FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Integrating fUML into Enterprise Architect

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Uwe Brunflicker, BSc
Matrikelnummer 0625178

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Mitwirkung: Dipl.-Ing. Dr. Tanja Mayerhofer, BSc

Wien, TT.MM.JJJJ

(Unterschrift Verfasserin) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Integrating fUML into Enterprise Architect

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Uwe Brunflicker, BSc
Registration Number 0625178

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.Prof. Dipl.-Ing. Mag. Dr.techn. Gerti Kappel
Assistance: Dipl.-Ing. Dr. Tanja Mayerhofer, BSc

Vienna, TT.MM.JJJJ

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Uwe Brunflicker, BSc
Erdbergstraße 160/49, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasserin)

Acknowledgements

I would like to use this opportunity to express my gratitude to all the people who have supported me in completing this thesis. I am thankful for their guiding ideas, as well as their advice on issues I could not have solved without their continued help.

Firstly, I want to express my warmest thanks to Ms. Tanja Mayerhofer, who has guided me through creating this thesis, has provided a number of solutions to problems and whom I could always count on for advice.

Secondly, I want to express my gratitude to Mr. Richard Deininger from LieberLieber Software GmbH. Mr. Deininger, the lead developer of the Advanced Modeling Using Simulation and Execution (AMUSE) plugin, has provided numerous insights into the inner workings of the plugin, and has supported me extensively in altering it so it can be used for the prototype. Furthermore, I would like to thank Mr. Daniel Siegl from LieberLieber Software GmbH for providing the infrastructure (both hardware and software) that was utilized to create the prototype itself.

Lastly, I want to thank Mrs. Kappel and her staff for supervising my thesis, and ensuring that the whole process went over smoothly.

Thank you!

Uwe Brunflicker

Abstract

The rise of Model Driven Development (MDD) has renewed the interest in the execution of models. The predominant modeling language applied in MDD is the Unified Modeling Language (UML). Unfortunately, UML lacks clear execution semantics. This has lead to a plethora of different interpretations both in academia and industry, hindering the interoperability of UML tools supporting the execution of models. A possible solution to this problem is the so-called *Foundational Subset For Executable UML Models (fUML)* standard published by the Object Management Group (OMG). fUML is an extension of UML, defining a standardized execution semantics for a subset of UML. fUML has, however, not yet been widely adopted in commercial tooling.

This thesis investigates the integration of fUML with existing commercial UML modeling tools to contribute to the adoption of fUML and identify challenges arising in the integration. To this end, this thesis aims to answer the following research questions:

1. Can fUML be integrated into a proprietary UML model execution environment? Which challenges arise in the integration?

2. Does the standardized fUML model execution environment provide the same functionality as a proprietary UML model execution environment?

3. Is the performance of the standardized fUML model execution environment comparable to the performance of a proprietary UML model execution environment?

To explore these questions, a prototypical integration of fUML into the commercial UML modeling tool Enterprise Architect (EA) has been implemented in this thesis. The goal of the prototype is to allow execution and debugging of UML state machines and sequence diagrams via EA's execution environment, in conjunction with the execution of UML activity diagrams via the fUML execution environment.

The evaluation of the prototype has shown that the integration of the execution environments has been completed successfully. The prototype is capable of executing and debugging state machines, sequence diagrams and fUML compliant activity diagrams in conjunction, while still providing the same functionality as the proprietary execution environment. The performance analysis has shown that the prototype is slower than the proprietary execution environment provided by EA. This is mostly due to the necessity of running two different execution environments in parallel.

Kurzfassung

Die Verbreitung von Model Driven Development (MDD) hat zu einem erhöhten Interesse an der direkten Ausführung von Modellen geführt. Die vorherrschende Modellierungssprache im MDD-Bereich ist die Unified Modeling Language (UML). Jedoch verursaucht das Fehlen einer vollständigen und präzisen Definition des Ausführungsverhaltens von UML Modellen Inkompatibilitäten zwischen Ausführungswerkzeugen und -umgebungen für UML. Eine mögliche Lösung für dieses Problem ist der *Foundational Subset For Executable UML Models (fUML)* Standard der Object Management Group (OMG). Dieser Standard ist eine Erweiterung des UML Standards, die das Ausführungsverhalten einer Teilmenge von UML definiert. fUML findet bislang jedoch nur geringe Anwendung in kommerziellen UML Werkzeugen.

Diese Diplomarbeit untersucht die Möglichkeit, fUML in ein existierendes UML Ausführungswerkzeug zu integrieren, um zur Verbreitung von fUML beizutragen. Insbesondere werden die Herausforderungen, die durch eine solche Integration entstehen, untersucht. Folgende wissenschaftliche Fragestellungen werden behandelt:

1. Ist es möglich, fUML in ein existierendes UML Ausführungswerkzeug zu integrieren? Welche Herausforderungen entstehen hierbei?
2. Stellt die standardisierte Ausführungsumgebung von fUML dieselbe Funktionalität wie eine proprietäre Ausführungsumgebung zur Verfügung?
3. Ist die Performanz der standardisierten Ausführungsumgebung von fUML vergleichbar mit jener einer proprietären Ausführungsumgebung?

Um diese Fragen zu beantworten, wurde ein Prototyp entwickelt, der fUML in das UML Ausführungswerkzeug Enterprise Architect (EA) integriert. Das Ziel des Prototypen ist es, das Ausführen und Debuggen von UML Zustandsautomaten und Sequenzdiagrammen mittels der Ausführungsumgebung von EA zu ermöglichen, und zwar in Kombination mit der Ausführung von UML Aktivitätsdiagrammen mittels der Ausführungsumgebung von fUML.

Die Evaluierung des Prototypen hat gezeigt, dass die Integration der Ausführungsumgebungen erfolgreich durchgeführt werden konnte. Der Prototyp ist in der Lage, eine Kombination von Zustandsautomaten, Sequenzdiagrammen und fUML-konformen Aktivitätsdiagrammen auszuführen und zu debuggen. Der verfügbare Funktionsumfang ist vergleichbar mit der proprietären Ausführungsumgebung von EA. Die Performanz-Analyse hat gezeigt, dass der Prototyp schlechtere Ausführungszeiten als die proprietäre Lösung von EA aufweist. Dies ist hauptsächlich auf die Notwendigkeit zweier paralleler Ausführungsumgebungen zurückzuführen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With the application of Model-Driven Development (MDD) [23], creating and using models is turning into a central part of the software development process. Models are used throughout the whole lifecycle. The main goal when using MDD is generating source code directly from the models themselves. Furthermore, MDD can be utilized within the entire software development process. The first step of this approach is utilizing models to specify the requirements of the system. These requirement models provide a very coarse overview of the system, which is then subsequently refined until it represents the whole architecture. The architectural models, in turn, provide the basis for the actual implementation of the system by generating the source code. Additionally, the models can be used for documenting the deployment artifacts and for documenting issues in production systems, further completing the application lifecycle. Because of this notion of using models as the core building block of the whole system, MDD also provides multiple techniques to validate the models in question, e.g., techniques for automatically executing behavioral models to analyze the behavior they describe.

The predominant modeling language in the field of MDD is the Unified Modeling Language (UML) [21]. It offers a large number of different modeling concepts and diagram types that cover a wide variety of different applications. However, herein also lies one of its main problems. Because of its general purpose nature, most modeling concepts are intentionally defined vague which, however, leaves room for ambiguities and misinterpretations. To counteract this, the Object Management Group (OMG) has released the standard called *Foundational Subset For Executable UML Models (fUML)* [22]. This standard deals with a subset of UML consisting of a subset of modeling concepts for defining activity diagrams and class diagrams. For these modeling concepts, the fUML standard defines a clear semantic meaning. This removes the ambiguities in the semantics of the modeling concepts included in fUML. The main purpose behind the fUML standard is to make UML models machine interpretable and executable. This enables additional use cases for the models, such as analysis via debugging, testing, and dynamic analysis methods. All of these use cases can be supported via the visualization of the execution of the model, for example by displaying object flows in UML activity diagrams via the visualization of movements of tokens flowing through the diagrams. The fUML standard defines the semantics of its subset of modeling concepts by defining a completely functional virtual machine, which is able to execute UML models that adhere to the standard. This fUML virtual machine applies a strictly interpreter-based approach. This means that the virtual machine defines all rules and processes for the execution of models. The models are loaded into this virtual machine and transformed into a runtime representation, which is then manipulated accordingly. The fUML virtual machine consists of an execution environment, which fulfills the actual execution of the model, and a so-called locus, which acts as a storage for the objects that are created and manipulated [22]. In the strictly interpretative approach, the models are the source code. This stands in contrast to another widely-used approach for executing models, which depends on generating and running additional source code to perform the model execution.

The aforementioned ambiguities in the UML standard have lead to the development of a plethora of different interpretations of UML models both in academia and industry, hindering the interoperability of UML tools supporting the execution of models. The formally defined semantics of fUML may aid in establishing interoperability of UML execution tools and eliminating the risk of non-conformance and vendor lock-in.

1.2 Problem Statement

Multiple open source and commercial UML execution tools support MDD. These tools provide model execution capabilities to analyse behavioral models. The fUML standard does not have a significant market penetration yet. UML execution tools mostly rely on proprietary and largely incompatible model execution environments, leading to migration and compatibility problems. Currently, only the UML modeling tool MagicDraw supports the fUML compliant execution of models via the Cameo Simulation Toolkit[1].

The main goal of this thesis is to contribute to the adoption of fUML by integrating the fUML virtual machine into an existing model execution tool and by identifying the challenges that arise from this integration.

This master's thesis focuses on the commercial UML tool Enterprise Architect (EA)[2], and the execution environment provided by the plugin AMUSE[3], which is a proprietary UML execution tool based on code generation. The goal of this thesis was to modify the execution environment of AMUSE such that fUML standard-compliant models are executed using the virtual machine specified in the fUML standard instead of using its own, proprietary engine. In doing so, the following challenges have been addressed in this master's thesis:

Integration with Existing Execution Environment

AMUSE uses a proprietary execution environment not only for simulating activity diagrams, but

[1]`http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html`, Accessed: 6.11.2014
[2]`http://www.sparxsystems.com/products/ea/index.html`, Accessed: 27.1.2015
[3]`http://www.lieberlieber.com/amuse/`, Accessed: 16.08.2014

also for state machines and sequence diagrams. Additionally, it provides mechanisms to actually realize the connections that can exist between the different diagram types. For example, a state defined in a state machine might have an attached activity diagram to specify its behavior. For these connections, AMUSE is able to provide a cohesive flow of execution through these diagrams. This means that the execution itself can be coherently visualized by dynamically switching between diagrams according to the execution progress. This causes one of the challenges in integrating fUML with the AMUSE execution environment. As already mentioned, fUML only covers activity diagrams, but neither sequence diagrams nor state machines, and thus also does not provide support for linking these different kinds of diagrams. Nevertheless, the intended prototype shall support the interaction of activities executed using the fUML virtual machine and state machines as well as sequence diagrams executed using the proprietary execution environment of AMUSE.

Model Interpretation vs. Code Generation

Model interpretation and code generation are different approaches for executing models. AMUSE uses a hybrid of interpreting a model and generating code for the actual execution, while fUML is a strictly interpretative approach.
Therefore, the code generated by AMUSE must be adapted to enable the interaction with the virtual machine of fUML when executing an activity diagram connected with other UML diagram types. This master's thesis also surveys the available literature on the topic of combining model interpretation and code generation, and investigates the challenges that result from the combination of these approaches. A mechanism for achieving this combination is implemented in the developed prototype.

Handling Dynamically Created Objects

One additional side-effect of the discrepancy between interpretative and code generation approaches is the different way in which objects are handled. AMUSE requires that the objects involved in the model execution are known before the execution starts, while fUML creates them on-the-fly, with no prior indication whether they will be created at all, or when that would be. Furthermore, the respective execution approaches use different internal representations of the created and manipulated objects. On the one hand, AMUSE generates code based on the class definitions contained in the UML model and uses it to directly instantiate the respective classes. The interpreter approach of fUML, on the other hand, provides no possibilities for creating actual instances of the defined classes. Instead, it stores class information and objects in a custom representation format. For example, class information (name, number and type of attributes, etc.) is stored in instances of the type *Class* of the API of the fUML virtual machine. Similarly, instances of type *Object*, which are also provided by this API, are used as representations of class instances. Nevertheless, the execution environment of AMUSE needs to be able to refer to these dynamically created objects and process the object representation of fUML. For instance, a state machine might trigger an activity diagram to create objects, and thereafter has to check a guard condition that refers to an attribute of one of these objects. It was also a goal of this master's thesis to elaborate a mechanism for handling differences in the creation and representation of objects between distinct execution environments.

fUML Interpreter

A valid implementation of the virtual machine specified in the fUML standard needs to be incorporated into the prototype and made fully accessible to AMUSE. As the specification of the fUML virtual machine is based on Java code and AMUSE is developed in C#, either the fUML virtual machine code needs to be converted or the language barrier needs to be bridged in another way. Implementing a full virtual machine in C# is not in the scope of this master's thesis.

1.3 Aim of the Work

The aim of this master's thesis is to contribute to the adoption of fUML through integrating fUML into a commercial UML modeling tool and exploring the challenges arising in the integration. Therefore a prototypical integration of the fUML reference implementation[4] with EA has been developed. In doing so, this master's thesis has aimed at answering the following research questions:

1. Can fUML be integrated into a proprietary UML model execution environment? Which challenges arise in the integration?
2. Does the standardized fUML model execution environment provide the same functionality as a proprietary UML model execution environment?
3. Is the performance of the standardized fUML model execution environment comparable to the performance of a proprietary UML model execution environment?

The prototype shall provide the functionality of executing and debugging state machines and sequence diagrams via EA's execution environment, in conjunction with executing activity diagrams via the fUML execution environment. The visualization and control of the resulting combined execution is handled by EA and the EA model execution plugin AMUSE. The AMUSE plugin is adapted to use the fUML reference implementation instead of its proprietary model execution environment for activity diagrams, while leaving the remaining functionality of AMUSE unchanged.
The integration of the fUML reference implementation represents a clear break in the homogeneous implementation of AMUSE's execution environment. As has been described in the previous section, both execution environments follow different approaches for executing UML models. Therefore, the prototype also includes functionalities to smoothly integrate both of these execution environments.

[4]`http://portal.modeldriven.org/project/foundationalUML`, Accessed: 27.1.2015

1.4 Methodological Approach

This master's thesis has been carried out in three steps:

1. **Literature Study**. In this step, the necessary knowledge for carrying out the subsequent steps was acquired, by exploring existing approaches on the following topics:
   - Integration of different UML diagram types
   - Combination of code generation and interpreter-based model execution approaches
   - Utilizations of the fUML standard (especially in commercial tools)
   - Approaches for conducting performance evaluations of model execution environments
2. **Conceptualisation and Realisation**. This step consisted of developing a prototype capable of executing fUML compliant activity diagrams with the fUML execution environment, and utilizing the AMUSE framework for the execution of state machines and sequence diagrams. Furthermore, the prototype was to enable the interaction between these two execution environments to provide a cohesive flow of execution between activity diagrams, state machines and sequence diagrams. A substantial part of the development of the prototype has dealt with adapting the existing functionalities of AMUSE (e.g., the user interface), which has been expanded and modified whenever the need arose.
3. **Evaluation**. In this step, a comparison of both features and performance provided by the prototype developed in Step 2 and those of the proprietary execution environment provided by AMUSE has been conducted. This comparison has served as a basis for evaluating the success of the integration of the execution environments, as well as the general applicability of the approach proposed by this master's thesis.

1.5 Structure of the Work

The remainder of this thesis is structured as described in the following.

In Chapter 2, we provide information on both the theoretical background, as well as the tools utilized in this master's thesis. In particular, we take a look at the fundamentals of both UML and fUML, as well as the functionality of the EA plugin AMUSE and the MOLIZ project.

In Chapter 3, we explore the current state-of-the-art related to this thesis. The focus lies on the topics of UML model execution, integration of UML diagram types and the formalization of UML.

In Chapter 4, we describes the developed prototype, starting with the overall goal of the implementation and then going into detail on some technical intricacies, steps taken to integrate the execution environments, and the tools used for developing the prototype.

In Chapter 5, we discuss both the approach, as well as the results of the evaluation of the prototype. The evaluation is done regarding functionality, performance, usability and stability.

Finally, in Chapter 6, we provide a conclusion of the thesis, as well as an outlook on possible future work.

CHAPTER 2

Background

This chapters discusses the necessary background information for understanding both the context and the functionality of the approach proposed by this thesis.

In Section 2.1, we provide an overview for the modeling language UML. Special focus is placed on the different behavioral diagram types within UML, as well as interactions between these behavioral diagram types. In Section 2.2, we discuss the two model execution technologies used by the prototype developed for this thesis, namely model interpretation and code generation. Furthermore, an introduction to both the fUML standard, as well as the model execution environment defined by it, is provided. Lastly, in Section 2.3 we present the software tools used by the prototype developed for this thesis. The main focus lies on EA and the fUML reference implementation.

2.1 Modeling with UML

This section introduces the basic principles of models and the modeling language UML. Furthermore, it takes an extensive look at the behavioral diagram types defined within UML, and at the interactions between the behavioral diagram types. The latter is important to fully understand the interaction of the execution environments of fUML and EA.

2.1.1 Introduction to UML

A model is an abstraction of an existing entity. Specifically, a model needs to fulfill three specific criteria [28]:

- The model applies to an entity.
- The model is a minimal representation of the entity that includes only the attributes relevant to the model.
- The model serves a specific purpose.

Naturally, with a purpose as broadly defined as that of the discipline of modeling, there exist a myriad of different approaches and methodologies which are utilized all over the world. Their variety, as well as their incompatibility to one another, can complicate a broad spectrum of tasks, such as comparing different models or switching between approaches.

The Unified Modeling Language (UML) is a graphical modeling language developed by the OMG [25]. UML is currently in version 2.4.1[1], and provides a number of different modeling concepts applicable to a variety of different domains and scenarios.

One of the main goals of UML is to unify all these different modeling approaches, thereby increasing their compatibility. The unifying character of UML can be seen on various aspects of the language itself. Firstly, the history of UML shows that it has been created in an effort to consolidate various existing modeling languages [25]. The consolidation also led to the integration of many proven strategies and best-practices into the resulting UML specification.

UML aims to be a general-purpose language that can be applied in a variety of different scenarios and situations, irrespective of their specific nature. Specifically, UML is independent of the following factors [25, p. 11]:

- Application domain
- Used programming languages and platforms
- Development process
- Internal concepts

As a result of the general-purpose nature of UML, the definitions contained within UML need to be kept intentionally vague. However, this vagueness results in a lack of clear specifications of the semantics of the modeling concepts defined within UML, which is especially problematic in the context of model execution [25, p. 11].

UML itself is defined by four parts [11]:

- The **Infrastructure**, which defines foundational language constructs[2].
- The **Superstructure**, which defines user level constructs[3].
- The **Object Contraint Language (OCL)**, which is a language for defining expressions on UML models[4].
- The **Diagram Definition**, which specifies a basis for modeling and interchanging graphical notations[5].

The following sections give a brief overview of the core UML modeling concepts, which are relevant in the context of this thesis. For a more detailed look into the inner workings, please refer to the UML standard itself [21].

[1]`http://www.omg.org/spec/UML/2.4.1/`, Accessed: 10.11.2014
[2]`http://www.omg.org/spec/UML/2.4.1/`, Accessed: 10.11.2014
[3]`http://www.omg.org/spec/UML/2.4.1/`, Accessed: 10.11.2014
[4]`http://www.omg.org/spec/OCL/`, Accessed: 10.11.2014
[5]`http://www.omg.org/spec/DD/1.0/`, Accessed: 10.11.2014

2.1.2 Overview of UML Diagram Types

Figure 2.1 illustrates the different UML diagram types, which correspond to different types of models that can be created. The figure provides an overview of all available diagram types—not just the ones that are in the focus of this thesis. All diagram types that are explicitly relevant to this thesis are discussed in detail in the following sub-sections.

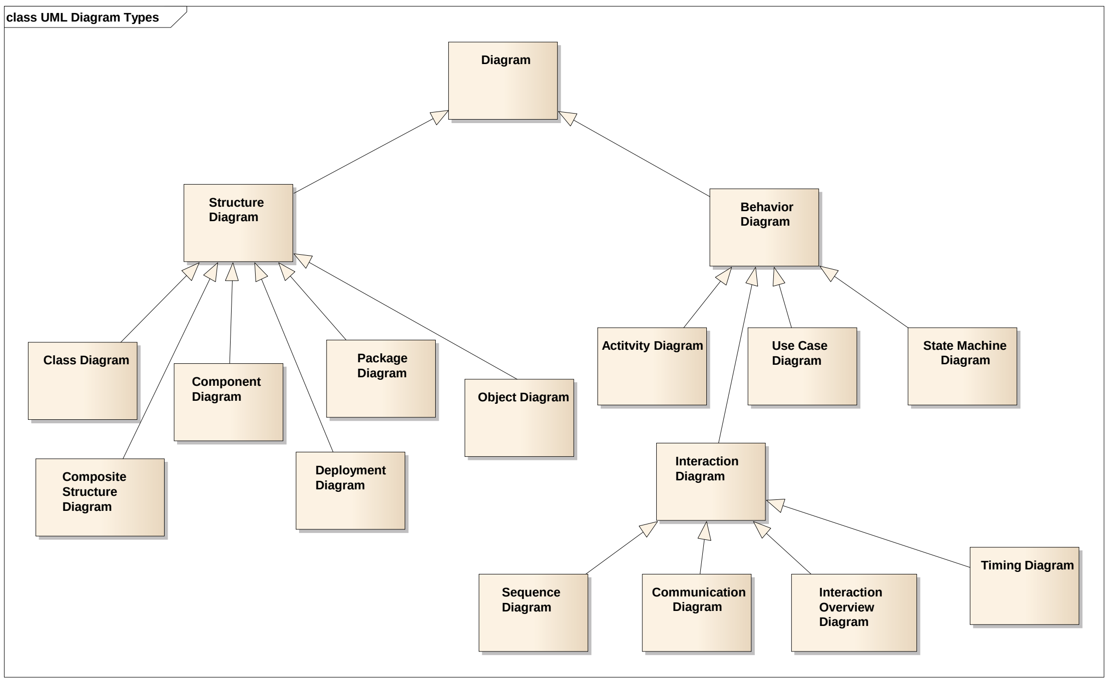


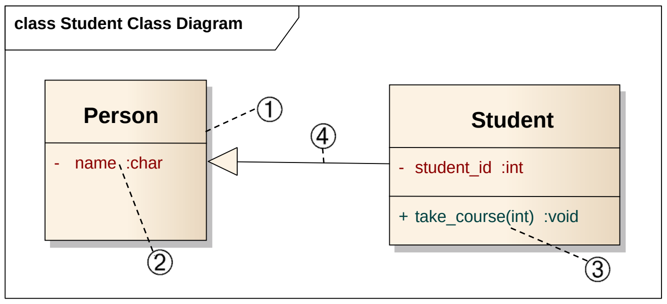
Figure 2.1: UML diagram types

There are two main types of diagrams: *structural* diagrams, which describe the static properties of the modelled object, and *behavioral* diagrams, which describe its dynamic parts. In the following, we highlight the properties of the diagram types which are relevant to this thesis, namely class diagrams, activity diagrams, sequence diagrams and state machine diagrams.

2.1.3 Class Diagram

Classes are templates for creating objects with specific characteristics. A class diagram describes a set of classes, as well as their relations. An example is shown in Figure 2.2.

The structural aspects of a class are specified by properties. A property defines a set of values, which may be directly owned by the instance of the defining class. Thereby, values may be instances of classes or primitive values. The class *Person* depicted in Figure 2.2 has a property *name* of type *char*.



Legend:
1.) Class 2.) Property 3.) Operation 4.) Generalization

Figure 2.2: UML class diagram

The behavioral aspects of the class are defined by operations, which specify name, type and parameters for invoking an associated behavior. The class *Student* depicted in Figure 2.2 has an operation *take_course*, which takes a parameter of type *int*.

The class diagram has multiple mechanisms for defining relationships between classes. In addition to using associations for defining horizontal hierarchies of classes, the class diagram also supports the specification of generalizations to define the inheritance of attributes and operations from a more general to a more specific class. Figure 2.2 depicts such a generalization relationship between the classes *Person* and *Student*, resulting in the inheritance of the *name* attribute by the class *Student*.

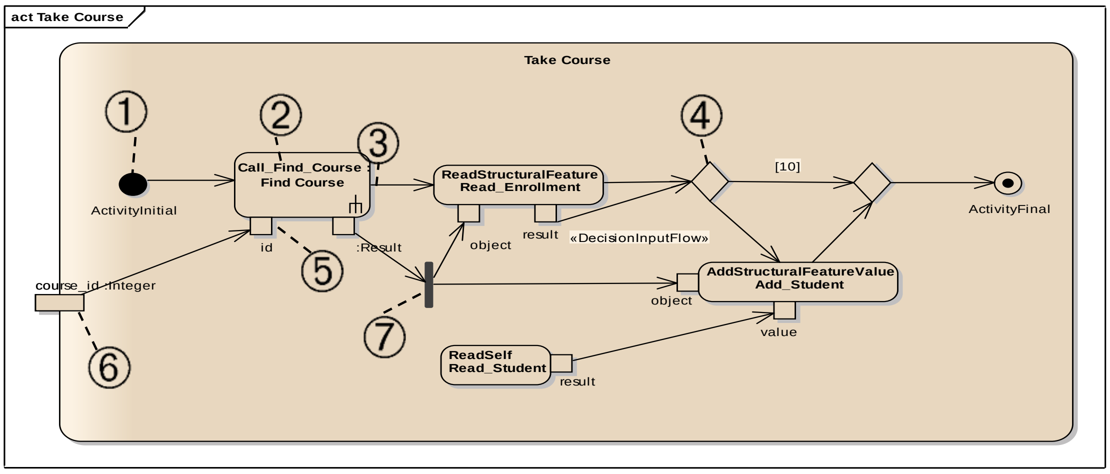
2.1.4 Activity Diagram

An activity diagram shows the behavior of a single activity, which is a model of the behavior of an object. This model consists of atomic actions that are executed in a pre-defined order. An example is shown in Figure 2.3.

Action

The core element defining an activity is the action, which is an atomic operation within the activity. Because of this broad definition of scope, there exists an equally broad range of interpretations of actions, which largely depend on the context. If the model is used to define, for instance processes in an easily human-readable form, the actions often contain plain text that describes the expected effects.

If the model is used in the context of execution, however, the actions need to contain additional information. Therefore, the UML standard defines a set of specific action types, forming a computationally complete action language. These actions have more clearly-defined semantics, which are expressed by additional properties and associations. The preciseness by which these actions are defined in the standard makes activity diagrams machine-interpretable, which is the reason why the fUML standard is largely based on them.



Legend:
1.) Initial node 2.) Action 3.) Control flow 4.) Decision node
5.) Pin 6.) Parameter 7.) Fork node

Figure 2.3: UML activity diagram

The example shown in Figure 2.3 represents such a machine-interpretable activity. and demonstrates a number of commonly-used action types:

- It calls a separate behavior named “Find_Course” via a call behavior action.
- A read structural feature action “Read_Enrollment” is utilized to determine the number of enrolled students of the course supplied by the “Find_Course” behavior.
- A read self action “Read_Student” is used to determine the student object that is performing the activity.
- An add structural feature value action “Add_Student” is used to add the student to the list of students enrolled in the course.

For passing input and output values, the actions define pins. In the example, the call behavior action “Find_Course” has an input pin to receive the course id, as well as an output pin to provide the result of the course lookup.

Control Nodes

An activity diagram can contain a number of different control nodes to allow for a more fine-grained control of the flow of execution. The example in Figure 2.3 contains the following control nodes:

- An initial node “ActivityInitial” to denote the starting point of the execution of the activity.

- A decision node to branch the further path of execution based upon the number of enrolled students determined by the read structural feature action “Read_Enrollment”.
- A fork node to pass on the course object to multiple recipients that require this object as input.
- A final node “ActivityFinal” to denote the end point of the execution of the activity.

Activity Edges

Actions and control nodes are connected by activity edges, which represent a specific relation between the connected elements. The actual semantics of the activity edge depend on its subtype, of which there are control flow and object flow. The visual reprsentation gives no clear indication as to what type of activity edge is represented—this information can only be inferred from the specific elements connected by the activity edge.

The control flow represents the flow of control. Each control flow connects either two actions or an action and a control node. The flow of control originates at the initial node, and usually consists of a path leading to one (or more) final nodes. There are multiple possibilities for introducing decisions and parallelisms into the flow of control, which are discussed in the section on control nodes above.

The object flow represents the flow of information within the activity. It connects either two pins of two respective actions, or a pin and a control node.

A vital component of both control flows and object flows is the concept of guard conditions. These conditions specify specific criteria, which have to be met in order for tokens to be passed along the control or object flow. In the provided example, a guard condition is utilized for the decision made by the decision node. The path with guard condition “10” will only be taken if ten students are already enrolled in the course.

Execution Semantics

The execution semantics of activity diagrams are based on token-flow semantics applied to a directed graph, consisting of vertices (actions and control nodes) and edges (the activity edges). The activity diagram defines two distinct graphs—one for the flow of control, the other for the flow of information.

The flow of control is a representation of the order of execution in the activity through control flow edges. It originates at the initial node of the activity, where a single control token is generated at the start of the execution. This token is then passed on to each following action, which keeps the token while performing its designated operations, and passes it onwards once it has finished, thereby enabling the execution of the next action. Finally, the final node consumes control tokens, ending the execution of the activity.

It has to be noted that the presence of fork nodes in the flow of control causes the creation and handling of multiple control tokens in parallel. Furthermore, actions without an incoming flow of control can fire instantly once all of their required input parameters are provided.

The main flow of execution within the activity is defined by the object flow. It passes along object tokens, which contain data that is processed or produced by actions and control nodes.

