Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat.)

durch die
Fakultät für Wirtschaftswissenschaften
Institut für Informatik und Wirtschaftsinformatik
Universität Duisburg-Essen
Campus Essen

vorgelegt von
Dipl.-Wirt.Inf. Moritz Balz
geboren in Essen

Essen (2011)

Tag der mündlichen Prüfung: 31. August 2011

Erstgutachter: Prof. Dr. Michael Goedicke

Zweitgutachter: Prof. Dr. Bruno Müller-Clostermann

Acknowledgements

I would like to thank all who supported me during the completion of the thesis. This thesis would not have been possible without the valuable feedback from my first supervisor, Prof. Dr. Michael Goedicke, and the freedom he gave me to pursue my interests. I would also like to thank Prof. Dr. Müller-Clostermann for agreeing to be my second supervisor.

I'm really grateful for all the helpful comments and hints I received from my friends and from my colleagues in the institute. My special thanks go to Michael Striewe: He was the first to grasp the idea of embedded models, and we discussed it intensely in the last years, which helped me to shape the approach described in this thesis. Michael especially contributed the application of graph transformations to embedded models and the related tools (cf. sections 4.4.3.3 and 4.4.7). I would also like to thank Malte Goddemeier who developed the state machine design tool in his master's thesis (cf. section 4.4.1).

Abstract

Models in software engineering are descriptive structures so that transformations can connect their contents at a semantic level. In model-based software development, algorithmic program code usually exists alongside models – derived from them or with the purpose to amend them. While thus both kinds of notations must be considered by developers, no consistent mapping is given since transformations between models and code are usually unidirectional for code generation. This impedes a continuous integration of both, limits the applicability of models, and prevents error tracking and monitoring at run time with respect to models.

In this thesis, the approach of *embedded models* is introduced. Embedded models define patterns in program code whose elements have formal relations to models and can be executed by reflection at the same time. Model specifications are thus embedded in implementations and can be accessed by bidirectional transformations for design, verification, execution, and monitoring. The thesis focuses on the development of such patterns and their precise description as well as on the connection to other program code surrounding embedded models. Implementations are described for two modeling domains, state machines and process models, including tools for design, verification, execution, monitoring, and design recovery. The approach is evaluated with two case studies, the modeling of a real-world load generator for performance tests and the development of model-based educational graphical scenarios for university teaching.

Both case studies show that the approach is valid and fulfills its purpose for a certain class of applications. Focusing on the integration in implementations, embedded models are thus a bottom-up approach for model-based software development.

Contents

Figures

Listings

Chapter 1

Introduction

This chapter outlines the background and motivates the work of this thesis in section 1.1. Based upon this we define the related problems and the goals that must be reached to solve them in sections 1.2 and 1.3. The approach of the thesis is summarized in section 1.4, followed by a justification that no prior research has solved the problems in section 1.5. We present the structure of the conceptual thesis part in section 1.6 and give a road map of the entire thesis in section 1.7.

1.1 Background and Motivation

The use of models for the specification of software is desirable when the software is comprehensible at different levels of abstraction. Models allow to consider aspects of a system under development from a domain-specific view. Often a visualization concept exists that eases development. An example for this are process models which allow to represent certain activities, their sequences, and decisions for different paths graphically. If the models are formally founded, they can be simulated and even be used to prove certain properties. A simulation is possible, for example, for state machines whose well-defined state space enables a thorough verification.

However, real-world software systems are usually larger and more multi-faceted than such domain-specific aspects. This means that such a system must at least be assembled from different domain-specific views. In many cases, some aspects of the system are not covered by modeling techniques at all. The reasons are manifold: (1) Software can be very specific to a certain purpose and thus very complex, e.g. due to the need for particular algorithms, communication with special hardware, or the use of certain programming interfaces; (2) software may use new or experimental technologies which are not covered by domain-specific models, e.g. new application frameworks, programming languages, or server components, so that appropriate algorithms are developed manually; (3) software can embrace legacy systems that are not covered by current modeling technologies.

Thus the information expressed in models at higher levels of abstraction must be broken down to achieve complete implementations. This can happen along different abstraction levels wherein the abstract information is gradually

supplemented with detailed specifications, e.g. with respect to frameworks or server software that will execute the complete application later on. This refinement, be it manual or automated with appropriate tools, leads to lower levels of abstraction containing detailed imperative statements that will be executed at run time. In most cases, general-purpose programming languages are used that can represent arbitrary execution logic. The fact that not every aspect of a system can be modeled does often lead to the situation that a large portion of functionality as well as implicit knowledge and documentation of the software is still represented within manually-written source code.

This leaves a semantic gap between high-level representations of software and the actual systems they abstract from: The desire to model different well-defined aspects of software contradicts the fact that parts of the software are not covered by appropriate models. Thus different levels of abstraction exist that have to be synchronized. Even worse, this gap becomes manifest with the existence of different explicit notations: Beside the general-purpose program code, description languages for model specifications exist that cannot be connected to implementations directly since they represent different abstraction levels. The need to express more and more aspects of software in modeling languages leads to the situation that these languages are becoming more and more complex, as has been observed by Martin Fowler in the context of the Model-Driven Architecture (MDA) [Mukerji and Miller, 2003]: "MDA uses a bunch of OMG standards (UML, MOF, XMI, CWM etc), these standards are every bit as much a platform as the Java stack (or the .NET stack for that matter)." [Fowler, 2003a] Such stacks of modeling languages exist in parallel to general-purpose programming languages, which are still needed to derive compiled byte code or machine code.

The current approaches to model-driven software development (MDSD) acknowledge these problems and provide assistance in working with different abstraction levels and different notations. However, some inherent problems remain that we will consider in the next section.

1.2 Problem Definition

When software systems are developed, several stages are passed through, in which different notations are used to describe certain overlapping aspects of the software under development at different abstraction levels and with different purposes. This is especially true when formal models are used to specify the software, since multiple abstraction levels are maintained in different notations in this case. The existence of different notations entails that important information is not available in a consistent way; even worse, different pieces of information in different notations are hard to synchronize if the software is developed and changed over longer periods of time. An overview of notations to encounter during development is given in figure 1.1. They can roughly be classified as follows:

At design time, the software architecture and its provided functionality is derived from the requirements and recorded in semi-formalized description languages or formalized models. Based on this, source code is derived from the specifications during implementation, either with manual programming or – in the case of some MDSD approaches – with code generation. The nota-

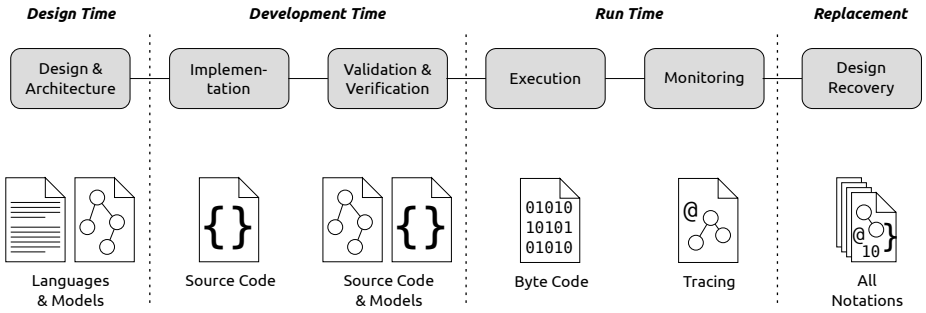


Figure 1.1: Different stages of development processes and notations that are used to describe the software or certain aspects of it. While all notations focus on different views on the same software, they are hard to synchronize over time.

tion of the source code belongs to certain programming languages which can be general-purpose or domain-specific languages. Design and implementation can be verified afterwards, either at the level of formal specifications or of the programming language. At run time, the software system is represented by compiled machine code or byte code which contains detailed imperative statements. Running systems can be monitored, which usually relies on the existence of meta data relating executable code to specifications at a higher level of abstraction. Finally, when a software system is to be replaced, design recovery activities can be applied to all existing information in all notations to transfer knowledge into follow-up systems.

These different notations are usually independent and cannot be synchronized automatically because they are too different and describe only specific aspects of the overall system. The notation of a general-purpose programming language is used in most cases to express algorithms in detail. This is even true if MDSD is used to derive the source code from high-level model specifications since most systems cannot be modeled completely, so that the derived source code has to be amended, tuned, or customized afterwards to fit special requirements, performance constraints, or new or unsupported frameworks in use. Therefore, MDSD approaches are often not appropriate to reduce the number of notations developers have to work with since some aspects must be expressed in program code that does not integrate seamlessly in high-level specifications. In this context, the following issues with respect to the use of models can be observed:

- Since a model considers only a limited set of aspects of a system, the notation and tools for this model may be very specific. Thus, it is hard to assemble a software system from different domain-specific views since the semantics and notations may not have well-defined interfaces or, worse, may be inconsistent.
- The need to assemble a software system from different specific views may require interface code that must be written manually.
- Not every aspect of a software system can be (efficiently) expressed at the abstract level of model specifications. Especially for fine-grained behav-

ioral semantics, this leads to source code being written manually to fulfill requirements.

- The integration of specific, innovative, or legacy components may require program code that is not covered by modeling technologies. The software under development will in this case consist of different parts that are developed at different levels of abstraction.

In this context, current MDSD approaches usually assume that source code can be derived from abstract semantic model representations, either manually or in an automated way. The semantic gaps are by this means bridged only unidirectionally. This leads to additional problems:

- Considering the issues listed above, generated program code is often customized or amended. In general, the derived system will thus consist of generated and manually-written program code at the same time.
- Changes or supplements to generated source code cannot easily be tracked back into the model, which impedes a continuous integration.
- At run time, the software is represented by low-level imperative statements in compiled code and not by sophisticated models. This makes it difficult to track errors with respect to model semantics.
- Because of different representations at development time and run time, it is also in many cases not possible to apply meaningful monitoring that is related to the semantics of models.

In summary, the advantage of model-driven software development – to work with domain-specific views at different levels of abstraction – is accompanied by differences and possible inconsistencies regarding semantics and notations. Thus the abstraction levels are not bridged in an unambiguous and consistent way. In the next section, we will outline the requirements of an approach that aims at overcoming such difficulties.

1.3 Requirements and Objectives

The origin of the problems described above are inconsistencies that can occur between different representations and abstraction levels. Hence the objective of our approach to MDSD must be to avoid them. This leads to the following requirements:

- The approach should provide *consistency between model specifications and source code*. The relation between model specifications and source code should thus be bidirectional so that it is not only possible to derive source code from a model, but also to reconstruct model semantics unambigiously from the source code.
- The approach should allow for *integration of arbitrary source code*, including interfaces to source code derived from other models as well as source code that is not based on models at all.

- The approach should provide *consistency between model specifications and executing program code*. The program code must hence at run time reflect the specifications derived from the model. This includes all structures and algorithms that are concerned when the origin of errors or unexpected behavior is to be found as well as error messages which must be interpretable in the context of the model.
- Based upon this, the approach should also provide *consistency between model specifications and monitoring*. In addition to tracking of errors, this affects all structures and algorithms that may represent the model semantics implicitly or explicitly.

In summary, the objective of our approach must be to find a common representation (or a set of common representations) that are capable of expressing the specifications of multiple classes of models as well as arbitrary fine-grained algorithms. We will now present the contribution of the thesis that aims to find such a common representation for different levels of abstraction.

1.4 Thesis Contribution

The inconsistencies described in section 1.2 exist due to different notations that are used to describe software under development with different views at different abstraction levels. Above we thus stated the goal to reduce the number of notations present in many software development activities. However, it is not desirable to reduce the number of views on the software since the various abstraction levels fulfill different purposes, especially when model specifications are used. Thus it is necessary to decouple notations and views by finding representations that are capable to represent more than one abstraction level or specific view on the software under development.

Considering the representations that are currently used during development, the source code is – despite all approaches to develop software at higher levels of abstraction – still a vital part of the set of notations in use. Even more, program code of modern programming languages became more and more expressive over time. In object-oriented programming languages, static structures exist that can be arranged according to (informal or formalized) patterns. Some of the languages support compiled, type-safe meta data that can be added to object-oriented structures to give them certain semantics. Related approaches, which are subsumed under the concept of *attribute-enabled programming* [Schwarz, 2004], interpret the meta data during development time and run time with respect to certain frameworks, e.g. for the definition of components. The use at run time is possible in environments that provide structural reflection to access information about program code during execution [Cazzola, 1998]. One can hence state that program code written in such languages also contains information at different levels of abstraction. By this means, the programs are more than just code since they reflect a certain amount of design information.

Based on this perception, the contribution of this thesis is to use a general-purpose programming language as a common representation for model specifications as well as detailed algorithms. The idea of design patterns [Gamma et al., 1995], which are specified informally, is extended for

this purpose: For abstract models of interest, a program code pattern will be defined that arranges fragments of program code inside this language so that they represent the model specifications. This bottom-up approach embeds model specifications inside arbitrary program code, thus creating what we will call an *embedded model*. The program code is hence comprehensible at different levels of abstraction: Besides the semantics that are inherent to the programming language, subsets of the program code can be interpreted regarding their embedded model specifications. By this means a tight coupling between specifications and implementation is given. This allows to work with embedded models for the following purposes:

- The program code including its meta programming attributes is the main notation for different abstraction levels. It can represent model specifications as well as program code that is written manually. The program code patterns that form the embedded model syntax can define interfaces to arbitrary program code outside the model. A software system is hence not assembled from different views. Instead, these specialized views can be extracted from the code on demand.
- For each embedded model, a lean execution framework is created that accesses the embedded model fragments by means of structural reflection since they are available at run time. Based on this, it invokes fragments of program code to create a sequence of actions matching the execution semantics of the model.
- Monitoring and debugging of applications with respect to their embedded models is thus possible by considering the program code pattern while it is executed.

The approach therefore fulfills the requirements defined in section 1.3. It aims to be applicable in cases when software is developed in terms of program code (and not only with notations representing abstract models), so that the code is a notation that is considered explicitly during development. Other use cases for the utilization of MDSD techniques are not covered by this approach, for example business process platforms that allow for re-configuration at run time or rule-based deployments of customized software products. We will now justify that for the given focus no prior research exists that fulfills the requirements.

1.5 Thesis Justification

The development of complex systems is based on a hierarchical understanding. This applies to software systems, too. The principle of hierarchic comprehension is for example described by Herbert A. Simon's theory of *Near Decomposability* [Simon, 1996]: A nearly decomposable system is constituted by clear hierarchy of single parts, which interact with well-defined interfaces.

Such clear hierarchies are not present during software development when we consider the issues defined in section 1.2: In MDSD, the transformations are usually unidirectional. At development time, information from a model notation is transformed into source code [Brown et al., 2006]. The translation of

abstract representations into low-level representations entails that different hierarchies are in use. When one modeling language is used and the source code is generated from it, the number of hierarchies in use is 2. When more modeling languages are used, for example domain-specific languages which have different semantics, the number of hierarchies is even higher. When source code is developed manually – to adapt generated code or to supplement it – the hierarchies will intersect partly. Since changes are possible in all hierarchies in parallel during development, a permanent synchronization is not completely possible, since the semantics can be inconsistent [Hailpern and Tarr, 2006; Demeyer et al., 1999]. When only semantics of lower abstraction levels, i.e. the program code, are considered, additional information can be gained, but only related to low-level specifications [Nierstrasz et al., 2007], so that inconsistencies between the hierarchies cannot be overcome. Models can also be transformed into executable systems at run time when model notations are interpreted [Luz and da Silva, 2004]. However, they must in this case be integrated into larger systems, which are specified in a different way, since the fact that low-level access to the hardware and software environment is often necessary contradicts the fact that models are intended to abstract from those details.

In embedded models, the program code is considered for model transformations and refinement in the same way as any other model notation. By this means only one hierarchy regarding notations exists: Parts of the program code can contain information at higher levels of abstraction, but are connected to program code with implementation details at the same time. The program code is thus structured hierarchically when some of its fragments are emphasized and have a semantics of their own, but also well-defined interfaces to the other program code.

However, the embedded models approach is not applicable to every software development task, but only to those where different abstraction levels are considered. It does not apply to cases where software is configured or assembled only with descriptive languages, for example business process models in service-oriented architectures (SOAs) [Brahe, 2007]. In such situations the internal structure of a program is not of interest, but only its descriptive service interfaces, so that only one abstraction level is concerned. However, situations like these are rare since they require limited domains of applications and a controlled infrastructure. The implementation tasks will most likely consider program code that is engineered in detail, so that embedded models are of interest. In summary, when different abstraction levels are concerned, the approach satisfies the requirements named above and provides a relevant contribution for the field of model-driven software development.

