Towards Integrating Modeling and Programming Languages: The Case of UML and Java[⋆]

Patrick Neubauer, Tanja Mayerhofer, and Gerti Kappel

Business Informatics Group, Vienna University of Technology, Austria
{neubauer, mayerhofer, gerti}@big.tuwien.ac.at

Abstract. Today, modeling and programming constitute separate activities carried out using modeling respectively programming languages, which are neither well integrated with each other nor have a one-to-one correspondence. As a consequence, platform and implementation details, such as the usage of existing software components and libraries, are usually introduced on code level only. This impedes accurate model-level analyses that take platform-specific decisions into account as well as the direct deployment of executable models on the target platform. In this work we present an approach for integrating existing software libraries with fUML models—an executable variant of UML models for which a standardized virtual machine exists—not only at design time but also at runtime. As a result of that, the modeler is empowered with the capabilities provided by existing software libraries on model level. Our approach is evaluated based on unit tests and initial case studies available in the ReMoDD repository that assess the correctness, performance, and completeness of our implementation.

1 Introduction

Back in 1966, the first object-oriented programming language was born. Its name is SIMULA and it combined modeling and programming in a unified approach, that is one of the strengths provided by object-orientation [4]. Carrying this idea further, the language BETA, which was developed based on SIMULA and DELTA, was designed for supporting both designing and programming systems. It even provided besides a textual syntax also a graphical notation for representing the same abstract syntax tree, such that the user could switch between both forms of representation. This was possible, since they had a one-to-one correspondence. Later, the graphical syntax of BETA was replaced by UML and, therewith, the one-to-one correspondence was broken referred to as impedance mismatch [4].

Today, the design of programming languages and modeling languages is largely separated from each other as it is carried out by different communities. Likewise, in today's mainstream object-oriented software development, modeling and programming are conceived as separate activities. This results in an impedance mismatch between modeling languages and programming languages. For instance, there is no one-to-one

[⋆] This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859 and by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the FFG BRIDGE program grant no. 832160.

correspondence between the modeling concepts provided by UML and the programming concepts provided by Java.

In model driven engineering, this issue can be overcome by generative software development approaches utilizing intelligent code generators that are able to bridge the gap between modeling and programming languages for generating complete and deployable program code from models. However, these code generators add platform and implementation details, such as invocations of existing software components and libraries, to the resulting code, which are not reflected in the models. Hence, in model-level analyses targeted at validation, verification, or optimization, they cannot be considered. Therefore, platform and implementation details that are to be considered in model-level analyses have to incorporated into the model. One approach to achieve this is envisioned by Seidewitz [10] who proposes to avoid the "programming gap" at all, by incorporating all implementation details into the model by utilizing UML's action language. While this allows to incorporate algorithmic details into UML models, platform and implementation details concerning the usage of existing software components and libraries cannot be incorporated into the model in this approach. For instance, it is not possible to integrate an existing datastore available in terms of a software library into a UML model. This impedes accurate model-level analyses taking into account the behavior of reused software components and libraries, as well as the direct deployment of models on the target platform as envisioned by Seidewitz. The need for making existing software available on model level was also highlighted by Selic [11]. He describes that one way to accomplish this is by allowing direct calls to such existing software from within the model.

This paper is concerned with enabling the integration of existing software components with UML models. This enables on the one hand to construct more accurate UML models enabling model-level analyses closer to the target platform, and on the other hand a direct deployment of models on the target platform. In this work, that emerged from first ideas by Mayerhofer *et al.* [5, 6] and subsequent work in Neubauer's master's thesis [7], we propose the integration of software libraries with executable UML models developed with fUML—a subset of UML whose execution semantics was standardized by OMG in terms of a virtual machine (VM). More specifically, at design time, the software library is reverse-engineered into a UML class model representing the library's API structure, which is in the following used by the modeler to reference library classes and operations. At runtime, these references are used by an *Integration Layer* to locate, instantiate, and modify compiled stateful library components. As a result of this, the modeler can take advantage of the full power provided by already existing libraries as well as re-use existing software components. Thereby, we aimed at fulfilling the following requirements: (i) making it as natural to the modeler as possible to use existing software libraries (i.e., the modeler does not need to touch any source code), (ii) making the usage of libraries transparent to the modeler (i.e., the modeler can interact with library components just like with any other natively defined fUML component), and (iii) not extending the fUML metamodel or modifying the fUML VM as this would break the conformance to the standard.

