it clutters the resulting diagram with additional fork nodes and object flows.

Figure 8 shows an activity with three call operation actions that are executed in sequence, thus demonstrating the proposed notation. The first one is realized as currently allowed by fUML: that is, by receiving an object instance on its target pin. The second and the third one show the proposed simplification. `Op2` is called on the instance that is currently present in the attribute *attr1* of the class that holds the activity. When the pin is disconnected and does not have any specific name, as in the `Op3` call operation action, it is assumed as

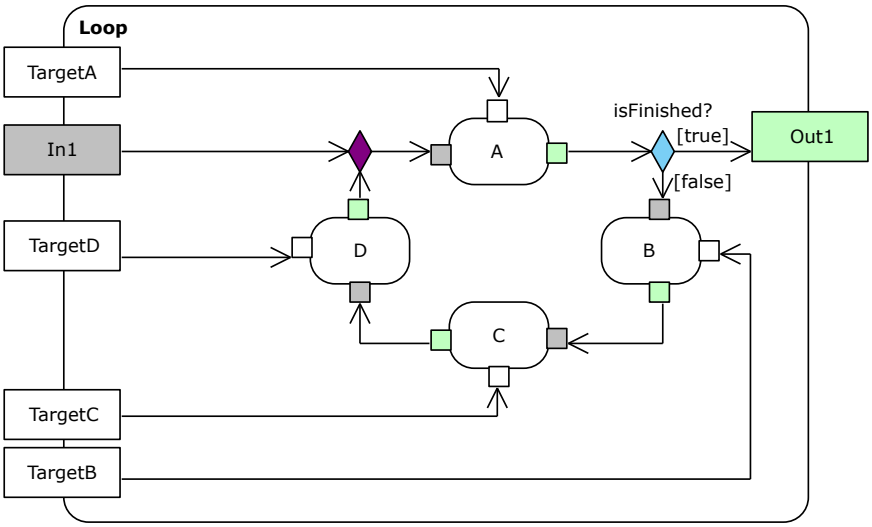


Figure 7: The same loop realized with the proposed persistent target pins. Once a token is received, it will not be consumed immediately but will remain available to the target pin.

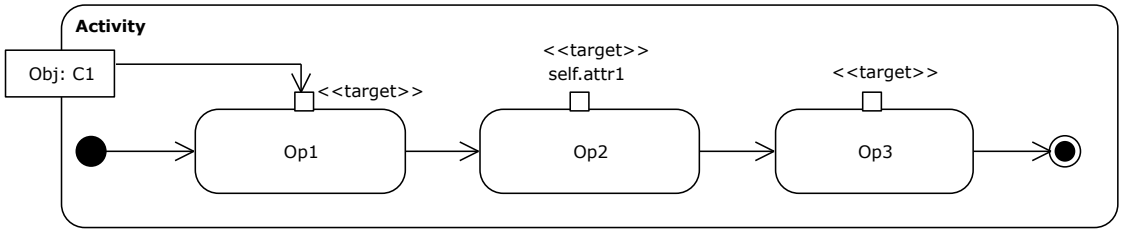


Figure 8: The class diagram shows three call operation actions whose target pins are shown on top of them. `Op1` works according to the current specification, whereas `Op2` and `Op3` show our proposed extension.

"self". This is then equivalent to call a *ReadSelfAction* (that retrieves the activity's context object) and to pass the result to the target pin.

In Figure 9 it is shown how `Op2` would need to be realized by using the current fUML's formalism. The merge and fork node realize an infinite source of tokens that provide to the target pin a persistent object token. The instance is then retrieved via the *ReadSelf* action that returns the instance of the context object of the action execution. The requested attribute is retrieved from this object by the *ReadStructuralFeature* action and provided to the target pin.

The OCL constraint shown if Figure 10 expresses that the call operation action must have at least one incoming control flow or object flow other than the one that reaches the target pin to terminate the activity life cycle correctly. Without this constraint, the activity can only be terminated if it contains at least a reachable activity final node.

Thanks to this notation, it would be possible to simplify the loop shown in Figure 6 to the one shown in Figure 11. This is not only a graphical simplification, but may lead to an execution speedup too. It allows to eliminate the overhead caused by input pins and parameter passing, and to execute the operation directly on attributes that are available to the activity anytime.

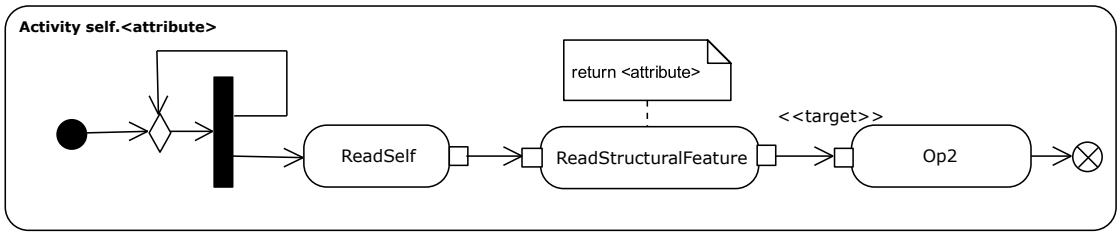


Figure 9: fUML equivalent to the proposed persistent *self.<attribute>* annotation.

```
context CallOperationAction
    inv otherInputs:
        self.incomingEdges->size() > 0 or
        self.input->select(self.incoming->size()>0)->size()>0
```

Figure 10: OCL constraint that allows a correct end of the execution of the activity containing an extended call operation action.