1.6 Structure of the Conceptual Thesis Part

The problem description in section 1.2 referred to several notations in use during development. As outlined in section 1.4, these notations are to be replaced by views, which are extracted from the program code on demand. This leads to two perspectives the conceptual part of this thesis will take:

The *vertical perspective* considers maintaining multiple abstraction layers in program code. Since the goal is to reduce the number of notations and use the

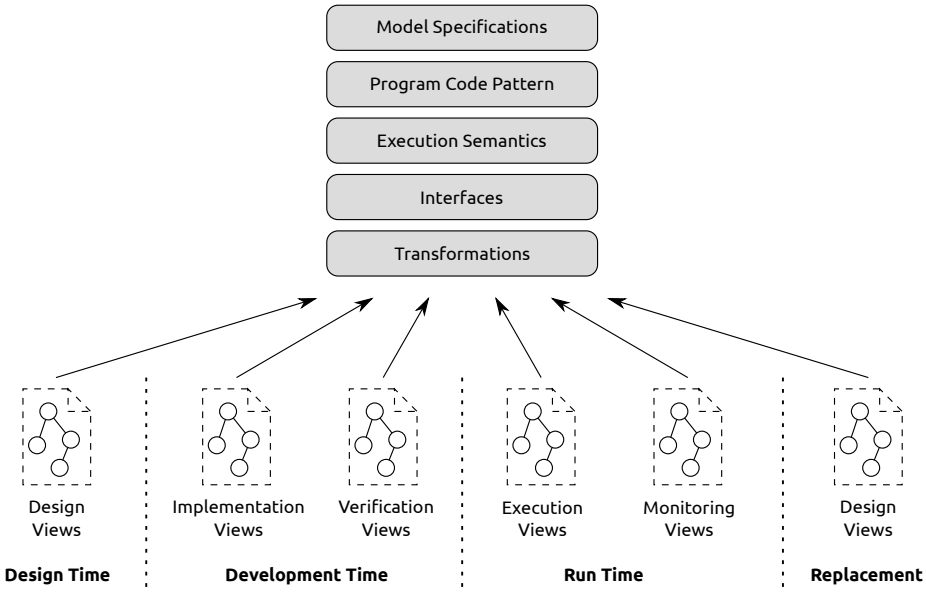


Figure 1.2: The structure of the conceptual thesis part.

program code for this purpose, the code must carry additional information. This leads to a stack that is illustrated at the top of figure 1.2: Based on the semantics of a formal model class, a program code pattern is developed that represents the syntax of model instances. The code of the pattern will be executed by reflection so that an appropriate execution semantics is defined that utilizes the program code pattern at run time. Since the program code pattern can be part of almost arbitrary other program code, interfaces between both exist that realize the actual embedding of the pattern in larger programs. Finally, model specifications and patterns are connected by transformations that can either extract information from the code or create code with respect to a model.

The stack has the purpose to be used throughout the development process. This is reflected in the *horizontal perspective* as shown at the bottom of figure 1.2: For different development activities, appropriate views are extracted from the program code with respective transformations. By this means the approach is useful since tool-based MDSD techniques can be applied.

In this thesis, both perspectives will be described in general first to explain the overall approach. Afterwards they are applied to two model classes, state machines and process models, to give specific information about applicability, implementation, and tools supporting the development with embedded models.

1.7 Thesis Road Map

The thesis is structured as follows:

Chaper 2 considers work that is related to the motivation and implementation of the embedded models approach. For this purpose, the top-down ap-

proaches to MDSD with their concepts, problems, and possible solutions will be described. Afterwards, a review of approaches to make program code interpretable at different abstraction levels is given. The chapter concludes with the definition of a gap in existing approaches that embedded models aim to fill.

The embedded models approach is introduced in **chapter 3**. First, the definition of the vertical perspective is explained in section 3.1, including the introduction of a related meta model and program code structures of object-oriented programming languages that are appropriate to represent embedded models. The horizontal perspective of the development process and the related tool support for embedded models is outlined in section 3.2. Based on this approach for the definition of single embedded models, section 3.3 considers principles of system architectures based on embedded models and their relations. This consideration is influenced by the fact that embedded models may be self-contained regarding the model semantics, but may at the same time rely on interfaces to other program code. Thus, the principles for interaction among embedded models as well as between embedded models and arbitrary program code are explained from the perspective of larger systems that can contain more than one embedded model.

Two classes of embedded models are derived from the meta model in **chapter 4**. We consider behavioral models here since they are more complex to connect to program code than static models. First, state machines are considered in section 4.1, which are useful for systems that need to be verified and simulated with respect to a finite state space. Then the approach is applied to process models in section 4.2 which focus on sequences of actions and related input and output data. Both model types are first defined precisely with respect to the meta model, before the related program code pattern is introduced. We also consider the execution semantics that are of interest when the embedded model code is executed by a framework, the interfaces to other program code, and transformations that allow to consider the code at different abstraction levels. A comparison of the two classes of embedded models is given in section 4.3. For both model classes we present tools that support the development of software with embedded models in section 4.4. At development time, (visual) design tools are of interest as well as tools that can verify embedded models or connect them to domain-specific verification tools by extracting the modeling information from the program code. At run time, the execution of the embedded model code is most important. In addition, tools for monitoring and debugging embedded models are also presented with the objective of supporting the discovery of errors. Finally, design recovery with embedded models is introduced with a tool that transforms program code with an embedded state machine model to an embedded process model.

In **chapter 5** the approach is evaluated. For this purpose criteria are defined at the beginning. These criteria are applied to two case studies: The first focuses on the development of a load generator application for performance tests wherein the load generation algorithm is modeled as a state machine and a process model. Both implementations are compared and then evaluated with respect to the criteria. The second case study reflects on the use of embedded models for teaching purposes within the programming learning environment Greenfoot. Greenfoot allows to create simple graphical simulations that contain entities interacting with each other. These entities are in the case study modeled as state machines so that the simulations can be verified. The case

study comprises a small user study performed with students in a master's course.

Chapter 6 considers the impact that embedded models can have on different aspects of software development and outlines areas of future work at the same time. The topics to be discussed are directions for additional evaluation based on program comprehension concepts, additional modeling domains, interaction of embedded models and meta models, and the question how modeling and implementation languages can be combined.

Finally, **chapter 7** summarizes the problem domain and the approach of the thesis. The novel contributions are emphasized and the applicability of the approach is considered.

The next section will now introduce related work that is important in the context of a bottom-up approach to MDSD.

Chapter 2

Related Work

The problems mentioned in section 1.2 are subject to various research activities, and different classes of related approaches exist. A brief overview of them will be given in this chapter. For this purpose we roughly classify these approaches into two groups:

First, *top-down approaches* focus on models and their computation-independent semantics. Their purpose is to define application logic without consideration of actual platforms and implementation details, thus decoupling the core functionality from underlying technical systems. In these approaches, executable systems are derived from the abstract models and at most amended with manually-written source code. The semantics of formal models are thus considered at a high level of abstraction only. These approaches will be considered in this chapter in section 2.1.

Second, *code-oriented approaches* center on source code that is not derived from other representations, but instead constructed during software development. These approaches focus on the fact that program code does often not only express algorithms in detail, but is at the same time related to abstract concepts also. In section 2.2 we thus describe approaches that try to relate source code to model specifications. This includes a consideration of design patterns and approaches to relate them to abstract specifications.

The consideration of all this related work leads to the question whether competing approaches exist. To determine this, the following criteria were derived from the objectives given in section 1.3:

- *Focus on formal high-level models*: We are interested in formal models that are developed to serve a certain purpose at a higher level of abstraction (e.g. state machines or process models). This excludes, for example, models for verifying low-level algorithms.

- *Consistency between model specifications and static program code structures*: Static program code structures defining the pattern must represent the syntax of the model. The code must therefore be interpretable at design time.

- *Integration in arbitrary program code*: The models must be seamlessly integrated in applications that are not based on the same modeling class or

any model at all, and therefore be appropriate to interact with application logic that is not part of the models.

- *Consistency between model specifications and executed program code*: The program code must at run time reflect the semantics of the model. Execution semantics will be defined that allow for execution by a framework that interprets the model syntax at run time. This consistency must also allow for monitoring and debugging.

During the more than three years of research for this thesis, no approaches fulfilling all these requirements were found. We will now explain the existing work that is related to the thesis and its differences to our concept. The chapter summary will provide a classification of the existing approaches and the embedded models approach.

2.1 Top-down MDSD Approaches

Top-down MDSD approaches assume a clear hierarchy of abstraction levels. At each level, the degree of abstraction is lowered by adding more detailed information about the software under development. Models are in this context represented in explicit notations. These notations carry the precise syntax of the models so that model specifications can always be read from these notations. However, this does not connect models to executable systems. We will now consider the technologies that are used to derive such systems from higher levels of abstraction. The principle is shown in figure 2.1: The model specifications are represented in their notations so that a bidirectional mapping between both exists; the program code is generated from them and in many cases supplemented with other program code that is written manually.

2.1.1 Modeling Languages

Many modeling languages exist that serve different purposes. No common theory is available that describes precisely what defines a modeling language and how its elements are described and connected [Favre, 2004]. Some modeling languages with appropriate meta models, like the Unified Modeling Language (see section 2.1.2), aim at being generic and thus applicable to different domains. In contrast, Domain-Specific Languages (DSLs) [van Deursen et al., 2000; Hudak, 1998] are designed to fulfill a certain purpose. Thus it takes more effort to create systems based on more than one DSL [Estublier et al., 2005]. The use of modeling languages with explicit formal notations implies that an abstract level is considered and executable systems must be derived. The following issues that can occur must be considered [Pastor and Molina, 2007, ch. 19]:

- *Lack of adequacy*: A high level of abstraction implies that a modeling language must exactly fit the problem it shall describe since it cannot compensate lacking adequacy with the ability to express arbitrary semantics. This is due to the fact that at higher levels of abstraction no facilities exist to express detailed algorithms, which would cover problems that cannot be expressed in an abstract model.

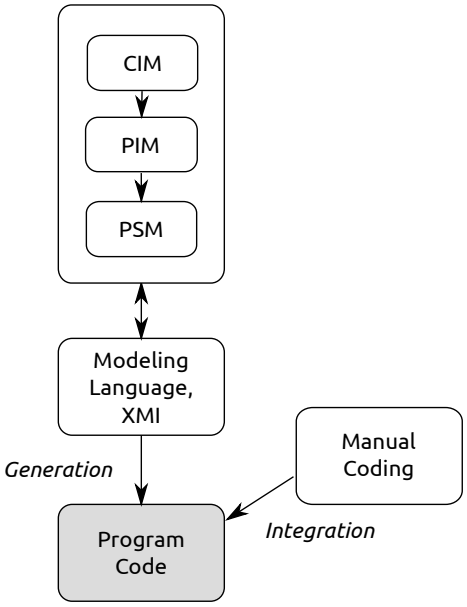


Figure 2.1: The principle of top-down MDSD approaches with terms as used by the MDA (cf. section 2.1.2).

- *Imprecise semantics*: Modeling languages are not necessarily defined completely in a precise and unambiguous way.
- *Strong versus weak formalisms*: Modeling languages based on strong formalisms may be avoided by developers because of their complexity.
- *Low abstraction level*: The aim of modeling languages to lower hurdles for practical application may lead to low abstraction levels, for example for certain frameworks, leading to a high coincidence between models and source code.
- *Action semantics*: The specification of detailed behavior is necessary for the creation of real systems, but hard to achieve at higher levels of abstraction.
- *Formulae*: Missing precise semantics in modeling languages are often balanced by the association of so-called formulae to model elements, for example constraints like OCL for UML (cf. section 2.1.2). However, the addition of such meta information does not always integrate in the modeling language consistently, which can lead to ambiguities.

2.1.2 UML and MDA

Higher levels of abstraction in top-down MDSD approaches are expressed in modeling languages. For these languages a procedure must exist that derives executable systems. Since a multitude of top-down MDSD approaches is available, we will consider here one specific stack of modeling languages and an architectural approach to derive software systems from it.

The most widespread modeling language is currently the Unified Modeling Language (UML) [OMG, 2010] which is standardized by the Object Management Group (OMG). The UML consists of certain so-called diagrams [Fowler, 2003b] which describe certain functional aspects of a system, for example packages, processes, or state charts [Crane and Dingel, 2005]. These diagrams are derived from a meta model, the Meta Object Facility (MOF) [OMG, 2006; Atkinson and Kühne, 2003]. The meta model is also related to an XML-based interchange format, XML Metadata Interchange (XMI) [OMG, 2007]. XMI is thus a notation for different UML and other MOF-based models.

The best-known UML diagrams are class diagrams which model object-oriented properties of software under development including attributes, operations, and relations. Since these diagrams can be mapped to code structures directly, they are easy to use and thus widely accepted. Modeling a complete program, however, includes modeling of its behavior, which is more difficult to express in diagrams. Thus UML offers two ways to specify behavior [Frankel, 2003, ch. 3]:

- *Design by Contract* principles define contracts for operations. The Object Constraint Language (OCL) [Beckert et al., 2007] allows to define constraints where they are appropriate in UML diagrams. This includes among others invariants and pre- and postconditions for operations. It is therefore appropriate to define interfaces and thus to describe the behavioral aspects, but not to specify them.

- *Behavioral models* describe the behavior of programs at a higher level of abstraction, e.g. with sequence diagrams, activity diagrams, or state charts. Because of the higher level of abstraction, the specification of the behavior in this diagrams does not cover all details, but in many cases only a simplified view on the software system that must be amended with detailed algorithms to be executable.

While the static diagrams like class or use case diagrams are easy to use, these concepts for behavioral specifications are harder to grasp and to apply, so their usage varies [Dobing and Parsons, 2006; Agarwal and Sinha, 2003]. Furthermore, UML diagrams are not completely formally founded and thus not in every case appropriate for precise modeling. Therefore, approaches exist that connect UML diagrams to formal modeling techniques like Petri nets [Saldhana and Shatz, 2000; Bernardi et al., 2002; Fernandes et al., 2007] or aim at supplementing the diagrams with formal semantics, for example for state charts [Jürjens, 2002; Gogolla and Presicce, 1998; Varró, 2002; Evans, 1998; McUmber and Cheng, 2001].

Since these techniques are not appropriate to model a complete system, behavioral languages are connected to UML that can help to express more detailed aspects of software under development. To connect different UML fragments, a language is needed that executes actions which are defined in UML models. For this reason, Action Semantics [OMG, 2001; Sunyé et al., 2002; Raistrick et al., 2004] are required that allow to specify the behavior of such actions in detail. UML therefore defines the function set that Action Semantic Languages (ASL) are expected to provide. However, the languages itself are not specified, so that different ASLs exist that are specific to tools they are used

in. With their ability to define almost arbitrary actions, ASLs resemble general-purpose programming languages to a certain degree.

The Model-Driven Architecture (MDA) [Bézivin and Gerbé, 2001; Mukerji and Miller, 2003; Frankel, 2003] is an approach using UML and the related specifications to create software by following the principle of separating functional and technical aspects. MDA proposes three levels of abstraction: The computation independent model (CIM) defines the mere functionality as domain or business models. It may consist of one or more models derived from the MOF. The platform independent model (PIM) is built afterwards by incorporating information about the software architecture of the system to develop. Finally, one or more platform specific models (PSM) are used to define actual implementations for concrete platforms [Wagelaar and Jonckers, 2005]. Thus, the degree of detail is increased at each level. The objectives of the MDA are to make development faster and more efficient by facilitating reuse and also to protect investments against changes in underlying platforms. While the MDA relies on the fact that some platform-dependent program code has to be written manually, the objectives should be accomplished by finding an appropriate balance between the amount of functionality that is modeled and the amount that is written manually.

For this thesis it is of interest that modeling of such systems comprises several different languages, including general-purpose programming languages for platform-specific tasks. For this reason the objective of platform independence is questionable: While the models are indeed independent from actual platforms like server environments, they depend on several specifications of the UML, modeling languages expressing the specifications, and also on specific tools [Fowler, 2003a]. The latter are necessary to create and maintain models as well as transformations and to derive executable systems from them. In addition, the MDA does not provide sufficient support for connections to arbitrary application logic since manually-written program code is only considered at lower levels of abstraction.

In contrast, the approach of this thesis facilitates the use of only one notation that is appropriate to maintain different levels of abstractions and different domain-specific views on the software to develop. As will be seen, the objective of the MDA to be more efficient is supported by embedded models with the provision of several views on the program which enable efficient and tool-supported development of domain-specific aspects within the program code. The MDA's objective of being platform independent is naturally in contrast to the use of a certain programming language. However, since the model syntax is completely available in an embedded model, the specifications can always be extracted and employed for several purposes, including transformation into a modeling language or into embedded models in other programming languages.

2.1.3 Code Generation

When models are defined at higher levels of abstraction, it is necessary to derive executable programs from them. These programs are not only required to follow the model semantics in the first place, but must also consider actual platforms, for example operating systems, programming languages, frameworks,

or server software. Top-down MDSD approaches acknowledge that general-purpose programming languages are appropriate to express all needed semantics and integrate them in actual platforms. The preferred way to propagate high-level representations into executable programs is therefore to generate such source code [Brown et al., 2006]. This implies that at different levels of abstraction, different kinds of semantics exist: Besides the abstract model semantics, code generation also introduces execution semantics that depend on the generation strategy, programming language, and also platforms to be used. The differences can for example be seen for the approach of generating Java source code from UML state charts [Niaz and Tanaka, 2003]: While parts of the state chart are represented as static structures in the program code and thus identifiable as part of the model, the generated code also contains fine-grained behavioral logic that cannot be related to the model fragments.