The remainder of this paper is organized as follows. First, we introduce fUML in Section 2. Thereafter, we describe our approach for integrating software libraries with fUML models in Section 3. In Section 4, we report on two initial case studies which we carried out for evaluating our approach. Related work is discussed in Section 5. Finally, Section 6 concludes the paper and provides an outlook on future work.

2 Foundational UML

fUML, introduced by OMG [8] in 2011, precisely defines the execution semantics of a subset of UML in terms of a virtual machine capable of executing fUML compliant UML models. Thereby, the UML subset considered by fUML includes UML class diagrams to define the structural aspects of a software system and UML activity diagrams to define its behavioral aspects. With the introduction of fUML, UML can be used as a programming language, that is nevertheless more abstract than existing third-generation programming languages [12]. Therewith, it seeks to overcome the model-code impedance mismatch by replacing source code entirely with fUML models.

However, fUML does neither provide nor foresees a functionality to provide access to existing software libraries. Instead, it intends to build its own library called *Foundational Model Library*. This library contains user-level elements, which can be referenced in fUML models. However, it is only composed of packages that define primitive types, such as Boolean, Integer, Real, and String, and a set of primitive functions for those data types (e.g., a function to concatenate two String values). While the first version of the fUML specification [8] already defines the Foundational Model Library, the latest version of the specification V1.1 still does not further extend the library's capabilities. When comparing the Foundational Model Library with libraries found in traditional third-generation programming languages, such as Java, the capabilities provided by, e.g., the Java Class Library (JCL) or any third-party library, are far beyond those of the Foundational Model Library. By looking at functionality provided by JCL, one can not only find features to build graphics and sound, access databases, and perform math operations, but also sophisticated abstractions on the underlying operating system and hardware to provide access to resources like the network or file system. While, from a technical standpoint, it is possible to extend the Foundational Model Library with the aforementioned functionalities, doing so is infeasible due to the required effort of adding them by the still rather small community currently using fUML. Furthermore, it is not only desirable to use JCL functionality in fUML models, but also to re-use already existing software components in fUML models, i.e., to re-use any existing software library.

3 Library Support for fUML

We presented the initial idea to integrate software libraries with fUML in [6]. In our approach, required classes of existing software libraries are integrated into the fUML model during design time by using reverse engineering techniques. During runtime, a dedicated Integration Layer is employed, which handles calls to any specified software library. Thus, the Integration Layer extends the fUML VM with the ability to handle access to software libraries. For this, it makes use of the command and event API developed for fUML within the moliz project [5]. This command and event API allows the

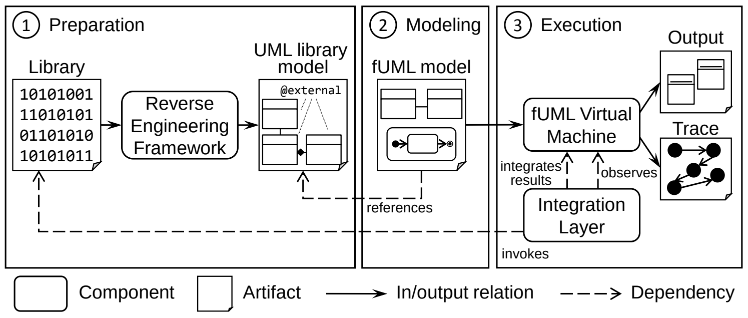

Fig. 1: Library support for fUML at a glance

Integration Layer to detect calls to software libraries by fUML models during runtime, and to re-integrate the result of the performed library call back into the fUML runtime environment. For performing the actual call to the software library, the Integration Layer makes use of reflection techniques.

The implementation of our Integration Layer[1] originating from Neubauer's master's thesis [7] currently supports the integration of Java libraries in fUML models. However, it is also possible to support other programming languages which provide reflection capabilities. The primary goal of the prototype is to provide the modeler with the possibility to instantiate Java library classes using CreateObjectActions of fUML, to modify Java library class instances using AddStructuralFeatureValueActions, and to call operations upon them via CallOperationActions. Additionally, the result of a library call, such as the return value of an operation call, is translated from Java to fUML and integrated into the fUML runtime environment.