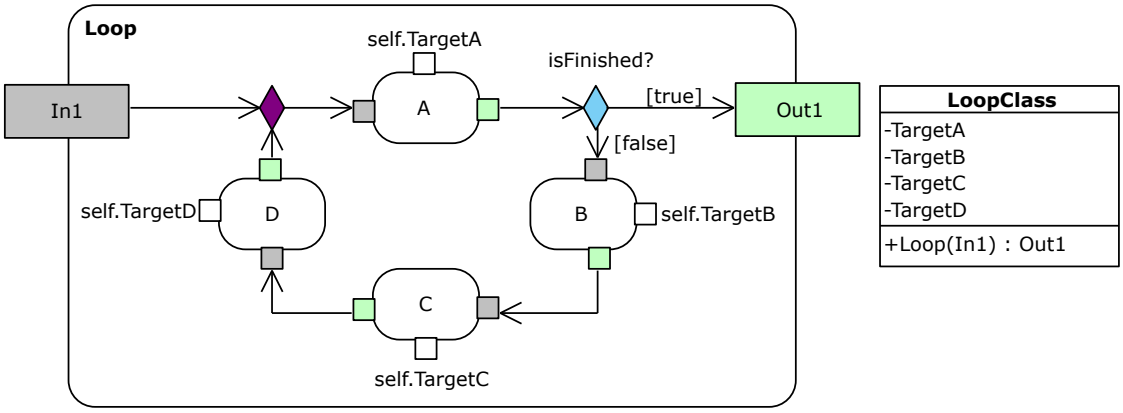


Figure 11: On the left side, the loop realized with the proposed simplified notation and the persistent target pins. On the right-hand side a UML class showing the corresponding method and attributes that will be accessed by the target pin using the simplified notation.

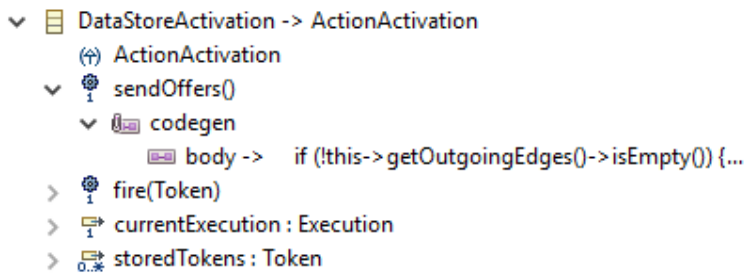


Figure 12: Example of the realization inside the ecore fUML model of the *DataStoreActivation*.

Table 1: Comparison between different loop realizations.

fUML (ms)	Simplified Target (ms)	Speedup (%)
41937,5	16562,5	2,53

3 IMPLEMENTATION AND VALIDATION

All the solutions and improvement proposed in this paper, except the simplified notation for target pins, have been realized in our fUML execution engine for C++ and were validated with use cases and integration tests. More details about the execution engine's performance regarding execution efficiency and memory occupied can be found in (Bedini et al. 2017).

The implementation has been realized inside an extended ecore model. The relevant elements have been added and implemented in C++.

An example for the solution presented in Section 2.2.1 regarding the *EClass* "DataStoreActivation" is shown in Figure 12. It contains the element's *EOperations* (in the shown example: `sendOffers()`) and *EReferences*. The code is defined inside the *EAnnotations*. Each annotation must hold at least one *details* entry, whose key can be either *includes* or *body*. The code inside the body annotation contains the C++ implementation of the standard UML behavior for the corresponding operation.

The ecore model has been generated to C++ and successfully compiled and executed. The optimization loops described in Section 2.2.2 have been generated and executed by our execution engine. The results were analyzed and compared to each other.

The execution output shows that the value obtained by the execution of the different proposed solutions are equal and the loop's elements were executed in the correct order. Table 1 shows the average of 10 executions of the loops shown in Fig. 6 and Fig. 7 for 2500 loop iterations. Each call operation action simply adds 1 to the given input value, until a value greater than 10000 is reached.

The obtained result shows that the diagram realized by using the simplified notation has a better performance than the one which is allowed in standard fUML thanks to the lower number of elements and unneeded passages of tokens.

4 CONCLUSION AND FUTURE WORKS

This paper has shown how highly-used modeling practices can be simplified by extending the fUML's specification to support more advanced elements. Loops realized with chains of call operation actions can be reduced to a simplified notation regarding their target pins or by the use of *DataStoreNodes*.