Generated code can be complete in cases where the environment is limited [Lindlar and Zimmermann, 2008]. However, since (meaningful) models must abstract from all details, the generated source code is likely not to be complete when it must be connected to complex environments or other parts of larger applications. Besides model-based enrichment at the PIM and PSM layers, it is often necessary to amend the generated source code manually. Such changes lead to the situation that generated source code has no completely formal reference to the higher-level model semantics. Models are in many cases related to developed systems only by the developer's knowledge [Tichy and Giese, 2003]. This prevents at first an automatic back tracking of changes [Hailpern and Tarr, 2006; Baker et al., 2005]. Even worse, when generated source code is amended or tuned after generation, a regeneration of parts of it after changes on the model levels defies a gradual integration [Vokáč and Glattetre, 2005] and incremental development.

This problem is considered by several approaches, for example the Generation Gap Pattern [Vlissides, 1996]. In the motivation it is stated that "having a computer generate code for you is usually preferable to writing it yourself, provided the code it generates is correct, efficient enough, functionally complete, and maintainable." Because of this, the pattern proposes to differentiate between generated and non-generated source code by using sub classes: "It encapsulates generated code in a class and then splits that class into two, one class for encapsulating generated code and another for encapsulating modifications." The approach has a limited applicability as some conditions must be met, especially that "generated code can be encapsulated in one or more classes" and "regenerated code [...] retains the interface and instance variables of the previous generation".

The OO-Method approach [Pastor and Molina, 2007] distinguishes between model-driven and model-based code generation and classifies itself as the latter. The basic idea is that the code is generated out of so-called Conceptual Schemas [Olivé, 2005] and therefore the properties of object-oriented artifacts as well as formal methods based on them can be considered. However, generation of executable source code is still necessary. MetaBorg [Bravenboer et al., 2006; Riehl, 2006] aims at "embedding" DSLs into general-purpose programming languages, however, this means that code in the general-purpose programming language is generated from DSLs and inserted at the right place in other program code.

In summary, code generation is a way to derive executable programs from

higher-level representations. However, the different explicit notations used at the different abstraction levels lead to inconsistencies since they are based on different semantics. While differences between the notations can be bridged unidirectionally from the technical perspective and as long as several restrictions as mentioned above are accepted, an automated mapping between the different semantics is currently not feasible. Considering the criteria defined above, code generation can therefore not accomplish the objectives of this thesis: A consistency between model specifications and program code is given neither at development time nor at run time. It is also difficult to integrate generated and non-generated program code systematically.

2.1.4 Management of Inconsistencies

Since such a mapping between different semantics is not directly available, approaches exist that aim at reconstructing model semantics from generated program code and therefore make a round-trip engineering with code synchronisation possible [Sendall and Küster, 2004]. However, these approaches still require manual effort and are thus error-prone since the required information is not available at all abstraction levels, so that they are mostly based on heuristics [Hailpern and Tarr, 2006; Demeyer et al., 1999]. A precise and automated round-trip engineering is only possible for models at the level of the semantics of the programming language like class diagrams or behavioral models describing the control flow in detail, like realized by FUJABA [Nickel et al., 2000]. The approach of mapping UML models to code structures by means of metadata inside the source code [Wada and Suzuki, 2005] eases the identification of code fragments that are subject to round-trip engineering, but still cannot avoid or fully automate it.

Similarly, reverse engineering approaches utilize patterns and models to recover design information from existing software systems. In the case of FUJABA, design patterns are detected by means of static analysis of source code, and behavioral models are inferred from run time traces [Wendehals et al., 2004], e.g. based on automata [Wendehals and Orso, 2006]. This is implemented by the tool Reclipse [von Detten et al., 2010]. However, neither informal design patterns nor low-level models are of interest in the context of this thesis. Instead, embedded models focus on an unambiguous connection between program code and abstract models in order to eliminate the need for reverse engineering.

Several approaches exist that aim at managing the inconsistencies that occur throughout the development process [Nuseibeh, 1996] or resolve them, for example by means of graph transformations between different so-called ViewPoints [Goedicke et al., 1999]. Both considerations assume that different notations are necessary and are thus only partly related to the approach of this thesis: If different notations including program code are used to specify design and implementation information at different abstraction levels, embedded models are applicable so that the number of notations can be reduced and the different abstraction levels are only views on one notation. In the case that different notations are necessary nonetheless, for example if different stakeholders use different tools during development, management of inconsistencies is still necessary and applies to the program code patterns of embedded models, too.

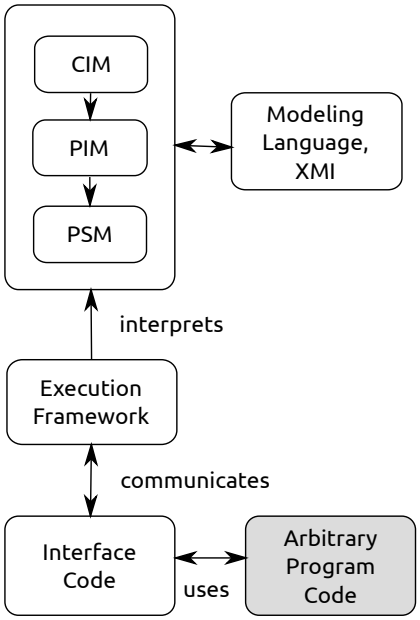


Figure 2.2: The principle of interpretational MDSD approaches.

In summary, the aim of model round-trip engineering and inconsistency management is to bridge the gaps between different abstraction levels by enabling transformations that will work regardless where changes were made. However, as long as ambiguities cannot be precluded, the related concepts are not appropriate to accomplish the objectives introduced in section 1.3.

2.1.5 Model Execution

A way to avoid the generation of source code is to execute model specifications directly, as for example done by Executable UML (xUML) [Luz and da Silva, 2004; Mellor and Balcer, 2002]. As usual for UML, the fundamental UML static class diagrams are used in the first place [Starr, 2001]. However, in order to create executable systems, xUML also employs several specifications related to UML, mainly diagrams, but also languages to express behavior and constraints [Raistrick et al., 2004] (see also section 2.1.2).

The principle of these approaches is outlined in figure 2.2. While it allows for a clean and model-centric view of systems, it either relies on the assumption that entire applications can be expressed as a model or must provide integration layers with arbitrary program code. The assumption as well as the provision of integration layers will in many cases fail, especially in heterogeneous environments, for certain reasons: (1) A model or the related modeling tool may not be sufficient to express all functional requirements in detail, especially for behavioral models; (2) existing specialized frameworks or legacy systems may cause special requirements that are not covered by models or modeling tools; (3) non-functional requirements, e.g. performance, may have high priority and demand a deviation from the structure an abstract model would imply.

The Executable OCL (xOCL) approach [Jiang et al., 2007, 2008] acknowledges that "UML is not defined precisely enough for unambiguous model execution. Therefore, the first and most important requirement for model execution is precisely modeling the actions." [Jiang et al., 2007] The approach tries to combine ASLs and OCL to reach a higher expressiveness. Similar to this is eXecutable OCL [XOCL, 2008]. While the scope of the executed models is by this means enhanced, the problems of executable models as stated above are not solved.

The UniMod tool [Gurov et al., 2005] connects executable UML to program code by defining a state machine model that executes Java code in every transition. While this allows to structure the program code and model the behavior of an application at different levels of abstraction, it requires at the same time different notations with possible inconsistencies. A verification and simulation of the model is also difficult because modifications of the system state in the arbitrary program code are not considered part of the model. The ModelTalk framework [Hen-Tov et al., 2008] executing domain-specific models alongside Java program code relies on different notations, too, as the models are specified in XML-based languages. At run time they are instantiated by the framework and their representations in Java objects are injected into other objects using the framework. Despite the integration in Java, the development of applications using this approach still requires different languages with different semantics and notations. The approach of Kermeta to weave executability into metamodeling languages [Muller et al., 2005] embraces explicit modeling notations for the meta models, for example OCL for constraints.

Several similar approaches exist. However, they do not satisfactorily answer the question how to deal with different notations. This means that all general-purpose executable model languages either imply that all details of programs must be modeled cumbersome in detail or that interfaces to arbitrary program code must be defined which are not seamlessly integrated in the modeling languages. They do therefore not fulfill the criterion of seamless integration in other program code, and are in many use cases not applicable in general since not every implementation detail can be efficiently expressed in abstract models.

2.2 Source Code Semantics and Design Patterns

So far we considered top-down approaches that focus on modeling languages and derive implementations (or at least parts of implementations) from them. Since embedded models are not only closely related to models, but also to program code, we will now also look at approaches that interpret program code with respect to the semantics it contains. Depending on the approach, these semantics can be considered at higher levels of abstraction, be formalized, and also be related to models.

2.2.1 Abstract Specifications in Program Code

Several concepts exist that enrich program code with specification information that is interpretable at different abstraction levels.

This thesis considers object-oriented programming languages that add additional abstraction layers to plain algorithms by defining the semantics of objects and their relations [Craig, 2001]. Thus, they are interpretable at multiple levels [Pastor and Molina, 2007]: Classes "appear in the statement of the problem in a more or less natural way". The meaning of their relationships depends on the "preciseness of the associated semantics": "class relationships have more semantics than those proposed by most methods, and the conceptual schema will be precise only if these relationships are clearly defined." [Pastor and Molina, 2007] Thus, object-oriented languages must be supplemented with formal specifications if their expressiveness should be increased.

For this purpose, formal specification languages and architectural patterns are appropriate. This is for example done in the approach of Specification-Carrying Code [Serugendo and Deriaz, 2005]. The semantics of methods are described as formal specifications, for example in Prolog [Kim, 1991], and stored in an appropriate XML-based notation. Implementing classes register themselves in a framework that provides services according to specification files. Clients can lookup service implementations by giving the specification file and make according method calls. While this enhances the formal description of object-oriented programs, the implementation is still independent from the formal specification, since notation and semantics of specification and implementation language are different. Thus, the correctness of an implementation can hardly be determined automatically. Similar, Software Reflexion Models [Murphy et al., 2001] connect program code fragments to high-level models, but do not facilitate a single-source strategy. Proof-Carrying Code [Necula, 2002] supplies program code that is to be executed in a specific environment with meta data proving that the code is working as intended and proposes verification mechanisms, too. However, this also relies on external notations and is not related to higher-level formal models, but to specification details instead.

Framework Specific Modeling Languages [Antkiewicz and Czarnecki, 2006] are similar to DSLs, but consider source code fragments that are related to certain domain-specific frameworks. Thus, the program code elements are clearly identifiable so that references to model elements can be extracted from the program code [Antkiewicz et al., 2007]. This makes a continuous round-trip engineering possible [Antkiewicz, 2007]. While it allows to maintain different abstraction levels in the program code, the objectives are different to our approach: Framework-specific models aim at increasing the quality of program code that is based on existing frameworks like Java Applets, Struts applications, or Eclipse plugins [Antkiewicz et al., 2009]. Thus, they consider only the semantics of frameworks and not that of formal models.

Introspective Model-Driven Development (IMDD) [Büchner and Matthes, 2006] has similar objectives as the Embedded Models approach proposed in this thesis: "[An] advantage of IMDD is the possibility to achieve symbolic integration between declarative models and imperative artifacts. This makes it easy to mix both programming styles, and get the benefits of modeling the declarative aspects on a high level of abstraction." This single-source approach also avoids the need to work with different explicit notations to derive executable systems. But, the references to model elements are generic and not

```
for (Tuple2 tuple2:
    squill
      .from(c, c.customer)
      .where(
        gt(c.customer.isActive, 0),
        notNull(c.percentSolved),
        notNull(c.refoundSum))
      .orderBy(desc(c.customer.id))
      .selectList(
        c.customer.lastName,
        c.percentSolved)) {
    System.out.println(
      "Customer␣" + tuple2.v1 + "␣has␣a␣complaint␣solved␣" + tuple2.v2 + "%");
  }
```

Listing 2.1: An embedded DSL example for the Squill framework [Squill].

related to specific properties of formal models, thus leading to another focal point than embedded models.

Similar to this, Internal DSLs "are particular ways of using a host language to give the host language the feel of a particular language." [Fowler, 2006a] This means that the semantics of a DSL are available inside program code written in general-purpose programming languages and at the same time that the related program code is interpretable with respect to the DSL [Fowler, 2006b; Bravenboer and Visser, 2004; Cuadrado and Molina, 2009; Hofer et al., 2008]. An example for this is the Squill framework [Squill] which arranges method calls inside Java so that they resemble SQL statements (see listing 2.1). Another example is LIQUidFORM [LIQUidFORM] which uses similar patterns to allow building queries for the Java Persistence API [Sun Microsystems, Inc., b]. While internal DSLs make source code interpretable with respect to the underlying domain concepts, their usage is limited to simple DSLs whose semantics can be represented in chains of statements [Freeman and Pryce, 2006], which precludes consideration of more sophisticated models. In addition, they are only partly represented by static structures that are accessible by reflection at run time. The concept of internal DSLs has therefore similar purposes, but does not accomplish all objectives of embedded models.

Design patterns [Gamma et al., 1995] propose unified program code structures for repetitive tasks. They establish relations between program code and mental models the code is based on [Storey, 2005; Schauer and Keller, 1998], but have limitations: Design patterns were originally intended to be easily adaptable for special situations and thus are not formalized. Approaches to describe intentions of patterns explicitly [Meffert, 2006] are informal and therefore not of interest with respect to our objectives. Approaches to formalize design patterns [Soundarajan and Hallstrom, 2004; Mikkonen, 1998; Meijler et al., 1997; Eden et al., 1997a,b] aim at specifying the relation between the pattern and the related program code more precisely. This often involves the definition of a Pattern Description Language (PDL) [Albin-Amiot et al., 2001] or a Pattern Specification Language (PSL) [Taibi and Ngo, 2003], respectively. Such PSLs can even be used for code generation [Taibi and Mkadmi, 2006]. The patterns themselves can also be specified more precisely by using a modeling language like the UML [Guennec et al., 2000; Mak et al., 2004] or a modeling language that is specific for this purpose [Mili and El-Boussaidi, 2005; El Boussaidi and Mili, 2007]. In addition, appropriate meta models can be de-

```
@Stateful
@Local(Service.class)
public class ServiceBean implements Service
{
  @EJB
  Reference ref;

  public void doSomething()
  {
    ref.doSomething();
  }
}
```

Listing 2.2: An Enterprise Java Bean example for attribute-enabled programming.

veloped [Albin-Amiot and Guéhéneuc, 2001] and connected to existing modeling languages, for example the UML [Kim et al., 2003]. Design patterns are thus different from embedded models since their primary goal is not to represent a formal specification. This is not different in cases when the patterns themselves are specified formally since this does not imply relations to formal specifications at higher levels of abstraction.

Attribute-enabled programming (AEP) [Schwarz, 2004] uses the capability of modern programming languages to incorporate type-safe, compiled meta data to annotate program code fragments. These annotations can be used to make program code semantically interpretable even at run time, which is already used by many frameworks. These frameworks are usually responsible for starting and invoking the related program code following the principle of inversion of control [Fayad and Schmidt, 1997]. In Java, the concept for expressing such meta data is called *annotations* [Sun Microsystems, Inc., 2004]. It is for example used in Enterprise Java Beans [Sun Microsystems, Inc., 2008] as shown in listing 2.2 with an example. With these few lines of code, the following functionality is achieved by the framework:

- The annotation `Stateful` defines that the class `ServiceBean` is the implementation of a server-side component that can be used by clients and keeps its state. Creation of a lookup name and management of references are tasks the framework is responsible for.

- Clients can access this component with an interface that is defined with the `Local` annotation, in this case the type `Service`.

- The component has a reference to another component of the type `Reference`. The dependency is injected in the field `ref`, i.e., when a component of type `ServiceBean` is instantiated, its dependencies are instantiated, too.

This simple example demonstrates the power of AEP: The few annotations and the few static object-oriented structures can be interpreted by frameworks by means of structural reflection [Demers and Malenfant, 1995; Cazzola, 1998] to provide complex functionality. This adds a level of abstraction and additional semantics to the related program code, but facilitates the use of only one source for this. At the same time, developers using such frameworks are

freed from repetitive tasks and can instead direct the framework by annotating their code according to the given rules. But, considering the context of this thesis, AEP does not imply the use of any formal model for the specification of program code elements. It is therefore a mechanism for maintaining different abstraction levels in program code, but not related to formal models by itself. Although AEP could be a tool for approaches that maintain different abstraction levels in the program code, no such approaches are currently known.

In summary, approaches exist that increase expressiveness of object-oriented source code. This is achieved by supplementing the code with information about its relation to high-level specifications or an arrangement of code fragments that allows to interpret them at different abstraction levels. However, none of these approaches accomplishes the goals of this thesis: Either they are not related to formal models, like for example design patterns, or they rely on the existence of external notations, like for example FSML. Internal DSLs and AEP accomplish the goals partly, but are so far not combined to enable embedding of complex model specifications in the program code. Thus no systematical approach exists that maintains different abstraction levels in the program code and makes them accessible not only at development time, but also at run time.

2.2.2 Detection of Patterns and Models

Besides the approaches to enrich program code with additional specification information, there is also continuous work in progress that aims at detecting semantics of patterns and models in arbitrary program code. This is of interest when architecture, design, and structure of program code are unknown and shall be related to higher levels of abstraction.