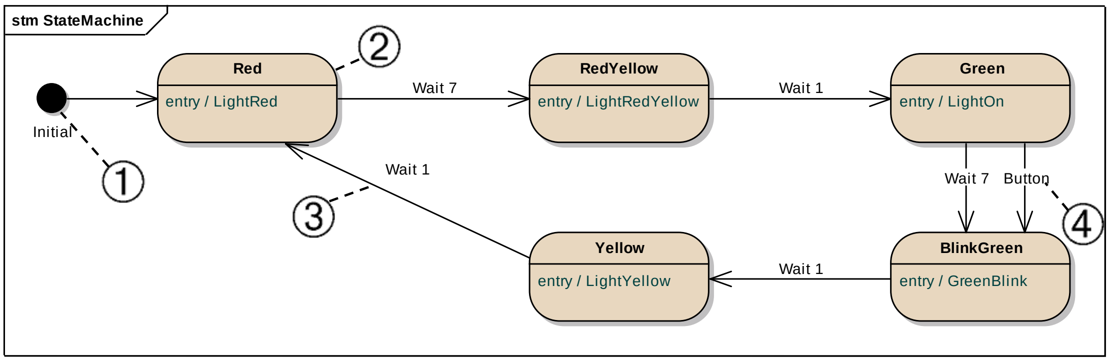
The data stored by these object tokens can range from primitive values to complex data structures. The flow of control serves as a support structure for the flow of information within the activity.

If we apply these execution semantics to the example provided in Figure 2.3, this would result in the following execution steps:

1. The course id is provided to the call behavior action "Find_Course".
2. The read self action "Read_Student" is executed, providing a reference to the context object (i.e., the *Student* object excuting the activity) to the add structural feature value action "Add_Student" once the flow of control reaches it.
3. The initial node creates a token of control and passes it to the call behavior action "Find_Course".
4. The call behavior action "Find_Course" is executed and passes the obtained *Course* object to the fork node. Furthermore, a control token is passed to the read structural feature action "Read_Enrollment".
5. The fork node produces two identical copies of the *Course* object received from the call behavior action "Find_Course". One of these objects is passed on to the read structural feature action "Read_Enrollment", the other is passed on to the add structural feature value action "Add_Student".
6. The read structural feature action is executed, taking the result of the call behavior action as input. The read structural feature action "Read_Enrollment" reads the number of enrolled students from the *Course* object and transfers it to the decision node as a decision input value. Additionally, the read structural feature action "Read_Enrollment" provides a control token to the decision node.
7. Depending on the value returned by the read structural feature action "Read_Enrollment", the decision node either passes a control token directly to the merge node or to the add structural feature value action "Add_Student".
8. Should the add structural feature value action "Add_Student" be executed, it adds the *Student* object to the list of *Students* enrolled in the *Course*. Afterwards, the add structural feature value action sends a control token to the merge node.
9. As soon as the merge node receives a control token, it passes it on to the final node "ActivityFinal".
10. The execution of the activity is finished by the execution of the final node "ActivityFinal".

2.1.5 State Machine

A state machine models the behavior of an object via a system of discrete states and transitions. The object switches between the states via the transitions when the conditions for the respective transition are met. The process of transitioning between the states serves as the description of the behavior of the object itself.



Legend:
1.) Initial state 2.) State 3.) Transition 4.) Guard

Figure 2.4: UML state machines

Figure 2.4 illustrates the functionality of a state machine based on a model of a simple traffic light with a button for pedestrians to signal that they want to cross the road.

State

Each state contained by a state machine represents a situation in which the object fulfills a predefined condition. In the example shown in Figure 2.4, each state corresponds to a distinct combination of lights shown on the traffic light. As with the definition of the activity diagrams discussed before, the behavior of a state can be described in a number of ways, reaching from prose to machine-interpretable commands. For the purpose of model execution, the behavior of a state may be defined via separate behaviors, such as activities. By dividing a state within a state machine up into orthogonal regions, it is given the possibility to execute multiple sub state machines, while still residing in the same parent state. This allows for a hierarchical decomposition of the state machine, which enables to detail a complex state into a set of sub states. Another distinct advantage of using regions is the introduction of parallelism, as each orthogonal region is executed in parallel and can independently react to events.

Transition

A transition connects either two states, or a state and a region. It denotes both the possibility of switching between the two connected elements, as well as the conditions that must be fulfilled for such a switch to occur.

The core information contained in the definition of a transition is the trigger which will cause the transition to occur. This trigger is a description of an event that is observed by the state machine, and which will cause the corresponding transition to fire. The example in Figure 2.4 shows three possible mechanisms for triggering a transition. Firstly, there is a transition from the initial state to the "Red" state without any condition. Secondly, all of the distinct color states are connected with transitions with time triggers. Lastly, the transition marked with the number 4 has the condition "Button", which refers to an event caused by pressing the pedestrian button connected to the traffic light.

Execution Semantics

The core mechanism by which the state machine operates is the processing of events. By processing events, the state machine advances from one state to the next until it reaches the final state, upon which the execution is completed.

One of the important properties of a state machine is that it is always in a consistent state during the execution. This is ensured by utilizing the mechanism of run-to-completion processing. This mechanism consists of executing sequential run-to-completion steps, with each execution of such a step only starting once the previous one has completely finished. The run-to-completion step itself is defined as the complete transition from one state to the next.

Additionally, the execution semantics of a state machine ensures that only a single event is processed at any given time. This accomplishes two major goals:

- The execution does not have to handle concurrency during the execution of a state machine.
- The state machine is always in a consistent state, as it is simply unresponsive to queries during the processing of an event.

Aside from guaranteeing that the state machine is always in a consistent state, the usage of this mechanism also reduces the complexity of handling the execution of state machines considerably.

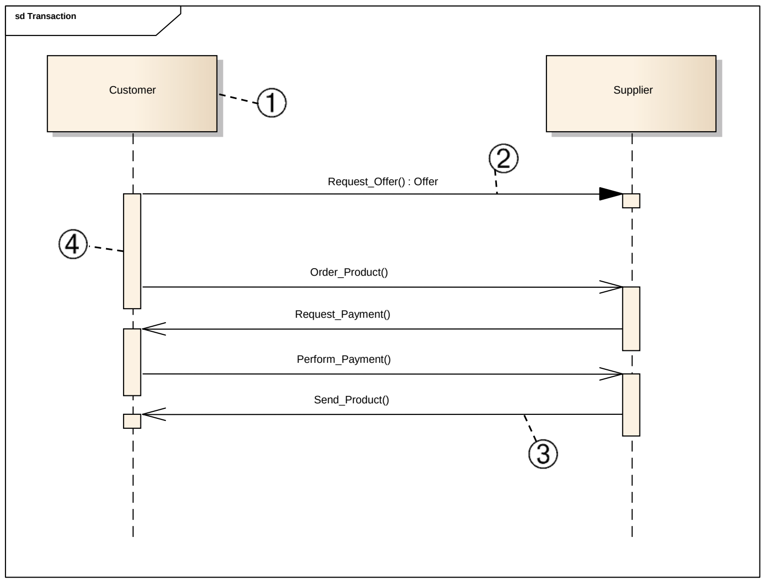
2.1.6 Sequence Diagram

Sequence diagrams are used to model interactions between objects. The interactions are represented by messages, which are exchanged by objects represented by lifelines.

The example modeled in Figure 2.5 represents a transaction between a customer and a supplier, consisting of acquiring an offer from the supplier and then performing a purchase.

Lifeline

The lifeline is the quintessential element of the sequence diagram. Each lifeline represents one of the objects participating in an interaction. In the case of the example shown in Figure 2.5, an object of type *Customer* and one of type *Supplier* are represented. The lifeline serves as a connection point for both incoming and outgoing messages. Furthermore, it represents the



Legend:
1.) Lifeline 2.) Synchronous message 3.) Asynchronous message 4.) Execution occurrence

Figure 2.5: UML sequence diagram

chronological order in which the messages shall be exchanged. In the visualization used in Figure 2.5, the chronological order is represented by the position of the message along the line component of the lifeline, with a position further down the line representing a message that is sent after one positioned further up. Normally, the sequence diagram only represents the order in which the messages are exchanged. However, the sequence diagram also supports message properties for more precisely defining time constraints.

Message

Each message represents a particular communication between the connected objects. There are two main types of message exchange mechanisms:

- **Synchronous** communication, in which the sender waits for the receiver's answer. The message "Request_Offer" in Figure 2.5 is sent synchronously, indicating that the offer is provided immediately by the supplier (e.g., because the process is conducted via an e-commerce portal).
- **Asynchronous** communication, in which the sender resumes his execution immediately after sending the message.

Messages can be used for directly invoking operations provided by the receiver. Such operation-based messages are used in Figure 2.5, with each exchanged message corresponding to an operation invocation. Furthermore, they can be used for exchanging signals, as well as creating and destroying objects. In the context of this thesis, only the message types for transmitting operation calls and signals will be considered, while message types dealing with the creation and deletion of objects will be omitted.

Execution Semantics

The execution of a sequence diagram consists of the exchange of messages between the objects denoted by the respective lifeline. As has been noted in the paragraph on lifelines above, they represent the chronological order in which the messages are sent, and therefore imply the order of execution of the sequence diagram. Additionally, the execution occurrences, which are denoted as thicker sections along the lifelines, indicate which of the objects currently is the active element in the context of the execution, and is therefore currently able to send messages.

2.1.7 Interactions Between UML Behaviors

As has been noted in the previous chapter, the fUML standard does not define the execution semantics of interactions between different behavioral diagrams because it only considers UML activity diagrams. To counteract this problem, different sources have been consulted on possible interaction mechanisms between different UML behavioral diagram types. An important source of information used for this purpose was the UML standard itself. This section goes into detail on the various mechanisms specified in the UML standard, which deal with interactions between different types of behaviors corresponding to the different types of behavioral diagrams.

Behavior

Before going into details on the specific types of behaviors defined in the UML standard, it is necessary to take a closer look into the parent metaclass of all types of behavior detailed below, which is the metaclass *Behavior*. Figure 2.6 shows an excerpt of the UML metamodel, which includes both the meta class *Behavior*, as well as its accociations to other metaclasses defining possible interactions between the defined modeling concepts. All of these interactions will be discussed in detail in the following sub-sections.

As shown in Figure 2.6, the metaclass *Behavior* does not define any particular forms of interactions between behavioral diagram types by itself. Indeed, the only universal property of all types of behaviors is the definition of a context object of type *BehavioredClassifier*, which is explained in the following section. All other interaction capabilities need to be introduced by the associations of the subclasses of the metaclass *Behavior*.

BehavioredClassifier

The metaclass *BehavioredClassifier* serves as a base class for various types of classifiers of objects that are usable within UML. The most prominent sub-class of *BehavioredClassifier* is *Class*.

A single behavior connected to the behaviored classifier is declared as the classifier behavior, i.e., the behavior which describes the behavior of the classifier itself. The actual behavior of this classifier is tied to the classifier itself, and to the respective objects. In the moment the classifier

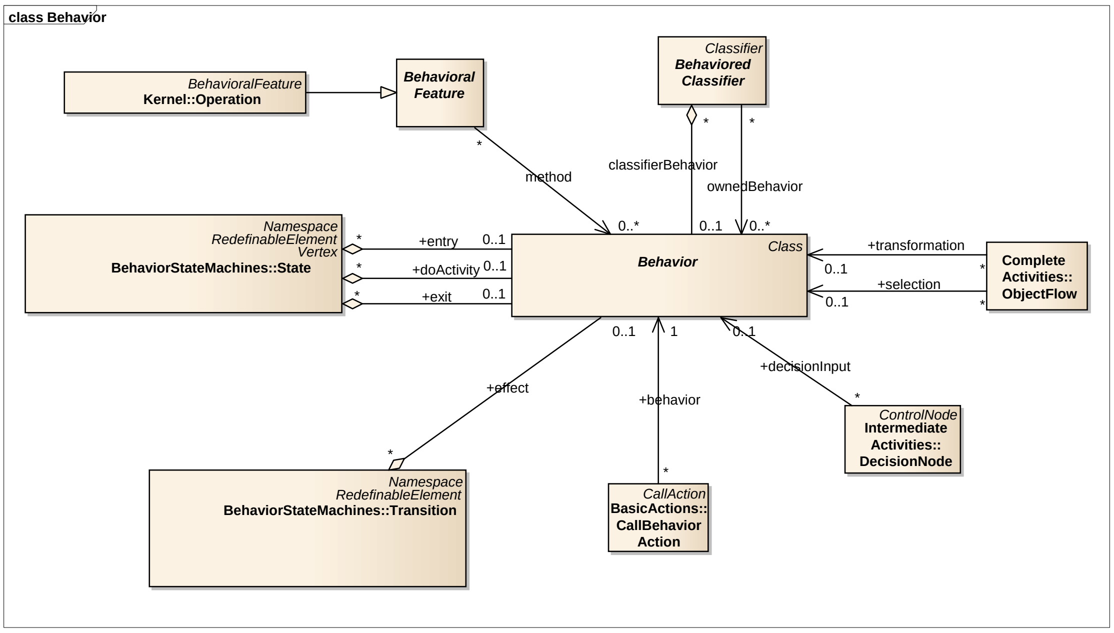


Figure 2.6: UML metaclass *Behavior*

is instantiated, the classifier behavior begins execution and controls the entire behavior of the instance, as well as its interactions with other objects. As such, the classifier behavior also represents the ”lifespan” of the object itself, which is discarded as soon as the classifier behavior has completed.

Additionally, a behaviored classifier is able to declare an arbitrary number of owned behaviors, with which it can interact. The UML standard does not contain a specification for when or how these owned behaviors are executed.

Class

The *Class* is a specialization of the metaclass *BehavioredClassifier*. The specification of *Class* also does not explain how the class interacts with its owned behaviors. A possible interaction mechanism is the execution of operations to which the owned behaviors are attached. Some additional information on this approach can be gathered from the fUML standard. Therein the following restriction for owned behaviors of a behaviored classifier are defined: “An owned behavior must be either the classifier behavior of or the method for an operation of its behaviored classifier.” [22]. As has been discussed in the previous chapter, the fUML standard only deals with classes and activies, which means that this restriction also applies to classes specifically. This implies that the method of invoking the owned behaviors via operations is indeed the correct approach.

Activity

There are three ways in which activities may interact with over behaviors. Firstly, the activity may contain actions that directly invoke other behaviors. Such an invocation can be performed directly by utilizing a call behavior action, or by invoking an operation via call operation action, which is in turn connected to a behavior. Both actions support synchronous and asynchronous execution, as well as using both input and output parameters.

A different approach to invoke behaviors from within an activity is to utilize actions for creating instances of behaviored classifiers. This is accomplished via a create object action. A similar, more roundabout approach based on instances is also available. As behaviors are themselves classifier behaviors, they can also be directly instantiated and then executed via a start object behavior action. This possibility allows for additional use cases, such as modifying the behavior object before execution. Similarly, an object may passed to a start classifier behavior action which automatically executes its associated classifier behavior.

Aside from directly invoking behaviors, the activity contains a variety of different mechanisms designed to support both the flow of control and the flow of information via the invocation of additional behaviors. An example is the decision node. The decision node can be provided with a number of inputs, based on which it will decide how to pass on the flow of execution. These inputs can be processed by an additional behavior to make this decision. Similarly, object flows can be equipped with additional behaviors to both transform or discard tokens before passing them on.

State Machine

The main mechanism with which a state machine invokes other behaviors is by defining the behavior associated with its states.

Each state may contain up to three different behaviors. Firstly, an entry behavior can be defined for the state, which is executed to completion before the state is entered through an arbitrary transition. Next, the state can define a behavior that is executed while the execution resides in the state. When such a behavior is defined, the execution resides in this state until either the execution of this behavior has been completed or externally terminated. After the execution of this behavior has finished, an executable transition is automatically triggered and the state is left. Finally, a behavior to be executed on leaving the state can be defined. This behavior will also be executed to completion before continuing on with the transition. Aside from invoking behaviors directly while residing in a state, each transition also has the capability to define an effect, which can also refer to an arbitrary behavior.

When considering all of the aforementioned possibilities for connecting to other behaviors available in the state machine, we need to reconsider the run-to-completion mechanics of the state machine. The run-to-completion mechanism requires that the state machine only performs complete run-to-completion steps, with each step transitioning from one consistent state to the next. During a run-to-completion step, the state machine is unresponsive to external events. While this mechanism reduces the complexity of handling state machines considerably, one should also consider the potential pitfalls and limitations this mechanism brings with it.

Consider, for example, the execution of the model depicted in Figure 2.7.

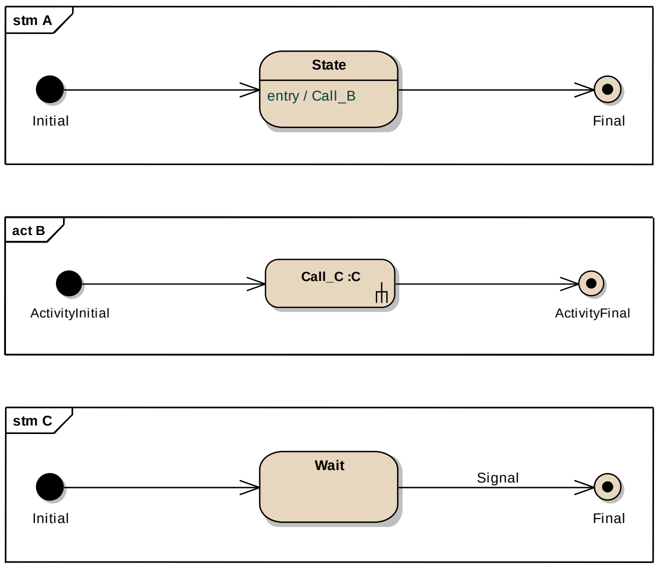


Figure 2.7: Model that depicts possible issues with the run-to-completion mechanics of state machines

Executing this model results in the following execution order:

1. State machine A contains state “State” with the entry action “Call_B”, which executes activity B.
2. Activity B synchronously invokes state machine C with the call behavior action “Call_C”.
3. State machine C waits for the occurrence of a signal to advance from state “Wait” to the final state.

The model depicted in Figure 2.7 is valid according to the UML standard. However, the run-to-completion mechanics lead to a potentially indefinitely blocking of state machine A, as it is trying to complete its entry-action. This, in turn, blocks the completion of the run-to-completion step of state machine A. These sort of interdependencies need to be considered when designing models.

Sequence

The UML standard does not specify any type of direct interaction betweem sequence diagrams and other behaviors. The only available mechanism for invoking behaviors is therefore to send a message requesting the start of an operation, which is connected to a behavior.

Invocation of Emergent Behavior

A number of diagram types within UML do not describe the behavior of specific entities. Instead, they describe *emergent behavior*, which refers to the behavior of a system that results from interactions of entities within this system. Two examples for such diagram types are interaction diagrams and use case diagrams. The execution semantics of these diagram types is not well defined by the UML standard and out of scope for this thesis.

Signals

Aside from directly invoking other behaviors, behaviors are also able to communicate with each other by utilizing signals. The UML standard defines these signals in terms of requests, which are exchanged by the instances capable of receiving signals. To further specify these requests, signals can hold additional data in the form of arbitrary attributes [21, p. 464].

Sending signals is a functionality that is reserved exclusively for activities. There are three separate types of actions for sending signals. Firstly, there is the send signal action, which generates a signal and sends it to a specified target. Secondly, the broadcast signal action also generates a signal, but transmit it to all possible recipients. Lastly, there is a variation of the broadcast signal action called the send object action. The send object action allows for sending an already existing signal object to all available recipients. Similarly to the start object behavior action discussed above, this allows for further manipulating the signal object before transferring it.

As for the execution semantics of these actions, all three transmit the signals asynchronously and immediately resume the execution. There is neither a possibility to wait for a reception notification, nor for a possible reply from the recipient(s).

Determining which types of model elements are able to respond to signals is rather difficult, as the UML standard does not contain a straightforward definition. Nevertheless, an analysis of the standard has revealed two mechanisms for receiving signals and reacting to them.

The first mechanism in place for handling signals relies on the concept of receptions. A reception indicates that a classifier can react to a signal [21, p. 463]. The snippet of the UML metamodel that defines the metaclass *Reception* is shown in Figure 2.8. The figure displays two pieces of key information: the metaclass *Reception* is a specialization of the metaclass *BehavioralFeature*, and its instances are owned by classes. The metaclass *BehavioralFeature*, in turn, is the base class for the metaclass *Operation*. Operations can be utilized by classes to execute arbitrary behaviors as discussed previously in the analysis of the execution capabilities of classes.

The second available mechanism for handling signals are triggers. A trigger specifies an event that leads to the execution of a behavior on a classifier instance. The UML standard defines three elements which incorporate triggers. They are listed and explained in Table 2.1.

The snippet of the UML metamodel that defines the metaclass *Trigger* is shown in Figure 2.9. The figure depicts the relation between the trigger and the event. The connection between the event and handling signals is defined by the metaclass *SignalEvent*, which defines an event that is caused by receiving a signal. As such, it can be argued that the trigger is able to respond

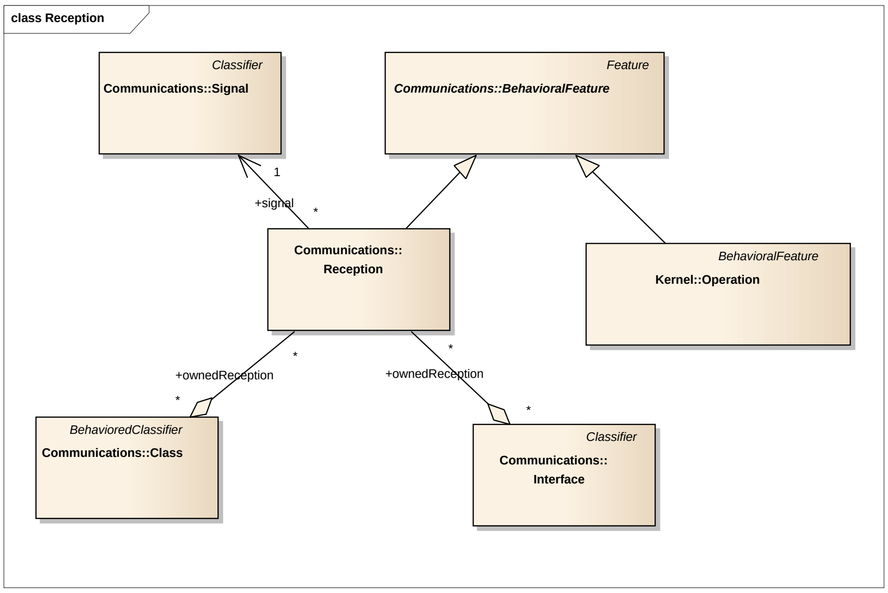


Figure 2.8: UML metaclass *Reception*

Control Node Sub-Type	**Description**
Accept event action	An accept event action defines a set of triggers to indicate which types of events it accepts.
Transition	Transitions within a state machine may define sets of triggers which cause the transition to fire.
State	A state holds a list of deferrable triggers. If events arrive for these triggers and are not immediately consumed by a transition, the state holds these events until they can be consumed.

Table 2.1: Elements which incorporate triggers

to signals by virtue of the fact that it is able to respond to any type of event. This functionality is not explicitly stated in the standard, however.

The functionality of triggers is especially relevant to this thesis. Even though the exact interpretation of the functionality is unclear, it is implemented in both EA and the AMUSE plugin. Thus, it is also implemented in the prototype integrating fUML with AMUSE.

In general, the execution semantics behind sending and receiving signals are not well-defined. For example, the following points of consideration are not explicitly defined [21]:

- Handling a signal with multiple possible recipients
- Handling a signal with no possible recipients
- Time intervals between event occurrences

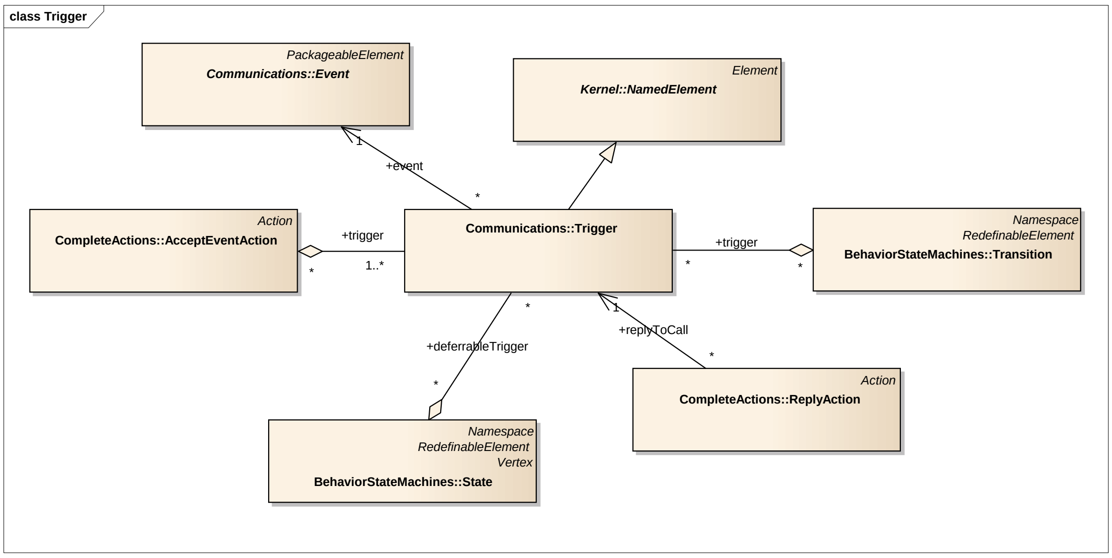


Figure 2.9: UML metaclass *Trigger*

2.2 Model Execution

This section explores various different mechanism for executing models in general and UML models in particular. Therefore, two of the main paradigms in model execution, interpreter-based execution and execution via code generation, are explored. This section only discusses pure implementations of both model interpretation and code generation. A more thorough discussion of the differences between model interpretation and code generation, as well as approaches of integrating and combining the respective approaches, is provided in Section 3.3.

In addition to to the basic approaches of model interpretation and code generation, we describe the the fUML standard in detail. Both the syntax employed by fUML, as well as the execution model and the execution semantics of fUML are discussed.

2.2.1 Model Interpretation

Model interpretation is an approach that utilizes a virtual machine to directly execute a given model[6]. A virtual machine is an abstract computing machine which holds objects in a designated memory space and performs a set of instructions on these objects. In the case of model execution, the objects held by the virtual machine are instances of modeling concepts, and the intructions manipulate these instances to simulate their behavior [24]. The complexity of the set of instructions provided by a virtual machine depends on the functionality. In the case of model execution, a complete virtual machine has to be able to interpret any type of model that is valid and executable according to the modeling language.

[6]`http://modeling-languages.com/executable-models-vs-codegeneration-vs-model-interpretation-2/`, Accessed: 16.08.2014

The technology for implementing such a virtual machine is in principle not bounded by any specific constraints. Any type of system that is able to apply the instructions for model execution is applicable. This form of model execution is applied by the fUML virtual machine.

Model and Object Instantiation

One of the main challenges in integrating the execution environments of fUML and AMUSE is the integration of their respective object representations. Therefore, the implementation of these object representations in both of the model execution paradigms will be discussed in more detail.

The object space of the virtual machine must be constructed such that it can hold all instances that are valid according to the model specification. Therefore, the data structure that holds these objects need to be suitably generic.

To illustrate a possible implementation of such an object space, we will consider the object space used by the fUML execution environment. Figure 2.10 depicts the classes used for maintaining objects within the fUML execution environment.

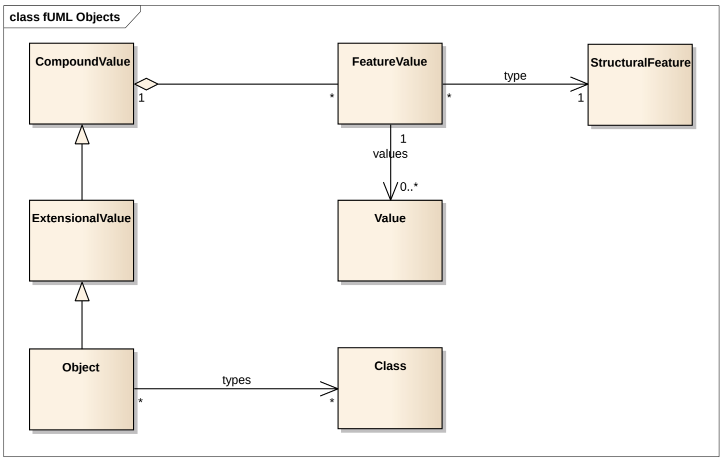


Figure 2.10: Class structures used by fUML objects

These classes fulfill the following roles:

- All objects are represented by instances of the type *Object*, which is a generic representation of an instance of a classifier.
- Each instance of type *Object* references an instance of type *Class* to represent the classifier it instantiates.
- As the class *Object* is an extension of the class *CompoundValue*, objects can hold a number of different instances of type *FeatureValue* to represent the properties of the instantiated classifier. A *FeatureValue* references a specific instance of type *StructuralFeature* that specifies the property itself, as well as a number of objects of type *Value* to denote the values.

To further illustrate the usage of the interpretative approach, we will reconsider the class diagram example introduced in Section 2.1 and depicted in Figure 2.2. Specifically, we will discuss instances of the class *Student*. This class holds two properties, namely the property *name* of type *char* and the property *student_id* of type *int*. Figure 2.11 depicts the objects maintained by the fUML execution environment to represent an instance of the class *Student*.