Thanks to these simplifications, it becomes possible to realize complex diagrams such as optimization loops with a reduced number of elements that are more pleasant to read and easier to generate. The execution performance benefited from the reduced number of generated elements that led to a smaller number of token activations.

Future Work The final goal of this work is to realize a modeling IDE for fUML that supports the definition of the constructs proposed in this paper, allowing the generation and the execution of the resulting model in C++. This will be achieved either internally generating the required fUML equivalent models introduced in this paper to keep the resulting tool fully fUML-compliant, or by directly generating the proposed elements in a code-efficient way. The source code for the C++ code generator and execution engine can be found at the MDE4CPP project page (Systems and Software Engineering Group 2016).

ACKNOWLEDGMENTS

This work was supported by the *Federal Ministry of Economic Affairs and Energy* of Germany [20K1306D], *Federal Ministry for Education and Research* of Germany [01S13031A], and a Ph.D. grant from the *Thuringian State Graduate Support* program.

REFERENCES

Baker, P., S. Loh, and F. Weil. 2005. "Model-Driven Engineering in a Large Industrial Context — Motorola Case Study". In *Proc. Model Driven Engineering Languages and Systems: 8th Int. Conf. (MoDELS 2005)*, edited by L. Briand and C. Williams, pp. 476–491. Montego Bay, Jamaica, Springer Berlin Heidelberg.

Bedini, F., R. Maschotta, A. Wichmann, S. Jäger, and A. Zimmermann. 2017. "A Model-Driven C++-fUML Execution Engine". In *5th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2017)*. accepted for publication.

Ellner, R., S. Al-Hilank, J. Drexler, M. Jung, D. Kips, and M. Philippsen. 2011. "A FUML-Based Distributed Execution Machine for Enacting Software Process Models". In *Modelling Foundations and Applications: 7th European Conference (ECMFA 2011)*, edited by R. B. France, J. M. Kuester, B. Bordbar, and R. F. Paige, pp. 19–34. Birmingham, UK, Springer Berlin Heidelberg.

Lazăr, I., S. Motogna, and B. Pârv. 2010. "Behaviour-Driven Development of Foundational UML Components". *Electronic Notes in Theoretical Computer Science* vol. 264 (1), pp. 91 – 105.

Mayerhofer, T., P. Langer, and G. Kappel. 2012. "A runtime model for fUML". In *Proceedings of the 7th Workshop on Models@run.time*, edited by N. Bencomo, G. Blair, S. Götz, B. Morin, and B. Rumpe, pp. 53–58. New York, Vienna University of Technology, Austria, ACM.

OMG 2015, March. "UML 2.5 Specifications". online.

OMG 2016, January. "Foundational UML 1.2.1 Specifications". online.

Rumpe, B. 2014. "Executable Modeling with UML. A Vision or a Nightmare?". *CoRR* vol. abs/1409.6597.

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

Wichmann, A., S. Jäger, T. Jungebloud, R. Maschotta, and A. Zimmermann. 2016, April. "Specification and execution of system optimization processes with UML activity diagrams". In *Proc. IEEE Systems Conference (SysCon 2016)*, pp. 1–7. Technische Universität Ilmenau.

AUTHOR BIOGRAPHIES

FRANCESCO BEDINI received the Master's degree in Research in Computer and Systems Engineering with distinction from Technische Universität Ilmenau in 2016. He is currently working towards the Ph.D. degree in the Systems and Software Engineering Group of TU Ilmenau. His main research interests include efficient generation of code from UML and fUML models and their validation. His email address is francesco.bedini@tu-ilmenau.de.

ALEXANDER WICHMANN received the Master's degree in Computer Science from the Technische Universität Ilmenau, Germany, in 2013. He is currently working towards the Ph.D. degree in the Systems and Software Engineering Group of Technische Universität Ilmenau. His main research interests include model-based specification and execution of system optimization processes. His email address is alexander.wichmann@tu-ilmenau.de.

RALPH MASCHOTTA received the Diploma degree in technical computer science from the University of Applied Sciences Schmalkalden, Germany, in 1999 and the Ph.D. degree from Technische Universität Ilmenau, Germany, in 2008. Since 2011, he has been a senior scientist and lecturer with the Systems and Software Engineering Group, Faculty of Computer Science and Automation, TU Ilmenau. His main research interests include object-oriented programming, modeling, and design of software systems, as well as image processing and image recognition in medical and industrial applications. His email address is ralph.maschotta@tu-ilmenau.de.

ARMIN ZIMMERMANN received the Diploma, Ph.D., and Habilitation degrees from Technische Universität Berlin, Germany, in 1993, 1997, and 2006, respectively. He has been a full Professor of systems and software engineering since 2008 and the director of the Institute for Computer and Systems Engineering since 2012 with Technische Universität Ilmenau, Germany. His research interests include discrete event system modeling and performance evaluation and their tool support with embedded systems applications. Armin Zimmermann is a member of the Industrial Automated Systems and Controls Subcommittee of the IEEE IES Technical Committee on Factory Automation. His email address is armin.zimmermann@tu-ilmenau.de.