Several approaches target the detection of design patterns [Shull et al., 1996; Dong et al., 2007; Niere et al., 2002; Philippow et al., 2005]. This leads to the creation of flexible reverse-engineering tools like PINOT [Shi and Olsson, 2006] that are able to recognize design patterns based on descriptions of the program code fragments to expect [Shi, 2007]. This is similar to Reclipse (cf. section 2.1.4). These approaches vary with respect to their course of action, for example with respect to the number of phases used during design pattern detection [De Lucia et al., 2007]. The objectives of the detection are also manyfold, one example is the extraction of patterns that relate to the UML [Niere et al., 2001]. These approaches do not fulfill the same purpose as embedded models since they are based on design patterns, which have no clearly defined syntax and semantics, and thus are based on heuristics to detect design information.

Similar approaches exist that do not target design patterns, but semantics of models. This applies to the Query-driven State Merging (QSM) algorithm for software behavior model induction [Dupont et al., 2008] which is used for Grammar Inference, for example for the domain of state machines [Walkinshaw et al., 2007]. The approach of example-driven reconstruction of software models [Nierstrasz et al., 2007] is intended for situations where little information about the properties of model elements to be searched is available, but similarities to existing program code are to be found. A different approach is taken by DiscoTect [Yan et al., 2004] which aims at recovering architectural information. This is not extracted from source code, but from compiled Java code at run time, which is for this purpose accessed through the Java Plat-

form Debugging Architecture [Sun Microsystems, Inc., a]. These approaches are certainly helpful for the given purpose, which is identification of semantics in program code unknown so far. Hence, they are based on heuristics. Their application has thus other focal points than this thesis which aims for an approach that allows to supplement program code with well-defined specifications that can be reconstructed unambiguously instead of relying on heuristics.

2.2.3 Considering Program Code as a Model

Since general-purpose programming languages are used to express algorithms in potentially large and very different programs, approaches exist that use their semantics with formal models, but at the level of the programming language. Thus, the program code itself is considered a model for validation and verification of related programs. Various techniques embed semantic information and modeling constraints into object-oriented source code [Beckert et al., 2007]. Examples are the Java Modeling Language (JML) [Leavens et al., 1999] offering an extensive syntax for specification annotations or the approach to use Smalltalk with its introspection capabilities as a meta language [Ducasse and Gîrba, 2006]. In contrast to such universal approaches we do not aim to present a notation for the specification of all possible systems, but only designated parts of it. Domain-specific models can thus be examined more thoroughly and with respect to a formally well-founded background.

Model Checking for such arbitrary source code is also possible. A tool for this is Java PathFinder [Visser et al., 2003] which instruments compiled Java byte code, produces inputs in ranges defined by the user beforehand, and validates constraints inserted in the program code. Similar verification is possible with external model checkers like Spin [Holzmann, 2003] when appropriate models are related to Java code [Holzmann et al., 2008]. Model checking techniques can also be used to improve validation without explicit specifications [Holzmann and Smith, 2001] and at compile time [Volanschi, 2006]. Explicit model specifications for program code are used by the approach of Bandera [Corbett et al., 2000] which extracts finite-state models from Java code. The extracted models represent single statements and fragments of Java programs and can be verified like other state machines [Peled et al., 2001]. In contrast to the goals of this thesis, these approaches work at the low-level semantics of the programming language. While this allows for verification of detailed algorithms, it does not contribute to the maintenance of different abstraction levels in the program code.

2.3 Chapter Summary

The evaluation of related work presented here was based on the objectives defined in section 1.3 and evaluation criteria defined at the beginning of this chapter. For the different classes of approaches we described for which reasons they do not fulfill the objectives. However, several approaches with similar objectives exist, which are depicted in relation to our approach in figure 2.3. This overview shows that these approaches cannot be considered competing approaches. The criteria taken for this visualization are the abstraction level and

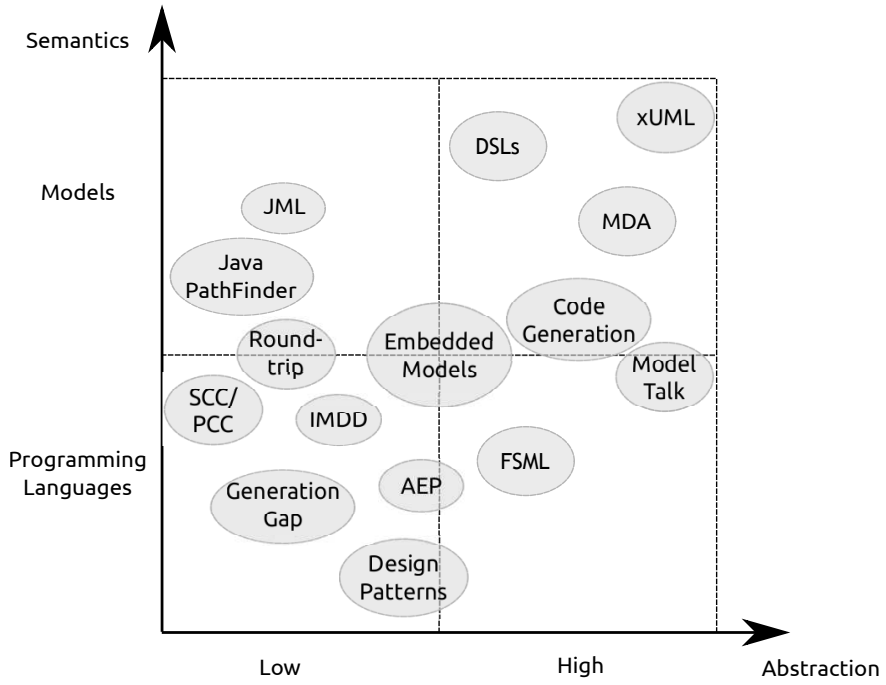


Figure 2.3: Overview of related work with potentially competing approaches.

the semantics being considered, that of high-level models or that of programming languages:

The *MDA* and similar approaches, *DSLs* and *xUML* as well as other executable model approaches, work at high levels of abstraction with model semantics, but do not connect seamlessly to arbitrary program code, thus leaving a semantic gap. *Code generation* and frameworks connecting models to program code like *ModelTalk* rely on different notations with different semantics. *Framework-specific modeling languages (FSML)* are not related to high-level models. *Design patterns* are related to abstract problem descriptions, but also not to high-level models. The *Generation Gap* pattern considers model semantics implicitly, but only so far as program code is designed to provide place holders for generated program code. *Specification-carrying code (SCC)* and *proof-carrying code (PCC)* are related to models, but only in so far as the correctness of the program code is concerned, similar to *introspective model-driven development (IMDD)*, which uses model descriptions only to describe the program code. *Round-trip engineering* considers program code as well as model semantics, but at a low level of abstraction in order to extract model semantics based on heuristics. Finally, approaches like *Java PathFinder* and the *JML* comprise the program code itself as the model, thus allowing to verify it, but only with respect to statements and semantics of the programming language.

In summary, the embedded models approach is different because it considers the semantics of formal models and a programming language at the same time. Thus, the same program code can be considered at higher and lower levels of abstraction on demand. This allows to include formal models as well as arbitrary program code in a single-source development approach. By this

means we achieve consistency between model semantics and program code at development and run time. In the following chapter we will introduce the approach in detail and afterwards describe the implementation for two different modeling domains fulfilling the objectives described above.

Chapter 3

The Embedded Models Approach

The consideration of related work shows that none of the existing approaches fulfills the requirements defined in section 1.3. Thus there is still need for an approach that bridges the gap between higher and lower levels of abstraction as well as between the semantics of models and of programming languages. In this chapter we will introduce the embedded models approach that embeds model specifications in arbitrary program code by defining appropriate program code patterns, and makes them executable by means of reflection. Keeping different abstraction levels in the program code allows to consider the code with several views in different stages of the development process. In this chapter the definition of embedded models is given and maintaining different abstraction levels in the program code is explained in section 3.1. The resulting development process that comprises different views on the program code that can be provided by appropriate tools is introduced in 3.2. Afterwards the architecture of software systems containing embedded models is considered in section 3.3 before we summarize the chapter in section 3.4.

3.1 Definition of Embedded Models

To create an embedded model, the specifications of a formal model are mapped to a program code pattern in a general-purpose programming language. If this mapping is unambiguous and bidirectional, the program code is interpretable at different levels of abstraction. The specification of the program code pattern depends on the semantics and expressiveness of the chosen programming language. We will now give the basic definition of embedded models and then describe the elements of an embedded model in detail.

3.1.1 Definition

For the definitions to give, we consider a model to be "[...] an abstraction over some (part of a) software product (e.g., requirements specification, design, code, test, call-flow graph). There is a variety of kinds of models [...]"

[Hailpern and Tarr, 2006]. The kinds of models are denoted *model classes* in the following. E.g., UML activity diagrams are a model class.

Above we stated the goal to represent such model specifications in program code patterns. Actual program code following the pattern is then a notation for a model. "Notation" is in this context defined as follows:

Definition 1 $N(A_M, L)$ is a *notation* for a model class M with

- A_M the abstract syntax of M,
- L a language with its elements, attributes of elements, and rules to combine elements,
- the notation $N(A_M, L)$ being the elements, attributes, and rules of L so that $\forall m \in M : \exists n_m \in N(A_M, L)$, $a_m \in A_M$ is the concrete syntax for a model instance $m \in M$, n_m is an instance of $N(A_M, L)$ that represents the elements of a_m. □

In embedded models, the connection between models and code is not only constituted by a notation, but also by appropriate execution semantics in the case of behavioral models:

Definition 2 Given A_M the abstract syntax of a model class M. $\alpha(a_m) = \langle \Lambda_m, \rightarrow \rangle, a_m \in A_M$ is a transition system representing the *execution semantics* of a model $m \in M$ with

- a_m the concrete syntax for a model instance m,
- $\Lambda_m = \{\alpha_0 \ldots \alpha_n\}$ a finite set of possible actions defined by the model m,
- $\rightarrow \subseteq \Lambda_m \times \Lambda_m$ the valid transitions between actions of Λ_m derived from the interpretation of a_m. □

This leads to the following definition for embedded models:

Definition 3 $E_{M,G} = \langle A_M, P \rangle$ is a *class of embedded models* with

1. A_M the abstract syntax of a model class M,
2. G a general-purpose object-oriented programming language (called "host language" in the following),
3. $P \equiv N(A_M, G)$ a program code pattern built from elements of G that is a notation for M so that a bidirectional mapping between A_M and P is given,
4. execution of $E_{M,G}$ being possible so that $\forall m \in M : \exists \alpha(p), p \in P$, p represents the elements of a_m, $\alpha(p)$ is bisimilar to $\alpha(a_m)$. □

An embedded model thus establishes a connection between different levels of abstraction, that of the formal model and that of a host language. These different abstraction levels are maintained in program code written in the host language G. By this means each embedded model is specific to G as well as to model specifications of M. In order to create embedded models according to definition 3, each class of embedded models must comprise the following five elements as illustrated in figure 3.1:

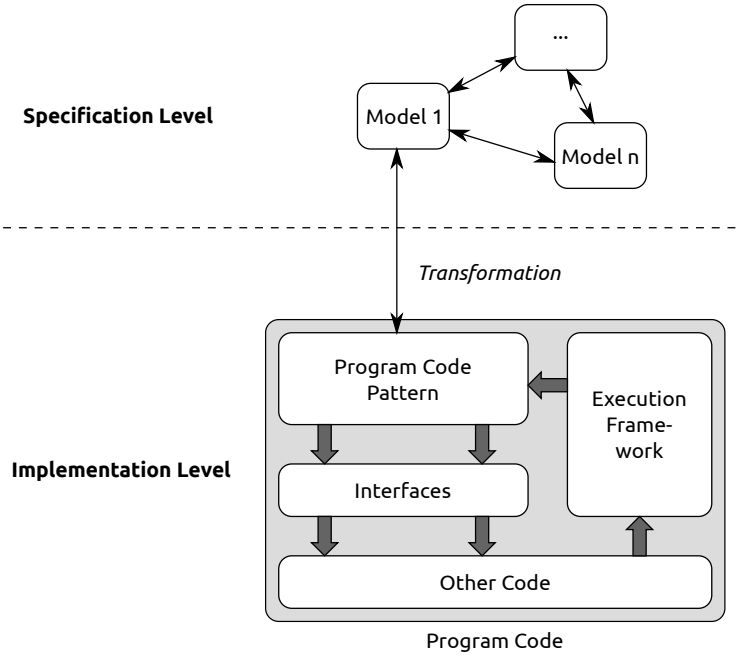


Figure 3.1: Different abstraction levels maintained in the program code. Code following the pattern definition is related to model specifications by transformations.

1. The well-defined semantics of the model class M.
2. The program code pattern P whose fragments are a notation for M and a valid program in the host language G at the same time.
3. Execution semantics that allow to interpret and invoke the code fragments at run time so that the resulting sequence of events and data flow matches the execution semantics of M.
4. Interface definitions between the pattern code and arbitrary other program code it is embedded in. These interfaces define which data can be exchanged and also bridge differences between abstraction levels.
5. A set of transformations that define how differences between abstraction levels can be bridged unambiguously. This allows for an automated provision of different views on the program code and the related model, especially with appropriate tools.

The detailed specification of these elements will now be given in sections 3.1.2 to 3.1.6.

3.1.2 Model Specifications

In order to find a mapping between model elements and design patterns, common characterizations of model structures must be determined.

This is for example done by Hailperrn & Tarr who define models to be "an annotated graph over a set of model nodes, a set of model edges, an alphabet of labels, and a function annotating nodes and edges", where "the annotation function maps either nodes or edges into labels" [Hailpern and Tarr, 2006]. Such static structures are already sufficient for example for data models like entity-relationship diagrams [Chen, 1976]. However, more complex behavioral models require additional facilities to describe dynamic aspects of software systems, which will thus be considered explicitly here. These can be expressed by connecting static structures with logical formulas. This applies for example to expressions in state machines or process models. In addition, labels are not that much of interest to connect elements inside the model, but instead to refer to elements outside the model. In general, connections between these static structures and formulas are realized as relations or functions.

Since formal models are usually domain-specific and not always backed by a common meta model, no unified definition of the elements of such a model can be given. But, we can categorize the elements to expect in such models. For the purpose of embedded models, the abstract syntax of a model specification is defined to be based on the following meta model:

Definition 4 $A_{Meta} = \langle S, \Sigma, L, F \rangle$ is the meta model for the abstract syntax of model specifications with:

- S a set of static elements of a formal model,
- Σ a logic for expressions in a formal model,
- L a set of labels that denote model elements or entities outside a model,
- F a set of functions that arrange and combine model elements. □

Instances of this meta model may have several instances of the members of the tuple, e.g. one set of entities and one set of relationships which are both sets of static structures. This meta model is merely a categorization, but will help to define systematically (1) the host language elements used for embedded models in object-oriented programming languages and (2) implementations for certain modeling domains.

3.1.3 Program Code Patterns

Based on this abstract categorization, developing program code patterns to embed model specifications into program code means to describe a subset of all possible program code structures in the host language that can be mapped to the abstract syntax of the model.

3.1.3.1 Host Language Requirements

Whether a program code pattern for embedded models can be defined depends on the expressiveness of the chosen host language. Considering related work, embedded models enhance two existing concepts: Internal DSLs, which provide a single-source approach for high-level specifications in the program

code, and AEP, which enables meta data in the program code to maintain different abstraction levels (cf. section 2.2.1). Therefore, a host language appropriate to embed model specifications with our approach must at least fulfill the following requirements:

- It must be object-oriented and thus provide the mechanisms to specify static structures for objects, their relationships, and functionality they can execute.
- It must be typed, i.e. the objects must be specified as classes that are contained in a type hierarchy.
- It must be capable of handling meta-information for its structural elements, e.g. for classes and methods.
- It must support structural reflection that allows to access information about structural elements at run time. This concerns classes, methods, and also the meta data attached to them.

3.1.3.2 Pattern Specification

Considering these features of host languages, we can use the related static elements to define program code patterns that are appropriate to represent the abstract syntax of formal specifications.

Definition 5 $P_{Meta} = \langle \mathcal{T}, \mathcal{K}, \mathcal{M}, \mathcal{MP}, \mathcal{A}, \mathcal{AP}, \mathcal{S}, \mathcal{R} \rangle$ is the meta model for program code pattern specifications with the following sets of static program code elements and relations between them:

- $\mathcal{T}$ a set of types in the programming language,
- $\mathcal{K}$ a set of packages in the programming language,
- $\mathcal{M}$ a set of methods in types,
- $\mathcal{MP}$ a set of method parameters,
- $\mathcal{A}$ a set of meta data annotations in the programming language,
- $\mathcal{AP}$ a set of annotation parameters,
- $\mathcal{S}$ a set of statements in expressions in the programming language,
- $\mathcal{R} = \{\xrightarrow{isOf}, \xrightarrow{contains}, \xrightarrow{returns}, \xrightarrow{has}, \xrightarrow{refers}, \xrightarrow{invokes}, \xrightarrow{constitutes}\}$ relations between the static elements. □

The single elements will now be explained in detail. Examples will be given for a small code fragment shown in Java in listing 3.1.