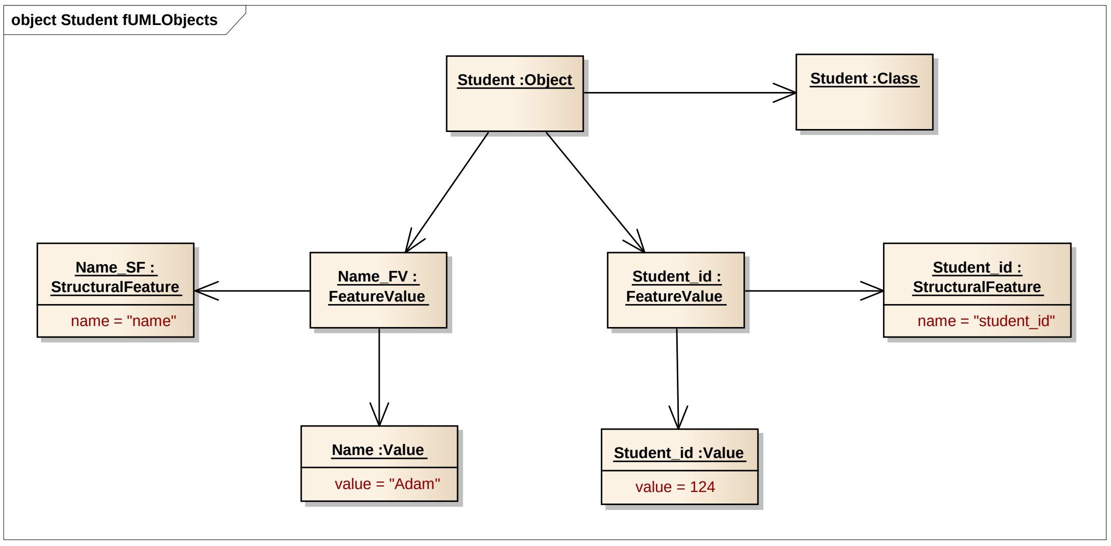


Figure 2.11: Class structures used by fUML objects

In particular, this involves the following objects:

- An object of type *Class* representing the class *Student*
- An object of type *Object* representing an instance of the class *Student*
- Two objects of type *StructuralFeature*, each representing a property of the class *Student*
- The *Student* object holds two instances of the class *FeatureValue* referenceing the two *StructuralFeature* instances. Each of the *FeatureValue* instances holds an instance of type *Value* which holds the value of the property.

2.2.2 Code Generation

Code generation is an approach that generates and compiles additional source code for each executed model, which can be run to perform the actual model execution. In contrast to the interpretation approach, the generated code is specifically tailored to only represent the functionalities of one specific model, which leaves a less complex, model-specific unit for the execution of the model. However, the complexity of the whole model execution system required for this execution method might not necessarily be smaller, as the component responsible for the generation of the necessary code still needs to be able to handle all possible models. The technology for generating code usually employs a templating engine, which is a piece of software that fills in a pre-built template file. The template file contains all static elements of the code that needs to be generated, as well as specific keywords to indicate positions at which the dynamic, model-specific information needs to be inserted. The resulting is the source code used for the execution. This form of model execution is applied by AMUSE.

Model and Object Instantiation

By generating code based on the classes defined in a specific UML model that is supposed to be executed, the code generation approach is able to create model-specific representations of the classes. The instances used during the model executions are then created by instantiating these generated classes.

To further illustrate the usage of the code generation approach, we will reconsider the class diagram example introduced in 2.1 and depicted in Figure 2.2, as it is processed by the AMUSE plugin. The source code generated for the class *Student* is depicted in Listing 2.1.

```
class Student
{
        int student_id;
        string name;
}
```

Listing 2.1: Code generated by AMUSE for the class *Student* (cf. Figure 2.2)

Instantiating this class results in a single object of type *Student*, with a property *name* of type *string*, and a property *student_id* of type *int*.

2.2.3 fUML

In addition to the UML standard discussed in the Section 2.1, the OMG has also released a document detailing the *Foundational Subset For Executable UML Models* (or fUML for short). This fUML standard is concerned with defining precise semantics for a subset of UML. The fUML specification consists of three major parts [16]: The syntax, the execution model and the model library. Each of these components is discussed in the following. Furthermore, we discuss the execution semantics which are defined by fUML in a separate sub-section.

Syntax

As fUML deals with a subset of modeling concepts of UML, the syntax used for creating fUML models is identical to the syntax used within UML. The fUML standard deals primarily with implementing the elements of the UML kernel, which includes classes, associations, and packages [21].

Classes are utilized for describing the structures used during the model execution—i.e., instances, their properties and their relations to each other. fUML does not include the full range of functionality provided by UML classes, however. For example, the fUML execution environment does not support using interfaces within class diagrams.

Behavioral features are modeled exclusively by activity diagrams. Again, a number of different specific features of UML activity diagrams are omitted within fUML. This includes all actions for using static variables, as well as the broadcast signal action.

The fUML syntax is restricted to these reduced versions of class diagrams and activity diagrams, and does not go into any further detail on the other types of behavioral diagrams. Furthermore, the fUML standard does not offer any insights on possible interactions with these behavioral diagram types. However, the aim of this thesis is not only to integrate fUML with AMUSE for executing activity diagrams, but also to enable the integration of the execution of activity diagrams with the execution of state machines and sequence diagrams. These interactions therefore are not governed by the fUML standard.

Execution Model

The execution model of fUML is a model that specifies how UML models are represented and interpreted for their execution. This model is written in fUML itself. The execution model of fUML is built as an extension of the UML metamodel. The execution semantics are added via the visitor pattern, a software pattern defined by Gamma et al. [8]. By applying the visitor pattern, the fUML execution model is able to add execution semantics to the definitions of UML modeling concepts, while leaving the original definitions of the modeling concepts unchanged. By extending the metaclasses of UML with this visitor pattern, the execution model is largely decoupled from the UML metamodel.

Locus. An important concept defined by the execution model is the *locus*, which serves as a persistent storage facility for all value specifications created or modified during the execution. The locus holds a single instance of the *executor* component, which provides the model execution facilities. This executor is linked to the locus instance, and can only operate upon the value specifications stored by it. The specification mentions the possibility of behaviors interacting across multiple loci, but does not specify a mechanism by which this could be accomplished. Furthermore, the locus may provide access to additional services and objects to be used by the executions. These additional services are defined by the *model library*.

Executor. The *executor* component provides the model execution facilities. It provides the following three key services [22]:

- Execute: Synchronously executes a behavior for given input parameter values and returns output parameter values.
- Start: Asynchronously starts the execution of a stand-alone or classifier behavior, returning a reference to the instance of the executing behavior.
- Evaluate: Evaluates a value specification, and returns the resulting value.

The functionalities provided by the executor are implemented by utilizing the visitor pattern discussed above. A separate *execution factory*, which also resides within the locus, provides the required visitor instances. The execution factory instaniates the respective visitors for the elements contained by the model to be executed. It provides instances of the following visitor types:

- Execution: An execution visitor performs the execution of a specified behavior. The following implementations are available:
  - ActivityExecution: Implements the execution semantics of activities.
  - OpaqueBehaviorExecution: Used to execute pre-defined behaviors, which are provided by the fUML execution environment.
  - FunctionBehaviorExecution: Basic functionality, such as comparison of objects and definitions of binary and unary operations.
- Activation: An activation visitor defines the behavior of a specific activity node. fUML provides an implementation of an activation visitor for each activity node type defined within UML, with the exception of the BroadcastSignalAction.
- Evaluation: An evaluation visitor is used to create a value instance based on a value specification. The value specification may represent a primitive value, such as an Integer or a String value, as well as links between objects.

Model Library

The *model library* provides supporting structures for the execution environment of fUML. This includes both primitive datatypes, as well as reusable behaviors, such as basic arithmetic functions. The fUML standard also mentions the possibility of providing access to external systems via such behaviors, but does not go into further detail on how these behaviors could be implemented.

Execution Semantics

The execution semantics fUML gives to activity diagrams are nearly identical to the execution semantics given to an activity diagram in the UML standard, albeit more precise. The execution model of fUML is based on token-passing semantics. Two different flows of tokens are implemented—the flow of control tokens, which signify which actions are currently being executed, and the flow of object tokens, which serve to store and transport information during the execution.

The execution of an activity is conducted in the following way [18]: As a first step, the available input parameter values are provided to the input parameter nodes of the activity, from where they can be passed on via object flows. Secondly, the initially executable action nodes are determined. This includes the initial node, as well as all activity nodes with no incoming control flows. All these nodes are provided with control tokens. When the initial control tokens have been distributed, the execution model continuously repeats the following steps until no additional activity node can be executed:

1. Determine all executable activity nodes. For a node to be executable, all incoming control flows must hold a control token. Furthermore, depending on multiplicity, all input pins must hold a corresponding object token.

2. Execute all executable action nodes. During the execution, each respective node:

   a) Consumes all incoming control and object tokens

   b) Performs its associated behavior

   c) Provides outputs of the behavior on output pins via object tokens, if applicable

   d) Generates a new control token on each outgoing control flow

Lastly, after the execution of all executable nodes has concluded, the output values generated by the activity are placed on the output parameter nodes.

Semantic Variation Points. The fUML standard introduces two *semantic variation points*. The fUML execution model contains explicit implementations of these functionalities, which may however be adapted by the concrete implementation.

- Event dispatch scheduling: The fUML execution model handles all events on a first-in, first-out basis.

- Polymorphic operation dispatching: In the case of multiple implementations of an operation, the fUML execution model decides which implementation to execute based upon both the context of the invocation and the target of the operation call.

For these semantic variation points, the fUML standard incorporates the *strategy* pattern, which allows interchanging different algorithms [8]. The strategy pattern is also used to incorporate non-deterministic behavior, e.g., choosing which clause body of a conditional node is to be executed if their respective tests are positive [22].

Furthermore, the fUML standard does not require a specific implementation of the following features:

- The semantics of time: The standard does not restrict the way time should be handled.

- The semantics of concurrency: The execution of activities according to the fUML standard is implicitly concurrent, as long as the token passing mechanics allow it. The standard does not require an implementation to actually execute multiple actions in parallel, however.

- The semantics of inter-object communications mechanisms: The methods of exchanging values and signals within the model execution are not specified. Furthermore, both reliability of the communication and possible delays are solely in the hands of the specific implementation.

2.3 Tools

The prototype created for this thesis is implemented as an integration of multiple existing tools. These tools include both the main integrated components (Enterprise Architect and the fUML reference implementation), as well as multiple additional tools and plugins that are utilized specifically for the integration. Each of the following subsections will provide a short overview of a specific tool, and will then present the functionality of the tool that is relevant to the prototype created for this thesis.

2.3.1 Enterprise Architect

Enterprise Architect (EA) is a commercial UML modeling tool developed by Sparx Systems. EA aims to provide a complete modeling tool for the UML standard, i.e., all elements described therein can be modeled.

The strictness with which EA adheres to the UML standard varies with the elements in question. A more detailed examination of this topic including the description of measures taken to overcome certain limitation of EA can be found in Section 5.2.3.

The prototype created for this thesis utilizes the following functionality of EA:

- Creation of models
- Validation of models before execution
- Generation of code

A core feature of EA is its inherent expandability via the creation of plugins. EA provides a fairly extensive API, which provides access to a large number of EA's functionalities. A plugin has access to all of the information stored in a model, and is also able to modify the user interface of EA, e.g., by adding custom user interface elements or context-sensitive menu entries.

In the professional edition and above editions, EA provides an extensive mechanism for executing models based on an interpretative approach. This model execution mechanism is capable of executing a combination of activity diagrams, state machines and sequence diagrams. This model execution mechanism, however, supports only limited interactions between specific behaviors. Communication between behaviors can only be realized via signals broadcasted by pre-defined triggers[7].

2.3.2 AMUSE

The toolkit called Advanced Modeling Using Simulation and Execution (AMUSE) is a plugin for EA developed by LieberLieber Software GmbH[8]. The purpose of AMUSE is to provide an alternative execution environment for behavioral models created within EA based on code generation.

[7] `http://www.sparxsystems.com/enterprise_architect_user_guide/10/model_simulation/model_simulation.html`, Accessed: 27.1.2015

[8] `http://www.lieberlieber.com/amuse/`, Accessed: 16.08.2014

Figure 2.12 depicts the components of AMUSE, as well as the basic relationships between them, which are both described in the following.

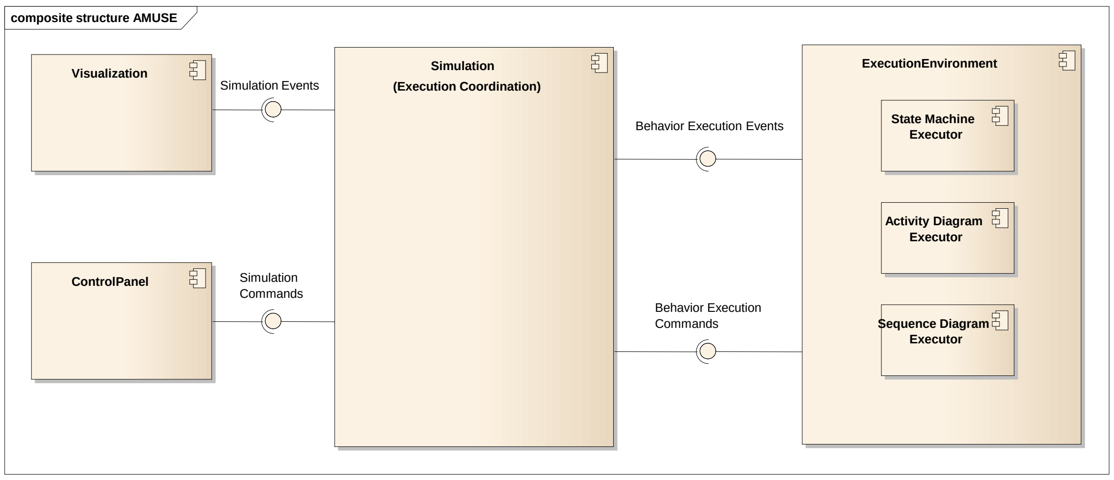

composite structure AMUSE
Visualization
Simulation Events
ControlPanel
Simulation Commands
Simulation (Execution Coordination)
Behavior Execution Events
Behavior Execution Commands
ExecutionEnvironment
State Machine Executor
Activity Diagram Executor
Sequence Diagram Executor

Figure 2.12: Main components of AMUSE

Simulation (Execution Coordination)

The *Simulation* component serves as the central component of the AMUSE plugin. The *Simulation* component accepts commands from the *ControlPanel* component, controls the *ExecutionEnvironment* component and continuously provides information about the current state of the model executions to the *Visualization* component. The *Simulation* component is also able to conduct several *Sub-Simulations*, with each *Sub-Simulation* providing the same functionality as the original *Simulation* component. When *Sub-Simulations* are present, the *Simulation* component propagates events between *Sub-Simulations*, which allows them to communicate with each other without being tightly coupled. In fact, the respective *Sub-Simulations* do not know of each other's existence at all.

Execution Environment

The *ExecutionEnvironment* component provides an implementation of the entire execution environment of the AMUSE plugin. This component is responsible for both the execution of behavioral diagrams, as well as the constant propagation of the current execution state, which is interpreted and further distributed by the *Simulation* component.

Supported Combination of UML Diagram Types. The execution semantics provided by the AMUSE execution environment are built for a very specific combination of structural and behavioral UML diagram types, which must be adhered to by the user for the execution to be performed successfully.

The core element of each execution conducted by AMUSE is a single *Class* element. The whole execution is centered around a single instance of this class, with all attached behavioral diagrams describing the behavior of this class. The *ExecutionEnvironment* component executes all behaviors in the context of this single instance. To distinguish this class within this description of the execution semantics, it is referred to as the *main class*, with the instance being referred to as the *main instance*.

A single state machine is used as the behavioral model for the main instance. This state machine represents the classifier behavior of the main instance, which implies that it represents both the entire behavior of the instance, as well as all forms of communication available to it.

Furthermore, the main instance may hold an arbitrary number of activity diagrams. These activity diagrams can be invoked by the state machine either as entry/do/exit actions on a state, or as an effect of a transition. The execution of such an activity diagram is conducted asynchronously, i.e., the state machine immediately resumes execution after the execution of an activity diagram has been invoked.

Furthermore, a single sequence diagram may be defined for the main instance. This sequence diagram only serves the purpose of creating objects in addition to the main instance, which will be used during the execution.

Execution Preparation. Before conducting the actual execution of the main instance, the *ExecutionEnvironment* component performs a preparation phase to set up the necessary structures for performing the execution. In this phase, the source code needed for the model execution is generated and compiled. The generated source code consists of the following components:

- Representations of all behavioral models used during the execution. Each of these representations only contained the model information that is specifically required for the execution, and is stored in a separate code file. A representation is generated both for the state machine and the sequence diagram connected to the main class, as well as for all activity diagrams that are referenced by this state machine.
- Representations of all classes that are required during the execution. This includes both the main class, as well as all classes that are linked to either the main class or one of the behavioral models that have been processed during the first step. An example for the latter would be the class referenced by a create object action within one of the activity diagrams.
- An implementation of an additional *BehaviorContainer*, which contains references to all objects required for the execution. This includes the main instance, as well as representations of the behavioral models generated in the first step. This *BehaviorContainer* serves as the object space used by the *ExecutionEnvironment* component.

Execution. As soon as the code generation has been completed, the *ExecutionEnvironment* component starts the execution of the main instance. The actual execution of each specific behavioral model is conducted by a separate *Executor* instance, which is instantiated by the *ExecutionEnvironment* component and then responsible for the execution of the behavioral model. Three separate types of executors are available to the *ExecutionEnvironment* component, each of which is responsible for a specific type of behavioral diagram, namely activity diagram, state machine and sequence diagram.

The execution performed by an *Executor* is strictly based on the representation of a behavioral model, which has been generated by the *ExecutionEnvironment* component beforehand. The implementations of the respective *Executors* are built to only serve their specific purpose in the previously described execution semantics of AMUSE.

The execution semantics for state machines implemented in AMUSE adhere to the execution semantics for state machines defined in the UML standard, as described in Section 2.1.

The execution semantics for activity diagrams supported by AMUSE are very limited. All of the control nodes are available for usage. However, only a single, generic action type is supported. The actual behavior of each action must be implemented through C# code.

The execution semantics for sequence diagrams are only built for the purpose of instantiating objects and assigning the created objects to specific attributes of other objects within the object space. Each instance defined with a separate lifeline within a sequence diagram will be instantiated within the object space. Links between these objects can be established by exchanging pre-defined messages. Additionally, the invocation of operations defined on the instantiated objects is also supported. However, this mechanism cannot be used to invoke behaviors attached to this operation as discussed in Section 2.1. Instead, it is only able to execute C# source code that has been attached to the specific operation within the model.

The combination of the *Executor* instances and the code files generated by the *ExecutionEnvironment* component are sufficient to provide the full execution environment of AMUSE. These components are compiled into a single library which is then dynamically loaded and used by the *Simulation* component. This library can also be used as a stand-alone execution environment for the main instance, which can be used independently of both EA and the AMUSE plugin.

Control Panel and Visualization

The *ControlPanel* provides functionalities for controlling the execution to the user via an additional user interface that is directly integrated with the user interface of EA. The available functionalities are:

- Starting and stopping the execution
- Advancing the execution by a single step
- Setting breakpoints
- Inspecting the current execution state (e.g., created objects and their respective values)
- Manual triggering of transitions and dispatching signals

The *Visualization* component displays the current state of the running executions to the user. The on-screen display of the execution is created by layering a Scalable Vector Graphics (SVG) [29] overlay over the display of the model itself, which is used to convey information about the current execution state, e.g., the currently active node. The propagation of user interface updates is done strictly via events. Therefore, the *Visualization* component receives updates on the current state of the executions from the *Simulation* component, which passes the corresponding events along from the *ExecutionContext* component.

2.3.3 MDE Plugin

The MDE Plugin is a plugin developed by LieberLieber Software GmbH that provides an object-oriented interface for accessing models created within EA. This interface encapsulates the information contained within the models in classes and objects that adhere to the UML standard.

Even though EA's API provides access to all of the data contained within a model, accessing them via the provided functionalities can be cumbersome. Especially accessing information on specific types of UML modeling elements, e.g., specific action types, requires querying the EA models directly via SQL queries. This information is made available by the MDE plugin in a more direct way.

At the time of developing the prototype for this thesis, the MDE plugin was still under development, and therefore did not encompass the whole UML standard yet. Therefore, the MDE plugin could not be used exclusively for accessing the EA database, and the prototype still needs to use direct SQL queries to the EA database to acquire a part of the required model data.

2.3.4 fUML Reference Implementation

The fUML reference implementation is an implementation of the fUML virtual machine as described in the fUML standard [22], which is provided by modeldriven.org[9]. The implementation is done in the Java programming language. For a more detailed description of the inner workings of the fUML execution environment, please refer to Section 2.2.3.

The purpose of the fUML reference implementation is to provide a reference for tool vendors seeking to implement the fUML standard. As such, the implementation covers all the features described in the standard, but includes no additional features aside from those explicitly specified therein.

The fUML reference implementation provides a single main functionality. It executes a UML conformant model specified in an XMI file and produces an execution trace as output. The execution is carried out as a single, synchronous method-call, with no possible points of interaction with the execution environment during the actual execution.

This holds the following key implications for utilizing the fUML reference implementation in the context of this thesis:

- Pausing / resuming the execution is not possible
- There is no way to influence the execution path in scenarios where there are multiple execution paths due to parallelism
- No feedback on the current state of the execution can be obtained during the execution
- No information about objects present at the locus can be obtained during the execution

[9]`http://portal.modeldriven.org/project/foundationalUML`, Accessed: 16.08.2014

2.3.5 MOLIZ fUML Debug API

Mayerhofer et al. have implemented an extension of the aforementioned fUML reference implementation [19] called the MOLIZ fUML Debug API, which aims to overcome the previously described limitations.

This implementation extends the original fUML reference implementation by utilizing Aspect Oriented Programming, and thereby adding additional functionality while leaving the original source code untouched.

Stepwise Execution

The MOLIZ API allows the stepwise execution of fUML models. This provides control over the actual execution, and allows observing the execution state (by querying the locus via its built-in access functionalities) after every step in order to react to it accordingly.

Multiple execution may be ongoing in parallel. The fUML reference implementation distinguishes these executions based on the fact that each execution is conducted by a separate *ActivityExecution* instance. The MOLIZ API assigns unique ids to each of these currently running executions. This provides an approach to both track and refer to specific executions. These unique ids are also used in controlling the execution. When advancing a single execution by a step, the unique id of the respective execution must be provided. The actual definition of a step is based on the concept of an *ActivityNodeActivation*, a singular object that encapsulates the execution behavior for a single node (either an action or a control node). For each step executed, one such *ActivityNodeActivation* is run.

Event Mechanisms

The MOLIZ API also incorporate event mechanisms, which propagate changes in the current execution state. The following types of events are provided:

- Begin and end of the execution of an activity
- Begin and end of the execution of an action
- Events pertaining to objects residing at the locus (e.g., the creation or modification of an object).

2.3.6 IKVM

IKVM[10] enables the invocation of Java libraries from within the .Net Framework. This is accomplished by applying a process to convert Java Archives (JARs) into Dynamic Link Libraries (DLLs, libraries for C# code). The created DLLs provide full representations of all classes and functionalities contained within the JARs. Whenever a method within one of these classes is invoked, the DLLs invoke a custom-built virtual machine that executes the original JARs.

[10]`http://www.ikvm.net/`, Accessed: 5.1.2014

The process of converting JARs to DLLs via IKVM requires the user to provide both the JARs that should be used, as well as all referenced JARs (including JARs referenced by the referenced JARs themselves). Unfortunately, the version of IKVM used for the development of the prototype does not provide a concise way of determining if all required references have been added, and instead requires the analysis of log-files produced during the conversion to determine which required JARs could not be loaded. The resulting process of determining the correct set of JARs is therefore both tedious and error-prone.

CHAPTER 3

State of the Art

This chapter provides an overview of the existing work which relates to this thesis, thereby outlining the relations between the existing work and the contributions of this thesis.

In Section 3.1, we discuss the various existing tools and approaches for executing both UML activity diagrams and state machines. In Section 3.2, we present the work that specifically deals with implementing interactions between UML behioral diagram types in the context of model execution. This section also focusses on the topic of formalizing UML, as specific formalizations also deal with formalizing the specific interaction mechanisms. Lastly, as has been discussed in the introduction, the prototype implemented during this thesis also has to overcome the technology gap between two major approaches to model execution, namely model interpretation and code generation. Therefore, we discuss the literature on this topic in Section 3.3.

3.1 Execution of UML Activity Diagrams and State Machines

The fUML standard seeks to provide a clear and concise semantics specification for a subset of UML. However, defining the semantics of UML has been a topic of research for a number of years before the creation of fUML, leading to a variety of different interpretations of the semantics of UML. The thesis of Hausmann [10, pp. 23–28] discusses 25 different formalizations, each built for a specific application purpose and built upon a specific formal technique.

The number of tools which support the execution of UML models has increased steadily over the past years. Aside from the fUML reference implementation and the AMUSE toolkit, which are both the focus of this thesis, a substantial body of work exists which includes multiple open source and commercial tools[1] with varying degrees of adherence to the UML standard.

There are a couple of commercial tools that deal with the execution of UML models. The most prominent example is IBM's Rational Rhapsody[2], which provides execution capabilities

[1] `http://modeling-languages.com/list-of-executable-uml-tools/`, Accessed: 27.1.2015

[2] `http://www-03.ibm.com/software/products/en/ratirhapfami`, Accessed: 6.11.2014

for UML state machines and activity diagrams via code generation. Another example is the Cameo Simulation Toolkit[3], which is an extension of the UML Tool MagicDraw[4] that includes a model execution framework based on fUML and the SCXML standards of W3C.

However, as the fUML standard has only recently been released in February 2011 [22], and the adoption of such a new standard in the industry usually requires considerable time and resources, we also look into the academic literature on this topic.

This section discusses both the literature, as well as the tools that either provide model execution capabilities based on fUML, or specifically deal with the execution of activity diagrams and state machines. Table 3.1 provides an overview of the discussed work.

Author	**UML Version**	**Diagram Types**	**Basis of Execution Environment**
Sarstedt [26]	UML 2	Activity diagrams	Abstract State Machines
Crane and Dingel [5]	UML 2	Activity diagrams	Compositions of state machines
Hoefig et al. [12]	UML 2	State machines	Model interpretation, code generation
Cameo Simulation Toolkit	UML 2, fUML 1.1	Activity diagrams, state machines	fUML standard, SCXML standard of W3C
Pópulo [7]	UML 2	Activity diagrams, state machines	UML Action Language

Table 3.1: Related work on execution of UML activity diagrams and state machines

The work of Sarstedt [26] defines an interpreter for activity diagrams similar to the one defined in the fUML standard. The approach is defined theoretically on the basis of Abstract State Machines [2], and a corresponding prototype implementation is provided. However, no evaluation of the interpreter and implementation concepts is conducted, leaving the actual practicability of this approach unproven.

The work of Crane and Dingel [5] introduces another proprietary implementation of an UML model execution tool for activity diagrams. This work is especially interesting in the context of this thesis because of its approach to formalizing the execution semantics of activity diagrams. The formalization of the lower-level components, which actions and activities are composed of, is based on compositions of state machines [5]. Therefore, even though the specific interpreter described in this work only deals with activity diagrams, it could theoretically also be expanded to incorporate state machines. The introduced prototype also provides multiple modes for executing the activity diagrams, including random and guided modes. The work does not provide an evaluation of the presented prototype.

The work of Hoefig et al. [12] discusses an implementation of two variants of UML model execution environments for executing state machines. One of these execution environments is

[3]`http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html`, Accessed: 6.11.2014

[4]`http://www.nomagic.com/products/magicdraw.html`, Accessed: 6.11.2014

based on model interpretation, the other one is based on code generation. The goal of the work of Hoefig et al. is to evaluate the performance differences between these two approaches of executing state machines. To accomplish this goal, both described approaches have also been implemented in prototypical form. Aside from the implemented prototype and the performance evaluation, this work also discusses a number of different challenges when executing state machines, as well as the accuracy of the specification of the execution semantics of some state types, e.g., the history state.

From the currently available tools for UML model execution, the functionality of the Cameo Simulation Toolkit most closely resembles the functionality of the developed prototype developed for this thesis. The Cameo Simulation Toolkit is a plugin for the commercial UML modeling tool MagicDraw. It provides model execution capabilities for both activity diagrams and state machines[5]. The remarkable feature of the Cameo Simulation Toolkit is its adherence to two specific standards for implementing the provided UML model execution capabilities. For the execution of state machines, the Cameo Simulation Toolkit utilizes an open-source execution environment based on the SCXML standard of W3C, which is in turn based in Harel's specification of statecharts [9]. The execution environment used for activity diagrams is a custom implementation based upon the fUML standard.

Another UML model execution tool that offers model execution features similar to those offered by the developed prototype implemented for this thesis is the Pópulo tool [7]. Pópulo defines a fully UML compliant interpreter for the UML Action Language, which is capable of executing actions and activities. The official website of the Pópulo tool[6] also indicates that the current version of Pópulo supports the execution of state machines. This functionality is discussed as future work in the original publication [7], but is not further defined in the available documentation of the tool. Pópulo offers full debugging support during the execution, such as step by step execution or breakpoint-based debugging. The tool provides a feature-set similar to the one offered by the developed prototype. Also, the tool relies on XMI files as a basis for the interpretation, which is a standardized storage format for UML models. This allows it to exist as a stand-alone tool, which is interoperable with every UML tool that can create a valid XMI representation of UML models.

3.2 Formal Specification of UML Diagram Types and Interactions

The tools discussed in Section 3.1 share a common weakness, which is also one of the topics covered in this thesis—they simulate a very limited range of diagram types, with most tools only focussing on one behavioral diagram type (usually activity diagrams or state machines). Therefore, these tools are not concerned with the mechanisms required for implementing communication mechanisms between different diagram types.

One of the core problems in successfully implementing the various communication mechanisms between different types of UML diagrams is the intentionally vague nature of the UML standard, which does not go into detail on the relations between diagram types. Nevertheless, a

[5]`http://www.nomagic.com/files/manuals/Cameo%20Simulation%20Toolkit%20UserGuide.pdf`, Accessed: 27.1.2015

[6]`http://caosd.lcc.uma.es/populo/index.htm`, Accessed: 22.12.2014

number of possible approaches have been proposed to give the elements of the UML standard a concise meaning. Most of these approaches, again, only deal with one specific diagram type, while only a few approaches look at a more broad spectrum. In this section, we focus on the literature that tries to interpret multiple behavioral diagram types in an integrated way.