Dynamic Class Loading and Reflection. The two major capabilities required by the Integration Layer to fulfill its goals are dynamic class loading and the reflection technique. The former ability builds upon the Java classloader that is part of the Java runtime environment and enables to dynamically load Java classes into the Java virtual machine. The reflection technique enables meta programming and hence allows the Integration Layer to instantiate and modify Java classes during runtime.

Figure 1 visualizes the entire approach in three steps from making libraries available to be referenced in the fUML model at design time to the point where the fUML model is executed. In the following, these steps are described in more detail.

Step 1: Preparation. In the *Preparation* step, the Java library to be used is prepared for being integrated with fUML models. Therefore, the structural information about the Java library's API is reverse-engineered into a UML class model (depicated as "UML library model" in Figure 1) either from the source code or a compiled JAR file using, e.g., the Eclipse plug-ins MoDisco[2] or Jar2UML[3]. To reference Java library classes and operations inside an fUML model, references from the fUML model to the obtained

[1] https://github.com/patrickneubauer/fuml-library-support
[2] http://www.eclipse.org/MoDisco
[3] http://soft.vub.ac.be/soft/research/mdd/jar2uml

UML library model may be created. Hence, the modeler can use the Java library in a natural way by referencing the reverse-engineered UML library model and does not need to deal with source code. To make the Java library usable at runtime, the UML library model is extended with UML comments indicating library classes as well as the location of the library's compiled JAR file.

Step 2: Modeling. Having obtained a UML library model from the Java library that is annotated with the location of the compiled JAR file, the library can be used in the development of an fUML model in the *Modeling* step. For this, any Eclipse-based UML editor like, e.g., the Eclipse UML Model Editor or the Papyrus UML Model Editor[4] can be used. Whenever the fUML model has to access the Java library, references from the fUML model to the UML library model are created. In more detail, to create an instance of a Java library class, a CreateObjectAction may be used having the classifier reference set to the UML class representing the respective Java class in the UML library model. Using an output pin, the created object can, e.g., be provided to the target input pin of a CallOperationAction. Such a CallOperationAction can be defined to call an operation on the Java library object. For this, the operation reference of the CallOperationAction has to be set to the respective UML operation representing the Java class' operation in the UML library model. To specify the value of an existing Java object's member field, an AddStructuralFeatureValueAction can be used referring to the respective UML attribute in the library model via its structuralFeature reference.

Step 3: Execution. To start the execution of an fUML model in the *Execution* step, the Integration Layer passes the fUML model to the fUML VM. Furthermore, using the command and event API provided by the moliz project [5], it registers itself as a listener to the fUML VM. Hence, the Integration Layer is notified about any event occurring during the execution of the fUML model and acts upon any required library access. For example, if the execution of a CreateObjectAction is completed, an *activity node exit event* referring to the CreateObjectAction is received. In case the action refers to a UML class contained by the UML library model, the Integration Layer detects that it has to instantiate the respective Java class. Similarly, modifications of existing Java objects via AddStructuralFeatureValueActions and Java operation calls via CallOperationActions are detected by the Integration Layer. The result of the library access, such as the instantiated Java object or the return value of an operation call, is integrated back into the fUML runtime environment by translating it from Java to fUML and providing it to the fUML VM. Thus, resulting values can be further processed during the fUML model execution. This means that an Java object instantiated by the fUML model via a CreateObjectAction may be modified by the model using *AddStructuralFeatureValueActions* and *CallOperationActions*. Thus, the Integration Layer makes state full Java objects accessible at runtime. As an example, we describe the handling of CreateObjectActions to instantiate Java classes in more detail. A detailed description of handling AddStructuralFeatureValueActions and CallOperationActions can be found in [7].

Handling a CreateObjectAction. When the Integration Layer receives an *activity node exit event* concerning a CreateObjectAction referring to a library class, it interrupts the model execution and performs the following steps.