Named Elements $Named = \mathcal{T} \cup \mathcal{K} \cup \mathcal{M} \cup \mathcal{MP} \cup \mathcal{A} \cup \mathcal{AP}$ is the set of elements that have a name, which is assigned to them with the function *name* so that $\forall n \in Named : name(n) \neq \epsilon$. They are thus appropriate to represent labels.

```
1  package sample;
2
3  public class SampleClass extends SuperClass
4  {
5    @Reference(AnotherClass.class)
6    public boolean someMethod(SomeClass s)
7    {
8      return s.getValue();
9    }
10 }
```

Listing 3.1: Sample code using the elements of P_{Meta}.

Types $\mathcal{T}$ is a set of types in the programming language that follow the principle of object orientation and are thus defined statically and explicitly. Each type has a unique name. Types have an inheritance hierarchy so that two types can be connected with a relation called *isOf*: $t_1 \xrightarrow{isOf} t_2, t_1 \in \mathcal{T}, t_2 \in \mathcal{T}, t_1 \neq t_2$ thus means that the type t_2 is a sub type of the type t_1. Since inheritance is transitive, it is true that $\forall t_1 \in \mathcal{T}, t_2 \in \mathcal{T}, t_3 \in \mathcal{T} : t_1 \xrightarrow{isOf} t_2 \wedge t_2 \xrightarrow{isOf} t_3 \Rightarrow t_1 \xrightarrow{isOf} t_3$. If no type is specified or all types are of interest, the placeholder *any* applies.

The type in listing 3.1 (line 3) is by this means described as follows: $\exists t_{Sample} \in \mathcal{T}$, $name(t_{Sample}) =$ `SampleClass`, $\exists t_{Super} \in \mathcal{T}$, $name(t_{Super}) =$ `SuperClass`, $t_{Sample} \xrightarrow{isOf} t_{Super}$.

Packages $\mathcal{K}$ is a set of packages that contain types so that $\forall t \in \mathcal{T} : \exists k \in \mathcal{K}, k \xrightarrow{contains} t$. Packages have a hierarchy so that two packages can be connected with the relation *contains*: $k_1 \xrightarrow{contains} k_2, k_1 \in \mathcal{K}, k_2 \in \mathcal{K}, k_1 \neq k_2$.

For the code in listing 3.1 with the package declaration in line 1, the following applies: $\exists k_{sample} \in \mathcal{K}$, $name(k_{sample}) =$ `sample` , $k_{sample} \xrightarrow{contains} t_{Sample}$.

Methods $\mathcal{M}$ is a set of methods that each belong to one type. Types are thus containers having a relationship called *contains* to methods, so that $\forall m \in M : \exists t \in \mathcal{T}, t \xrightarrow{contains} m$.

Each method takes a possibly empty set of parameters which is expressed with the relation *has*, so that $\forall m \in \mathcal{M} : \exists \mathcal{MP}_m \subseteq \mathcal{MP}, \forall p \in \mathcal{MP}_m : m \xrightarrow{has} p$. Each parameter is of a certain type and thus connected to this type with the relationship *isOf*, so that $\forall p \in \mathcal{MP} : \exists t \in \mathcal{T}, p \xrightarrow{isOf} t$. Parameters can not only represent a single entity, but also sets of entities. In this case, the number of entities is specified with $num(p)$. However, the default case is that $num(p) = 1$. Each parameter belongs to exactly one method so that $\mathcal{MP}_{m_1} \cap \mathcal{MP}_{m_2} = \emptyset, m_1 \in \mathcal{M}, m_2 \in \mathcal{M}, m_1 \neq m_2$.

Methods can optionally return an instance of a certain type which is defined by the relationship *returns*, so that in this case $\exists t \in \mathcal{T} : m \xrightarrow{returns} t, m \in \mathcal{M}$.

Methods can contain expressions in the propositional logic $\Sigma_{Propositional}$. They consist of single statements so that $\forall m \in \mathcal{M} : \exists \mathcal{S}_m \subseteq \mathcal{S}, \forall s \in \mathcal{S}_m : m \xrightarrow{contains} s$. Statements may contain method invocations so that $s \xrightarrow{invokes} m, s \in \mathcal{S}, m \in \mathcal{M}$. Statements also constitute parameters in method invocations

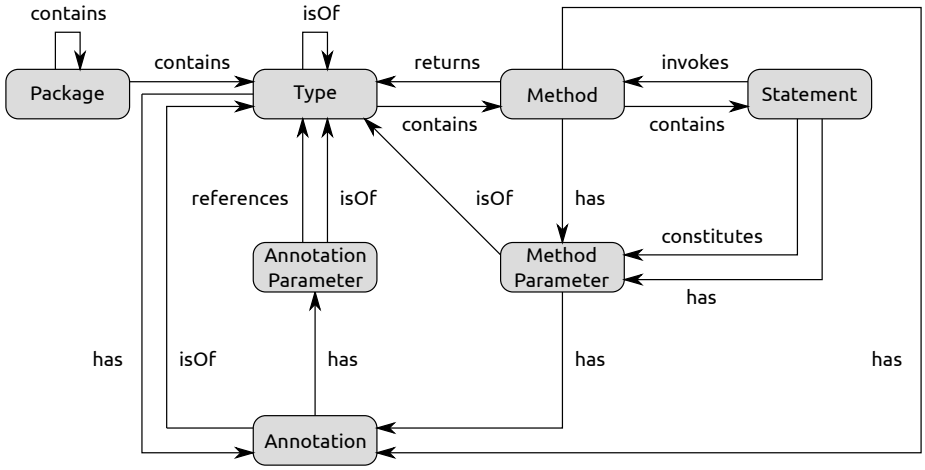


Figure 3.2: The basic elements and their relations used to assemble the program code patterns.

so that $s \xrightarrow{constitutes} p, s \in \mathcal{S}, p \in \mathcal{MP}$. In addition, statements may use parameters of the method they are contained in so that $s \xrightarrow{has} p, s \in \mathcal{S}, p \in \mathcal{MP}$.

The method signature in listing 3.1 in line 6 is described as follows: $\exists m_{someMethod} \in \mathcal{M},\ name(m_{someMethod}) = \texttt{someMethod},\ t_{Sample} \xrightarrow{contains} m_{someMethod},\ m_{someMethod} \xrightarrow{returns} t_{Boolean},\ \exists p_s \in \mathcal{MP},\ name(p_s) = \texttt{s},\ p_s \xrightarrow{isOf} t_{SomeClass} \in \mathcal{T},\ m_{someMethod} \xrightarrow{has} p_s$. For the statement in line 8, the following applies: $\exists s \in \mathcal{S}, \exists m_{getValue} \in \mathcal{M}, s \xrightarrow{invokes} m_{getValue}, m_{someMethod} \xrightarrow{contains} s$.

Meta Data Annotations Meta data annotations can be attached to classes, methods, and method parameters. This is also expressed with the relation *has*. Thus it is true that $\forall e \in (\mathcal{T} \cup \mathcal{M} \cup \mathcal{MP}) : \exists \mathcal{A}_e \subseteq \mathcal{A}, \forall a \in \mathcal{A}_e : e \xrightarrow{has} a$, however, it is possible that $\mathcal{A}_e = \emptyset$.

Each meta data annotation has a name and a (possibly empty) set of parameters: $\forall a \in \mathcal{A} : \exists \mathcal{AP}_a \subseteq \mathcal{AP}, \forall p \in \mathcal{AP}_a : a \xrightarrow{has} p$. The values can be literals in the host language or references to types so that $p \xrightarrow{references} t, p \in \mathcal{AP}, t \in \mathcal{T}$.

Listing 3.1 contains a meta data annotation in line 5 that can be described as follows: $\exists\, a_{Reference} \in \mathcal{A}, name(a_{Reference}) = \texttt{Reference}, \exists\, p \in \mathcal{AP}, p \xrightarrow{references} t_{AnotherClass} \in \mathcal{T}, a_{Reference} \xrightarrow{has} p, m_{SomeMethod} \xrightarrow{has} a_{Reference}$.

Summary By means of these elements, the structures from A_{Meta} can be represented in P_{Meta}: Static elements from S with static programming language elements in $\mathcal{T} \cup \mathcal{K} \cup \mathcal{M} \cup \mathcal{MP} \cup \mathcal{A} \cup \mathcal{AP} \cup \mathcal{S}$; the logic Σ with $\Sigma_{Propositional}$; labels in L with the names of elements in $Named$; and functions from F with relations in $\mathcal{R}$. An overview of the static language elements and their relationships is given in figure 3.2. They are the framework for any implementation of embedded models in object-oriented programming languages. However,

the detailed implementation depends on semantics and expressiveness of such programming languages. Examples will be given for two modeling domains in chapter 4.

3.1.3.3 Implementation Languages

A host language that is appropriate to express embedded models with program code patterns as defined above is Java in version 5 or higher [Gosling et al., 2005]. It provides classes, interfaces, methods, and expressions in propositional logic as described in section 3.1.3 and thus can represent the elements in $\mathcal{T}$, $\mathcal{M}$, and $\mathcal{MP}$. When methods are used to encapsulate classes, naming conventions apply following the *JavaBeans* specification [Englander, 1997]: Methods that are used to retrieve a hidden class attribute are named with the prefix `get` (in the following called "get methods"); methods that set a hidden class attribute are named with the prefix `set` (in the following called "set methods"). Metadata annotations ($\mathcal{A}$, $\mathcal{AP}$) are mapped to Java annotations [Sun Microsystems, Inc., 2004]. Their parameters can be literal values (including strings), nested annotations, enumerations, and references to Java class definitions (extending `java.lang.Class`). The source code is compiled into Java bytecode [Lindholm and Yellin, 1999] that is executed by Java Virtual Machines (JVMs) at run time. In the bytecode, the static structures of classes, methods, parameters, and annotations are available. Differences exist at the level of method bodies: The Java expressions from the source code are compiled to a reduced instruction set that is processed sequentially, for example with the replacement of `for` and `while` loops by `goto` statements. While the execution semantics of both representations are the same, less structural information is available at run time. Depending on this, Java's reflection mechanisms allow to access classes, methods, parameters, and annotations, but not method bodies.

Virtually the same capabilities and limitations apply to the languages running on top of the Common Language Infrastructure (CLI) [ECMA International, 2006b] provided by the .NET platform, for example C# [ECMA International, 2006a]. For information hiding the concept of methods is supplemented with so-called *properties* that publish class attributes, so that no naming conventions are needed in the source code. In the compiled bytecode, however, the properties are realized as methods with naming conventions similar to JavaBeans. The meta data annotations are called *attributes* and take the same parameter types, i.e. literal values including strings, nested attributes, enumerations, and references to type definitions (extending `System.Type`).

In summary, general-purpose programming languages exist that fulfill the requirements for notations of embedded models as sketched in figure 3.2.

3.1.3.4 Pattern Instantiation

When instances of a program code pattern are created they contain information about the instance of one formal model. At the same time, the program code is expected to be accessible from outside, since at least the execution framework must use it for execution. For this reason, instantiation of embedded models must consider two aspects of program code patterns:

First, program code patterns follow certain rules, some of which are given by the definition of program code fragments. These provide a frame for implementations and make the program code following the pattern accessible from outside, which is required for execution by means of a framework using reflection. Such rules may imply that certain types or meta data annotations may be used that can be identified accordingly. These are *pre-defined*, i.e. they are specified and implemented when an embedded model definition is being created. Afterwards they are shared among implementations of this embedded model and are used by the appropriate tools. Types are identifiable by fully-qualified names in host languages. They can define methods to be implemented when a pattern instance is created. In addition, interfaces or abstract classes can act as so-called "marker interfaces" [Newkirk and Vorontsov, 2002] that do not require extending types to implement methods, but exist only to mark them with their fully-qualified name. Meta data annotations have the purpose of being accessible for interpretation, so they are appropriate to decorate other static elements of the program code. In Java and C#, meta data annotations can be applied to classes, methods, and method parameters. The program code being connected to any pre-defined program code fragments is by such means accessible at development time and run time and can serve as entry points to the embedded model instance. This is the foundation not only for execution at run time, but for any tool that accesses the program code pattern instances to consider them at different abstraction levels.

Second, inside the frame of the pre-defined program code, implementations of embedded models consist of program code that is created *per instance*. This program code is itself not interpretable with respect to an embedded model, but gains additional semantics within the context information of the pre-defined program code. For this purpose it must follow the rules that make it interpretable at different abstraction levels within this context.

An example for pre-defined and per-instance code fragments can be seen in listing 3.1: We can assume that the annotation `Reference` is pre-defined and allows to identify methods of interest. In contrast, its parameter referring to another class describes the current instance of the embedded model. Complete examples for program code pattern instances will be shown in chapter 4.

3.1.4 Execution Semantics

As defined above, the program code specified by embedded models carries multiple abstraction levels at the same time. Sections 3.1.2 and 3.1.3 introduced the possible structures for model specifications and representations in the program code that are to be mapped to each other. This already allows to represent models in the program code that are based on static structures only. However, when models specify behavior also, this behavior must be realized at run time when the programs containing embedded model instances are executed.

For this reason each embedded model specifying behavior is accompanied by an execution framework that accesses, interprets, and invokes the related program code fragments. For this purpose, appropriate execution semantics for the formal model must be defined. The static structures of the model are used to determine the behavior of the application at run time and denote single actions which can be executed in the host programming language. The execu-

tion semantics thus denote *sequences of actions* that correspond to the semantics of the formal model.

The execution semantics of the embedded model are based on the pre-defined program code fragments of the pattern since they can provide appropriate entry points for the interpretation of the program code. They also indicate the location of methods containing behavioral logic in expressions that are defined per instance. The execution framework therefore accesses program code in the context of the pre-defined code and realizes behavioral execution with the invocation of appropriate methods. We thus distinguish two types of program code: Some of the code following the patterns is *interpreted*, i.e. the static structures are read by the execution framework to make decisions. This entails the use of pre-defined program code fragments to enable interpretation. Some of the code is *invoked*, i.e. it contains statements that are executed. This can either apply to statements which are interpretable at higher abstraction levels, too; or it can apply to interfaces which abstract from program code that does not belong to the model.

It must be considered that execution will not rely on the source code, but on compiled bytecode. Therefore the information being available is more coarse-grained than in the source code as described in section 3.1.3. The embedded model must thus be defined such that single actions in the execution semantics are accessible at run time. This entails that the parts of the pattern to be interpreted are completely available at run time, too. The program code that will be invoked is not required to be represented in the same detail as in the source code, since invocation of method considers the method bodies black boxes.

3.1.5 Interface Definitions

Program code specified by embedded models is part of arbitrary other program code. This arbitrary other program code can be based on other embedded models, design patterns, or not on any abstract specification at all. This co-existence of fragments based on different specifications in the same set of program code is the foundation for the objective of embedded models to reduce the number of notations used to describe the software to develop.

It also entails that program code of embedded models is not self-contained, but must have well-defined interfaces between the program code pattern representing the model syntax and the arbitrary program code to allow for data exchange and application logic invocation. For the definition of interfaces, the semantics of the formal model must be considered. In section 3.1.2, labels were introduced which are part of the model specification, but refer to the world outside the actual model. Such labels would be represented by program code elements from *Named*. By this means, parts of the program code pattern serve as *interfaces* to arbitrary other program code.

3.1.5.1 Interface Semantics

These interfaces play an important role in the maintenance of different abstraction levels in the program code: When arbitrary program code is connected to the embedded model with labels, the interfaces realize the abstraction that allows to interpret parts of the program code with respect to the abstract specifications of a model. Interfaces can be known to the execution framework

so that it can control communication between the embedded model code and other program code if reasonable. Two properties of interfaces can be distinguished:

First, interfaces can be *data-oriented*, meaning that they are used to exchange data with arbitrary other program code. Such data exchange must be defined with respect to the model semantics since incoming and outgoing data may be defined explicitly in the model or implicitly assumed by the execution framework controlling the communication. Abstraction to other program code can be realized in data-oriented interfaces when data is aggregated. By this means complex data structures can be simplified if necessary to be handled by an abstract model.

Second, interfaces can be *action-oriented*. In this case they provide entry points from an embedded model into other program code that is to be invoked. By this means execution of arbitrary application logic can be initiated from the embedded model, even if its specification is not covered by the related formal model (or any model at all). For such actions the model specifications can define contracts (cf. section 2.1.2) in an embedded model: *Pre-conditions* restrict the circumstances when actions are invoked; *post-conditions* demand that invoked actions leave the overall application in a certain state with respect to the model specification. Both can be necessary to realize an abstraction when interaction with arbitrary program code is desirable.

The distinction between data-oriented and action-oriented interfaces is not exclusive since interfaces or parts of them can have one or both properties.

3.1.5.2 Implications for Other Program Code

Embedded models do not consider the content and semantics of application logic connected to them. By this means their flexibility will be maximized since no limitations for their application are given, except that the interfaces must be served by the application's program code. However, it is important to note that embedded models assume that such arbitrary program code is compliant to the semantics of the model as defined by the interfaces. While they cannot force any compliance, implications regarding the nature of the arbitrary program code are possible.