As discussed in Section 2.1, one of the key problems of UML in the context of model execution is the ambiguity with which modeling concepts and their semantics are defined within the UML standard. The semantics are mostly defined in English prose, and therefore cannot be directly utilized for machine-based execution of UML models. The fUML standard defines precise semantics for classes and activity diagrams, but does not address the other behavioral diagram types. Additionally, the fUML standard has only been published in 2011. Therefore, there is a large body of work on formalizing (parts of) UML with the goal of enabling model execution. One of the main sources for this section is the thesis of Hausmann [10, pp. 23–28], which discusses 25 different formalization of UML.

The literature discussed in this section is especially relevant to this thesis because each work considers multiple types of behavioral diagram types which have been integrated with each other. Such an integration, in turn, is also one of the goals of the prototype created for this thesis. Table 3.2 provides an overview of the discussed work.

Author	**UML Version**	**Diagram Types**	**Approach**
Kohlmeyer and Guttmann [15]	UML 2	Activity diagrams, state machines	Abstract State Machines
Jürjens [13]	UML 1	Activity diagrams, state machines	Abstract State Machines
Börger et al. and Cavarra [3] [4]	UML 1.4	Activity diagrams, state machines	Abstract State Machines
Kirshin et al. [14]	UML 2	Activity diagrams, state machines	General model execution framework
Nitto et al. [20]	UML 1.3	Class diagrams, activity diagrams, state machines	Process models
Baresi et al. [1]	MADES UML	State machines, sequence diagrams	TRIO metric temporal logic

Table 3.2: Related work on formal specification of UML diagram types and interactions

The work of Evans et al. [6] provides a very useful starting point in exploring formalization of the UML standard. It discusses a number of different approaches to formalizing models. The main presented ideas are either to try to formalize the existing, informal definitions, or to use an existing formal mechanism for creating models. Evans et al. propose the implementation of a variant of UML called *Precise UML* (PUML). The focus of PUML is the formalization of core UML concepts, as well as the creation of a formal reference manual [6]. Unfortunately, it seems that the work on this interpretation has ended, as the website of the Precise UML project[7]

[7] `http://www.cs.york.ac.uk/puml/`, Accessed: 27.1.2015

indicates no new publications after the year 2000.

The work of Kohlmeyer and Guttmann [15] describes the existing mechanisms in UML for defining interactions between different diagram types, such as associating an activity with the entry action of a state or a call behavior action. It also defines formal semantics for these interactions, again using Abstract State Machines. Because of the formal nature of the specification, the resulting interaction model is platform-independent and could theoretically be integrated with fUML as well as the proprietary execution environment of AMUSE.

The work of Jürjens [13] takes a strictly formal approach for defining the semantics underlying UML behavioral diagrams. The formalization is done by transforming all relevant concepts to a formalism based on Abstract State Machines. The proposed formalism handles state machines, activities and a newly introduced modeling concept called a "subsystem". A subsystem consists of a class diagram, an activity diagram, as well as a set of state machines describing the behavior of single instances. Furthermore, a subsystem also includes all the relevant interfaces and messages relevant for connecting the aforementioned diagram types. The work includes an outline of the proposed formalisms for state machines and activity diagrams, as well as a number of specific examples to illustrate the fundamentals. It goes into detail on possible applications of this formalism, such as proving behavioral equivalence based on the generated Abstract State Machines. This work is still based on the notion of activity diagrams being a specialized version of state machines defined in the UML versions 1.x. Nevertheless, it shows a very clear-cut approach to formalize UML behavioral diagrams and breaking them down to a common denominator. Unfortunately, the work contains no discussion of the practicability of the introduced approach, and therefore only serves as a theoretical guideline. As for the context of this thesis, the approach of modifying the complete execution mechanism for all behavioral types far surpasses the scope, and can instead only be seen as a possible alternative approach.

The work of Börger et al. and Cavarra [3] [4] presents a formalism designed to formalize UML behavioral models (specifically state machines and activity diagrams) by converting them to Abstract State Machines. A complete model for this conversion is provided. This conversion is also applicable to actions based on their definition in UML 1.4. The semantics of activity diagrams have been changed in UML 2.0, however. The semantics are no longer tied to state machines as in UML 1.4. Therefore, the approach presented in this work is not applicable to the current version of the UML standard. An interesting advantage mentioned in the work of Börger et al. is that Abstract State Machines are directly executable by a variety of different tools. Therefore, a complete conversion of all behavior diagram types might possibly be sufficient for enabling a homogenous execution of models containing the different behavioral diagram types.

From a conceptual point of view, defining model execution semantics is also discussed by Kirshin et al. [14]. They describe a general model execution framework, which can be adapted to fit to execute models conforming to an arbitrary modeling language (such as UML). It operates on the basis of so called "Instances", which are the run-time representations of the model elements. However, as the described approach is a very general-purpose one, it resides on a very high level of abstraction, and can only be used as a conceptual framework in the context of this master's thesis.

The work of Nitto et al. [20] introduces a very specific use case for formalizing UML. They use UML for defining process models. Process models provide a description of a process, which can then be executed fully automatically by a process executor. The work of Nitto et al. introduces specific restrictions for a model consisting of class diagrams, activity diagrams and state machines, with each representing a specific component of the process description. A model that conforms to these specific restrictions can be directly converted into code, which is in turn interpretable by the process executor OPSS.

The work of Baresi et al. [1] introduces a formalization of a combination of UML class diagrams, state machines and sequence diagrams. The formalization is conducted by translating a model consisting of these diagram types into formulae adhering to the TRIO metric temporal logic, which has well-defined execution semantics. The main goal of the work of Baresi et al. is to provide automatic verification of UML models based upon this conversion, which can be accomplished by using a satisfiability checker tool on the formulae resulting from translating the UML models to the TRIO metric temporal logic.

3.3 Model Interpretation vs. Code Generation

One of the key challenges this thesis addresses is integrating two separate execution environments with a different approach to executing models, namely model interpretation and code generation. For an introduction to these model execution environments, please refer to Section 2.2. In this section, we present the literature which deals with both the differences between model interpretation and code generation, as well as the integration of these two approaches.

3.3.1 Comparison of Model Interpretation and Code Generation

The differences between model interpretation and code generation, as well as their respective merits, are an ongoing topic of discussion in the industry[8] [9]. In the following, we contrast them regarding functionality, performance and UML standard compliance.

Functionality

A wide variety of different implementations for both model interpretation and code generation are available, making an exhaustive comparison of their respective functionalities rather pointless. Therefore, an alternative approach for this analysis needs to be taken.

A theoretical comparison of the expressiveness of both model interpretation and code generation can be conducted based on the concept of turing-completeness. Turing-completeness refers to a set of properties of a program, which enable it to complete an arbitrary computation. Specific implementations of both model interpretation [17] and code-generation [27] have been shown to possess this specific property. Therefore, both approaches can theoretically provide equivalent functionality.

A practical approach for comparing model interpretation and code-generation is to apply a specific implementation of the respective approaches to a pre-defined set of case studies. This

[8] `http://modeldrivensoftware.net/profiles/blogs/model-driven-development-code`, Accessed: 16.08.2014

[9] `http://www.linkedin.com/groups/CodeGeneration-is-better-than-50539.S.54241224`, Accessed: 16.08.2014

approach enables the verification of the functional equivalence of the approaches based on these case studies. Furthermore, a direct performance comparison can be conducted based on the execution of these case studies. This approach has been used in the evaluation of the prototype created for this thesis.

Performance

The work of Höfig et al. [12] deals with both model interpretation and code generation. It provides an extensive analysis of both model execution environments in the context of UML state machines. Furthermore, it gives a detailed description of an implementation of a UML interpreter, which is compared to an equivalent code-generation implementation.

The findings of this work show a distinct lack of existing tools or benchmarks for performing an actual performance comparison between the model interpretation and the code generation approach. The benchmarking process that has been used in this work relies on taking time-stamps at specific execution points in the implementations of both approaches. This approach of benchmarking is also used for the evaluation of the prototype created for this thesis.

The performance evaluation conducted by Hoefig et al. [12] has found the following results in comparing the execution of code generated for the execution of models with interpreting the model directly:

- the interpreted approach is about 3 to 460 times slower (with a noted average of 20)
- the interpreted approach utilizes 60 to 80 times more memory

The authors of this study have not provided specific information on the execution time and memory capacities required for generating the source code they were utilizing in their tests.

Unfortunately, both specific data, as well as general research on comparing the performance of model interpretation and code generation is scarce.

Standard Compliance

The work of Riehle et al. [24] introduces a difference between model interpretation and code generation that is exclusive to the context of model execution itself. The code generation approach generates source code classes representing the model classes it needs to instantiate. The model element and generated code, however, are not directly connected to each other. Therefore, the source code classes exist outside of the definitions of the UML standard. Conversely, with an interpretative approach, it is possible to interpret the UML model utilizing only concepts defined within UML itself, thereby providing a causal connection between the modeled classes and the resulting instances [24]. Furthermore, by utilizing the behavioral semantics defined within UML, even the execution of the model can be built strictly with concepts defined within UML, further ensuring the UML-compliance of the model execution approach.

3.3.2 Combination of Model Interpretation and Code Generation

In practice, a combined approach is often employed, which utilizes both model interpretation and code generation—albeit in different phases of the development process. A possible configuration is to utilize model interpretation in the early stages of the modeling process, while still utilizing the code generation approach for generating the software artifacts for operational use[10].

As for the approach that is followed in this thesis—which is the application of both approaches side-by-side (only for the execution of different diagram types), the literature study has not found a significant amount of work on this specific topic.

The framework proposed by Kirshin et al. [14] deals with the issue of different model execution techniques. Specifically, it includes two different approaches: one is strictly interpreter-based, while the other one uses code generation. The most interesting aspect of this framework in the context of this master's thesis is that it is able to apply both code generation and interpretation simultaneously. The work mentions the possibility to apply interpretation for some model elements, while utilizing code generation for the rest. A similar mechanism for switching between interpretation and code generation is also implemented in the prototype developed for this thesis. However, the prototype developed for this thesis switches between interpretation and code generation based on the type of the behavioral diagram, and not based on the specific model element.

[10] `blog.abstratt.com/2010/08/07/model-interpretation-vs-codegeneration-both/`, Accessed: 16.08.2014

CHAPTER 4

Prototype

In Chapter 2, we have extensively discussed the UML standard, with a specific focus on interactions between specific behavioral diagram types. Furthermore, we have discussed the topic of model execution. We have introduced two different model execution techniques — interpretation and code generation. All of the findings presented in Chapter 2 are based on a theoretical evaluation of both the UML standard, as well as the fUML standard. The gained knowledge built the foundation for integrating fUML with EA in a prototype. This prototype utilizes both the knowledge on the integration of different behavioral diagram types, as well as different model execution techniques discussed in Chapter 2.

In this chapter, the prototype is discussed in detail. In Section 4.1 gives a general overview of the created prototype, present its main goals, and discuss the ways in which the prototype utilizes external tools. In Section 4.2, we provide a detailed look into the functionalities the prototype provides, as well as technical details of the implementation and important design decisions. Finally, in Section 4.3 we discuss the adaptations that were performed on the used external tools to complete the integration of EA and the fUML reference implementation.

4.1 Overview

This section provides a general overview of the prototype created for this thesis. The main goals achieved by the prototype are discussed, as well as the expected results of creating the prototype. Lastly, external tools used by the prototype are summarized, and the context in which they are used is presented.

4.1.1 Goal

The main goal of the prototype was to adapt the EA model execution plugin AMUSE so that all executions of activity diagrams are done via the fUML reference implementation, while leaving the remaining functionality of the AMUSE plugin intact. This goal has been accomplished by replacing the proprietary model execution environment AMUSE utilizes for executing activity diagrams with a fUML standard compliant model execution environment in the form of the fUML reference implementation. Thereby, the execution of activity diagrams via AMUSE is made fully fUML compliant. Furthermore, this allows for increased interoperability of AMUSE with other tools which also fulfill the fUML standard.

The rational behind basing the prototype on the AMUSE plugin instead of the model execution capabilities of EA is two-fold. Firstly, the AMUSE plugin incorporates a larger number of interaction mechanisms between behavioral diagram types, such as exchanging parameters between behavioral diagrams, which are currently not available in the model execution environment of EA. Secondly, adapting AMUSE for the prototype has given us the opportunity to combine an interpretation-based execution environment with a code generation-based execution environment and explore the implications of combining these two model execution approaches.

Supported Activity Diagram Elements. By utilizing the fUML reference implementation, the prototype supports all model elements of activity diagrams depicted in Table 4.1. The fUML reference implementation provides support for all model elements defined within the fUML standard and therefore supports more model elements than those supported by the prototype. The main reason for this discrepancy in supported model elements is that EA does not support the modeling of a number of specific action types.

Executor Integration. As discussed in Section 2.3, the AMUSE plugin incorporates an *ExecutionEnvironment* component that instantiates separate *Executors* for each behavioral diagram, with specific types of *Executor* available for each behavioral diagram type. The main goal in implementing the prototype was therefore to replace the proprietary *Executor* type AMUSE uses for activity diagrams with a custom implementation that uses the fUML reference implementation for executing the activity diagram, which is referred to as the "fUML executor". The fUML executor must provide the following functionalities to correctly fulfill the specification of an *Executor* component within the AMUSE plugin:

- Execution of an activity diagram with a given context object and a list of parameters
- Mechanisms to pause, continue and abort a running execution
- Notifications for both the start and the end of the execution, as well as the entry and exit of a specific action

To enable the fUML reference implementation to execute activity diagrams modeled within EA, both the activity diagrams and the class diagrams of classes instantiated therein must be converted from EA's internal format into a format interpretable by the fUML reference implementation. For this conversion to succeed, the prototype must also be able to determine which specific class diagrams and activity diagrams need to be known to the fUML reference implementation before the execution starts.

All instances available within the AMUSE object space during the model execution also have to be made available to the fUML reference implementation and vice versa. This includes, for instance, objects instantiated by an activity diagram call operation. For this purpose, the fUML executor must provide a mechanism to convert instances in one of the format into a usable

Model Element	**Model Element Type**
Create object action	Action
Value specification action	Action
Add structural feature value action	Action
Call behavior action	Action
Send signal action	Action
Read self action	Action
Read structural feature action	Action
Remove structural feature value action	Action
Reduce action	Action
Input pin	Object node
Output pin	Object node
Activity parameter node	Control node
Initial node	Control node
Final node	Control node
Fork node	Control node
Join node	Control node
Decision node	Control node
Merge node	Control node
Expansion region	Structured acivity node
Control flow	Activity edge
Object flow	Activity edge

Table 4.1: Model elements of activity diagrams supported by the prototype

representation in the other format. This also involves instantiating objects in both formats, which requires a substantially different approach in each object space due to the usage of both model interpretation and code generation. To provide constant synchronization of the respective object spaces, the prototype must monitor both object spaces for any occurring changes originating from model executions, and immediately replicate any changes in the other object space.

To provide the necessary functionalities for controlling the currently running executions, the prototype must expose the corresponding functionalities of the MOLIZ fUML Debug API. Furthermore, the fUML executor must propagate an event both at the start and end of the execution of the activity diagram, as well as at the entry and exit of a model element. The event fired at the entry and exit of a model element needs to contain a reference to the correct EA model element. The MOLIZ fUML Debug API allows the prototype to observe the corresponding events for element entry and exit during the execution in the fUML reference implementation. However, it is necessary to determine which element in the EA model corresponds to the element that has been entered or exited during the execution in the fUML reference implementation. For this purpose, a complete mapping between all EA model elements and their representations in the fUML reference implementation must be maintained by the prototype.

4.1.2 External Tools

A number of external tools are used by the prototype created for this thesis. This section specifically discusses how these tools are utilized in implementing the prototype, as well as the used versions of the tools. For a more detailed explanation of the tools, please refer to Section 2.3.

The UML modeling tool EA is used for creating the models used for the model execution. The prototype has been evaluated with version 10 of EA. The AMUSE plugin requires at least version 7.5.850 of EA. Every license version of EA is compatible with the AMUSE plugin, and therefore also compatible with the prototype.

The EA plugin AMUSE is used and modified for conducting the model execution. The prototype is based upon version 2.5 of the AMUSE plugin.

The MDE plugin for EA is used to simplify the access to the EA database and to access the data stored therein via UML compliant data structures. The prototype uses a development build of the MDE plugin that is deployed as part of AMUSE 2.5. The used version for the prototype is the version shipped with the AMUSE version 2.5.

The fUML reference implementation is used for executing activity diagrams from within the AMUSE plugin. The prototype uses a version of the fUML reference implementation based on fUML 1.1.0.

The MOLIZ fUML Debug API developed by Mayerhofer et al. is used as an extension of the fUML reference implementation. This API is used to enable the stepwise execution of activity diagrams via the fUML reference implementation, and to continuously monitor the execution state. The prototype uses a development build of the MOLIZ fUML Debug API, which was downloaded on August 3, 2014.

4.1.3 Expected Results

By only replacing one specific *Executor* implementation of the AMUSE plugin, the prototype implemented for this thesis is able to transparently execute activity diagrams using the fUML reference implementation, while leaving the remaining functionality of the execution environment of AMUSE unchanged. The prototype provides a complete implementation of the *Executor* component, including both commands and events. Therefore, all of the model execution features of AMUSE are also applicable to the execution of activity diagrams via the newly implemented *Executor*, and therefore to the execution of activity diagrams via the fUML reference implementation.

Firstly, this allows the *ControlPanel* component of AMUSE to control the execution of activity diagrams via the fUML reference implementation. The *ControlPanel* component is able to both execute the activity diagram stepwise, as well as set breakpoints on the execution.

Secondly, the *Visualization* component is able to correctly display the execution of activity diagrams based on the events propagated by the fUML executor. This includes marking the currently active model element, as well as displaying the diagram that is associated with the currently running execution.

4.2 fUML Executor Implementation

The fUML executor is the main component of the prototype developed for this thesis.

As discussed in Section 2.3, the name *Executor* in the context of the AMUSE plugin refers to a component which is instantiated by the *ExecutionEnvironment* component of AMUSE, and which is responsible for the execution of a specific behavioral diagram. The role of the fUML executor is to replace the proprietary *Executor* implementation AMUSE instantiates for executing activity diagrams with an implementation that conducts an fUML compliant execution of activity diagrams via the fUML reference implementation. Figure 4.1 provides a high-level overview of the role the fUML executor fulfills. The figure depicts both the main functionality provided by the fUML executor to AMUSE, as well as the functionality provided to the fUML executor by other components.

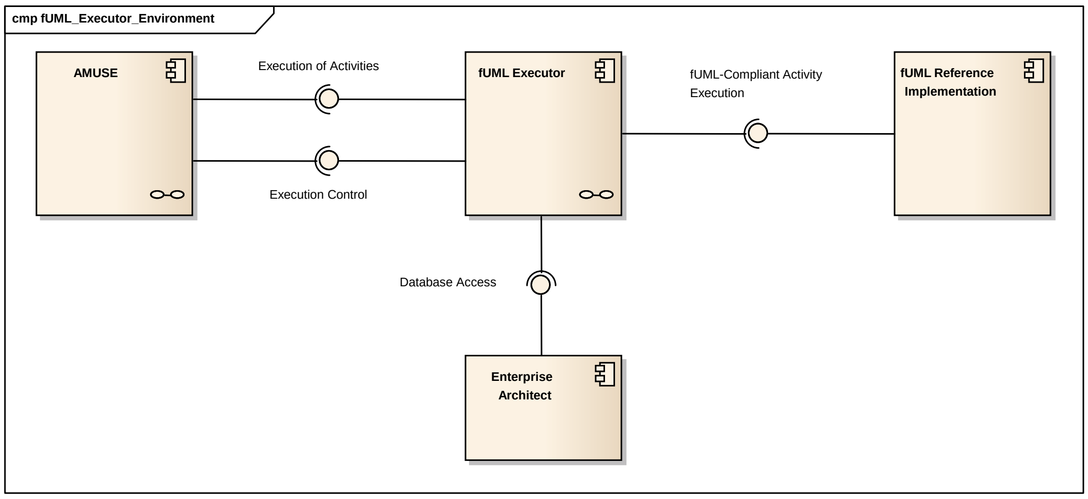


Figure 4.1: Environment of the fUML executor

The main functionality provided by the fUML executor is the execution of an fUML compliant activity diagram modelled in EA. The fUML executor does not directly implement the fUML compliant execution of an activity diagram. Instead, it utilizes the fUML reference implementation to conduct the execution. For this purpose, it loads the models from the EA database and converts them into fUML models.

The fUML executor is composed of separate components, each of which fulfills a specific purpose in the context of the model execution. These components are depicted in Figure 4.2. Note that, while AMUSE may instantiate multiple instances of the fUML executor itself to execute multiple activity diagrams in parallel, these components are shared by all instances of the fUML executor. Figure 4.3 provides an overview of the specific functionalities of each of the components of the fUML executor and will be referenced throughout the following sections.

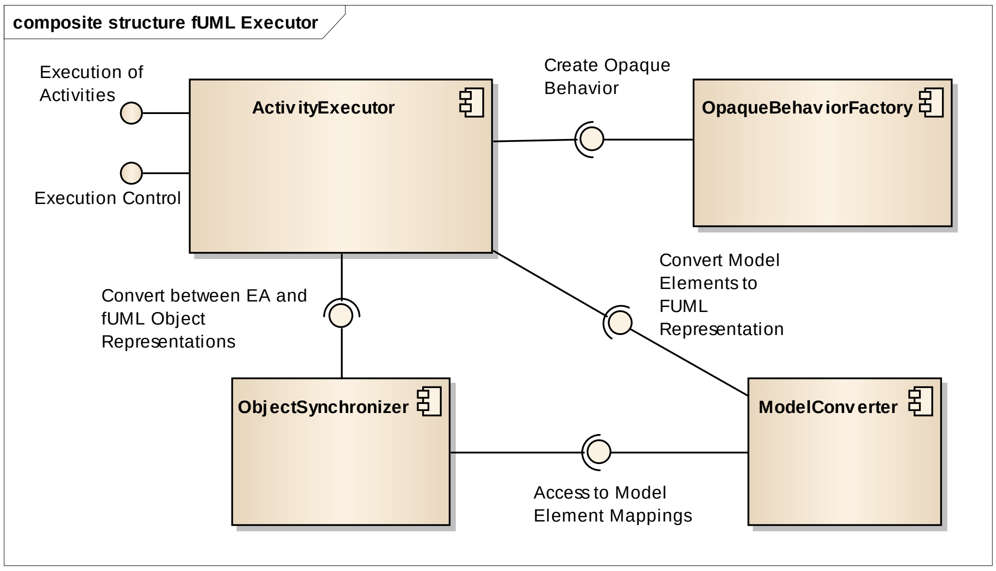

composite structure fUML Executor
Execution of Activities
Execution Control
ActivityExecutor
Create Opaque Behavior
OpaqueBehaviorFactory
Convert between EA and fUML Object Representations
Convert Model Elements to FUML Representation
ObjectSynchronizer
ModelConverter
Access to Model Element Mappings

Figure 4.2: Components of the fUML executor

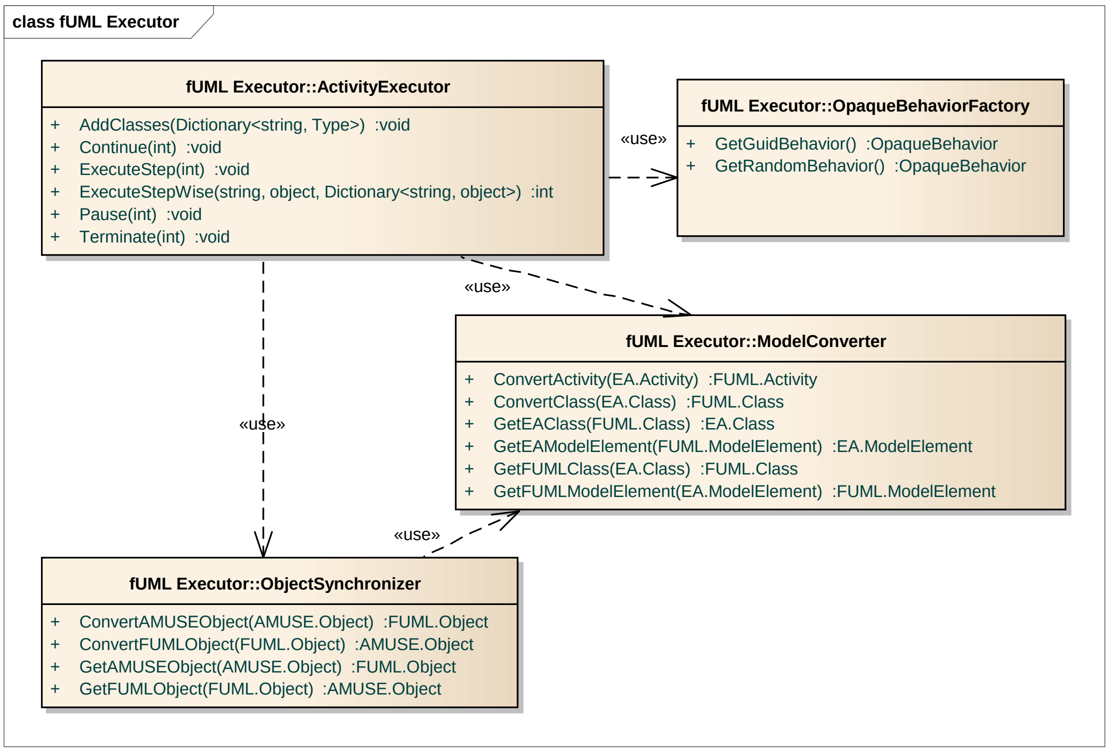

class fUML Executor
fUML Executor::ActivityExecutor
+ AddClasses(Dictionary<string, Type>) :void
+ Continue(int) :void
+ ExecuteStep(int) :void
+ ExecuteStepWise(string, object, Dictionary<string, object>) :int
+ Pause(int) :void
+ Terminate(int) :void
«use»
fUML Executor::OpaqueBehaviorFactory
+ GetGuidBehavior() :OpaqueBehavior
+ GetRandomBehavior() :OpaqueBehavior
«use»
fUML Executor::ModelConverter
+ ConvertActivity(EA.Activity) :FUML.Activity
+ ConvertClass(EA.Class) :FUML.Class
+ GetEAClass(FUML.Class) :EA.Class
+ GetEAModelElement(FUML.ModelElement) :EA.ModelElement
+ GetFUMLClass(EA.Class) :FUML.Class
+ GetFUMLModelElement(EA.ModelElement) :FUML.ModelElement
«use»
«use»
fUML Executor::ObjectSynchronizer
+ ConvertAMUSEObject(AMUSE.Object) :FUML.Object
+ ConvertFUMLObject(FUML.Object) :AMUSE.Object
+ GetAMUSEObject(AMUSE.Object) :FUML.Object
+ GetFUMLObject(FUML.Object) :AMUSE.Object

Figure 4.3: Functionalities of the components of the fUML executor

The *ActivityExecutor* component provides the activity diagram execution facilities within the fUML executor, and utilizes the remaining components to conduct the activity diagram executions. The implementation of this component is presented in Section 4.2.1.

Both AMUSE and the fUML reference implementation use different representations of model elements. The fUML executor loads the diagrams stored within EA, and converts them into a format which is interpretable by the fUML reference implementation. This representation is then executed by the fUML reference implementation. This conversion is conducted by the *ModelConverter* component and explained in Section 4.2.2. Furthermore, both AMUSE and the fUML reference implementation have separate object spaces, each which their own respective formats for objects. The fUML executor synchronizes both of these object spaces so that AMUSE has access to the object that reside on the locus of the fUML reference implementation and vice versa. This synchronization is conducted by the *ObjectSynchronizer* component. The implementation of this component is explained in Section 4.2.3.

The execution of an activity diagram via the fUML executor is started immediately and continuously. However, the fUML executor also provides a number of operations for controlling a currently running execution, which are also provided by the *ActivityExecutor*. These operation include pausing and resuming the execution, or setting stopping points on specific model elements via breakpoints. We discuss the implementation of these features in Section 4.2.4.

The fUML executor propagates specific information about currently running model executions. Firstly, every change of the status of the execution is propagated, e.g., the start of the execution. Additionally, the fUML executor indicates whenever the execution of a specific model element is started or stopped. All of this information is both collected and propagated by the *ActivityExecutor* component. We discuss this functionality in Section 4.2.5.

Lastly, in addition to executing diagrams modelled directly within EA, the fUML executor provides a number of additional opaque behaviors which may be utilized within the diagrams modelled within EA. These opaque behaviors provide specific pre-implemented behaviors in addtion to the opaque behaviors available via the fUML reference implementation, such as generating a unique identifier. The *OpaqueBehaviorFactory* component manages the instantiation of these additional opaque behaviors. The implementation of this component is presented in Section 4.2.6.

4.2.1 Execution of Activities

The fUML executor invokes the *ActivityExecutor* component to conduct the execution of a specific activity diagram. Similarly to the fUML reference implementation itself, there is only a single *ActivityExecutor* instance which conducts and coordinates all the concurrent executions of activity diagrams which are requested by the different fUML executor instances. Additionally, the *ActivityExecutor* component provides functionalities to give commands to and monitor the status of all running executions.

The *ActivityExecutor* component represents a clear interface between the AMUSE and fUML representations of both model elements and objects. All parameters passed to the *ActivityExecutor* component are given in AMUSE's representations of model elements and objects. Internally, the *ActivityExecutor* component converts these model elements and objects into a format interpretable by the fUML reference implementation by utilizing the *ModelConverter* component and the *ObjectSynchronizer* component, conducts the execution of the activity diagram and transforms all feedback during the execution via events into a format interpretable by AMUSE.

When invoking the execution of a specific activity diagram by the *ActivityExecutor* component via the *ExecuteStepWise* operation, the fUML executor must provide a number of parameters:

- The **unique id** of the activity diagram, which is required to load the model from the EA database.
- The **context object**
- A list of **input parameters** values

All of this information is provided to the fUML executor by the execution environment of AMUSE.