[4] `http://www.eclipse.org/modeling/mdt`

1. **fUML Placeholder Object Retrieval.** As a result of executing the CreateObjectAction, the fUML VM created an fUML object of the referenced UML class, which is contained by the UML library model. This fUML object represents a stub or placeholder object of the Java object to be instantiated.
2. **Java Object Creation.** The Java class to be instantiated is determined by examining the classifier reference of the CreateObjectAction referring to the UML class that represents the Java class in the UML library model. While the namespace and name of this UML class uniquely identify the Java class, the location of the JAR file containing the Java class is captured in a UML comment owned by that UML class. Being supplied with both the unique identification and location of the Java class, the Integration Layer uses the reflection technique to create an instance of that exact Java class.
3. **fUML Object Creation.** To integrate the instantiated Java object into the fUML runtime environment, it is first transformed into an fUML object. For this, a new fUML object is created and its feature values are set according to the Java object's member field values. For example, in case the Java member field is of type int, a corresponding fUML IntegerValue is assigned to the fUML object. After translation, the fUML placeholder object retrieved in the first step is replaced by the translated fUML object.
4. **CreateObjectAction Result Assignment.** Finally, the fUML object created in the previous step is assigned to the CreateObjectAction's result output pin in form of an object token containing a reference to this fUML object.
5. **Object Bookkeeping.** In order to use the instantiated Java object in later stages of the fUML model execution (e.g., by using a CallOperationAction to call a library operation upon this object), both the Java object and the fUML object are added to a dedicated map data structure.

4 Critical Discussion

Succeeding the development of the prototype, we carried out initial case studies that are available in the ReMoDD repository[5]. Two of them are presented in this section. Additional case studies are discussed in [7]. The main research questions we aimed to answer through these case studies are as follows:

- **Correctness.** Is the developed prototype correct? In particular, does the prototype work as expected? If not, what causes the malfunction?
- **Performance.** How is the performance of executing an fUML model accessing a software library compared to executing a similar plain Java application? How much overhead is imposed by the prototype on top of the fUML VM performance?
- **Completeness.** Can every library be used? What are the limitations?

Mail Case Study. In this case study, we built an fUML model that uses a software library to compose and send an e-mail to an existing e-mail address. The library

[5] `http://www.cs.colostate.edu/remodd/v1/sites/default/files/fuml-extlib-examples.zip`

used in this case is Apache's Common Email library[6]. Used fUML action types include ValueSpecificationAction, CreateObjectAction, and CallOperationAction. Within the fUML model, a *SimpleEmail* object is created and multiple calls upon it are made to specify various e-mail parameters, such as its subject and receiver. The parameter values itself are specified through multiple ValueSpecificationActions. At the end, the last operation call finally transmits the created e-mail to the specified recipient.

Petstore Case Study. In the Petstore case study, we developed a model that creates, stores, and retrieves objects through the Google App Engine datastore. To realize the access to the datastore, the Objectify API[7] was used. The fUML model created for the Petstore case study behaves as follows. Initially, domain entities, such as *Customer*, *Address*, and *Order*, are registered through the Objectify service. Then, for testing purposes, the datastore is setup to run on the local machine rather than in the cloud. Next, both a *Customer* and *Address* object are created and multiple of their features (e.g., first name, last name, and city) are specified. While all of those features are simple Strings, assigning the created *Address* instance to the *Customer* requires handling a CallOperationAction receiving a complex input parameter. Afterwards, the created *Customer* instance is saved in the datastore. In order to verify the success of the previous operation, the same *Customer* is looked up in the same datastore.

Correctness Results. To ensure the correctness of the prototype, we built unit tests for the individual prototype components. The correctness was also confirmed by the case studies. For the case studies, we built a Java application that implements the same functionality as the respective fUML model, and compared the results obtained by executing both. On one hand, in the Mail case study, the composed e-mail was correctly sent and received by the specified recipient. On the other hand, in the Petstore case study, both the instance of the *Customer* and *Address* were created, the *Address* was added as a complex field to the *Customer*, and the *Customer* was populated to the local datastore. Finally, the same *Customer* was retrieved from the datastore showing the Integration Layer's correct treatment of the Petstore case study. Hence, the developed prototype successfully obtains the desired result in both case studies.