These implications are based on the fact that program code connected to interfaces of embedded models can be considered as slices [Weiser, 1981] with the interface methods being the entry points. Depending on the requirements of specific embedded model instances, assertions about single slices can be made. While the execution of certain actions itself is not of interest, changes to the state space of the application can influence the embedded model and are thus of interest. Based on the interface semantics introduced above, the the following assertions are possible:

- *Expected changes*: Post-conditions of action-oriented interfaces are assertions which changes to the state space of the application a certain slice has to perform in order to conform to the model specifications.
- *Reproducible reading*: When embedded models access data from the state space of the surrounding application, this may be expected to be reproducible. A related slice may thus be required not to change the state space, but only to read variables.

- *Exclusive writing*: An embedded model may be responsible for storing data in the state space of the surrounding application. In certain cases, it can be desirable that such data is not modified by other parts of the application. Any slices without that for outgoing data are then forbidden to modify the related parts of the state space.

The implications are used for verification in two ways: The interfaces and the execution framework must be designed in a way that a validation of exchanged data is possible where necessary at run time. A validation can also occur with appropriate tools considering the program code during the development process as will be explained in section 3.2.2.

3.1.6 Transformations

The aim of embedded models is to consider programs at different levels of abstraction, while only one notation – that of the program code – is necessary. This entails that different views on this program code must exist to facilitate working with the different abstraction levels. These views may consider only parts of the program code, interpret it with different semantic specifications, and utilize external tools. To enable this, transformations between the program code pattern and the views must exist. These must be unambiguous to allow for automatic transformations and tool support, which is necessary for embedded models to be fully beneficial. We consider two different types, external and internal transformations.

External transformations transform model information between the program code pattern and an external notation. This can be used for example to create a model description in the notation of an external verification tool. External transformations have the following properties:

- The model must be transformed completely to the extent that the transformed information is useful.
- A single transformation is unidirectional, although the opposite way may be supported by a separate transformation as well, of course.
- They are performed as one action with a clearly defined beginning and ending.
- When a model is being transformed and a model definition already exists in the target, the model information must be merged. This may, for example, happen when a model definition was extracted from program code, modified in an external tool, and is to be written back to the program code. For this case, collision detection strategies must exist.

In contrast, *internal transformations* do not rely on a separate notation, but provide non-persistent views on the program code. This is for example the case with editors that arrange fragments of the program code pattern directly according to the model specifications. Internal transformations have the following properties:

- Only the information that is currently needed by the target of the transformation must be extracted.

- The transformation can be bidirectional so that model information is exchanged between two participants.
- The transformation can be a process that has single steps over time and thus no clearly defined beginning and ending.
- Ideally, model information is transferred instantly when it is needed, so that both participants always access recent information.

In general, different technologies may be of use to implement transformations. Since transformations will likely occur between completely different representations, one transformation may involve two technologies. So far we identified these concrete technologies:

- *Program code access in IDEs*: Integrated Development Environments (IDEs) like Eclipse [Eclipse] provide programming interfaces to access source code in projects managed by the IDE, for example Eclipse's Java Development Tools (JDT) [The Eclipse Foundation, 2008]. This is of interest at development time for internal and external transformations.
- *Graph Transformation*: Triple graph grammars and graph transformations [Ehrig et al., 2006] are applicable to embedded models because program code structures can be accessed as trees [Striewe et al., 2010b; Striewe, 2008], for example in the Document Object Model (DOM) provided by JDT. When converted to triple graph grammars, they are eligible for every graph transformation. This is applicable at development time, when the program code is completely accessible, for external transformations that create another explicit representation of the model specifications. A possible output format of the transformation is XML which can be used to create input files for different XML-based modeling tools, for example those that rely on XMI.
- *Reflection*: Structural reflection mechanisms of the host language can be used to access the program code pattern to a certain degree at run time. Limitations caused by compilation into bytecode apply as described in section 3.1.3.3. The use of reflection is appropriate at run time since instantiations and the state of objects can be considered. It can be used for both internal and external transformations.
- *Debugging*: At run time, debuggers for the host language can be used, if available, to access additional information. In addition to static structures, debuggers can read variable values as well as expressions in methods and invoke methods at every time and thus access the application's state space. The use of debuggers is of interest at run time for internal and external transformations.
- *Byte Code Access*: For standardized byte code formats like that of Java, tools exist that provide access to structural information. By this means the compiled code can be analyzed to the degree of detail supported by the byte code. With the use of decompilers, it is in principle possible to reconstruct even expressions that were not retained as such by the compiler.

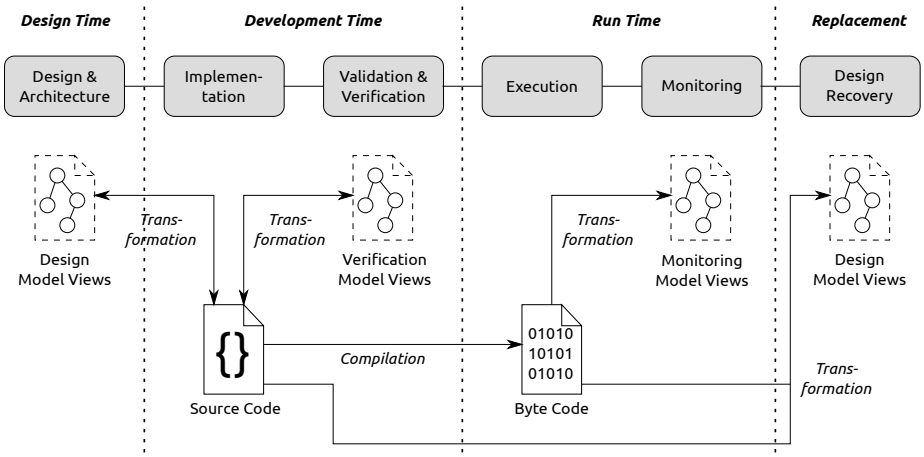


Figure 3.3: The development process that is supported by tools for embedded models. Depending on their purpose, transformations can be unidirectional, e.g. for extraction of information for monitoring, or bidirectional, e.g. for design views that read and write source code.

Two transformations are needed in every case, which we will refer to as *default transformations* in the following: One that considers the source code to extract the complete model elements from it, and one that considers compiled byte code to extract the model elements from it that are still available after compilation. These are needed for the basic representation of the model specifications during development time and run time. However, apart from these, a number of specific views can exist that are each realized by special transformations.

The use of transformations throughout the development process will now be explained in section 3.2 and later discussed with examples in chapter 4.

3.2 Views on Embedded Models

Following the definition of embedded models as given above, program code can carry information at different abstraction levels. Thus the number of notations in use during development can be reduced as shown in figure 3.3: Source code and compiled program code are the main explicit notations, with the compiled code being derived from the source code. During design, verification, execution, monitoring, and design recovery, the models are only specific views on the program code that are extracted from this notation on demand. The program code is thus interpretable at different levels of abstraction: that of the programming language itself and that of the formal models, leading to different views for different purposes. The views that are applicable to different stages of the development process will now be explained; examples will be given in chapter 4.

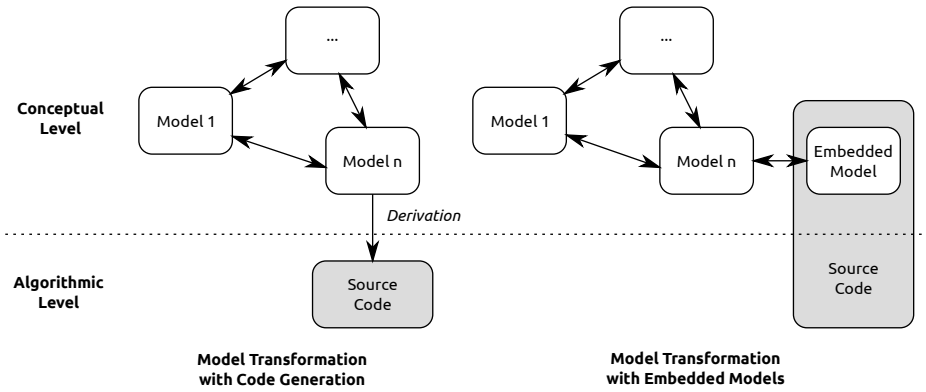


Figure 3.4: The integration of embedded models in model transformations.

3.2.1 Design and Implementation

The most-perceived benefit of MDSD is the ability to design software for certain domains graphically. For many models appropriate visualizations exist. They support the objective of MDSD to create software more efficiently. Design tools can be created for different user groups, for example determined by the technical know-how. One could imagine that a domain model is created by an operating department with a simple tool and connected to existing program code afterwards by developers with a more sophisticated tool. Appropriate design tools can thus guide the design of software and facilitate program comprehension.

When the program code carries design information as well as the actual implementation, the related activities in the development process are not clearly separated. During design, we can expect two use cases for embedded models:

First, models can be defined using dedicated notations, for example in modeling tools. It is usual that such models are refined and/or exchanged between tools, which is realized with appropriate model-to-model (M2M) transformations [Czarnecki et al., 2009]. These are based on the fact that both notations, source and sink, carry the semantics of the models so that the transformation is unambiguous. Program code is usually not involved since it contains only detailed execution logic, so that it is only derived from high-level specifications, as illustrated at the left hand of figure 3.4. However, with embedded models and the resulting expressiveness of program code, the code can participate in model transformations as illustrated at the right hand of figure 3.4: Program code structures can be created as a result of model transformations, and modeling information can be extracted from the code and used in other notations. By this means external transformations (cf. section 3.1.6) can be applied: Modeling information can be stored in the program code as a result of model transformations in a way that it constitutes a part of the program's functionality instead of being only pure meta data.

Second, modeling can be used to create the program code directly. This is useful if the design is not that complex that it requires external notations. In this case design tools can be used that create program code directly from visual representations. The design information is thus only a different view on

the system to develop and by this means an internal transformation between the model specifications and the program code.

At the same time, the implementation of other program code must be considered. With embedded models, the development of model specifications and other program code can happen in parallel. Consequently it is much easier to align the different views than in cases where different notations are used. At the same time, the separation of different concerns must be represented in the program code, for example with modularization concepts.

3.2.2 Verification

When the program code has been designed and implemented with embedded models, the resulting source code contains the modeling information in the program code pattern. This allows for model verification at different abstraction levels. However, model verification relies not only on the existence of a model, but also of a specification the model is verified against. Thus appropriate views must exist for this purpose that provide model specifications and interpret the program code and the embedded specifications accordingly. The modeling information or subsets of interest are then extracted for verification from the program code. Any abstraction or refinement explicitly happens alongside interfaces or pre-defined code fragments. The verification tools can either work on the program code directly with internal transformations, or be separate tools that are comprised into the development by external transformations providing data in the notation of such tools.

Verification is also possible for slices of program code connected to the embedded model (cf. section 3.1.5.2). Views for static program code analysis can determine if the arbitrary program code is valid with respect to the model specifications it is interacting with. The search space can be reduced considerably so that starting points for slices [Tip, 1995] can be provided if the program code pattern defines entry points from the model specifications into arbitrary program code. In addition, different approaches and tools for automated verification are applicable. Static analysis is possible based on rules, e.g. with PMD [Copeland, 2005] proving certain structural properties, or with advanced approaches like the Hoare logic [Hoare, 1969] proving correctness of algorithms. Dynamic analysis and model checking [Holzmann et al., 2008], e.g. with Java PathFinder [Visser et al., 2003], can be used to validate data exchange and state spaces of the program at run time. Depending on the model, static or dynamic analysis are enabled since the model specifications reduce the search space with the definition of slices and provide assertions and constraints that would otherwise have to be inserted manually.

While design and verification are different tasks, it is likely that a smooth transition between the related tools will exist: Even a simple design tool can verify basic properties of the model, and a verification and simulation tool will in most cases visualize the model appropriately. A sophisticated tool thus may cover both stages of the development process since the related information is consistently available in the notation of the source code.

3.2.3 Execution

Since the program code structures defined by embedded models are not directly executable, but instead only descriptions of modeling information, they are processed by execution frameworks at run time as described in section 3.1.4. The execution frameworks read and interpret the structures of the embedded model code and invoke methods if appropriate. The execution is by this means a specific view on the compiled program code that accesses a part of the modeling information to create the required sequences of actions.

3.2.4 Monitoring

The compiled code is also interpretable at run time for tracing and monitoring with respect to the abstract model [Maoz, 2009; Bodden, 2007], even if related documentation is not available. This is of interest for embedded models since they are not isolated, but interact with arbitrary application logic. Thus they may act differently than expected from the results of verification views, since verification will likely focus on certain aspects and not on all possible state spaces that can be introduced with real data at run time. For monitoring, traces in terms of class instantiations, method calls, and changes of variable values can be generated. Two kinds of monitoring are distinguished in the context of embedded models:

Active monitoring is controlled by the execution framework sending information to possible listeners. Due to compilation, the degree of detail is limited as explained in section 3.1.3.3. For example, reflection in Java allows access to structural elements, but not to details of method bodies, whose structures have different semantics in byte code than in source code. The advantage of such approaches is that they can be realized at the level of the framework. However, the overhead introduced by interpretation and emission of information must be considered for production systems.

Passive monitoring does not require modifications to or support by the execution framework. Instead it relies on instrumentation techniques provided by the platform. In Java, for example, the debugging interface [Sun Microsystems, Inc., a] is appropriate to gather information about running software. Since the program code patterns define well-known entry points and markers that identify program code as part of the model, instantiations and invocations of related code structures can be surveyed. With debugging, the monitoring can be applied to all elements of the programming language, including method bodies. However, such instrumentation is likely not to be available in production systems for performance reasons. In the case of debugging, the monitoring requires two running systems: The system that is debugged, and the debugger itself that controls the execution and gathers the data. Passive monitoring is therefore much more appropriate to inspect the running embedded model in detail, but harder to apply to production environments.

3.2.5 Design Recovery

Finally, after years of service a system may need to be replaced. However, most software systems are not simply abandoned. In many cases, data and

application logic defined in them are to be transferred to successional systems. However, experience shows that documentation is often incomplete, out of sync with the actual system, not known for sure to be in sync, or not existent at all. Especially modeling information would in such cases be of help since model specifications are often used to describe essential parts of the system that may be of interest for reengineering or can support the recovery of design in the software.

If the source code of software is still available, the modeling information of embedded models can be accessed directly. As described for execution and monitoring, the specifications of models are retained in executable systems to a certain degree, too. This information is appropriate for reengineering and design recovery since it is no external meta data, but constitutes (a part of) the actual system that is to be replaced. It is desirable to apply information used in the other views for design recovery too. Thus embedded models can support design recovery as long as the basic information is available about the fact that an embedded model is in use.

3.3 Software Architecture

The approach as explained so far is valid for the creation of embedded models for one modeling domain. However, when large systems and a respective architecture are developed, embedded models must be considered in a broader scope. We will in this section outline general properties of architectures embracing (multiple classes and instances of) embedded models while leveraging the characteristics of large, module-based software systems.

3.3.1 Principles

When large software systems are created, a software architecture [Buschmann et al., 2007, ch. 6] has to be developed that defines basic rules facilitating a structured development. In addition, some technological aspects have to be considered:

- Software systems can be modeled with more than one model class.
- Module concepts [Szyperski, 2002, ch. 4] or plug-in mechanisms can be used to assemble software systems not completely at development time, but at deployment time, when they are put into operation.
- Software systems can be dynamically configured for example with process model platforms so that their behavior is defined not until run time and can be adapted on demand.

A larger software system can thus be expected to consist of modules and dependencies between them. Properties of single modules as well as relations between models may be subject to dynamic configuration even not until the system is executed. To consider software systems with characteristics like these, the following has to be taken into account for embedded model definitions:

1. The *scope of model definitions* defines functional criteria for integrating embedded models in larger applications. If multiple modeling classes are used in an application, interaction between different model specifications is of interest.

2. Possible *requirements of program code patterns* with respect to platforms, libraries, or architectural decisions can influence the module dependencies, for example with the introduction of modules that provide predefined elements of the pattern.

3. Characteristics of the *execution semantics* must be obeyed when an embedded model is executed as part of a larger application. This is especially true since the execution frameworks are intended to control the model execution, which can contradict other platforms or frameworks, for example if they are based on inversion of control [Fayad and Schmidt, 1997].

4. An enhanced view on *interface definitions* is needed: On the one hand, the interfaces defined by an embedded model can have the purpose to define interaction between modules. In module-based systems, these interfaces are required (i.e., must be satisfied by other modules) by the embedded model. On the other hand, when embedded models are contained in modules, they must also provide interfaces to be accessible. The latter is especially related to the fact that model execution by means of the execution framework must be initiated, so that provision of such entry points is necessary.

5. Since module-based systems are not always completely assembled at development time, but can also be configured just before they are put into service, the development process and the related tools that are realized by *transformations* must be adapted to comprise the resulting dynamic changes.

So, the points of interest are related to the elements of an embedded model definition, which must be extended accordingly. We will describe the detailed implication for the single elements now in sections 3.3.2 to 3.3.6.

3.3.2 Model Definition Scope

As defined in the approach, embedded models are applicable in cases where programs are written that consist of program code which is (at least partly) written manually. In such cases they have the advantage that different abstraction levels can be maintained in one notation. Thus they are not intended to be used in cases where models are used at an abstract level only. In a larger software architecture as described above, we propose for this reason to classify the fields of application for models into *per-module models* for single modules and *assembly models* for complete systems consisting of multiple modules.