Assigning input parameters when executing an activity via the fUML execution environment presents a specific challenge, as AMUSE and fUML identify specific parameters differently. In AMUSE, parameters are referenced by name, while in fUML, one needs to refer to a specific parameter by reference. Therefore, a strict one-to-one mapping can not be automatically accomplished by the prototype. However, each of the parameters in fUML specifies a name, which is assumed to be unique in the context of the activity it is defined in. Based on this assumption, the prototype performs a name-based matching of the parameters and provided input parameter values accordingly.

The execution of an activity diagram conducted by the *ActivityExecutor* consists of a multi-step process:

1. Load EA representation of the activity diagram which should be executed from the database.
2. Convert the loaded EA representation of the activity diagram into an fUML-interpretable format, using the *ConvertActivity* operation of the *ModelConverter* component.
3. Convert the context object, as well as all received input parameter values into an fUML-interpretable format, using the *ConvertEAObject* operation *ObjectSynchronizer* component.
4. Start the execution of the activity diagram on the fUML reference implementation via the MOLIZ fUML Debug API.
5. Perform a stepwise execution of the activity diagram via the MOLIZ fUML Debug API until the execution is completed.

Note that the fUML executor only invokes the execution of a single activity diagram, which may in turn invoke the execution of several additional activity diagrams. Should the execution encounter an additional activity diagram, the entire process described above is repeated for this activity diagram before the execution continues.

Additionally, note that the execution process for an activity diagram does not involve determining or converting the required classes for representing the instances used during the execution. Instead, the classes are loaded separately during the initialization of the execution environment of AMUSE.

The MOLIZ fUML Debug API assigns a unique id to each invoked execution of an activity diagram. Whenever a fUML executor wants to interact with a specific execution via the *ActivityExecutor* component, it must provide the unique id which has been assigned to this execution. Furthermore, all events generated by the MOLIZ fUML Debug API carry the unique id that has been assigned to the execution from which the event originates. Therefore, the *ActivityExecutor* returns the unique id of the invoked execution to the fUML executor. The fUML executor stores this unique execution id so that it is able to invoke specific control commands for this execution, which are discussed in the Section 4.2.4.

4.2.2 Conversion of Model Elements

The *ModelConverter* component provides functionality for converting a model defined within EA into a format interpretable by the fUML reference implementation. It provides both the *ConvertActivity* operation for converting an entire activity diagram, as well as the *ConvertClass* operation for converting classes. Before we explain these conversions, we discuss the conversion method chosen in the prototype implementation.

Conversion Method

For realizing the conversion of class diagrams and activity diagrams created with EA to fUML, we identified 4 approaches. In the following, we discuss these approaches, including their advantages and disadvantages. Subsequently, we elaborate on the chosen approach for implementing our prototype.

XMI Export. The arguably most straightforward approach would be to utilize EA's XMI export for the integration of the fUML reference implementation. This method would allow us to use the interface given by the fUML reference implementation. Thereby, the amount of supported functionality would be determined by the extensiveness and correctness of the supplied XMI input. However, work conducted by LieberLieber in the context of the ARTIST project[1] has shown that the XMI export provided by EA is inconsistent with the standard. Furthermore, due to the additional degree of separation resulting from converting the model elements to and from an XMI file during the conversion process, there is no trivial way of establishing correspondences between the EA model elements and the created fUML model elements, which is critical for the correct operation of the fUML executor.

Own Converter Instantiating fUML Representation Directly. A second option that allows direct control over the conversion for a given EA model is to let the *ModelConverter* directly

[1]`http://blog.lieberlieber.com/2012/11/13/export-your-ea-models-for-eclipse/`, Accessed: 16.08.2014

instantiate the fUML model element representations. By using this method, the *ModelConverter* has total control over all model elements that are instantiated on the fUML execution environment, and can implement any correspondence between the EA model elements and their representations within fUML. Conversely, this approach requires a complete, manual implementation of the mapping mechanism between the EA representation and the fUML representation of the model elements.

Converter Utilizing MOLIZ fUML Debug API. The third option is a slight variation of the second option. The MOLIZ fUML Debug API includes the *ActivityFactory*, which is used for creating instances of commonly used fUML model elements and is used extensively in unit-testing the API itself. The *ActivityFactory* provides features for instantiating all common model elements found in activity diagrams. By utilizing these features, it is possible to instantiate complete and extensively tested representations of activity diagram model elements, while still being able to implement the correspondence between the EA representation and the fUML representation of the model elements. In essence, this approach is very similar to option number two, but reuses functionality developed for the MOLIZ fUML Debug API that would otherwise need to be reproduced. The disadvantage of this approach is that instantiating a model element not covered by the *ActivityFactory* must be achieved by another method, leading to an implementation that requires multiple conversion mechanisms to work in parallel.

Based on the analysis results of the available approaches listed above, the decision was made to implement the creation of fUML objects via option 3, namely by utilizing the *MOLIZ ActivityFactory*. The lack of trivial ways to establish the correspondence between EA and fUML model elements had eliminated the XMI-based approach 1, and the existing quality assurance of utilizing the already unit-tested methods of the *ActivityFactory* was the main reason to implemented option 3 instead of option 2.

An important consideration when choosing between the aforementioned approaches are the performance penalties incurred by each one. A key difference here is that the first approach leaves the initialization to the fUML reference implementation (by only passing in XMI information), while the other approaches build up the required model element structures themselves, and pass them to the *Executor* of the fUML reference implementation directly. However, at this moment, no direct performance comparisons between these model element conversion mechanisms has been conducted due to the fact that the direct-XMI approach is not usable for our prototype because of the discussed disadvantages. The performance difference is assumed to be not significant enough to warrant the extra implementation effort to utilize this approach in the context of the prototype.

Conversion of Classes

The conversion of a class conducted by the *ConvertClass* operation of the *ModelConverter* component involves creating a *Class* instance in the fUML model and initializing its associations and generalization relationships. Most of the properties can be directly mapped from the EA representation to the fUML representation. The only conversion that needs to be explicitly done by the *ModelConverter* component is to determine the corresponding primitive type that should be given to the fUML attribute based on the type information assigned in the EA representation.

Due to the fact that EA supports a number of types for attributes that are not available in the fUML reference implementation, multiple EA types are currently not supported by the prototype. This includes, for example, the *byte* type, which has no direct representation within the fUML reference implementation.

The conversion of all classes required for the execution of a particular activity diagram is directly invoked by the *ExecutionEnvironment* component of AMUSE during its own initialization and before any actual execution of a behavioral diagram is conducted. The *ExecutionEnvironment* component determines all classes that are relevant for the *ActivityExecutor*. This includes both class of the executed instance, as well as all classes that are either linked to this class or one of its behavioral diagrams. Such a link may be represented by an association between the classes, or the additional class may be used as a classifier in one of the behaviors that are associated with the class of the executed instance.

Conversion of Activities

The conversion an activity diagram from its EA representation to an fUML interpretable format conducted by the *ConvertActivity* operation of the *ModelConverter* component starts by creating an *Activity* instance, and then subsequently generating fUML representations of the nodes contained within the activity diagram. During the conversion of a single activity diagram, no further linked activity diagrams are considered for conversion. Instead, the conversion of additional activity diagrams is conducted only when these particular activity diagrams are executed. With this conversion approach, only activity diagrams that are actually executed are converted, which potentially avoids unnecessary conversions in larger models. The results of each conversion are cached by the *ActivityExecutor* component. Therefore, multiple executions of the same activity diagram do not lead to multiple conversions.

Pins. The conversion of most available types of nodes from their EA representation to their fUML representation involves a simple one-to-one mapping of the respective properties. However, during the conversion of action nodes, the conversion of their pins needs to be handled separately. Each of the action types available in UML has a specific configuration of input and output pins. An action holds separate variables for each pin, e.g., the implementation of the *AddStructuralFeatureValueAction* holds an input pin named *object* and a output pin named *result*. During the conversion process, the pins need to be converted into their fUML representation and then assigned to these separate variables.

Expansion Region. The only element that may be contained within an activity diagram that needs to be handled separately is the expansion region. The expansion region defines a kind of "sub-activity" within an activity diagram. This sub-activity is invoked for a collection of instances, and is run once for each instance in the collection. When converting such an expansion region, all model elements contained therein need to be assigned explicitly as part of the expansion region itself, and not as part of the parent activity. This enables the fUML reference implementation to determine which nodes need to be carried out in the context of the expansion region. An additional issue needs to be considered when converting expansion regions. EA does not offer the possibility to mark input and output expansion nodes of an expansion region as such. Only a generic expansion node type is provided. However, in the context of the fUML reference implementation, this distinction is required for the execution to work correctly. Therefore, each expansion node of an expansion region modeled within EA must be marked either with the "Input" or "Output" stereotype to indicate which type of node it represents. These specific stereotypes are then considered when converting the nodes attached to an expansion region.

Mapping

Each model element might potentially need to be converted multiple times with no possible distinction between the first and following conversion requests. Therefore, every time a model element is converted from the EA representation to the fUML representation, a correspondence between the two repesentations is stored in a cache held by the *ModelConverter* component. This serves a dual purpose: Firstly, it results in a considerable performance increase. For model elements that have already been converted, the *ModelConverter* component is able to retrieve the resulting fUML model element from the previous conversion and the conversion does not need to be repeated. Secondly and more importantly, multiple conversions would result in possible problems with the referential integrity of the system. Each model element is converted separately during the conversion of an activity diagram. However, multiple EA model elements may contain a reference to the same model element. In each scenario in which the EA model elements A and B reference model element C, it is vitally important that the fUML representations of A and B also reference exactly the fUML representation of C. This property is ensured by the caching mechanism — whenever model element C is referenced a second time during the conversion of the activity diagram, the cache returns a reference to the already existing fUML representation of C.

The *ModelConverter* component provides separate operations for querying the maintained cache. These operations are used for determining which EA model element has been mapped to which fUML model element. This type of lookup is used, for example, when converting fUML events to their AMUSE representation, or during the synchronization of the object spaces. For this purpose, the *ModelConverter* component provides the following operations:

- *GetEAClass* retrieves the EA class an fUML class is based on.
- *GetFUMLClass* retrieves the fUML class an EA class has been converted into.
- *GetEAModelElement* retrieves the EA model element an fUML model element is based on.
- *GetFUMLModelElement* retrieves the fUML model element an EA model element has been converted into.

4.2.3 Synchronization of Object Spaces

Both AMUSE and the fUML reference implementation hold separate object spaces, which are data structures in which they create and modify objects. As these object spaces are independent of one another, the execution environments are unable to utilize objects that are stored in the object space of the other execution environment. The *ObjectSynchronizer* provides mechanisms to synchronize these object spaces, thereby allowing each execution environment to interact with objects which have not been originally created within their own object space. For this purpose, the *ObjectSynchronizer* component constantly monitors the object spaces of AMUSE and the fUML reference implementation to detect the creation or modification of objects. Whenever either the creation of a new object or the change of a feature value of an object is detected, the operation to update the object in the other object space is invoked. Depending on the object space in which the change was detected, either the *ConvertAMUSEObject* operation is invoked to convert the AMUSE object to an fUML object, or the *ConvertFUMLObject* operation is invoked to convert the fUML object into an AMUSE object, which is automatically added to the target object space. Note that the mapping mechanism employed by the *ObjectSynchronizer* component ensures that each AMUSE object is represented by one fUML object and vice versa. Should the conversion be triggered multiple times for the same object, only the feature values are newly propagated.

Creating New Objects

Creating new objects in the respective object spaces requires considerably different approaches, which is mostly due to the different representations of objects within the respective object spaces. The differences between the representations of objects in both the AMUSE and the fUML object space have been discussed in Section 2.2. The main difference is that AMUSE creates objects by instantiating generated classes, while objects in fUML hold generic objects of type *Object*.

The process of creating a new fUML object starts with creating a new *Object* instance at the locus. This new object has no connection to a class and no feature values. In the next step, the fUML class instantiated by this object is determined by invoking the *GetFUMLClass* operation of the *ModelConverter* component. This fUML class is then used as a basis for generating the necessary feature values of the created object.

When creating an AMUSE object based on an existing fUML object, the first step is to determine and load the correct class to be instantiated for the AMUSE object. The *ObjectSynchronizer* component first determines the instantiated fUML class, which is referenced by the fUML object it is trying to convert. Based upon this fUML class, the *ObjectSynchronizer* component determines the corresponding class EA class by performing via the *GetEAClass* operation of the *ModelConverter* component. For instantiating this class, it accesses the C# implementation of this class that has been generated by the *ExecutionEnvironment* component of AMUSE. For this purpose, the *ExecutionEnvironment* component provides a correspondence of the modeled EA classes to their generated and compiled implementations to the *ActivityExecutor* component via the *AddClasses* operation before the model execution commences, which subsequently passes it on to the *ObjectSynchronizer* component. After the correct C# class has been determined, the AMUSE object can be created by instantiating it.

As soon as the instantiation of the object has been completed, the *ObjectSynchronizer* can conduct the propagation of the feature values from the existing object to the newly created object.

Propagating Feature Values of Objects

When propagating the feature values of an AMUSE object to an fUML object, the first step is to extract the values stored in the attributes of the AMUSE object based on the attribute definitions contained in the C# class. As these attributes are not known at compile-time, they need to be accessed via reflection[2]. Reflection in the C# programming language enables accessing attributes of objects based on their names. After the EA feature values have been determined, they need to be converted to an fUML value specification. Based on the type of the attribute within the instance of the fUML class, a specific sub-type of the *ValueSpecification* class needs to be instantiated, which holds the primitive value of the feature value of the AMUSE object. The last step is to assign the resulting value specification to the feature value of the fUML object.

Similarly, the transfer of attributes from the fUML object to the AMUSE object starts by extracting the values of all structural features of the fUML object based on the fUML class. Afterwards, these values are converted to their EA representation. Lastly, the resulting EA representations of the EA feature values are assigned to the attributes of the AMUSE object via reflection.

Each feature value may consist of a set of primitive values. These primitive values require no further conversion, as they are represented equally in both object spaces. Note that, due to the C# implementation of the classes AMUSE uses to instantiate objects, structural feature that are modelled to only include a single feature value cannot be extended by additional feature values. Therefore, should additional feature values be added to such an object in fUML, this change cannot be propagated to the AMUSE object space.

Additionally, the feature value may be connected to different feature values via *links*, which are tuples of values that represent a particular association. Changes within these links are also propagated between the object spaces.

Mapping

Whenever the *ObjectSynchronizer* component creates a fUML object based on an AMUSE object or vice versa, it stores a correspondence between these object in a cache.

This serves a dual purpose. Firstly, it increases the performance of the prototype by avoiding multiple conversions. Secondly, the caching mechanism ensures that each object in the AMUSE object space is only represented by a single object in the fUML object space and vice versa. *ObjectSynchronizer* component provides the following operations for accessing the cached correspondences:

- *GetAMUSEObject* retrieves the AMUSE object an fUML is connected to.
- *GetFUMLObject* retrieves the fUML object an AMUSE is connected to.

Detecting Changes in Object Spaces

The *ObjectSynchronizer* immediately propagates all changes it detects in one object space to the other object space.

[2]`http://msdn.microsoft.com/en-us/library/ms173183.aspx`, Accessed: 16.08.2014

Changes within the fUML object space can be easily detected. The MOLIZ fUML Debug API triggers events that inform about any changes that have occured within the locus of the fUML execution environment. These events also include information on creation or modification of either an object or a link. Therefore, changes in the fUML object space can be immediately propagated to the AMUSE object space by invoking the *ConvertFUMLObject* operation of the *ObjectSynchronizer* component for the modified objects.

Detecting changes within the object space of AMUSE is not trivially possible. The main instance executed by the plugin has individual events that inform the *ExecutionEnvironment* about any changes in attributes, which are then visualized by the *ControlPanel* component. These events are not propagated to the individual *Executors*, however. Furthermore, additional objects created by the AMUSE plugin do not propagate events for changed feature values at all. This means that changes in the object space of AMUSE can not be immediately propagated to the fUML object space. The fUML executor is only able to detect changes of objects of the AMUSE object space once it uses them for its own execution. For this purpose, the *ActivityExecutor* component automatically invokes the *ConvertEAObject* operation for the context object as well as all input parameter values on invocation of the *ExecuteStepWise* operation. Thereby, the objects within the fUML object space are always correctly updated before an execution.

4.2.4 Controlling the Execution

In addition to providing the functionality for executing activity diagrams, the *ActivityExecutor* component also provides mechanisms to control the currently running executions. The following features are available:

- Execute a single step of the execution of an activity diagram via the *ExecuteStep* operation
- Pause the execution via the *Pause* operation
- Resume the execution via the *Resume* operation after a pause
- Abort the execution via the *Terminate* operation

The definition of breakpoints on which the execution stops automatically is implemented directly in the AMUSE execution environment and therefore also supported by the prototype. Whenever the execution reaches a model element with a defined breakpoint, AMUSE automatically invokes the *Pause* operation of the *ActivityExecutor* via the fUML executor, and invokes the *Resume* operation when the user wants to resume the execution.

The implementation of all of these execution control features within the *ActivityExecutor* component is straightforward, as all of the execution control functionalities are directly provided by the MOLIZ fUML Debug API. The fUML executor invokes the corresponding functionalities of the API when asked to pause, resume or abort an execution. Each of these features is invoked by the fUML executor that has invoked the specific execution, which also must provide the corresponding unique execution id.

4.2.5 Propagation of Events and Signals

The fUML executor is able to propagate all major event events that are produced by the MOLIZ fUML Debug API. This includes:

- Start and end of the execution of an activity
- Start and end of the execution of an action
- Events pertaining to objects residing at the locus (e.g., the creation or modification of an object).

However, the fUML executor does not directly propagate the events received by the MOLIZ fUML Debug API. Instead, re-implementations of all of these events are provided, which contain references to AMUSE objects instead of fUML objects. To determine the corresponding AMUSE object, the fUML executor performs a reverse-lookup of the EA object in the cache of either the *ModelConverter* or the *ObjectSynchronizer* component.

Similarly, the fUML executor does not directly propagate signals sent by the fUML reference implementation. Whenever a signal is sent within the fUML reference implementation, the fUML executor determines the corresponding target object within the AMUSE object space and also sends a signal to this AMUSE object with the signal mechanism of AMUSE. Should the original signal contain parameters which reference fUML objects, the corresponding AMUSE object are determined via the *GetAMUSEObject* operation of the *ObjectSynchronizer* component and given as parameters for the propagated signal.

4.2.6 Implementation of Additional Opaque Behaviors

A core feature of fUML is the model library, which provides opaque behaviors. These opaque behaviors are stored at the locus and can be executed by invoking them via call behavior actions.

To further explore the possibilities of directly extending the fUML reference implementation, the fUML executor incorporates additional opaque behaviors. The implementation of these behaviors works by directly extending the original *OpaqueBehaviorExecution* class provided by the fUML reference implementation (the implementation of these opaque behaviors is discussed in more detail in Section 4.3.4). Via this mechanism, the behaviors described in Table 4.2 are provided as part of the prototype. The prototype currently only provides a very limited set of additional opaque behaviors. Each of the implemented opaque behaviors has been built specifically because one of the evaluation scenarios requires the provided functionality.

Both the original opaque behaviors provided by the fUML reference implementation, as well as the additional opaque behaviors implemented within the prototype are hardcoded within the respective implementation. Therefore, these opaque behaviors cannot be directly referenced when creating a model within EA. Instead, each opaque behavior is invoked by declaring an atomic action in the EA model (an atomic action in the context of EA is an action without a specific action type) and assigning a particular stereotype to this atomic action. The stereotypes required for executing the opaque behaviors provided by the prototype are also listed in Table 4.2.

Opaque Behavior	**Action Stereotype**	**Description**
GuidBehaviorExecution	Guid	Generates a new Global Unique Identifier and returns it.
RandomBehaviorExecution	Random	Returns a single, randomly selected value from all input values.
ReduceAddBehaviorExecution	ReduceAdd	Returns the sum of an arbitrary list of numeric input values. Used as a replacement for the reduce action, which cannot be modelled within EA.
ReduceCountBehaviorExecution	ReduceCount	Returns the count of an arbitrary list of numeric input values. Used as a replacement for the reduce action, which cannot be modelled within EA. This opaque behavior is an alternative implementation of the *ListSize* opaque behavior of the fUML reference implementation.

Table 4.2: Opaque behaviors provided by the fUML executor

Discussion of Including Opaque Behaviors in EA

The implementation of the opaque behaviors provided by the prototype at the moment is hard-coded and offers only a limited set of behaviors, with no possibility for the user to include other behaviors or to modify the existing ones. A far more accessible approach would be to include the existing opaque behaviors within the AMUSE plugin and enable the user to refer to them via call behavior actions when needed.

Because of the direct access to the EA database, the prototype would easily be able to inject completed opaque behaviors into a model. However, defining a clear interface with which to describe how the opaque behaviors should be represented in the model is not trivial. Ideally, the models for the opaque behaviors would be implementable as regular activity diagrams and which are indicated as opaque behaviors via a stereotype application. Then, the opaque behavior could be re-used in the whole model (or even be provided to other EA models).

A second possible approach would be to let the user only define the name and the parameters of the opaque behavior, in conjunction with the code that is run on execution of the opaque behavior. This approach would be more in line with the actual definition of how an opaque behavior is supposed to work according to the UML standard.

However the case, the exploration of these possibilities is out of the scope of this thesis, and they are therefore only mentioned as points for future consideration.

4.3 Integration

The previous section has introduced the fUML executor, a newly developed *Executor* component for the AMUSE plugin which enables AMUSE to execute activity diagrams via the fUML reference implementation. This section covers the necessary steps to fully integrate the fUML executor with both the AMUSE plugin, as well as the fUML reference implementation.

4.3.1 Integration of the fUML Executor into AMUSE's Execution Environment

As has been noted in Chapter 2.3, the AMUSE plugin instantiates separate *Executor* components for executing behavioral diagrams. The plugin is set up so that the *ExecutionEnvironment* component can utilize multiple *Executor* instances together cohesively and transparently. Therefore, the prototype replaces the *Executor* for activity diagrams with the fUML executor, while leaving the rest of the AMUSE plugin in its original state as far as possible.

The replacement of the proprietary activity diagram *Executor* of AMUSE with the fUML executor is largely straightforward. AMUSE uses a factory class for instantiating the executors. This factory class has been modified to instantiate the fUML executor instead of the propriety activity diagram executor. The main challenge of the integration lies within adapting the mechanism AMUSE uses to provide the executor with the required information so that it can correctly perform the execution.

An executor has access to the following generated C# classes for executing an activity diagram:

- C# implementations of all classes that are used during the execution (please refer to Listing 2.1 for an example)
- A C# representation of the specific behavioral diagram to execute

As the execution semantics for activity diagrams supported by AMUSE are limited to control nodes and a single generic action for executing C# code, the executor only requires very limited set of information about the executed activity diagram. The generated code contains a list of objects of type *Node*, which only contain a reference to a piece of C# which they execute as their behavior, as well as separate lists of incoming and outgoing edges. These edges are represented by objects of type *Edge*, which denote objects of type *Node* as their source and their target.

In the following, we discuss the source code generated by AMUSE for the “Check Activity” activity diagram depicted in Figure 4.4, which is part of the *Behavior As Activity* case study introduced in Section 5.1.1.

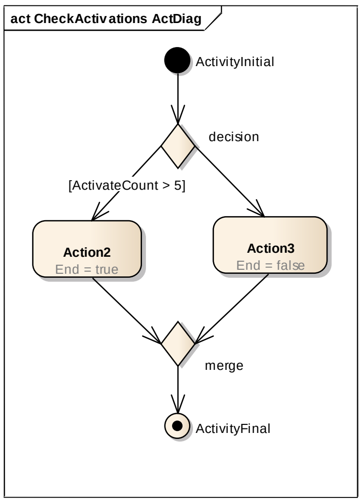


Figure 4.4: Activity diagram “Check Activity” of the *Behavior As Activity* case study

Listing 4.1 displays a part of the code AMUSE generates for the “Check Activity” activity diagram depicted in Figure 4.4. The main component of the generated code is the operation *Init_CheckActivations*, which instantiates an activity object of type *IBehavior* for the execution, as well as the following nodes and activity edges:

- The action node “Action_2” (line 6). The parameters are a reference to the activity, the name of the node, a unique identifier, as well as the type of node.
- A merge node (line 9).
- An edge between the node “Action_2” and the merge node (lines 12 to 15).

All remaining nodes and activity edges are initialized in a similar way.

Additionally, the generated source code contains the operation “Action_node_Action2”, which contains the C# code assigned to the action “Action_2” in the EA model and which will be subsequently executed as its behavior. A similar operation is also generated for the action node “Action_3”. These operation are able to directly manipulate attributes of the context object during the execution.

```
1  public partial class ActivityTest
2  {
3      public IBehavior Init_CheckActivations()
4      {
5          GenericBehavior behavior = new GenericBehavior(this, "
               CheckActivations_Act", "{66DCC393-C70B-43f4-A5E8-0
               C6C31B1265D}");
```

```
        Node node_Action2 = new Node(behavior,"Action2","{
            D6E92D2A-60D9-404a-AAD9-678F3F72DD2C}", NodeTypes.
            Node);
        node_Action2.Action = Action_node_Action2;
        node_Action2.Incoming = 1;
        Node node_merge = new Node(behavior,"merge","{37C4AE93
            -6F89-4059-B81E-267BB3A8AE08}", NodeTypes.Merge);
        node_merge.Incoming = 2;
        //Snipped remaining node initializations
        Edge edge = new Edge("", "{12244833-DA88-4f0e-8211-4
            EFBC5CA6C77}");
        node_Action2.Outgoing.Add(edge);
        edge.Source = node_Action2;
        edge.Target = node_merge;
        //Snipped remaining edge initializations
    return behavior;
    }

    private void Action_node_Action2(object parameter)
    {
        End = true;
    }

    //Snipped behavior operation of node "Action_3"
}
```

Listing 4.1: Example C# code generated to represent an activity diagram

The fUML executor, however, requires significantly more information about the activity diagram than is provided in these generated C# files. The generated C# code needs to provide the following information so that the fUML executor can correctly execute the activity diagram.

- Each type of action must be represented by objects of a specific class that contains the properties of the specific action type, e.g., an object representing a read structural feature action must contain a reference to a structural feature.
- Pins must be represented in the generated code.
- Control nodes must be represented by objects of specific types so that they can be correctly distinguished.
- Parameters must be stored in the generated code.

Multiple possible approaches to overcome this challenge were considered, each with advantages and drawbacks discussed in the following.

The arguably least-intrusive approaches to integrate the fUML executor would be to modify the generated C# code so that it provides all the information the fUML executor requires for the model execution. The advantage of this approach is that the implementation of AMUSE can be left largely intact, resulting in a reduced integration effort. The main challenge of this approach is that the generation of code for the execution of activity diagrams needs to be completely reworked to incorporate the additional required information. Furthermore, the fUML executor only has access to the C# representation of the specific activity diagram it should execute. However, the fUML reference implementation may trigger the execution of additional activity diagrams during the execution of the first activity diagram. Therefore, the fUML executor must be able to access the C# representations of all activity diagrams that may possibly be executed during the execution of the first activity diagram for the purpose of converting them to fUML.

A different approach would be to implement the fUML executor as a purely interpretative component, which does not rely on generated code at all. For this approach to work correctly, the *ExecutionEnvironment* component of AMUSE must be able to provide access the fUML executor with access to the EA database. The main advantage of this approach is that no C# code of the activity diagrams needs to be generated. Instead, the fUML executor can load the the model directly from the EA database. Additionally, this allows the fUML executor to load information at any time during the execution. Conversely, this approach makes the generated autonomous execution environment unable to execute activity diagrams at all. This implies that the autonomous usage of the generated execution environment is no longer possible.

Both approaches constitute a considerably different approach to integrating the fUML executor, and also have major implications on the actual implementation of the fUML executor itself. Ultimately, we have decided to use the second approach and implement the fUML executor as a purely interpretative component with direct access to the EA database. The reason is twofold. Firstly, the purely interpretative approach fits better with the nature of the fUML reference implementation as an execution environment. Secondly, a cursory evaluation of both approaches has shown that adapting the generated C# representation of activity diagrams to fit the requirements of the fUML executor would require significantly more changes to the AMUSE plugin than enabling the fUML executor to access the EA database.

4.3.2 Exchange of Signals

In the context of model execution, signals serve as an important mechanism for communication between behaviors. They are used both to transfer information, as well as possibly advance the execution of a behavior, e.g., because it is waiting for the activation of a trigger. Therefore, the prototype must enable the propagation of signals between the execution environments of fUML and AMUSE.

Propagation of Signals to AMUSE

The fUML executor propagates all signals sent within the fUML execution environment to the corresponding recipient in the AMUSE execution environment. Thereby, the distribution of signals is supposed to take place asynchronously according to the UML standard [21]. However, as discussed in Section 2.2, the semantics of concurrency are left as a semantic variation point in fUML, and consequently, in the fUML reference implementation. Therefore, the transmission of signals is handled in separate threads for each signal, which is also equivalent with the signalling mechanics within AMUSE.

Broadcast Signals via fUML

For sending signals to a defined object, fUML supports the implementation of the send signal action, but not the broadcast signal action. AMUSE, on the other hand, only supports broadcasting signals.

AMUSE broadcasts signals by directing a signal directly to the *ExecutionEnvironment* component itself. The *ExecutionEnvironment* component automatically distributes the signal to all elements known to it, which includes all currently running executors as well as the parent *Simulation*. The *Simulation*, in turn, passes the signal to all known *Sub-Simulations*, which effectively guarantees the distribution of the signal to all known objects in the current execution. The fUML executor incorporates this mechanism by interpreting broadcast signal actions as send signal actions aimed at the *ExecutionEnvironment* component.

4.3.3 Visualization

Large parts of the visualization of the execution of fUML activity diagrams is conducted by utilizing the existing mechanisms of AMUSE. This section discusses the necessary steps for integrating the fUML execution visualization based on the *Package Center* case study presented in detail in Section 5.1.3.