Performance Results. Since the performance of the Mail case study heavily depends on the current state of the network, the performance has been evaluated with the Petstore case study, in which the datastore is placed on the local machine and, hence, is not liable to network interference. Table 1 shows the performance results of executing both the fUML model and a corresponding Java implementation. Every benchmark has been performed five times and the visualized results represent the median of all executions. When looking at both execution time and memory one can see that executing the fUML model takes up a considerable longer time and larger amount of memory. However, the Integration Layer implementation only imposes a very small additional overhead on top of the fUML VM. In case of 50 consecutive executions, the measured overhead amounts to approximately 3% in terms of execution time and less than 1 MB in terms of memory (not depicted in the table) constituting less than 1% in overall memory overhead. The remainder of the overhead is imposed by the model execution, i.e., the fUML VM itself resulting from the fact that the fUML VM is not yet as optimized

[6] `http://commons.apache.org/proper/commons-email`
[7] `https://code.google.com/p/objectify-appengine`

as the Java virtual machine. For example, while the fUML standard foresees concurrent threading of the execution, the Java implementation of the fUML VM does not (yet) provide such capabilities.

Completeness. The implemented prototype currently supports only a subset of fUML's actions. In detail, the following fUML actions have been taken into consideration during the development of the prototype: CreateObjectAction, CallOperationAction, and AddStructuralFeatureValueAction. While the prototype supports all kinds of primitive data structures offered by the fUML standard, complex data types are only supported in a subset of possible use cases. Moreover, for example, the Java reflection library as well as Java libraries making use of dependency injection cannot be directly used as there are no corresponding concepts in the fUML standard. Additionally, static field and static operations are not supported. A list of explored limitations can be found in [7].

# of Executions	Time (Java)	Time (fUML)	fUML Overhead	Integration Layer Overhead	Memory (Java)	Memory (fUML)
1	583 ms	830 ms	42.37%	2.08%	5 MB	75 MB
10	936 ms	1260 ms	34.62%	4.03%	2 MB	77 MB
50	1199 ms	2541 ms	111.93%	2.75%	2 MB	88 MB

Table 1: Performance Results

5 Related Work

As discussed in Section 2, fUML defines its own library called Foundational Model Library. With this library, fUML provides primitive data types and associated primitive behaviors. Furthermore, this library can be extended by implementing for each data type or function to be added, a dedicated interface and registering this implementation at the fUML VM. Considering the vast amount of existing software libraries, which have to be added to the Foundational Model Library, in order to enable the development of fUML models executable on the target platform, and the still rather small community using fUML, this approach is currently infeasible from a practical point of view. Furthermore, re-using existing software components would require considerable additional implementation effort in this approach.

Nevertheless, some extensions of the Foundational Model Library exist. The Alf standard [9] provides a textual notation for fUML models and extends fUML with an additional library that provides further data types (e.g., Natural and Collection types) and corresponding functions (e.g., Collection functions). Cuccuru *et al.* [1] propose an extension of Alf with the Value Specification Language (VSL). VSL is a language standardized by the OMG for defining the values of non-functional properties of real-time and embedded systems in UML models. As part of this extension, additional data types and functions for these data types were added to Alf's library.

The Cameo Simulation Toolkit provided by No Magic, Inc. constitutes a commercial implementation of fUML. In this tool, it is possible to access software libraries in fUML models by embedding Java source code in the body of fUML OpaqueActions. This approach fundamentally differs from our approach as it requires the modeler to write actual source code. Furthermore, the result of executing source code embedded in the fUML model can only be accessed by source code embedded in another part of the fUML model. However, it is not possible to further process the result using (other) native fUML actions, such as the AddStructuralFeatureValueAction. A similar approach as the one implemented by the Cameo Simulation Toolkit is also provided by the commercial UML tools IBM Rational Rhapsody, IBM Rational Software Architect, and Enterprise Architect. While they do not fully conform to fUML, they support embedding source code in UML actions and thus enable access to software libraries.

A different approach is taken by the UML execution and debugging tool Pópulo [3]. In this tool, the supported action language can be extended using UML profiles. Therefore, the execution semantics of the introduced stereotypes have to be implemented by Java classes inheriting from dedicated classes of Pópulo's API. This approach is very similar to the approach of extending the fUML Foundational Model Library.