Single modules are likely to consist of software that is developed by writing program code, which may be based on models. These modules can at the same time be related to details of hardware platforms, frameworks, libraries, and server environments, so that the high-level specifications must be supplemented with appropriate detailed program code. Embedded models are in

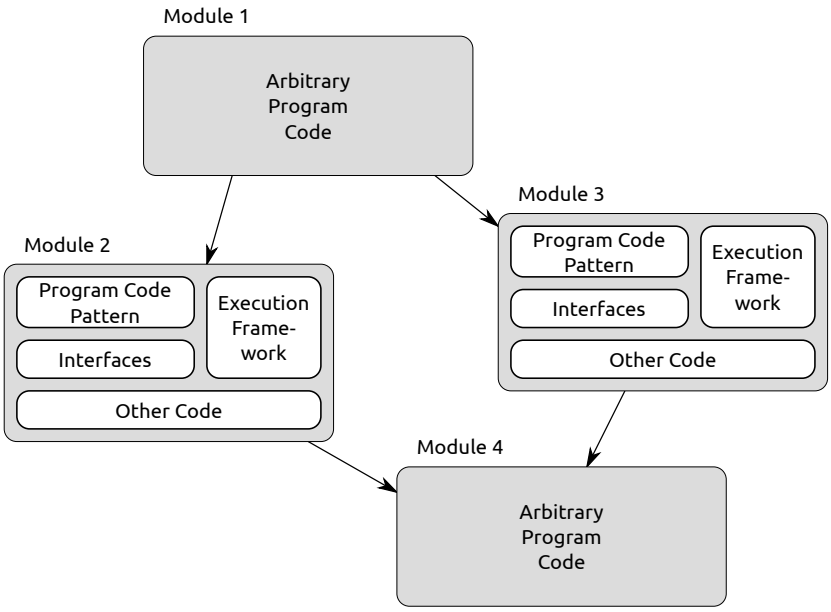


Figure 3.5: Embedded models in modules.

this case applicable. Considered in the context of a larger architecture, modules containing embedded models will be integrated as sketched in figure 3.5: From the outside, all modules appear as black boxes that are self-contained sets of program code and provide interfaces according to the purpose of the module. It is foremost not of interest if the program code is based on embedded models or not. Thus, in general, all modules are appropriate to be engineered with embedded models, since no assumptions regarding their implementations are made.

The use of modules implies that coarse-grained structures of large applications are considered at higher levels of abstraction, with module bodies as black boxes providing well-defined interfaces. By this means assembly and configuration of the system is possible without considering program code and environment details. For this purpose models can be used that work with module interfaces at higher levels of abstraction. In general, such models are defined in service-oriented computing [Bichler and Lin, 2006] with an appropriate formal foundation that allows for design and verification tool support. These models follow rules that are different from the models used to engineer program code for single modules:

- They are *dynamic*, i.e. they are not intended to create fix structures, but to handle frequent changes.
- They are *descriptive* because they abstract from program code semantics completely and focus on the description of interfaces instead. Thus they fulfill the requirement that they can be accessed at run time and changed frequently.
- These interface descriptions may be founded on the semantics of pro-

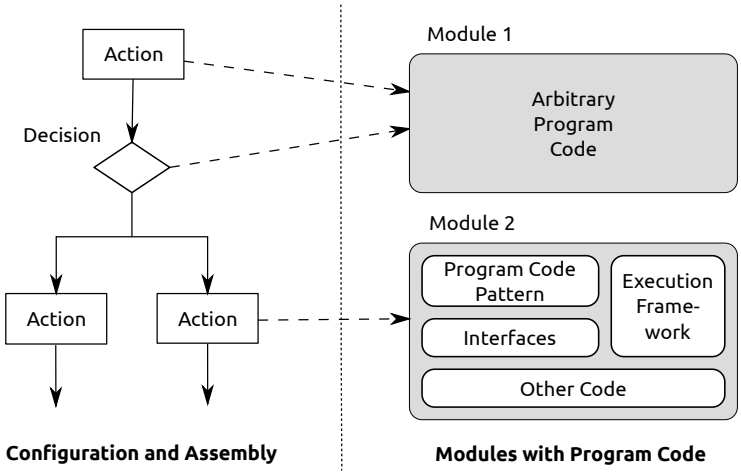


Figure 3.6: Types of models in a system architecture. Inside modules, implementations can be based on embedded models. In contrast, models that are used for dynamic configuration, e.g. the process model at the left hand, only invoke modules and are loosely coupled with the implementations, so that embedded models are not applicable.

gramming languages, but can as well be provided by middleware like CORBA [Szyperski, 2002, ch. 13] and its interface description language, or by services in service-oriented architectures. They must be expressive enough to provide all information of interest at run time.

When these requirements are fulfilled, large systems can be assembled and configured with appropriate models, for example business processes in service-oriented architectures [Brahe, 2007]. In this case such models exist in parallel to embedded models: Implementations inside modules are based on program code so that embedded models can be used here. In contrast, models that compose large systems out of these modules are only indirectly connected to program code when they control, configure, or invoke modules; they are instead appropriate for dynamic configuration, even at run time, so that embedded models are not applicable. An example for this can be seen in figure 3.6: The interaction of the modules and thus the overall behavior of the system is controlled by a process model as seen at the left hand. This model invokes module interfaces to initiate actions or access data for making decisions. The modules themselves can consist of arbitrary program code or be based on embedded models. The models used inside the modules and the models for system configuration are thus completely separated.

3.3.3 Program Code Pattern Requirements

When a system architecture is created that incorporates embedded models, it is important to account for the types of program code that the pattern consists of as described in chapter 3.1.3: Pre-defined and per-instance code.

Since the pre-defined code is intended to be applicable to all embedded model intances for one model class, it must not be included in each module. In-

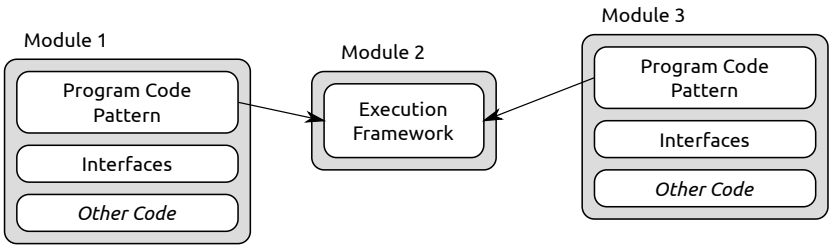


Figure 3.7: Requirements of program code patterns. Pre-defined program code and execution framework can be shared by multiple model instances.

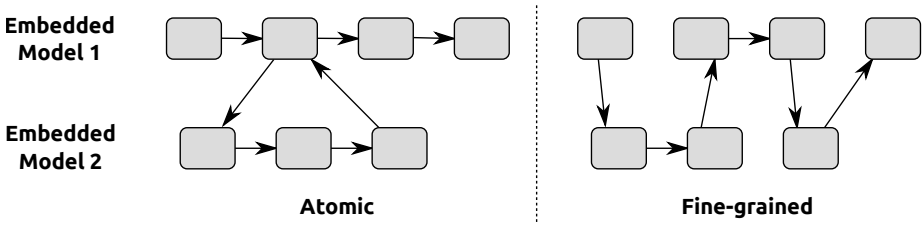


Figure 3.8: Atomicity and granularity of embedded model execution.

stead, it can be provided as a separate module all related embedded model instances depend on. This may introduce additional nodes in the module graph as shown in figure 3.7, where two embedded models of the same modeling class access a shared framework.

3.3.4 Execution Semantics

When a single embedded model is executed, its descriptive structures are interpreted so that a sequence of actions results that matches the model semantics (see section 3.1.4). In a larger context, the same applies to a system architecture as proposed here: The frameworks, run time environments, or models that control the overall system initiate a sequence of actions which may be executed by different modules and/or different embedded models.

The system architecture thus must ensure that all components of the system produce sequences of actions that are appropriate for the purpose of the system. This includes the execution semantics of each embedded model that is involved. These considerations depend on the underlying formal model, so that no general rules can be established. However, in order to categorize the execution semantics of different embedded models, the following criteria are of interest:

- *Atomicity*: The model execution can appear to other modules as one or more steps. If only one step is visible, the model is executed completely and control is returned to the invoking module when the model execution has finished completely. If more than one model is executed at the same time, a clear hierarchy exists. This is depicted at the left hand in figure 3.8.

- *Granularity*: When the model execution consists of more than one step, the model execution can overlap or intertwine with other model executions. In this case it is of interest how detailed models can intertwine. This is depicted at the right hand in figure 3.8.

- *Cycles*: When embedded models are executed, the program code that can be invoked in interfaces by the embedded model can concern other modules, too. This means that cycles can occur if the same module is invoked again, directly or indirectly. This can be desired and happen in a predictable way. However, since large module-based systems are probably not assembled until run time, unintended cycles can occur. Thus, when a system is assembled, the contained embedded models and their interfaces have to be validated to be cycle-free.

3.3.5 Interface Definitions

Embedded model definitions specify interfaces between the pattern program code and arbitrary program code at a technical level. These interfaces are even more important in a system architecture as proposed in this chapter [Szyperski, 2002, ch. 5].

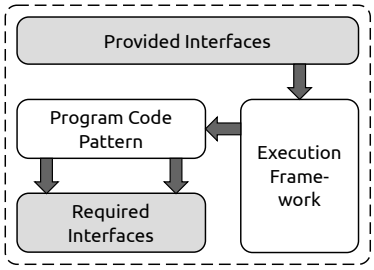


Figure 3.9: The interface types of an embedded model.

To begin with, it is important to note that each embedded model has two types of interfaces as depicted in figure 3.9. The interfaces that have been defined part of an embedded model are *required*, i.e. the embedded model needs to access them in order to fulfill its purpose. But, every embedded model does also have an interface that is visible from outside the model and can start its execution. This interface is *provided* to other program code respectively other modules. While the actual classes that initiate the model execution may technically belong to the execution framework and not to one specific model instance, they are still part of the functionality the embedded model provides to its environment.

Since embedded models can be included in arbitrary program code, they can access application logic contained in a module or program code that accesses other modules and invokes application logic there. If appropriate, the required interfaces can not only be considered abstractions inside the program code, but also abstractions between models. The principle is sketched in figure 3.10: The embedded model contained in the upper module accesses a second module that provides arbitrary program code with the required interfaces. When this is desired, the execution framework for an embedded model can be

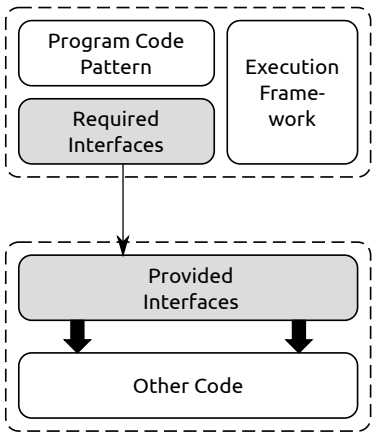


Figure 3.10: Interfaces connecting embedded models.

adapted so that it looks up modules automatically and thus integrates them better in the overall architecture. However, this is optional since the fact that interfaces exist is not a sufficient reason to separate the program code into different modules. In addition, a strength of embedded models is that they can be integrated in arbitrary program code, so that a separation in modules must not be the default case.

3.3.6 Development Process

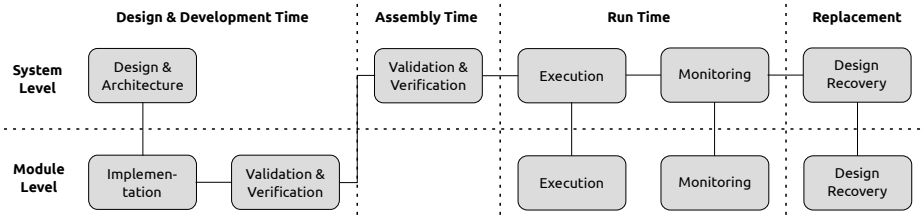


Figure 3.11: The development process for the software architecture with embedded models as proposed here.

The development process that was defined for single embedded models in section 3.2 must be enhanced according to the principles for the architecture as defined in section 3.3.1. The enhanced development process is sketched in figure 3.11. The general difference is that, according to the different model types introduced in section 3.3.2, two levels are considered, that of single models and that of the whole system.

The first step for the system architecture proposed here is the general design at the system level. Afterwards, single modules can be designed and verified separately. When all modules are developed, a new phase in the development process is reached: The assembly of the whole system. As explained in chapters 3.3.2 and 3.3.4, the complete system is assembled and configured after the actual development. Since the combination of modules that constitutes the

system is not known beforehand, the assembled system must be validated and verified again. This step ideally considers the model semantics of the system models and the embedded models. Afterwards, the whole system is executed, which means that embedded models in modules are also executed as part of the overall program flow. For the monitoring of the system, appropriate tools can be developed that incorporate the monitoring capabilities of single embedded models, too. Execution and monitoring therefore happen constantly at both levels of the development process.

3.4 Chapter Summary

This chapter described the general approach of embedded models. At the beginning a definition was given that relates model specifications to program code in a host language. Based on this, five elements were explained that are necessary for each embedded model specification to maintain different abstraction levels in the program code: The precise definition of a formal model is the starting point for the creation of an embedded model; the program code pattern makes the syntax of the formal model available in the host programming language; execution semantics define how behavioral aspects of the model are executed as sequences of actions during run time; interfaces between the program code pattern and arbitrary program code are considered because an embedded model will be part of larger applications and the resulting interactions must be carefully adjusted with the model semantics; transformations can be created that read the model information from the predominant representation in the program code pattern and provide different views on it.

Considering this "stack" of elements that makes different abstraction levels usable in the same program code, the application to different stages of the development process was explained. For design and implementation, program code can be created based on formal model specifications and their visualizations. A verification of this program code is possible with views that provide appropriate specifications. At run time, a view realizes the execution semantics, and monitoring with respect to the model specifications is also possible. Finally, design recovery activities can be applied to program code containing embedded models when a software system is to be abandoned. All these activities rely on the fact that the transformations between the abstraction levels are unambiguous and automated, so that appropriate tools can be created.

Based on this, software architectures were considered that are applicable to large, modularized software systems and embrace embedded models systematically. The main characteristic of the the proposal is that it classifies two types of models in this architecture: Embedded models are used to realize single modules based on program code, whereas system models can be used to assemble and configure the software system at run time. For such architectures, the elements of embedded model specifications are adapted: The scope of the formal model must be used where applicable; pre-defined program code used in embedded models can be systematically included in the module dependency graph to facilitate reuse; execution semantics must be considered to ensure proper execution of different modules, especially if they are based on models that intertwine; interface definitions can be considered from an advanced perspective and also be used to structure the system more fine-grained,

if appropriate; finally, the development process must cover the additional abstraction level considering the whole system. While these principles are rather general, they are a starting point for the creation of such software architectures. However, since the semantics of formal models can be very specific, an actual software architecture will have to be developed individually for each system and consider the embedded models in detail.

We have thus laid the foundation for the creation of embedded models. In the next chapter we describe two implementations for state machines and process models, including tools that enable continuous working at different abstraction levels in program code during the development process.

Chapter 4

Implementation

In chapter 3 the foundations for embedded models have been laid. We especially described the elements an embedded model specification must comprise. In this chapter, we use these general definitions to build concrete embedded models for the domains of state machines and process models.

These modeling domains are of interest for several reasons: The models themselves are applicable to many situations where a software is considered to switch between different states under well-defined circumstances. A model defining such states is appropriate to control larger parts of an application. This applies to embedded models in particular since transitions between states or actions in states can imply that application logic is executed which is not part of the model. For this reason, behavioral models have to cover structural aspects as well as the execution of low-level algorithms. While state machines emphasize the description of the state space, process models focus on the data flow. Both kinds of models are easy to visualize, and tools exist that verify and simulate model instances. In addition, state machines and process models were chosen for this thesis because studies show that behavioral models are used infrequently in MDSD projects in comparison to static models like class diagrams [Dobing and Parsons, 2006]. The reason is that they are not as easy to map to program code structures. We thus decided to consider behavioral models as non-trivial cases to determine the influence of embedded models here.

The structure of this chapter follows the general structure given for embedded model definitions. In sections 4.1 and 4.2, the embedded models for state machines and process models are defined. This definition comprises the five elements as introduced in section 3.1: Model definition, program code pattern, execution semantics, interfaces, and transformations (sections 4.1.1 to 4.1.5 and 4.2.1 to 4.2.5). Afterwards the tools are described that were developed for embedded state machines and embedded process models in section 4.4. Their presentation follows the stages of the development process as introduced in section 3.2: Design & implementation, verification, execution, monitoring, and design recovery. The chapter is then summarized in section 4.5.

4.1 State Machines

With state machines, a domain of behavioral models has been chosen for implementation in this thesis that has been subject to research in software engineering for decades. We will now use the characteristics of state machines to describe the behavior of applications containing appropriate instances of program code patterns.