Execution State

As the visualization of the actual execution is strictly based on events pertaining to the entry and exit of a specific behavior or element, the fUML executor needs to properly indicate these events. As discussed in Section 4.2, the fUML executor uses re-implementations of the events propagated by the MOLIZ fUML Debug API to accomplish this. The resulting visualization is shown in Figure 4.5.
The figure shows the visualization of 4 concurrent sub-simulations done by the AMUSE plugin. In each sub-simulation the currently active element is highlighted with a red color. The inactive elements are displayed in blue. Figure 4.5 depicts the following concurrent sub-simulations:

- The top left depicts the execution state of the package center object, which is executing an expansion region within an activity diagram. The single element within the expansion region is active.
- The bottom left depicts the execution state of the post office object, which is starting the execution of an activity diagram. As the execution is processing the initial state, no node is marked as active.
- The top and bottom right depict two separate postman objects. Both of the postmen objects are executing their main state machine, both of which are within the idle state.

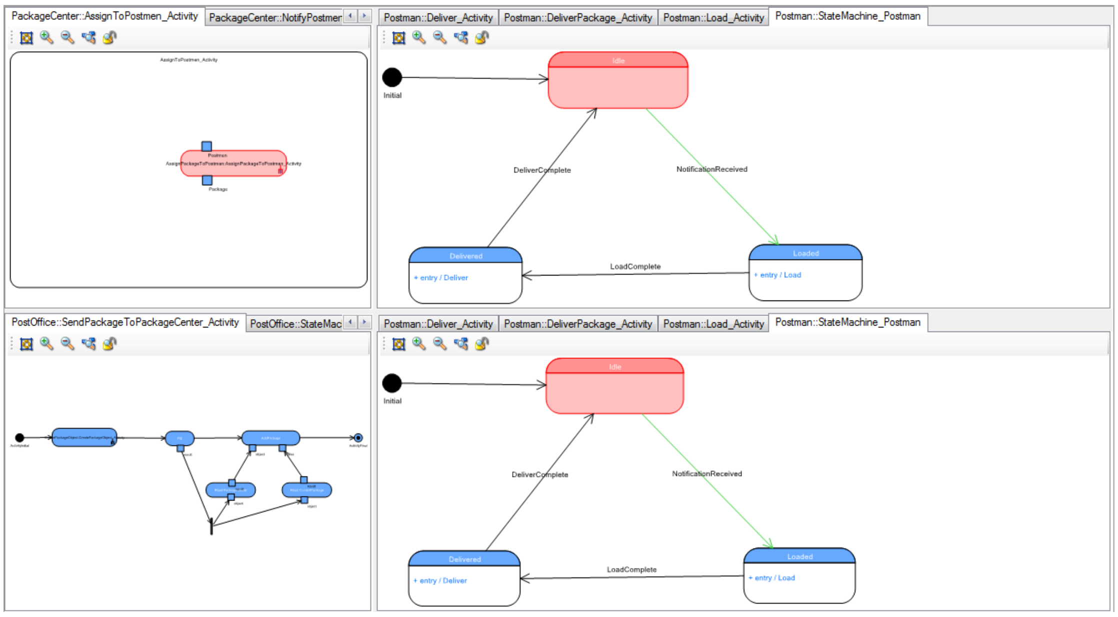

PackageCenter::AssignToPostmen_Activity
PackageCenter::NotifyPostmen
Postman::Deliver_Activity
Postman::DeliverPackage_Activity
Postman::Load_Activity
Postman::StateMachine_Postman
AssignToPostmen_Activity
Postmen
Package
Idle
Initial
DeliverComplete
NotificationReceived
Delivered
+ entry / Deliver
LoadComplete
Loaded
+ entry / Load
PostOffice::SendPackageToPackageCenter_Activity
PostOffice::StateMac

Figure 4.5: Visualisation of the execution state in AMUSE

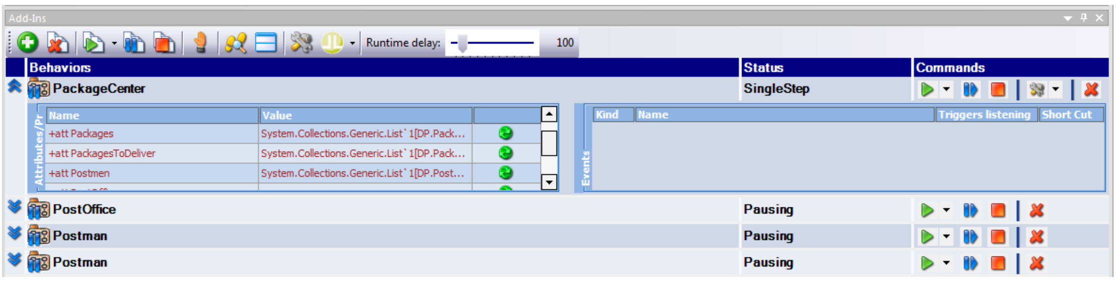
All 4 concurrent sub-simulations are triggered by the start of the simulation of the package center object itself.

Object Space

Aside from visualizing the state of the currently executing behaviors, it is equally important to show the current state of the objects created and modified during the execution. Therefore, the AMUSE simulation control panel provides a periodically updated display of objects maintained by the current execution. As the fUML executor also has access to these objects and might possibly modify them, it has to propagate any object manipulation accordingly. This is accomplished by synchronizing the fUML object space and the AMUSE object space as described in Section 4.2.3.

The visualization of the object space is shown in Figure 4.6. The figure shows the AMUSE control panel, which depicts all currently running executions in a list. When expanding one of these executions, the control panel displays information on all the objects that are present in the object space of this execution.

Furthermore, a new panel has been introduced to the AMUSE visualization for the purpose of providing more detailed information on the objects stored in the object space. Figure 4.7 shows the newly introduced panel. All objects are displayed in a tree-based visualization, which can be expanded to show feature values of the objects as well as links to other objects. The visualization is updated based on the events propagated by the fUML executor, which indicate changes or creations of objects.

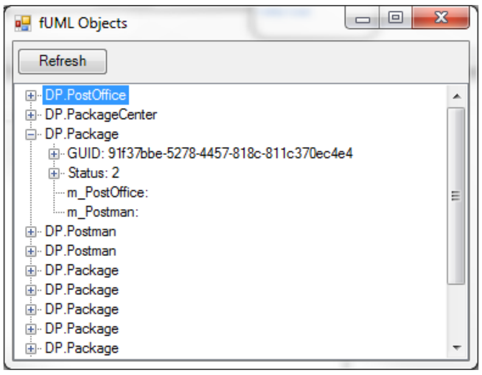


Behaviors	Status
PackageCenter	SingleStep
PostOffice	Pausing
Postman	Pausing
Postman	Pausing

Name	Value
+att Packages	System.Collections.Generic.List` 1[DP.Pack...
+att PackagesToDeliver	System.Collections.Generic.List` 1[DP.Pack...
+att Postmen	System.Collections.Generic.List` 1[DP.Post...

Kind	Name	Triggers listening	Short Cut

Figure 4.6: Visualization of executions and objects in AMUSE control panel

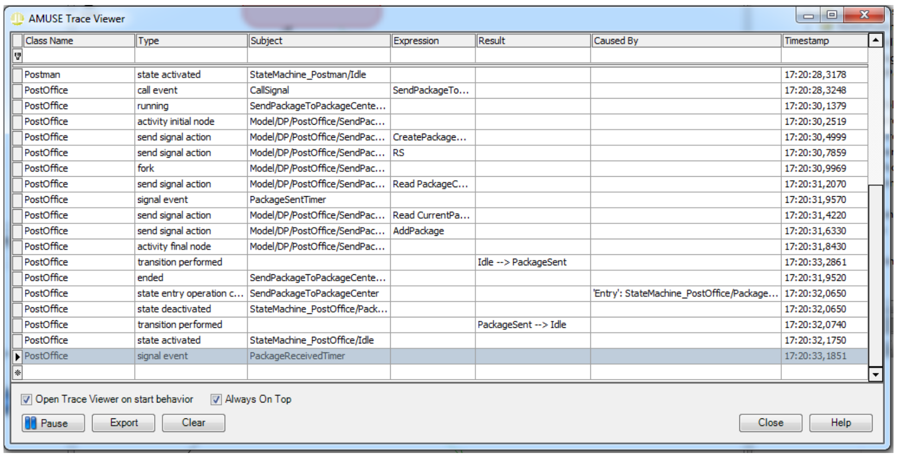


```
DP.PostOffice
DP.PackageCenter
DP.Package
    GUID: 91f37bbe-5278-4457-818c-811c370ec4e4
    Status: 2
    m_PostOffice:
    m_Postman:
DP.Postman
DP.Postman
DP.Package
DP.Package
DP.Package
DP.Package
DP.Package
```

Figure 4.7: Detailed visualization of the object space in AMUSE

Tracing Information

Lastly, the AMUSE framework is also able to provide extensive tracing information for an execution both during the execution itself, as well as in a persistent log file. Due to the utilization of the AMUSE mechanisms for indicating both element activations and object changes, the fUML executor can also be consistently traced. Figure 4.8 shows the trace output when executing the *Package Center* case study.


AMUSE Trace Viewer

Class Name	Type	Subject	Expression	Result	Caused By	Timestamp
Postman	state activated	StateMachine_Postman/Idle				17:20:28,3178
PostOffice	call event	CallSignal	SendPackageTo...			17:20:28,3248
PostOffice	running	SendPackageToPackageCente...				17:20:30,1379
PostOffice	activity initial node	Model/DP/PostOffice/SendPac...				17:20:30,2519
PostOffice	send signal action	Model/DP/PostOffice/SendPac...	CreatePackage...			17:20:30,4999
PostOffice	send signal action	Model/DP/PostOffice/SendPac...	RS			17:20:30,7859
PostOffice	fork	Model/DP/PostOffice/SendPac...				17:20:30,9969
PostOffice	send signal action	Model/DP/PostOffice/SendPac...	Read PackageC...			17:20:31,2070
PostOffice	signal event	PackageSentTimer				17:20:31,9570
PostOffice	send signal action	Model/DP/PostOffice/SendPac...	Read CurrentPa...			17:20:31,4220
PostOffice	send signal action	Model/DP/PostOffice/SendPac...	AddPackage			17:20:31,6330
PostOffice	activity final node	Model/DP/PostOffice/SendPac...				17:20:31,8430
PostOffice	transition performed			Idle --> PackageSent		17:20:33,2861
PostOffice	ended	SendPackageToPackageCente...				17:20:31,9520
PostOffice	state entry operation c...	SendPackageToPackageCenter			'Entry': StateMachine_PostOffice/Package...	17:20:32,0650
PostOffice	state deactivated	StateMachine_PostOffice/Pack...				17:20:32,0650
PostOffice	transition performed			PackageSent --> Idle		17:20:32,0740
PostOffice	state activated	StateMachine_PostOffice/Idle				17:20:32,1750
PostOffice	signal event	PackageReceivedTimer				17:20:33,1851

☑ Open Trace Viewer on start behavior ☑ Always On Top

Figure 4.8: Visualization of tracing information in AMUSE

The trace output that is displayed contains a single entry for each occurence of one of the following events:

- Feature value of an object has been updated
- The execution of a new behavior has started or ended
- The execution of an element of a behavior has started or ended

4.3.4 Integration of the fUML Executor with the fUML Reference Implementation

The MOLIZ fUML Debug API has been used for integrating the fUML reference implementation with the fUML executor. The difference in programming languages between the fUML executor implemented in C# and the MOLIZ fUML Debug API implemented in Java needed to be overcome. This has been accomplished by utilizing the IKVM toolkit to convert the entire MOLIZ fUML Debug API into DLLs, which are then utilized by the fUML executor.

Interestingly, the C# classes contained within the DLLs produced by the IKVM toolkit for the Java API can not only be executed, but also extended by C# classes, as shown in Listing 4.2.

```
public class RandomBehaviorExecution : OpaqueBehaviorExecution
{
    public override Value new_()
    {
        return new RandomBehaviorExecution();
    }

    public override void doBody(ParameterValueList
        inputParameters,
    ParameterValueList outputParameters)
    {
        ValueList valueList = new ValueList();
        int count = inputParameters.getValue(0).values.size();
        int selection = new Random().Next(count);
        Value selectedValue = inputParameters.getValue(0).values.
            get(selection) as Value;
        valueList.addValue(selectedValue);
        outputParameters.getValue(0).values = valueList;
    }
}
```

Listing 4.2: C# implementation of the “Random” behavior

Listing 4.2 shows the definition of the execution semantics of a new OpaqueBehavior, specifically the *RandomBehaviorExecution*, as described in Section 4.2.6. As shown, both the C# classes (*RandomBehaviorExecution*) and the Java classes originating from the fUML reference implementation (*ParameterValueList*, *ValueList* and *Value*) can be used in conjunction with the C# language constructs, such as the random number generation via *new Random().Next(count)* .

Due to utilizing the MOLIZ fUML Debug API implemented in Java via the IKVM toolkit, both debugging and exception handling in the implementation of the fUML executor written in C# are difficult.

In principle, the IKVM toolkit is able to provide detailed debugging information and even enables stepping through the associated Java code. However, as has been discovered during debugging the fUML reference implementation, the debugging capabilities are not fully stable, and cannot be used to completely step through the execution of the fUML reference implementation. This issue is discussed in more detail in Section 5.2.6.

Analyzing exceptions occurring in the Java code can be also be cumbersome, as IKVM can often not determine the exact type of an exception that has occurred in the Java code. Fortunately, IKVM provides a mechanism to print information about the original Java exception, which can be used to determine the exception source based on the original JAVA code.

CHAPTER 5

Evaluation

In Chapter 2, we have introduced the theoretical foundations for integrating fUML with EA, with a special focus on the integration of both different behavioral diagram types and different model execution techniques. Based upon these theoretical foundations, a prototypical integration of fUML within EA has been implemented. This prototype has been discussed in detail in Chapter 4. The created prototype has been extensively evaluated to determine the applicability of the proposed approach of integrating fUML into a proprietary model execution tool. In the evaluation we compared the created prototype to the model execution capabilities of AMUSE. The evaluation has been conducted based on three specific case studies, which are introduced in detail in Section 5.1.

Based upon these case studies, the prototype has been evaluated under multiple aspects, each of which is presented in a separate section within this chapter. Section 5.2 focusses on the model execution functionalities provided by the prototype, and compares them to the functionality AMUSE provides. Section 5.3 presents the results of a performance analysis that has been conducted to compare the two approaches both in terms of execution speed, as well as memory consumption. Section 5.4 presents a general comparison of the two approaches in terms of usability, with a focus on the user interface of EA. Section 5.5 presents an analysis of the stability of the model execution of the two approaches.

Finally, Section 5.6 provides an overall analysis of the presented evaluation results, and provides answers to the research questions proposed in Section 1.3 based upon these results.

5.1 Case Studies

This section presents the main case studies that are used as the basis of the evaluation.

5.1.1 Behavior As Activity

In this case study, we examine one of the example models distributed with the AMUSE plugin.

Figures 5.1, 5.2 and 5.3 illustrate the diagrams contained by this model. Thereby, *Behavior As Activity* refers to the name of this model.

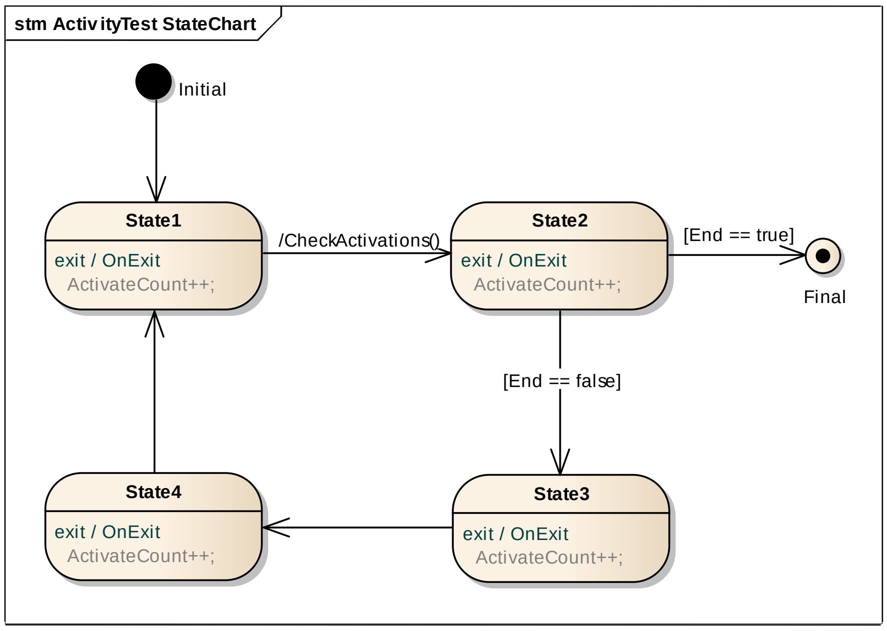


Figure 5.1: State machine of the *Behavior As Activity* case study

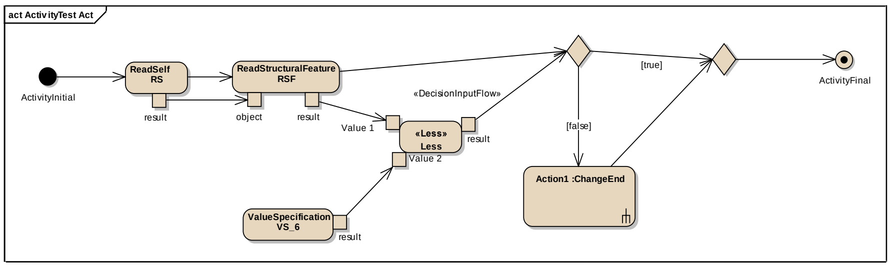


Figure 5.2: Activity diagram *Check Activations* of the *Behavior As Activity* case study

This model represents one of the introductory examples of AMUSE, and therefore only consists of the most basic model elements supported by AMUSE. In the state machine shown in Figure 5.1, each state increases a counter variable “Activate Count” by 1. The transition between states 1 and 2 calls the activity diagram “Check Activations” shown in Figure 5.2 to check if the end condition (a counter variable value above 5) has been met. In this case, the end-variable “End” is set, and the state machine can proceed to its final state.

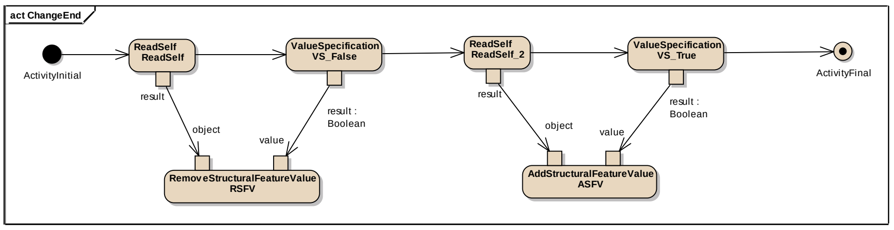
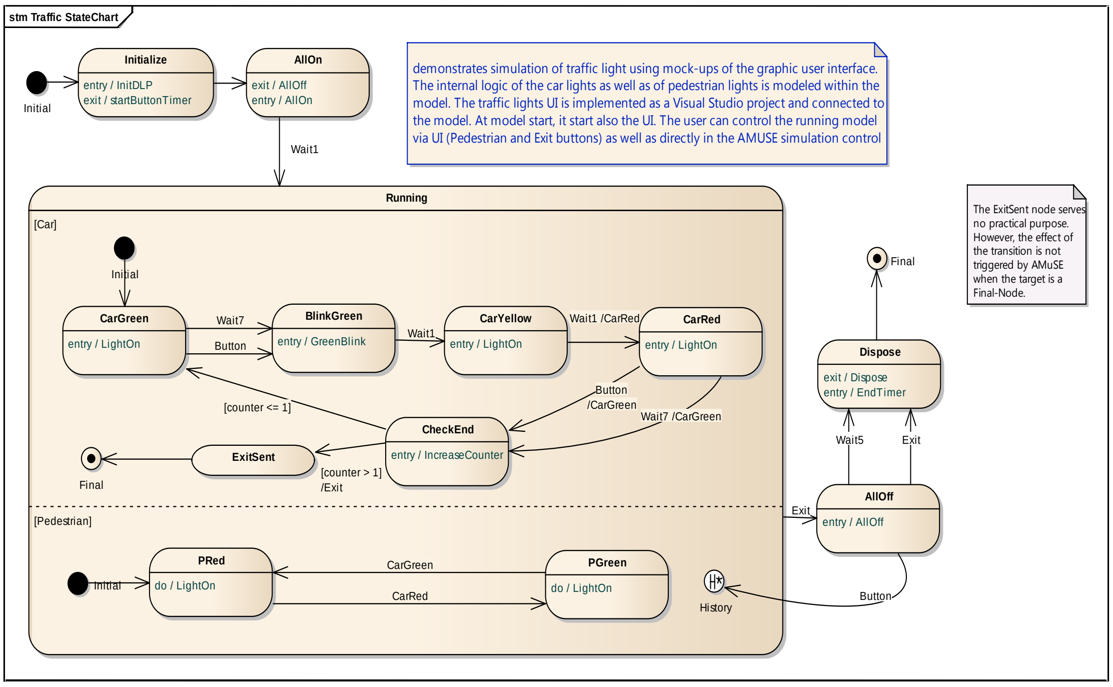


Figure 5.3: Activity diagram *Change End* of the *Behavior As Activity* case study

5.1.2 Traffic Light

The *Traffic Light* model is one of the more complex examples distributed with the AMUSE plugin. Figure 5.4 depicts the state machine defined in this model. The state machine represents a combination of a vehicle and a pedestrian traffic light, which are simulated by two sub-state machines running in parallel. These sub-state machines use activities to broadcast signals, which are then received by the respective other state machine. The signals are used to synchronize the light-phases of the traffic lights, in order to guarantee that only one traffic light shows green at any given time.


demonstrates simulation of traffic light using mock-ups of the graphic user interface. The internal logic of the car lights as well as of pedestrian lights is modeled within the model. The traffic lights UI is implemented as a Visual Studio project and connected to the model. At model start, it start also the UI. The user can control the running model via UI (Pedestrian and Exit buttons) as well as directly in the AMUSE simulation control

The ExitSent node serves no practical purpose. However, the effect of the transition is not triggered by AMuSE when the target is a Final-Node.

Figure 5.4: State machine of the *Traffic Light* case study

A custom user interface, which is depicted in Figure 5.5, is connected to the *Traffic Light* state machine. This user interface illustrates the current light configuration of the two traffic lights modeled in the state machine. Furthermore, the user interface contains a button that indicates to the state machine that a pedestrian wants to cross. When this button is pressed, the state machine immediately begins the process of switching the car traffic light first to green-blinking, then to yellow and then to red before finally switching the pedestrian traffic light from red to green.

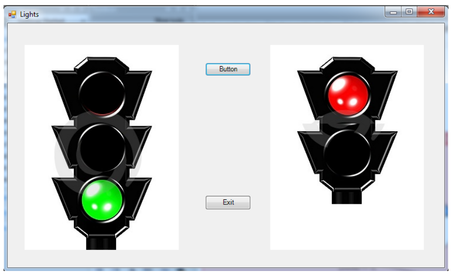


Figure 5.5: User interface provided by the *Traffic Light* case study

5.1.3 Package Center

While the two aforementioned case studies cover the basic functionality of activity diagrams, they are not complex enough to fully illustrate and test all the functionalities provided by the prototype. Therefore, the *Package Center* case study has been constructed.

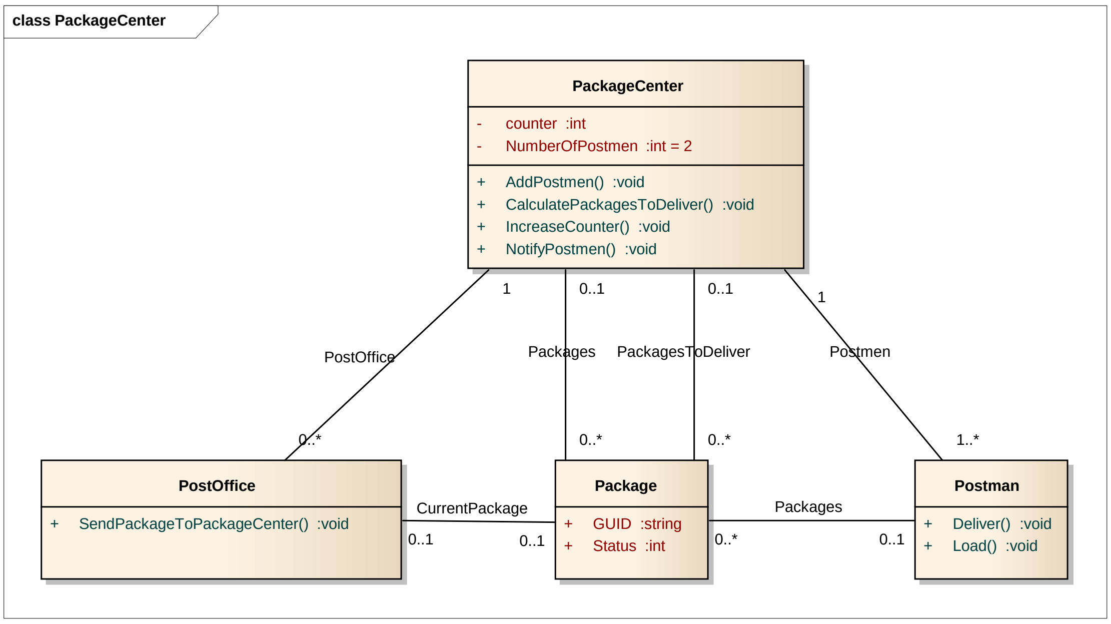


Figure 5.6: Domain of the *Package Center* case study

The *Package Center* case study is a model describing the basic delivery process of packages as it is used by postal services world-wide. Figure 5.6 shows the class diagram describing the domain used for the definition of this case study.

While the exact definitions of the models are omitted for brevity, the following sections give a brief overview of the inner workings of the state machines and activity diagrams defining the behavior of the classes.

Post Office. The post office is responsible for creating the packages to be processed. Its behavior is defined by a state machine which loops between the following states:

1. The state machine of the post office resides in an *Idle* state. After a fixed period of time, the state machine triggers the transition to the "Send Package" state.
2. Upon entry of the "Send Package" state, the "Send Package" activity diagram is invoked. This activity diagram creates a new package, assigns a new GUID and the status 0 (new package) to it, and finally adds it to package center's list of stored packages via an add structural feature value action.
3. Upon completion of "Send Package" activity diagram, the state machine returns to the "Idle" state.

Package Center. The package center has the role of processing the packages generated by the post office and assigning them to postmen for delivery. The state machine of the package loops between the following states:

1. The state machine of the package center resides in an *Idle* state. After a fixed period of time, the state machine triggers the transition to the "Calculate Packages" state.
2. Upon entry of the "Calculate Packages" state, the "Calculate Packages" activity diagram is invoked. This activity diagram creates a list of all packages stored packages which shall be delivered via the postmen based on the list of all stored packages.
3. Upon completion of the "Calculate Packages" activity diagram, the state machine transitions immediately to the "Assign Packages" state.
4. Upon entry of the "Assign Packages" state, the "Assign Packages" activity diagram is invoked. This activity diagram assigns each package which shall be delivered to a randomly chosen postman.
5. Upon completion of the "Assign Packages" activity diagram, the state machine transitions immediately to the "Notify Postmen" state.
6. Upon entry of the "Notify Postmen" state, the "Notify Postmen" activity diagram is invoked. This activity diagram sends a signal to all postmen, which informs them that they can now start with their respective deliveries.
7. Upon completion of "Notify Postmen" activity diagram, the state machine returns to the "Idle" state.

Postman. The postman carries out the task of delivering all the packages that have been assigned to him, which is represented in the model by the assignment of a different status to the package. The state machine of the postman loops between the following states:

1. The state machine of the postman resides in an *Idle* state. After receiving a signal from the package center which indicates that each package has been assigned to a postman, the state machine triggers the transition to the “Load” state.
2. Upon entry of the “Load” state, the “Load” activity diagram is invoked. This activity diagram assigns the status 1 to each package, which indicates that the package has been successfully loaded into the delivery vehicle.
3. Upon completion of the “Load” activity diagram, the state machine transitions to the “Deliver” state.
4. Upon entry of the “Deliver” state, the “Deliver” activity diagram is invoked. This activity diagram assigns the status 2 to each package, which indicates that the package has been successfully delivered to the recipient.
5. Upon completion of “Deliver” activity diagram, the state machine returns to the “Idle” state.

Package. The package object represents the physical package. It is only used for storing the state of the package, as well as a unique identifier. A package object does not have a behavior.

Object Initialization. The *Package Center* case study includes a sequence diagram used for initializing the used instances before running the simulation. This sequence diagram is shown in Figure 5.7.

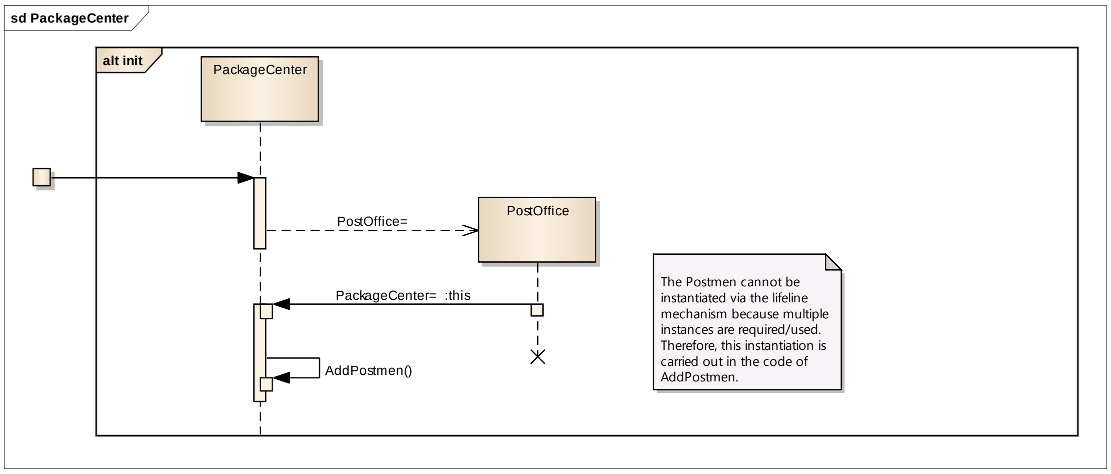

sd PackageCenter
alt init
PackageCenter
PostOffice=
PostOffice
PackageCenter= :this
AddPostmen()
The Postmen cannot be instantiated via the lifeline mechanism because multiple instances are required/used. Therefore, this instantiation is carried out in the code of AddPostmen.