Another related approach we want to mention is Umple [2], which is a model-oriented programming language provided with a user interface that allows both graphical and textual modeling as well as programming in parallel. For modeling, Umple supports many UML modeling concepts used in UML class diagrams and UML state machines. Imperative code to be added to the model, such as code for class operation bodies, is written in one of the target programming languages supported by Umple, such as Java and PHP. In a code generation step, code for the respective target language may be generated. This approach differs from our approach in that it creates code from models which is then compiled and executed, while in our approach executable models are directly interpreted by the fUML VM.

6 Conclusion

In this work, we have presented an approach to integrate software libraries with UML models. In this approach, an existing software library can be integrated with fUML compliant UML models at design time, as well as at runtime during model execution. Therefore, the library to be used is reverse-engineered into a UML library model that represents the library's API structure. This UML library model can be referenced by an fUML compliant UML model through CreateObjectActions for creating stateful instances of library classes, AddStructuralFeatureValueActions for modifying these instances, and CallOperationActions for calling operations on these instances. During the fUML model execution, a dedicated Integration Layer operating on top of the fUML VM handles accesses to compiled stateful library class instances using the reflection technique and dynamic class loading.

The conducted initial case studies evaluated the feasibility of the approach particularly focusing on correctness, performance, and completeness. The correctness of the prototype has been successfully evaluated by unit tests and confirmed by the case studies. Regarding performance, we found that the total overhead caused by the Integration

Layer sums up to approximately 3% in execution time and less than 1% in memory. The remaining overhead is caused by the fUML VM, which suggests an optimization of the fUML VM. In terms of completeness, it was found that the implementation supports a subset of available fUML actions and libraries. As future work we plan to increase the set of supported fUML actions in the prototype, to conduct new case studies eventually discovering further limitations, and to revise the implementation for overcoming these limitations.

With our approach, fUML is enhanced with the rich power of existing software libraries. In particular, it enhances fUML with further capabilities, such as accessing operating system functionality as well as reusing existing software components. Thus, it allows to build more accurate models that are closer to the target platform and, hence, enables more accurate model-level analyses for validation, verification, or optimization purposes. Furthermore, our approach enables to develop fUML models that are executable on the target platform, such that the compilation of models to source code through code generation becomes obsolete.

References

1. Cuccuru, A., Gérard, S., Terrier, F.: Defining MARTE's VSL as an Extension of Alf. In: Proc. of 14th Int. Conference on Model Driven Engineering Languages and Systems (MODELS'11), LNCS, vol. 6981, pp. 699–713. Springer (2011)
2. Forward, A., Badreddin, O., Lethbridge, T.C., Solano, J.: Model-Driven Rapid Prototyping with Umple. Software: Practice and Experience 42(7), 781–797 (2012)
3. Fuentes, L., Manrique, J., Sánchez, P.: Pópulo: A Tool for Debugging UML Models. In: Companion of 30th Int. Conference on Software Engineering (ICSE'08). pp. 955–956. ACM (2008)
4. Madsen, O.L., Møller-Pedersen, B.: A Unified Approach to Modeling and Programming. In: Proc. of 13th Int. Conference on Model Driven Engineering Languages and Systems (MODELS'10), LNCS, vol. 6394, pp. 1–15. Springer (2010)
5. Mayerhofer, T., Langer, P., Kappel, G.: A Runtime Model for fUML. In: Proc. of 7th Int. Workshop on Models@run.time (MRT'12). ACM (2012)
6. Mayerhofer, T., Langer, P., Wimmer, M.: Towards xMOF: Executable DSMLs based on fUML. In: Proc. of 12th Workshop on Domain-Specific Modeling (DSM'12). ACM (2012)
7. Neubauer, P.: Integration of External Libraries into the Foundational Subset of UML. Master's thesis, Vienna University of Technology (2014), `http://publik.tuwien.ac.at/files/PubDat_227306.pdf`
8. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0 (2011)
9. Object Management Group: Action Language for Foundational UML (ALF), Version 1.0.1 (2013)
10. Seidewitz, E.: UML: Once More with Meaning (2013), `http://www.slideshare.net/seidewitz/uml-once-more-with-meaning`
11. Selic, B.: The Pragmatics of Model-Driven Development. IEEE software 20(5), 19–25 (2003)
12. Selic, B.: The Less Well Known UML - A Short User Guide. In: Proc. of 12th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'12). LNCS, vol. 7320, pp. 1–20. Springer (2012)