Figure 5.7: Sequence diagram used for initializing the execution of the *Package Center* case study

The sequence diagram shown in Figure 5.7 is the first behavioral diagram that is executed during the overall execution. It initializes all required instances, namely one package center, one post office and multiple postmen objects. Furthermore, the sequence diagram establishes the correct links between these objects.

5.2 Functionality Analysis

The functionality of the prototype has been evaluated based on the presented case studies, as well as a number of additional test cases for specific features. The main goal of the functionality evaluation was to compare the activity diagram execution capabilities of the prototype with those provided by AMUSE. Furthermore, the activity diagram execution capabilities of the prototype have been compared with those of the fUML reference implementation.

5.2.1 Approach

The first step in evaluating the functionality of the prototype is the creation of a reasonable set of different models that cover all the expected functionalities. This set includes several basic models, which each cover a very specific functionality. These models have been also used as the basis for implementing the prototype itself in a Test-Driven-Development approach. The implemented test cases are shown in Table 5.1.

Model Contents	**Functionality Under Test**
Two initial nodes connected by control flows	Execution of invalid models
Single initial node	Running a basic activity diagram
Two groups of create object action, value specification and add structural feature value action	Creation of objects and assignment of feature values
Activities of the *Behavior As Activity* case study	Changing feature values, calling an activity from another activity
Add structural feature value action that adds a new value to the existing values of a feature value	Manipulation of collections of values
Expansion region that applies an add structural feature value action with a static value	Expansion region with all expansion node types
Send signal action	Sending signals with object parameters to a specified recipient

Table 5.1: Basic test cases used for the evaluation of the prototype's functionality

This collection of tests seeks to replicate the tests available for the MOLIZ fUML Debug API, and therefore covers every basic piece of functionality that is provided therein. However, the MOLIZ fUML Debug API provides much more in-depth tests for specific features, which are not explicitly tested for the prototype.

An important point to consider is that while all of these tests consist of models constructed directly in EA, not all of them can be run directly from within it, as they are encapsulated test cases with all external factors mocked away. Instead, these models can only be used via the Model Test Execution Framework described in the following.

Model Test Execution Framework

The basic idea behind the Model Test Execution Framework is to enable the creation of regular unit tests based on the MSTest framework integrated in Microsoft Visual Studio. These tests are able to execute a complete UML model created with EA using the developed prototype and assert the correctness of the model execution. To accomplish this, the Model Test Execution Framework accesses the same functionalities for executing fUML models used by the prototype, and conducts the same basic process:

1. Load a specified diagram from an EA model
2. Execute it via the fUML executor
3. Observe all fired events

The Model Test Execution Framework has access to all information about the execution that is also available to the prototype. Most importantly, the Model Test Execution Framework has the possibility to process all the events propagated by the fUML reference implementation. These events are used to verify the correct execution order of model elements, such as actions and control nodes. Additionally, the Model Test Execution Framework also has access to both the EA and the fUML representation of all objects created and updated during the execution.

The current implementation is based on performing the execution of the test in its entirety and only verifying the results afterwards. This solution was chosen to keep the structure of the tests equivalent with their implementation in the MOLIZ fUML Debug API.

5.2.2 Provided Features

The current version of the prototype is able to successfully execute all basic test cases described above, as well as all the case studies described in Section 5.1 consistently and without error. Table 5.2 provides a detailed comparison between the activity diagram execution capabilities of the prototype and AMUSE.

As shown in Table 5.2, the prototype supports all model elements of activity diagrams which are also supported by AMUSE. Additionally, the prototype even supports model elements currently not supported by AMUSE in the form of call behavior actions and expansion regions.

During the development of the prototype as well as testing it, several limitations of the integrated tools have been identified. These limitations are discussed in the following sections.

Model Element	**Model Element Type**	**Supported by the Prototype**	**Supported by AMUSE**
Create object action	Action	Yes	Yes
Value specification action	Action	Yes	Yes
Add structural feature value action	Action	Yes	Yes
Call behavior action	Action	Yes	No
Send signal action	Action	Yes	Yes
Read self action	Action	Yes	Yes
Read structural feature action	Action	Yes	Yes
Remove structural feature value action	Action	Yes	Yes
Reduce action	Action	Yes	Yes
Input pin	Object node	Yes	No
Output pin	Object node	Yes	No
Activity parameter node	Control node	Yes	Yes
Initial node	Control node	Yes	Yes
Final node	Control node	Yes	Yes
Fork node	Control node	Yes	Yes
Join node	Control node	Yes	Yes
Decision node	Control node	Yes	Yes
Merge node	Control node	Yes	Yes
Expansion region	Structured acivity node	Yes	No
Control flow	Activity edge	Yes	Yes
Object flow	Activity edge	Yes	Yes

Table 5.2: Comparison of activity diagram execution capabilities of the prototype and AMUSE

5.2.3 Limitations of Enterprise Architect

Previous chapters have mentioned various deviations of EA from the UML standard. In the following, we will discuss these deviations, as well as the workarounds applied in the implementation of the prototype.

Value Specifications

One of the most glaring deviations is the representation of value specifications in EA. The UML standard [21] specifies multiple different types of value specifications, each having the capability of specifying a value of a specific type. In EA, however, there is no direct representation of these value specifications. Instead, for example, the value specification action simply provides an input field for the value specified, with no further restrictions or any selection options for the type of value. This discrepancy requires the prototype to approximate the type of a value based on the value itself, which is not only cumbersome, but may also lead to possible errors during the model execution.

Missing Features of Specific Elements

A couple of features of fUML cannot be explicitly modelled in EA. Both the features, as well as the workarounds used in the prototype, are listed in Table 5.3.

Missing Feature	**Workaround**
Output pin of the write structural feature action	Custom pin with name "output" is recognized as the output pin.
Reduce action	Generic actions tagged with specific stereotypes are interpreted as calls to specific opaque behaviors. For example, a generic action tagged with the "ReduceAdd" is interpreted as a call to the opaque behavior "ReduceAdd". This mechanism is explained in more detail in Section 4.2.6.
Decision input flow	Object flows that are assigned the stereotype "DecisionInputFlow" are interpreted as decision input flows by the prototype
UnlimitedNatural type	Currently not supported by the prototype.

Table 5.3: Features of the fUML standard missing in EA

General Problems in Utilizing EA for the Prototype

The version of EA used for the prototype, namely version 10, has a significant problem with generating code for collections. EA supports specifying that any attribute of a class represents a collection. This collection mechanism is used by defining setting an additional "collection" marker for the attribute which should be handled as a collection. However, the C# code generated for such an attribute is incorrect, and, in fact, does not even compile. The reason for this is an error in the code generation template used for the attribute.

While it is possible to fix the code generation template as part of the prototype, this approach has the possibility of interfering with custom changes the user has made to the code generation templates. The workaround applied in the case studies is not to use collection attributes at all. Instead, the type of the attribute is declared as "List". This leads to the generation of an attribute of type *list*, which is the name of the list implementation within C#. The generated code is correct when applying this approach and the created classes and instances are interpretable by the prototype.

5.2.4 Limitations of AMUSE

Differences in Complexity of Models of AMUSE and fUML

When using AMUSE, the behavior of an activity is mostly defined by attaching C# code to generic actions. However, in fUML, the action types for manipulating objects and performing calculations, have to be used. Due to this difference, the structure of activity diagrams executed by AMUSE is fundamentally different from activity diagrams executed by the prototype. This difference results in a large difference in the complexity of models used by both approaches. The *Behavior As Activity* case study is revisited here to illustrate this difference. The fUML conformant models for this case study (depicted in figures 5.2 and 5.3) consist of 11 actions, 20 activity edges and 15 pins in total.

For comparison, consider the AMUSE conform model for the activity diagram, which is depicted in Figure 5.8. It consists of 2 actions, 6 activity edges and 0 pins. However, note that this model also contains 3 additional C# statements for depicting the behavior of 2 actions and a guard condition.

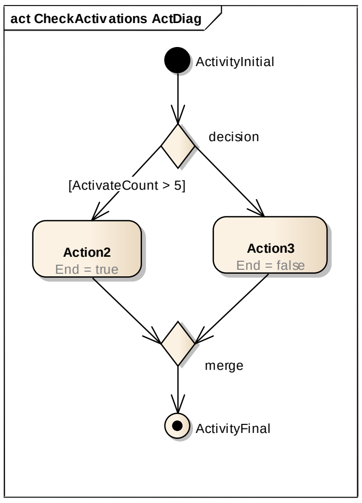


Figure 5.8: Activity diagram “Check Activity” of the *Behavior As Activity* case study

Limited Support of Behavior Call Mechanisms

The version of AMUSE utilized for the prototype, namely version 2.5, does not support some behavior call mechanisms, either because of limitations of EA itself, or because the generated code does not support it. Table 5.2 shows the mechanisms currently not supported by AMUSE. The most prominent omission of AMUSE is that it does not support using call behavior actions within activity diagrams. This mechanism is used extensively within the package center case study, however. Therefore, AMUSE is unable to interpret the model of the *Package Center* case study in its original form. To work around these limitations, a replica of the *Package Center* case study has been created, which only uses the behavior call mechanisms supported by AMUSE instead of call behavior actions.

Limited Functionality of Sequence Diagrams

AMUSE does not support calling behaviors from sequence diagrams. In fact, the implementation for the sequence diagrams fulfills only a very specific functionality , namely initializing additional objects before the execution is started. For a detailed description of this mechanism, please refer to Section 2.3.2.

For an illustration of a model that incorporates this functionality, we kindly refer to Figure 5.7. Aside from illustrating the functionality given to the sequence diagrams in AMUSE, this figure also illustrates one of the drawbacks of the implementation. EA imposes the restriction of only allowing a single lifeline corresponding to a class. Therefore, the modelled initialization routine is unable to define an initialization of multiple instances of the same class, as required in the *Package Center* case study. To overcome this limitation, the package center class defines a method with attached C# code to manually initialize the objects. This method is invoked from within the sequence diagram used to initialize the package center object.

As sequence diagrams were not within the primary scope of the prototype, this implementation has been left unchanged and was used for the purpose it was designed for, namely the initialization of the *Package Center* case study as explained in Section 5.1.3. Aside from this specific use case, sequence diagrams have not been dealt with further in the course of the evaluation.

5.2.5 Limitations of the fUML Reference Implementation

Many of the problems in utilizing the fUML reference implementation, such as the inability to observe a running execution, have been successfully solved through the MOLIZ API. However, the nature of the fUML reference implementation still caused problems in the execution of models.

Validation Capabilities

The most critical weakness of the fUML reference implementation in the context of the prototype is its inability to correctly validate models before execution.

The fUML reference implementation expects—according to its own definition—a "conformant UML model". A reasonable assumption might be that this would refer to a model that is created according to the fUML specification. However, a model can be created that invalidates this hypothesis. For instance, a model containing an empty expansion region would be assumed to be fUML conformant, but such a model causes an undesired behavior of the execution environment.

When executing the model, the fUML reference implementation crashes when trying to interpret the empty expansion region, and produces an exception. This exception, stays *unhandled* by the entire execution environment, and is instead passed on directly to the entity triggering the execution. The key observation here is that it performs *no validation of models before execution*.

The point of this observations is not to expect the fUML reference implementation to be completely bug-free and stable at all times. In fact, the described case likely represents an oversight in the implementation itself. However, it shows that the fUML reference implementation is lacking in mechanisms to deal with errors both in the provided models, as well as in its own implementation. Overall, this severely hinders the prototype—or, in fact, any user of the fUML reference implementation—in providing meaningful user feedback in the case of errors in the model, which in turn impacts the usability of the whole approach negatively.

Retrieval of Execution Identifiers

A specific technical problem in utilizing the MOLIZ API is the assignment and discovery of execution ids, which the API assigns to each running execution to be able to separate them. While this allows for a greater amount of control over the execution, the actual discovery of the associated execution IDs is not trivial. When invoking the execution of an activity diagram via the MOLIZ fUML Debug API, the unique execution id assigned to this execution is not immediately indicated. Instead, the fUML executor is required to explicitly query the list of executions that are currently being executed by the fUML virtual machine after the invocation to determine the execution id. This implementation is not only cumbersome, but might also lead to using wrong execution ids during parallel executions. A method of directly determining the execution id of a started execution would be a desirable improvement of the MOLIZ fUML Debug API.

5.2.6 Complications from Utilizing IKVM

Overall, the IKVM toolkit fulfills the task of bridging the gap between C# and Java remarkably well. The prototype is able to access, instantiate and even extend the classes of both the fUML reference implementation and MOLIZ transparently and without issue.

However, the debugging capabilities of IKVM incur problems when used with such a large number of different libraries as used by this prototype. Both the fUML reference implementation and the MOLIZ API utilize a variety of different dependencies to other libraries, each of which must be converted to DLLs as well.

IKVM provides functionality to debug the converted code. In the case of the prototype, these debugging functionalities could not be utilized at all during the first development cycles because IKVM could not fully interpret one of the dependencies of the fUML reference implementation. These problems were reported to the lead developer of IKVM, and subsequently fixed in a later version. However, even when the debugging capabilities were functional, they could only be used very scarcely. Specifically, the debugger oftentimes disconnected itself from the virtual machine while stepping through the execution for only a minor number of steps (between five and ten), which made fully tracing the fUML execution largely impossible.

The actual performance overhead of utilizing IKVM (for example in contrast to utilizing the fUML reference implementation from Java code) has not been analyzed in the context of this thesis. However, as the tests in Section 5.3 note a total execution time of few milliseconds, the actual overhead is assumed to be insignificant for the purposes of this prototype.

In general, even though IKVM seems to be the most widely distributed C# / Java interconnectivitiy tool, it is still a tool with only a handful of developers, and should therefore not be expected to have the same kind of robustness and support as a commercial tool.

5.3 Performance Analysis

An extensive analysis of the performance of both the implemented prototype, as well as the AMUSE plugin, has been conducted. This analysis has focussed on two main aspects, namely the model execution time and the memory consumption during the execution. The main goal of this analysis was to perform a comparison of the performance of both model execution approaches (interpretation vs. code generation). The analysis is based on the execution of the case studies presented in Section 5.1, which are executed by both the prototype and the AMUSE plugin to produce comparable performance results.

5.3.1 Approach

The first step in setting up the performance analysis was determining an appropriate number of executions. The main concern was to generate a usable amount of data, while still allowing for a reasonable timeframe of the analysis. Therefore, the decision was made to conduct each execution 10 times, and expand the number of runs for specific case studies if necessary. The data of different runs has been shown to be largely consistent. The only exception is the *Traffic Light* case study, which shows large variations in memory consumption between each different execution. These fluctuations are inherent to the *Traffic Light* case study due to the frequent usage of signals and the external user interface. Thus, an increase in the number of executions would not yield any additional information. Therefore, no additional executions have been conducted at this time.

As for the method of combining the results of the separate executions, the decision was made to rely on the median. The reason is that during preliminary testing, the first run of each simulation was shown to have significantly higher execution times, which would skew the results if averages were taken as basis for the evaluation.

All test runs for the performance evaluation have been conducted on the development machine itself. The specifications are shown in Table 5.4.

Component	**Specification**
Processor	Intel I7 @ 2,67 GHz
Video Card	AMD Radeon HD 5700
Memory	4 GB
Operating System	Windows 7, 64 Bit

Table 5.4: Hardware used for the performance evaluation

5.3.2 Analysis of Execution Time

For the evaluation of the execution time, both the prototype and the AMUSE plugin have been extended with performance measuring capabilities. As the *ExecutionEnvironment* component of AMUSE is responsible for both triggering and tracking the execution, it is the most suitable point to integrate the performance logging mechanism. Both the start and end of the code generation phase, as well as the start and end of the execution are under direct control of the *ExecutionEnvironment* component.

Performance Tests within EA

The first phase of the performance evaluation was conducted by executing the models of the three different case studies described in Section 5.1 directly in EA. The results of these executions are shown in Table 5.5, 5.6 and 5.7. Each presented table contains the following information about the specific executions:

- **Total preparation time**. This measurement indicates the execution time required by the *ExecutionEnvironment* component to prepare the model execution. For the AMUSE plugin, this includes generating the source code for all behavioral diagram types. For the prototype, this includes generating the source code for state machines and sequence diagrams, as well as the initialization of the fUML executor. The initialization of the fUML executor comprises converting all classes required for the execution to their fUML representation so they can be used by the fUML reference implementation.

- **Total model execution time**. This measurement indicates the execution time required for performing the model execution after the preparation has been completed. This measurement is calculated by measuring the total time difference between the start and the end of the overall execution, and subtracting all known delays that occur when executing the specific model (e.g., the *Traffic Light* state machine waits a specified amount of seconds before switching lights).

- **Total execution time**. A sum of total preparation time and total model execution time.

- **Number of executed elements**. This measurement indicates the number of different actions and states that were encountered and executed during the overall model execution. This number is always significantly higher for the prototype compared to the AMUSE plugin, as the AMUSE plugin uses actions with attached C# code for the execution, while the prototype uses the action types provided by fUML. This difference is discussed in more detail in Section 5.2.4.

Table 5.5 shows the results of the execution time evaluation of the *Behavior As Activity* case study. The table shows the basic premise of all following performance evaluation results. Both the preparation and the model execution time in the prototype are significantly higher compared to those of the AMUSE plugin.

Table 5.6 shows the results of the execution time evaluation of the *Traffic Light* case study. Unfortunately, the performance impacts of the external UI used by the *Traffic Light* model cannot

	fUML	AMUSE
Total preparation time	1.776 ms	761,5 ms
Total model execution time	572,5 ms	181,5 ms
Total execution time	2.340 ms	931 ms
Number of executed elements	70	42

Table 5.5: Execution time of the *Behavior As Activity* case study

be accurately determined. Therefore, the evaluation results for this case study are not as relevant as the results for the other two case studies. Nevertheless, the difference in pure execution time is suitably minimal when comparing the prototype and AMUSE, which is a result of the structure of the case study—only the send signal actions can be executed in fUML, while the remainder of the behavioral diagrams are executed identically by both the prototype and AMUSE.

	fUML	AMUSE
Total preparation time	2.806 ms	2635,5 ms
Total model execution time	7.573 ms	7.298 ms
Total execution time	10.388 ms	9.924,5 ms
Number of executed elements	72	62

Table 5.6: Execution time of the *Traffic Light* case study

Table 5.7 shows the results of the performance evaluation of the *Package Center* case study. The *Package Center* case study is the biggest case study used for this evaluation, and therefore shows the performance discrepancies between the approaches most clearly. The difference in preparation time shows that both approaches seem to scale relatively evenly, with the prototype requiring about 50 percent more time than AMUSE. The difference between model execution times, however, is significant both by itself, as well as in comparison to the other case studies. A key factor that could be observed during the execution was that the constant re-rendering of the newly added detailed object display seemed to significantly slow down the execution.

	fUML	AMUSE
Total preparation time	4.291 ms	2.792 ms
Total model execution time	31.807 ms	288 ms
Total execution time	36.118,5 ms	3.072,5 ms
Number of executed elements	206	48

Table 5.7: Execution time of the *Package Center* case study

A decisive problem of the prototype shown by all three case studies is the performance overhead induced by the preparation phase, as it requires both generating the source code for state machines and sequence diagrams, as well as the initialization of the fUML executor. The comparison of the preparation times shows that the time required to initialize the fUML executor is significantly higher than the time required to generate and compile the additional source code for the activity diagrams.

The difference in pure execution time is an expected side effect of the virtual machine. As shown by the evaluation of the three presented case studies, the difference in total performance is already over 1000 percent. However, aside from the inherent problems of the fUML reference implementation in comparison to the code generation approach, there are also additional factors that effect the performance of the prototype, namely the constant synchronization of objects between the fUML and AMUSE execution environments and the detailed visualization of the current objects. Finally, the larger number of nodes in the fUML conformant model also leads to a higher model execution time.

Peformance Tests outside of EA

The second step taken in evaluating the execution time of the prototype was to execute the models outside of EA. This removes the following factors influencing the performance measurements:

- Performance overheads caused by EA
- Performance overhead caused by visualizations in EA (diagram visualization, execution state, objects view)
- Simulation delays for correct visualization
- Code generation and interpretation of the models

Unfortunately, it is not possible to utilize the code generation facilities of EA without a running EA instance. Therefore, this evaluation focusses strictly on the model execution and neglects the preparation step. For executing the models, the model execution framework described in Section 5.2 is used. For this performance analysis, the Model Test Execution Framework was modified such that it logs both the start and the end of the execution at the same points the AMUSE framework does.

Table 5.8 shows the results of the evaluation of the case studies outside of EA. Comparing these results to the results obtained in EA shows the expected reduction in execution time through the elimination of the factors mentioned above.

The execution times for the *Behavior As Activity* case study show a significant reduction of execution time for both the prototype and AMUSE. However, it should be noted that the reduction is much higher for AMUSE, which retains only a very low execution time.

	fUML	**AMUSE**
Execution time of the *Behavior As Activity* case study	460 ms	35,5 ms
Execution time of the *Traffic Light* case study	7.815 ms	7.349 ms
Execution time of the *Package Center* case study	838,5 ms	65 ms

Table 5.8: Execution time evaluation of the case studies outside of EA

The external UI used by the *Traffic Light* case study is retained in this form of model execution. Comparing the results of this execution to the one conducted in EA shows no significant impact on the total execution time, which may support the assumption that the external UI is causing the performance problems of this particular case study.

The execution times for the *Package Center* case study show the biggest differences when compared to the evaluation conducted directly in EA. Through isolating the model execution from EA, the measured execution times of both AMUSE and the prototype are reduced significantly. While the overall relation between the execution times measured for fUML and AMUSE remains the same, both times are reduced by at least 90 percent, indicating that most of the performance overhead is not caused by the respective execution environments.

5.3.3 Performance Comparison with Hand-Written Code

The last step of the evaluation was to determine a general measure of overhead imposed by simulating models compared to executing C# code implementing the same functionality. To illustrate this approach, Listing 5.1 depicts a unit-test equivalent to the *Behavior As Activity* case study, which was implemented in C#.

```
[TestMethod]
public void BehaviorAsActivityCoded()
{
    PerformanceLog performanceLog = new PerformanceLog("
       Coded_BehaviorAsActivity", false, 0);
    performanceLog.Start();
    performanceLog.AddEntry("Execution started.");
    int i = 0;
    bool end = false;
    while (true)
    {
        i++;
        if (i%4 == 1)
        {
            if (i > 5)
            {
                end = true;
            }
        }
        if (i%4 == 2)
        {
            if (end)
            {
                break;
            }
        }
    }

    performanceLog.AddEntry("Execution completed.");
    performanceLog.Finish();
}
```

Listing 5.1: C# implementation of the *Behavior As Activity* case study

Table 5.9 shows the measured execution time for the implemented C# code. Comparing these results with the previously shown evaluation shows that the model execution approaches of both the prototype and AMUSE induce significant performance overheads. While the reduction to only the execution-mechanics performed in the previous section shows massive performance increases, the re-implementation by hand shows that both approaches still induce significant overheads.

This is especially true in the case of the *Behavior As Activity* case study, for which none of the executions takes even a single millisecond. Also, the coded implementation of the *Traffic Light* case study allows us to evaluate the actual impact of purely loading and controlling the user interface without the event mechanism, as this coded version directly manipulates the UI by calling the methods otherwise triggered by events.

	Execution-Time
Behavior As Activity	0 ms
Traffic Light	237,5 ms
Package Center	4,5 ms

Table 5.9: Performance Evaluation of the coded representations of the case studies

5.3.4 Analysis of Memory Consumption

Measuring performance strictly based on execution time neglects a very crucial performance factor, namely the memory consumption.

Both Java and C# are programming languages that employ garbage collection mechanisms. Any approach that tries to determine memory consumption has to take this mechanism into account, and is therefore prone to errors resulting from inaccurate measurements.

Nevertheless, the Model Test Execution Framework attempts to estimate the memory consumption of the respective of both the prototype and AMUSE by implementing the following approach:

1. Dispose all unused objects
2. Measure current memory allocation
3. Start execution
4. Measure memory allocation every time the execution enters a new element

Figure 5.9 shows a comparison of the memory consumed during the execution of the *Behavior As Activity* case study. As can be seen in the figure, the memory consumption of the prototype is significantly higher than that of AMUSE. Furthermore, the memory profile of the prototype shows a far more volatile behavior, with two major peaks in memory consumption that seem to largely coincide with the invocations of the fUML executor.

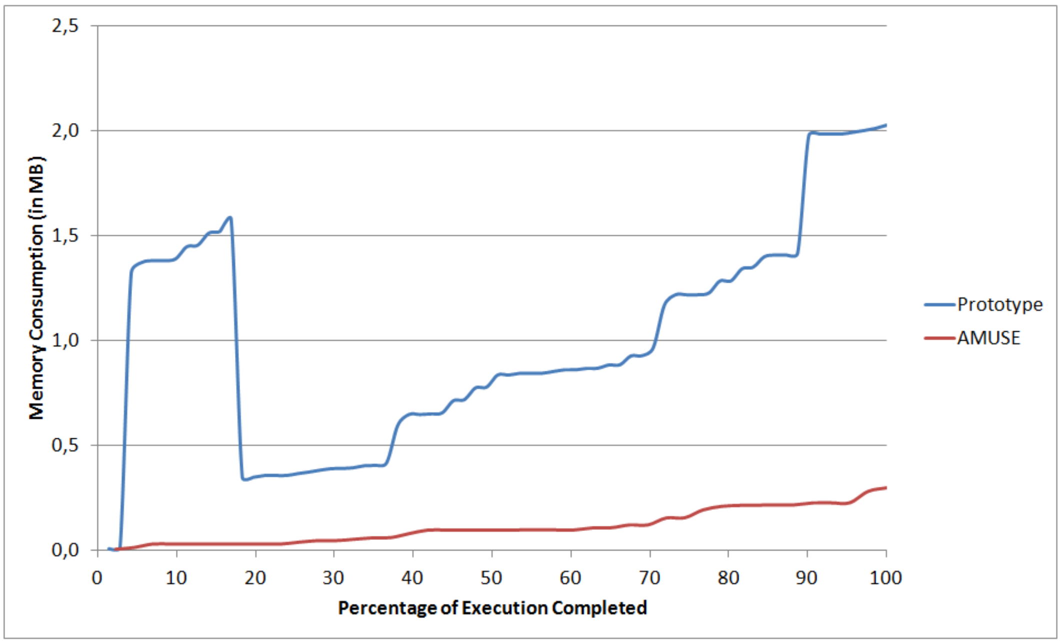


Figure 5.9: Memory comsumption of the *Behavior As Activity* case study

Figure 5.10 shows the comparison of consumed memory during the execution of the *Traffic Light* case study.

The memory consumption of the *Traffic Light* case study is inherently volatile due to the implementation of the UI component, which relies on polling mechanisms and constant triggering of events to communicate with the state machine. Each significant peak of memory consumption during the execution coincides with a change of one of the lights of the UI component. Furthermore, both the prototype and AMUSE show a very similar memory consumption pattern. The shift between the peaks in memory consumption stems from the different number of model elements that are contained in the models executed by the respective approaches. However, an analysis of the memory consumption during the individual executions of the *Traffic Light* case study has shown large differences in consumed memory at all measurement points. Therefore, the memory consumption analysis of this particular case study is only of limited meaning.

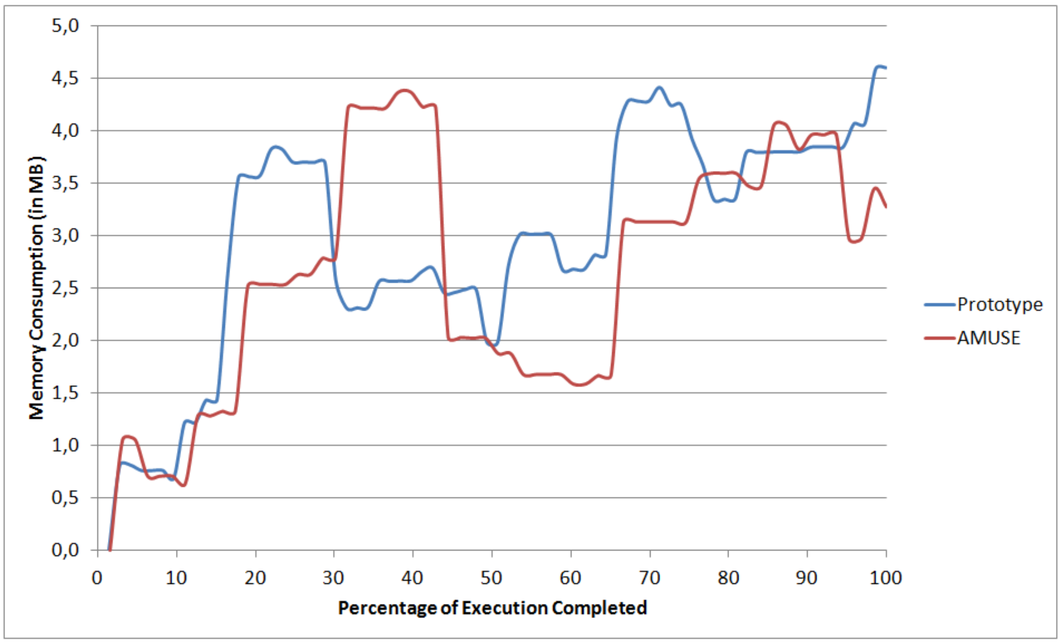


Figure 5.10: Memory comsumption of the *Traffic Light* case study

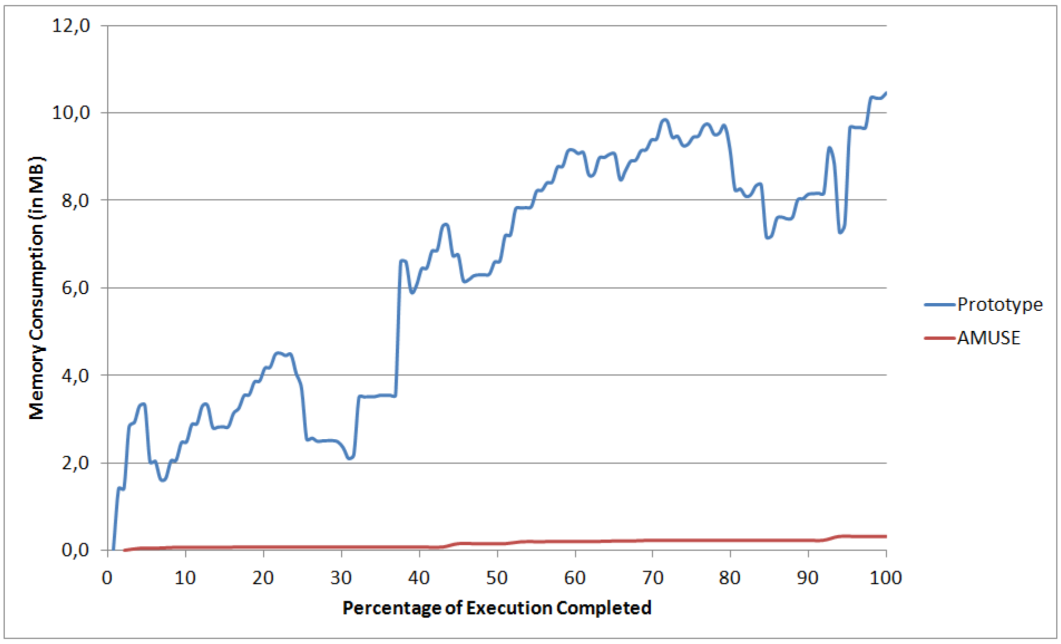


Figure 5.11: Memory comsumption of the *Package Center* case study

Figure 5.11 shows the comparison of consumed memory during the execution of the *Package Center* case study. It shows that while the code generation approach of AMUSE seems to require only very little memory in performing the execution of this case study, the prototype requires a far greater amount.

The reason for the high amount of required memory lies in the constant parallel execution of several activity diagrams. The fUML reference implementation allocates a minimum of 400 KB of memory for each newly started activity diagram execution. Furthermore, this memory is not always deallocated directly after the execution of the activity diagram has been completed. This behavior of the fUML reference implementation is directly reflected by the measured memory consumption during the execution of the *Package Center* case study. The package center object executes two activity diagrams for each package that must be delivered. Each of these executions consecutively increases the memory consumption. This increase starts when the execution of the *Package Center* case study has reached 50 percent completion, and is depicted in Figure 5.11. Additionally, all package objects created during the execution of the *Package Center* case study are stored until the end of the execution, requiring additional memory.

A preliminary evaluation of the *Package Center* case study with a reduced number of postman objects (1 or 0, respectively) has shown a decrease of 1,5 MB of memory consumed per postman, providing further evidence that the conduction of multiple parallel activity diagram executions causes the increased memory consumption.

5.4 Usability

The usability of both the prototype and AMUSE have been evaluated based on the presented case studies. The goal of this evaluation was to compare the usability of the respective tools, and their utilization of EA.

5.4.1 Approach

As has been noted in Section 5.2, AMUSE and the prototype utilize significantly different implementations of activity diagrams. Therefore, two separate versions of the activity diagrams used within the case studies have been created for the evaluation. The evaluation of the usability is based upon the experiences we have gained while creating these diagrams.

5.4.2 Results

Insufficient Expressiveness of EA Models

As has been discussed extensively in the previous chapters, fUML is a very precise modeling language which requires models to be very detailed. With the exception of the problems mentioned in Section 5.2.3, EA supports the creation of such models. However, it lacks the capabilities to represent certain key properties of fUML model elements in an easily perceivable manner.

Consider, for example, the activity diagram shown in Figure 5.12, which is part of the *Package Center* case study.

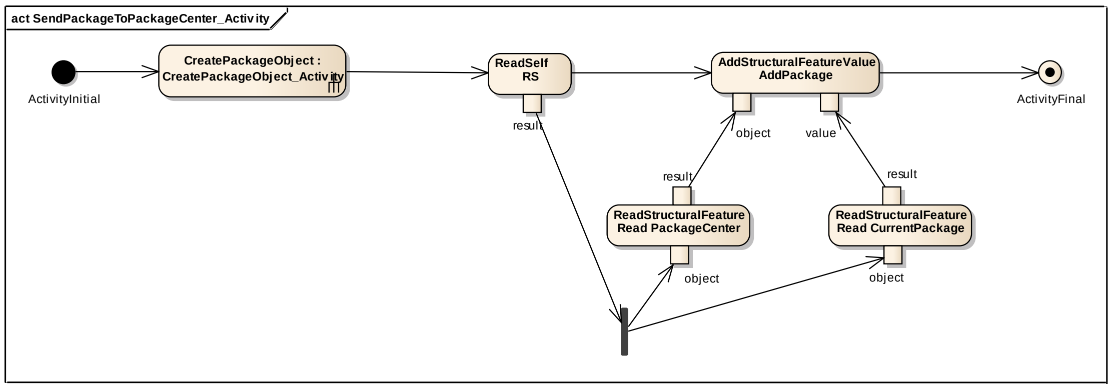

act SendPackageToPackageCenter_Activity
ActivityInitial
CreatePackageObject : CreatePackageObject_Activity
ReadSelf RS
result
AddStructuralFeatureValue AddPackage
object
value
ActivityFinal
result
ReadStructuralFeature Read PackageCenter
object
result
ReadStructuralFeature Read CurrentPackage
object

Figure 5.12: Activity diagram *SendPackageToPackageCenter* of the *Package Center* case study

The following information is not visible from the diagram, and must be inferred from determined by opening separate properties windows for each action:

- The structural feature referenced by the add structural feature value action is not shown
- The structural feature referenced by the read structural feature action is not shown
- Both control flows and object flows are displayed identically. The type of an activity edge can only be determined via the properties window in which this type is defined, or inferred from the model elements the activity edge connects.

Similar problems exist in a majority of the diagrams used for the *Package Center* case study. For example, input and output expansion nodes of expansion regions cannot be visually distinguished. Their only differentiating factor is the applied stereotype, which is not displayed in the diagram, and can only be seen in the "stereotype" sub-window of the properties window.

Inefficiency of the EA User Interface

While the user interface of EA covers all of the functionalities needed for specifying the models used for evaluating the prototype (aside from those covered explicitly in Section 5.2.3), the general experience of using EA to create the models has been perceived as being cumbersome.

The main reason lies in the design of the user interface itself. Commonly used model elements, such as classes and their corresponding attributes, can be easily and intuitively defined. Configuring the more specific model elements that are used for an fUML compliant activity diagram, such as an add structural feature value action, however, requires the navigation of various properties windows for each model element. Additionally, these properties windows vary based on the actual type of model element, requiring extensive knowledge of the menu structure of EA for creating fUML models.

Consider, for example, the following list of steps required for modeling an add structural feature value action correctly:

1. Drag a generic action onto the model
2. Select "AddStructuralFeatureValueAction" as the type
3. Drag the pins of the add structural feature value action onto the action from the project browser. The pins are automatically created upon creation of the add structural feature value action, but are not automatically added to the diagram
4. Navigate to the entry "Advanced—Set Structural Feature" of the context menu of the action
5. Select the correct from a list of structural features contained in the model

As can be seen, the definition takes a multitude of steps, each of which needs to be carried out correctly in order for the model to be valid according to fUML. A similar procedure needs to be applied for all other action types, as well as for some of the control nodes.

Inconsistencies Between the EA User Interface and Stored Models

Aside from the need to extensively use properties windows to define fUML conform models, creating models with EA mostly consists of dragging and dropping elements onto the modeling canvas and connecting them. However, for model execution, the visual representation is largely irrelevant, as only the model elements stored in the database and displayed in the project browser are processed.

One particularly frustrating case that is related to this discrepancy is the modeling of expansion regions. When dragging elements onto an expansion region, they are not actually registered as being owned by the expansion region itself, which leads to an incorrect model execution. The only way the user can fix this is to manually drag the respective elements onto the expansion region in the project browser so they are properly registered as children thereof.

Model Execution Visualization Issues

The version of AMUSE utilized in the prototype uses a custom SVG layer to display the state of running executions. The actual visualization needs to be implemented manually for each model element. However, as the AMUSE plugin only supports a limited set of model elements for activity diagrams, the visualization for the model elements required for an fUML compliant activity diagram are not sufficiently implemented. Unfortunately, this leads to an incomplete visualization of the execution state of activity diagrams by the prototype.

Consider, for example, the differences between the model of the *SetupForDelivery* activity, which is shown in Figure 5.13, and its visualization during the execution, which is shown in Figure 5.14.

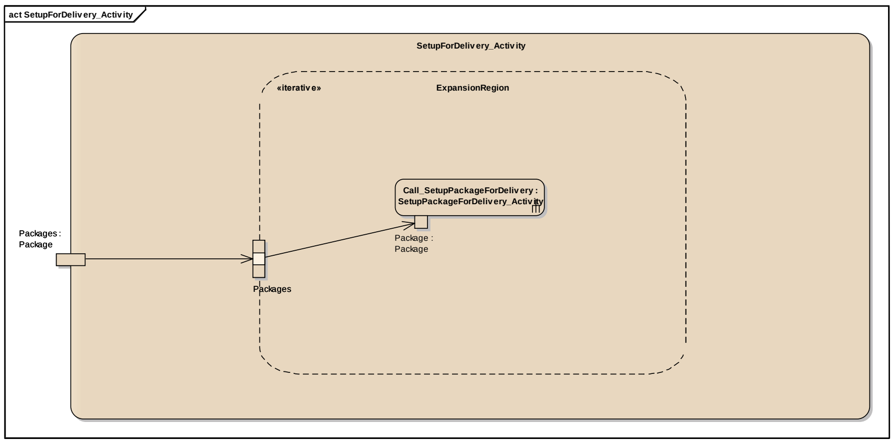

act SetupForDelivery_Activity
SetupForDelivery_Activity
«iterative»
ExpansionRegion
Call_SetupPackageForDelivery : SetupPackageForDelivery_Activity
Packages : Package
Package : Package
Packages

Figure 5.13: SetupForDelivery activity diagram of the *Package Center* case study

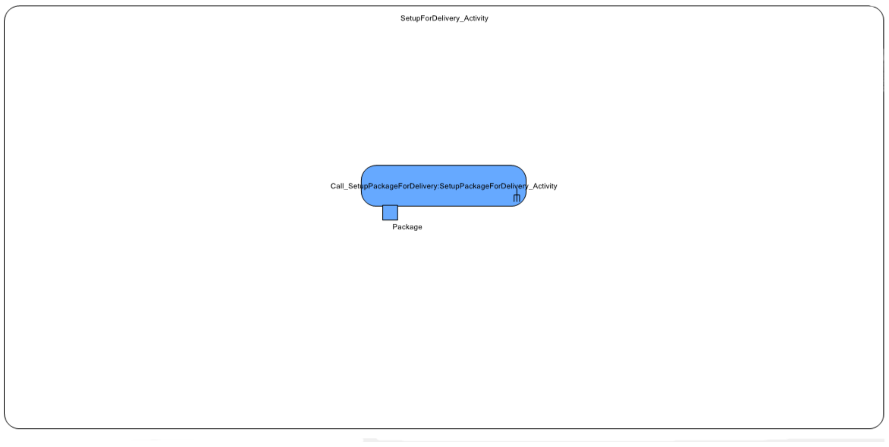

SetupForDelivery_Activity
Call_SetupPackageForDelivery:SetupPackageForDelivery_Activity
Package

Figure 5.14: Visualization of the execution state *SetupForDelivery* activity diagram of the *Package Center* case study

The activity diagram contains an expansion region, which in turn contains a call behavior action to be executed for every element passed on to the expansion region. However, a number of elements included in the model cannot be correctly visualized. This includes:

- The activity parameter node of the activity
- The expansion region
- The input and output expansion nodes of the expansion region, as well as all activity edges connected to them

After the analysis of the current implementation and a consultation with the lead developer of the AMUSE plugin, the necessary implementation effort to resolve these visualization issues has been deemed out of scope for the prototype. However, the version of AMUSE currently under development completely revamps the visualization aspects so that it may reuse the internal visualization of EA. The possibility of incorporating the new version of AMUSE into the prototype is discussed in Section 6.2.

5.5 Stability

The stability of the prototype has been analyzed based on the test cases covered by the Model Test Execution Framework described in Section 5.2, as well as the presented case studies. The goal of this evaluation was to determine whether the respective tools are able to execute a given set of models repeatedly without errors and with identical results.

5.5.1 Approach

The stability analysis is based on logging information of both the prototype and AMUSE. This includes both the detailed model execution trace information that is produced by both tools, as well as the additional logging capabilities that have been implemented for both the functionality analysis, as well as the performance analysis. Additionally, the stability of the protoype has been further examined by using the Model Test Execution Framework to conduct and verify repeated automated executions of the case studies.

5.5.2 Results

Stability of the fUML Executor

During all model executions conducted for the evaluations discussed in this chapter, no errors or inconsistencies could be detected in the model executions carried out by the fUML executor. All test runs yielded the expected results (both positive and negative) consistently and without any errors.

Stability of AMUSE

The extensive utilization of the AMUSE plugin has shown that it has fundamental problems with consistently repeating the execution of a combination of complex behavioral diagrams and producing the exact same result.

The AMUSE plugin reuses an existing execution environment for multiple executions of the same model. However, multiple executions conducted in this way retain some of the information from previous runs. This leads to exceptions for duplicate keys in some internal structures after an unpredictable number of runs. It should be noted that this behavior is not caused by the fUML prototype itself, as it also appears when using AMUSE's activity diagram executor. A second indicator that the execution environment is not completely reset after each execution is the appearance of OutOfMemoryExceptions after a around 20 executions. However, both of these issues can be resolved by deactivating the reuse of existing execution environments. Therefore, they have not caused any problems during the evalutation.

A far more serious issue has shown up during the evaluation of the *Package Center* case study. Out of the ten executions via the prototype, two did not completely execute the full process depicted by the model, and have terminated prematurely without reporting any errors. The classifier behavior of the package center is modelled as a state machine, which is executed directly by AMUSE. Therefore, the cause for the premature termination of the execution most likely lies within the implementation of this executor, although we have been unable to determine the specific cause.

5.6 Discussion of the Evaluation Results

The results of the evaluation of the prototype discussed in this chapter provide us with enough information to assess the applicability of our proposed approach of integrating fUML into existing UML modeling tools, and to provide answers to the research questions defined in Section 1.3.

The first research question investigated by this thesis was whether fUML can be integrated into a proprietary UML model execution environment and which challenges arise from such an integration. The prototype created for this thesis serves as proof that such an integration is indeed possible. The main challenges that have been identified for the integration are converting the model representation of the tool into a format that is interpretable by the fUML reference implementation and synchronizing the respective object spaces of the integrated model execution environment with the one of the fUML virtual machine.

The second research question was whether the standardized fUML model execution environment provide the same functionality as a proprietary UML model execution environment. The functionality evaluation presented in Section 5.2 has shown that the prototype provides model execution capabilities for activity diagrams equivalent to those of the AMUSE plugin. The prototype is even able to provide some functionalities which are currently not supported by the AMUSE plugin, such as the invocation of additional activity diagrams during the execution of an activity diagram and the execution of expansion regions.

The last research question was to determine if the created prototype has similar performance characteristics as a proprietary UML model execution environment. The performance analysis presented in Section 5.3 has shown that the prototype has both a significantly higher overall execution time than the the AMUSE plugin, as well as a significantly higher memory consumption. Both of these factors can be mainly attributed to the fact that the prototype uses both model interpretation and code generation in combination, as the prototype needs to maintain both the execution environment of AMUSE as well as the execution environment of the fUML virtual machine in addition to constantly synchronizing them. AMUSE, on the other hand, only needs to maintain its own proprietary execution environment.

CHAPTER 6

Conclusion and Future Work

6.1 Summary

Creating and using models is turning into a central part of the software development process. MDD [14] — the approach of utilizing models during every phase of the process — has become more and more wide-spread over the last years. The predominant standard in this field is UML [12]. It offers a wide variety of different modeling concepts and diagram types for a wide variety of different applications. However, to fulfill this goal, UML's semantics are intentionally kept vague, leaving many specifics up to interpretation. This vagueness, however, interferes with the growing need to execute models for analysis purposes. The modeling tools available on the market today try to solve this problem by applying a multitude of specific, often proprietary semantics. In February 2011, UML has been extended by the standard called *Foundational Subset For Executable UML Models* (or fUML for short) [22]. fUML defines precise and machine-interpretable semantics for a specific subset of UML. The standard includes a fully-functional execution environment, which is capable of interpreting fUML compliant models. However, neither the standard nor its execution environment have been widely adopted in existing modeling tools. Therefore, the goal of this thesis has been to integrate the fUML standard into an existing modeling tool, with the goal of identifying the challenges arising in this integration.

As the fUML standard only deals with class diagrams and activity diagrams, it only supports a limited number of modeling concepts. Therefore, integrating it into an existing tool usually requires adapting the existing execution environment instead of replacing it entirely. The theoretical part of this thesis therefore dealt with clearly defining all possible interaction points that exist between the modeling concepts covered by fUML, and the remaining modeling concepts provided by UML. In particular, the focus was on the interactions between activity diagrams and state machines. These interaction points served as basis for defining the interfaces between fUML's execution environment and existing model execution environments.

The main contribution of this thesis is a prototypical integration of fUML into Enterprise Architect, a commercial UML tool. The prototype utilizes the EA plugin AMUSE, which is able to execute combinations of activity diagrams, state machines and sequence diagrams. AMUSE provides visualization, debugging and tracing support during the model execution. The AMUSE plugin is adapted to utilize fUML for executing activity diagrams while still utilizing its own execution environment for executing state machines and sequence diagrams. Furthermore, the prototype integrates both of these execution environments so that they work together seamlessly.

6.1.1 Evaluation

An extensive evaluation of the prototype has been conducted with the aim of determining the validity of the general approach of integrating fUML with an existing existing UML execution environment, as well as the quality of the implemented prototype. The evaluation has investigated the following aspects:

- Functionality
- Performance
- Usability
- Stability

The evaluation of the prototype has shown that the integration of the execution environments has been completed successfully. The prototype is capable of executing and debugging state machines, sequence diagrams, and fUML compliant activity diagrams in conjunction, while still providing the same functionality as the proprietary execution environment. The performance analysis has shown that the prototype is less performant than that of the proprietary execution environment provided by AMUSE. This is mostly due to the necessity of running two different execution environments in parallel. The usability evaluation has shown that the created prototype can be used to used to effectively execute any combination of behavioral models the prototype supports. Thereby, it provides debugging, tracing and visualization capabilities for all executed diagrams. However, the modeling capabilities of EA are cumbersome to use for defining precise and fUML compliant activity diagrams. In combination with the limited feedback of the fUML execution environment concerning the validity of activity diagrams, finding errors in a model can be quite tedious. The stability evaluation has shown that the integrated execution environment of the prototype is able to execute the supported combinations of models consistently and without error. However, this part of the evaluation has shown that AMUSE suffers from memory leaks, and sometimes misinterprets transitions in state machines running in parallel, leading to inconsistent results within multiple executions of the same models.

6.1.2 Challenges

This master's thesis has discussed a number of different challenges arising from the integration of fUML into an existing UML execution environment. The largest challenge encountered when integrating fUML with AMUSE comprised inconsistencies between the UML standard and the EA support thereof. It has been shown that full support for modeling all fUML elements according to the exact specifications contained in the fUML standard is a requirement for a full integration. Currently, one must rely on unintuitive workarounds. This is a major point of concern when deciding to implement the approach suggested in this master's thesis, because it possibly hampers both the functionality and the usability of the resulting integrated model execution environment.

The approach to map the model representation used by a UML tool to the representation format of the fUML reference implementation by hand has been shown to be cumbersome, but effective in dealing with inconsistencies between the tool and the standard. This is because it gives full control of the mappings to the developer. Therefore, we recommend taking this approach when integrating fUML with a similarly inconsistent UML tool. Should the model execution tool adhere to the standard more rigidly, however, it may be promising to utilize one of the other approaches discussed in Section 4.2.2.

Lastly, a difference in programming languages between the model execution tool and the fUML reference implementation possibly requires the application of additional tools. In the case of this master's thesis, the IKVM toolkit was used to make the Java implementation of fUML usable in AMUSE, which is implemented in C#. Such toolkits, however, increases the complexity of the resulting model execution tool. The implementation created during this master's thesis has shown minimal problems resulting from the involvement of different programming languages.

Overall, the prototype created in the course of this master's thesis has shown that the approach of integrating fUML into an existing tool is theoretically sound and technically feasible. With the fUML reference implementation acting as a mostly stable and adequately performant execution environment, the resulting prototype shows that employing the approach proposed in this thesis can enhance an existing model execution tool with fUML standard compliant model execution capabilities for class diagrams and activity diagrams.

6.2 Future Work

Both the created prototype, as well as the evaluation thereof can be expanded in several meaningful ways. In the following, we discuss these points for future work in more detail. Furthermore, we also give an overview of steps needed for taking the prototype to operational use.

6.2.1 Integration of Sequence Diagram and Activity Diagram

The AMUSE execution environment only provides a limited amount of functionality for sequence diagrams. In particular, it only utilizes them as an initialization means for the execution. This initialization is conducted by creating objects for the defined lifelines and assigning properties via operations called by messages exchanged between the lifelines. At the moment, these exchanged messages need to be in a specific format, which is then converted directly into code. In a first step, this functionality could be expanded to also support calling fUML activities both for creating objects and setting properties. As soon as this step is completed, research can be conducted into fully utilizing sequence diagrams in conjuction with the fUML activity diagrams, e.g., for calling sequence diagrams from activity diagrams.

6.2.2 Execution of Opaque Behaviors Defined in the Model

The prototype supports the usage of opaque behaviors, which are pre-implemented behaviors that can be invoked with a call behavior action. In addition to the opaque behaviors provided by fUML, the prototype provides additional behaviors, which are used in the conducted case studies. All of these provided opaque behaviors consist of hard-coded behaviors which are instantiated at runtime. With the prototype, a model must contain an opaque action annotated with a specific stereotype that corresponds to the specific opaque behavior to invoke it. Ideally, this mechanism would be replaced by providing a library of models of these opaque behaviors with the fUML prototype, which could then be invoked via a normal call behavior action. This would allow editing the opaque behaviors via the modeling UI, and remove the need for the stereotype applications.

6.2.3 Integration of New Versions of Used Tools

Due to continuing problems introduced by frequently updating the used versions of all tools and libraries required by the prototype, the versions were frozen during the development of the prototype and have not been updated since. The visualization errors present in the current implementation of the prototype could be possibly solved by incorporating new versions of AMUSE. An updated calculation of execution ids could be obtained by updating the used MOLIZ fUML Debug API version, which would further improve the stability of the activity diagram execution mechanism of the prototype. In the case of EA, the evaluation was conducted with version 10, which has already been succeeded by Version 11. This update would be worth looking into for UI improvements, as well as the inclusion of action types that were not supported by version 10, such as the reduce action.

6.2.4 Continuation of the Evaluation

Continued Usability Study

As this thesis was mainly concerned with technical aspects of integrating fUML with UML tools, the usability aspects of using fUML via the interface of EA have not been thoroughly investigated. Doing so could provide further insights into the applicability of the prototype and fUML in general.

Comparison of Integration with Other Tools

The evaluation compared the functionality, as well as performance measures of the developed prototype with AMUSE. Comparing the prototype also with other UML tools supporting model execution would provide further insights into the practicability of fUML.

6.2.5 Necessary Steps for Creating a Shippable Version of the Prototype

Both the functionality and the performance of the current version of the prototype are suitable for the prototype to be shipped as a modified version of the AMUSE plugin. Nevertheless, creating a shippable version of the prototype requires a few modifications to the current implementation. Firstly and most importantly, the prototype must be upgraded to the current version of EA and AMUSE, as these versions are the ones distributed to the customers. Secondly, it is necessary to improve the error-handling and logging capabilities of the prototype significantly beforehand. The prototype is currently not fit to communicate exceptions that occur during the execution in a user-friendly manner, which can diminish its usability for the end user. As soon as these two challenges are overcome, the prototype could be delivered to customers for operational use and further evaluation.

Abbreviations

AMUSE	Advanced Modeling Using Simulation and Execution
EA	Enterprise Architect
fUML	Foundational Subset For Executable UML Models
MDD	Model-Driven Development
OMG	Object Management Group
UML	Unified Modeling Language

Bibliography

[1] Baresi, L., Morzenti, A., Motta, A., and Rossi, M. A Logic-based Semantics for the Verification of Multi-diagram UML Models. *SIGSOFT Softw. Eng. Notes 37*, 4 (2012), 1–8.

[2] Börger, E. The Abstract State Machines Method for High-Level System Design and Analysis. In *Formal Methods: State of the Art and New Directions*. Springer, 2010, pp. 79–116.

[3] Börger, E., Cavarra, A., and Riccobene, E. On Formalizing UML State Machines using ASMs. *Information and Software Technology 46*, 5 (2004), 287–292.

[4] Cavarra, A., Riccobene, E., and Scandurra, P. A Framework to Simulate UML Models: Moving from a Semi-formal to a Formal Environment. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (2004), SAC '04, ACM, pp. 1519–1523.

[5] Crane, M. L., and Dingel, J. Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds* (2008), CASCON '08, ACM, pp. 8:96–8:110.

[6] France, R., Evans, A., Lano, K., and Rumpe, B. The UML as a Formal Modeling Notation. *Computer Standards & Interfaces 19*, 7 (1998), 325–334.

[7] Fuentes, L., Manrique, J., and Sánchez, P. Execution and Simulation of (profiled) UML Models using Pópulo. In *Proceedings of the 2008 International Workshop on Models in Software Engineering* (2008), MiSE '08, ACM, pp. 75–81.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[9] Harel, D., and Kugler, H. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*, vol. 3147 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 325–354.

[10] HAUSMANN, J. H. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2005.

[11] HITZ, M., AND BERNAUER, M. *UML@ Work: Objektorientierte Modellierung mit UML 2*. Dpunkt. verlag, 2005.

[12] HÖFIG, E., DEUSSEN, P. H., AND SCHIEFERDECKER, I. On the Performance of UML State Machine Interpretation at Runtime. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2011), SEAMS '11, ACM, pp. 118–127.

[13] JÜRJENS, J. Formal Semantics for Interacting UML Subsystems. In *Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems V* (2002), FMOODS '02, Kluwer, B.V., pp. 29–43.

[14] KIRSHIN, A., DOTAN, D., AND HARTMAN, A. A UML Simulator Based on a Generic Model Execution Engine. In *Proceedings of the 2006 International Conference on Models in Software Engineering* (2006), MiSE '06, Springer, pp. 324–326.

[15] KOHLMEYER, J., AND GUTTMANN, W. Unifying the Semantics of UML 2 State, Activity and Interaction Diagrams. In *Perspectives of Systems Informatics*, vol. 5947 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 206–217.

[16] LAURENT, Y., BENDRAOU, R., AND GERVAIS, M.-P. Executing and Debugging UML Models: An fUML Extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (2013), SAC '13, ACM, pp. 1095–1102.

[17] MA, Z., SHENG, Z., GU, L., WEN, L., AND ZHANG, G. DVM: Towards a Datacenter-scale Virtual Machine. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (2012), VEE '12, ACM, pp. 39–50.

[18] MAYERHOFER, T. Breathing New Life into Models. Master's thesis, Vienna University of Technology, Vienna University of Technology, 2011.

[19] MAYERHOFER, T., AND LANGER, P. Moliz: A Model Execution Framework for UML Models. In *Proceedings of the Second International Master Class on Model-Driven Engineering: Modeling Wizards* (2012), MW '12, ACM, pp. 3:1–3:2.

[20] NITTO, E. D., LAVAZZA, L., SCHIAVONI, M., TRACANELLA, E., AND TROMBETTA, M. Deriving Executable Process Descriptions from UML. In *Proceedings of the 24th International Conference on Software Engineering* (2002), ICSE '02, ACM, pp. 155–165.

[21] OMG. Unified Modeling Language (UML), Version 2.4.1. `http://www.omg.org/spec/UML/2.4/`, 2010.

[22] OMG. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1. `http://www.omg.org/spec/FUML/1.1/`, 2011.

[23] PASTOR, O., ESPAÑA, S., PANACH, J. I., AND AQUINO, N. Model-Driven Development: Piecing Together the MDA Jigsaw Puzzle. *Informatik-Spektrum 31*, 5 (2008), 394–407.

[24] RIEHLE, D., FRALEIGH, S., BUCKA-LASSEN, D., AND OMOROGBE, N. The Architecture of a UML Virtual Machine. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2001), OOPSLA '01, ACM, pp. 327–341.

[25] RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[26] SARSTEDT, S. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. PhD thesis, Fakultät für Informatik, Universität Ulm, 2006.

[27] SIEK, J., AND TAHA, W. A Semantic Analysis of C++ Templates. In *Proceedings of the 20th European Conference on Object-Oriented Programming* (2006), vol. 4067 of *Lecture Notes in Computer Science*, Springer, pp. 304–327.

[28] STACHOWIAK, H. *Allgemeine Modelltheorie*. Springer, 1973.

[29] W3C. Scalable Vector Graphics (SVG). `http://www.w3.org/TR/SVG12/`, 2011.