4.1.1 Model Definition

The first step for the creation of an embedded model is a clear definition of the related high-level model and its semantics that are to be embedded. Based on existing definitions of state machines [Peled et al., 2001] and with respect to the meta structures for models introduced in section 3.1.2, our state machine model is defined as follows:

Definition 6 $A_{StateMachine} = \langle S_{States}, S_{Transitions}, L_{Actions}, \Sigma_{Propositional}, F_{Guards}, F_{Updates}, L_{Channels}, F_{Channels}, F_{Actions}, F_{Initial}, s_0 \rangle$ is the abstract syntax derived from A_{Meta} for a model class of finite state machines $M_{StateMachine}$ with

- S_{States} a finite set of states,
- $S_{Transitions} \subseteq S_{States} \times S_{States}$ a finite set of transitions,
- $L_{Actions}$ a finite set of action labels including at least one element ϵ,
- $F_{Actions}$ a function that assigns sequences of actions to transitions,
- $\Sigma_{Propositional}$ the propositional logic, including a finite set of variables $L_{Variables}$ and their domains,
- F_{Guards} and $F_{Updates}$ functions that assign logical formulas for guards and updates to each transition,
- $L_{Channels}$ a finite set of labels representing channels that enable communication between state machines,
- $F_{Channels}$ a function that assigns channel labels to transitions,
- $F_{Initial}$ a function that gives an initial assignment for all variables defined in the logic,
- $s_0 \in S_{States}$ the initial state. □

4.1.1.1 States

State machine models reduce the state space of real applications, which is in theory infinite, to certain well-defined states representing situations in the execution of a program that are reached under certain conditions. As such, states have a name that allows to distinguish them. With respect to the classification given in A_{Meta} (cf. section 3.1.2), the states are a set S_{States} of static elements. In our model, three types of states can be found:

- Exactly one state $s_0 \in S_{States}$ is the *initial* state denoting the start of the state machine flow.
- Some states are *final* states. They have only incoming transitions. At least one final state exists in each model if the state machine is expected to terminate. This is especially important for embedded state machines since they are expected to return control to the surrounding application.
- States between the initial and final state have incoming and outgoing transitions.

4.1.1.2 Transitions and Actions

The transitions in the set $S_{Transitions}$ are also considered static elements. They are a subset of all possible transitions $S_{States} \times S_{States}$ and represent a switch between states in a program. Thus each transition has a source state and a target state.

Transitions switch between states and thus modify the state space of the application. We define therefore that all activities happen when transitions fire. For this reason, action labels are attached to transitions by a function. Actions are represented by a set of labels $L_{Actions}$. Sequences of actions are assigned to transitions with the function $F_{Actions}$ so that each $\forall t \in S_{Transitions}$: $\exists L_{Actions_t} \subseteq L_{Actions}$. It is also possible that $L_{Actions_t} = \emptyset$.

4.1.1.3 Variables, Guards, and Updates

In this state machine model, a set of variable labels $L_{Variables}$ and their data types is defined as part of the propositional logic $\Sigma_{Propositional}$. When the state machine is started, the function $F_{Initial}$ assigns an initial value to each variable. $\Sigma_{Propositional}$ itself is used to build expressions that access these variables. In the state machine model, expressions and variables are used for two purposes: Guards decide which outgoing transition to choose in the current state. Updates represent the changes to the state space that occured when transitions fired and the related actions were initiated.

Guards are assigned to transitions with the function F_{Guards}. They consist of an expression that is evaluated to a boolean value. The basic expression has three components:

- a left side which is always a variable identifier,
- an operator out of $\{==, != , <, <=, >, >=\}$,
- a right side which is a literal value or another variable identifier.

Basic expressions can be aggregated to complex expressions with union (||) and intersection (&&) operators.

Updates are assigned to transitions with the function $F_{Updates}$. They consist of a set of expressions that assign new values to variables. The left-hand side of each assignment is always a variable. The right-hand side consists of expressions that follow the same rules as guards. However, updates are expected to allow for comparisons with the point in time before the transition fired in order

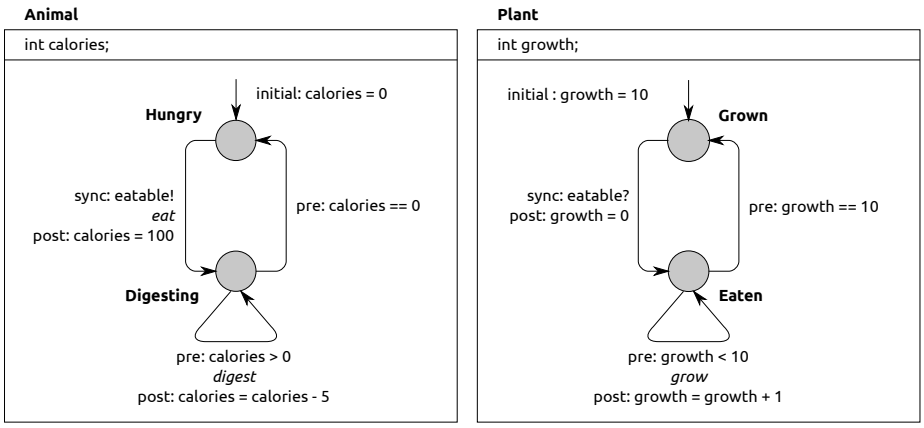


Figure 4.1: An exemplary state machine system with states, transitions, guards, updates, and a channel.

to represent changes. For this reason, variable identifiers in the right side of the assignment will be interpreted as representations of the old variable value.

An example for an update is a transition $s \rightarrow s'$ that increases the value of a variable `v` by 1, so that $v' = v + 1$. The related expression would be written as `v = v + 1`, since the left-hand side and right-hand side allow to distinguish between old and new variable values.

4.1.1.4 Channels

When complex systems are modeled, the models must scale to represent the complexity. State machines support scaling with communicating state machines, which allow to model and execute several aspects of a system separately and synchronize the execution during transitions. When state machines are connected with channels, we refer to the resulting set of state machines as a *state machine system*. For this kind of synchronization, channels are defined as labels in $L_{Channels}$ that denote synchronization points.

Channels can be assigned to transitions with the function $F_{Channels}$ that also denotes if a transition will send or receive on a channel. During the state machine flow, whenever a transition in the current state has a channel label assignment, it can only fire if another state machine provides the counterpart on the same channel at the same time.

4.1.1.5 Example

Figure 4.1 shows an exemplary state machine system with two state machines, *Animal* and *Plant*. Variables are defined for each state machine with data type and name. States and transitions are represented graphically. Other elements of the model are attached to transitions: The keyword *initial* denotes the initial assignment. The keyword *sync* describes synchronization between state machines and thus the use of channels for transitions. The labels printed in italic represent action labels. The keywords *pre* and *post* describe expressions for guards and updates of transitions.

The Animal has two states, *Hungry* and *Digesting*. The initial state is *Hungry*. The state machine uses the integer variable *calories*, which is initially set to 0. The variable value is set to 100 after the transition to the state *Digesting* has fired, as denoted with the post condition. However, this depends on the channel *eatable*: The transition will fire only if the state machine can send on this channel, as expressed with the exclamation mark. In addition, during this transition an action *eat* is initiated, which is denoted with the according action label. In the state *Digesting*, two transitions are available. As long as the number of calories is larger than 0, a transition is chosen that has an action label *digest*, which leads to the post condition that the number of calories is reduced by 5. If the number of calories has reached 0, a transition to the state *Hungry* will fire.

The Plant state machine has an initial state *Grown* where the initial value of its integer variable *growth* is 100. If the state machine can receive on the channel *eatable*, it switches to the state *Eaten*. According to the post condition, the growth is set to 0 after this transition. As long as the growth is less than 10, a transition is chosen here that initiates the action *grow* and increases the growth by 1. If the growth has reached 10, a transition to the state *Grown* will fire.

In summary, the state machine system describes two interacting entities: Animals can eat plants if they are fully grown, and digest afterwards without the desire to eat any plants, until they are hungry again. So far, the model is rather simple. However, if such models are used to implement software – in this case for example as part of a graphical simulation of animals and plants – the state machine is not sufficient to describe a complete implementation. Therefore, the action labels are available for abstraction from any detailed implementation logic. If this state machine would be connected to program code, the actions *eat*, *digest* and *grow* would have to be connected to implementing methods. In addition, the channel is assumed here to be always available. In a more realistic example, the channel could be used for more detailed decisions if animals and plants can interact; in a graphical simulation, one can imagine that this decision is based on the location of animals and available plants. The creation of the program code pattern will therefore have to provide appropriate interconnections to the implementation.

4.1.2 Program Code Pattern

The program code pattern considers the state machine model defined above and defines program code structures that represent the abstract syntax of the state machine and provide an appropriate behavior at run time. Based on our decision to use Java to realize the embedded model, it is defined as follows:

Definition 7 $E_{StateMachine,Java}$ is a class of embedded models for $M_{StateMachine}$ in the Java programming language. □

4.1.2.1 States

States are static elements of the model and must thus be represented by static program code fragments. They are in addition the entry point to the program code pattern, since all other model elements are attached to transitions which

are in turn attached to states. For these reasons states will be represented by type definitions, called *state classes* in the following. According to the rules for pattern instantiation as defined in section 3.1.3.4, they must be identifiable so that pre-defined program code must exist to annotate them as state classes. In this case a marker interface is implemented by every state class. By this means, state classes are identifiable type-safely. The name of the class represents the name of the state.

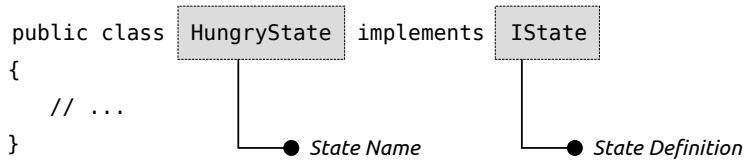


Figure 4.2: State definition in the program code pattern.

The state class for the state *Hungry* from the example given above can be seen in figure 4.2. The precise rules for the creation of state classes are the following:

- $\exists t_{State} \in \mathcal{T}, name(t_{State}) =$`IState`$, \nexists m \in \mathcal{M}, t_{State} \xrightarrow{contains} m$
- $\forall s \in S_{States} : \exists t \in \mathcal{T}_{States}, \mathcal{T}_{States} \subseteq \mathcal{T}, name(t) = name(s), t \xrightarrow{isOf} t_{State}$

In Java, the marker interface is a Java interface with the name `IState`.

4.1.2.2 Transitions and Actions

Transitions have the purpose to indicate paths between states and to provide labels for actions that make changes to the state space. Since transitions are each connected to at least one state, they are represented by methods inside state classes. These so-called *transition methods* are implemented in the state class of the state the transition emanates from, and have a pre-defined meta data annotation that refers to the class definition of the target state. This leads to the following rules for the creation of transition methods:

- $\exists a_{Transition} \in \mathcal{A}, name(a_{Transition}) =$ `Transition`
- $\forall t \in S_{Transitions} : \exists m_t \in \mathcal{M}_{Transition}, \mathcal{M}_{Transition} \subset \mathcal{M}, \exists s \in \mathcal{T}_{States}, s \xrightarrow{contains} m_t$
- $\forall m_t \in \mathcal{M}_{Transition} : m_t \xrightarrow{has} a_t \in \mathcal{A}, a_t \xrightarrow{isOf} a_{Transition}$
- $\exists p_{Target} \in \mathcal{AP}, name(p_{Target}) =$ `target`$, p_{Target} \xrightarrow{references} t_{Target} \in \mathcal{T}_{States}, a_{Transition} \xrightarrow{has} p_{Target}$

The implementation of actions to be invoked during transitions is not part of the model, however, we stated the goal to connect the state machine models to implementations directly. For this reason, each embedded state machine

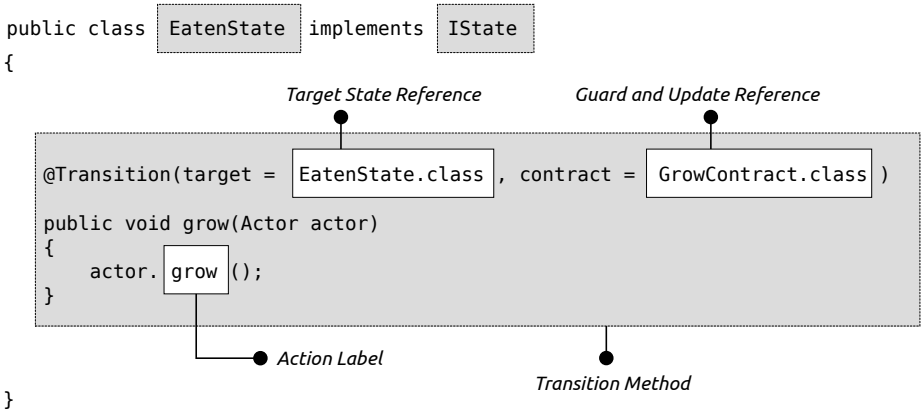


Figure 4.3: A transition in the embedded state machine pattern.

instance contains a type that acts as a facade to the application logic, in the following called *actor*. It contains methods that have no paramaters. The names of these methods are interpreted as action labels. To be used during run time, the actor is passed as parameter to each transition method. The transition methods then contain calls to the actor methods and represent a set of action labels by this means. The rules for actors are the following:

- $\exists t_{Actor} \in \mathcal{T}$
- $\forall a \in L_{Actions} : \exists m_a \in \mathcal{M}, name(m_a) = a, t_{Actor} \xrightarrow{contains} m_a, \nexists p \in \mathcal{MP}, m_a \xrightarrow{has} p$
- $\forall m \in \mathcal{M}_{Transition} : \exists p_{Actor} \in \mathcal{MP}, p_{Actor} \xrightarrow{isOf} t_{Actor}, m \xrightarrow{has} p_{Actor}$
- $\forall m \in \mathcal{M}_{Transition} : \exists L_{Actions_m} \subseteq L_{Actions}, \forall a \in L_{Actions_m} : \exists s \in \mathcal{S}, m \xrightarrow{contains} s, s \xrightarrow{invokes} m_a \in \mathcal{M}, t_{Actor} \xrightarrow{has} m_a, name(m_a) = a$
- $\forall m \in \mathcal{M}_{Transition} : \nexists s \in \mathcal{S}, m \xrightarrow{contains} s, \neg s \xrightarrow{invokes} m' \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m'$

The resulting program code is illustrated in figure 4.3. The transition method is named `grow`, although the name itself is not interpreted here. It has an annotation referring to the target state `EatenState`. The transition thus has the same source and target state as defined in the exemplary model in figure 4.1. The parameter of the transition method, which is by convention named `actor`, is of the type of the actor facade. The according instance will at run time be passed to this method by the execution framework. Inside the transition method, the actor method `grow` is called, so that this transition method has one action label `grow`. With these elements, the states, transitions, and actions are defined completely in the program code. In figure 4.3, the annotation contains an additional parameter `contract` used for guards and updates, which will be explained in the next section.

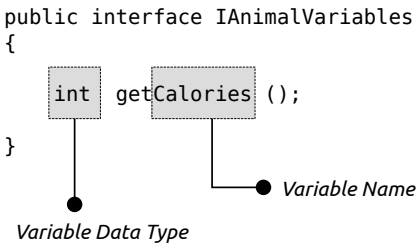


Figure 4.4: A variable facade type defining a variable `calories` as used in the example in section 4.1.1.5.

4.1.2.3 Variables, Guards, and Updates

One of the focal points of the state machine model introduced above is the consideration of a state space defined by variables that have a name and a data type. This state space is on the one hand interpreted by guards in order to decide which transitions are to fire. On the other hand, it is modified by updates to denote changes to the state space that occur during transitions. The consideration of the state space is of special interest in the context of embedded models: In contrast to the formal model itself, they are not self-contained, but instead part of larger applications. By this means the state space of the embedded state machine is only part of the larger state space of the overall application. However, the overall state space is not defined within the context of state machines, or may even consist of data types and value ranges that are not appropriate for use in the context of state machines. For this reason, the abstraction defining the limits between model and other program code must be reflected to focus on the subset of the state space that is of interest to the model accordingly.

Similar to actions, variables are defined in a facade type, too. This is realized by an interface containing a method for each variable that defines its name and data type as shown in figure 4.4. For this purpose the following rules apply:

- $\exists t_{Variables} \in \mathcal{T}$
- $\forall v \in L_{Variables} : \exists m_v \in \mathcal{M},\ name(m_v) = v,\ t_{Variables} \xrightarrow{contains} m_v,\ m_v \xrightarrow{returns} t_v \in \mathcal{T}, m_v \xrightarrow{contains} \epsilon$

$F_{Initial}$ is by this means defined by the return values of all methods of the variables facade instance at the point in time when the state machine is executed.

In Java, the methods in the variables interface are defined as get methods (cf. section 3.1.3.3).

As shown in figure 4.3, guards and updates are assigned to transitions by a reference to a so-called "contract class" in the meta data annotation. Contract classes define two methods containing the expressions for guard and update. Both return a boolean value. Classes are denoted contract classes with the implementation of a pre-defined interface named `IContract` that defines the guard and update methods, which are named `checkCondition` and `validate`. Guards and updates both consist of expressions with formulas in $\Sigma_{Propositional}$

that are implemented in methods as defined for expressions in section 3.1.3.2. In these expressions, the variable labels are denoted by invocations of the related methods of the variables type. This leads to the following definitions for the representation of guards and updates in the program code pattern:

- $\exists t_{Contract} \in \mathcal{T}$
- $\exists m_{Guard} \in \mathcal{M}, name(m_{Guard}) = \texttt{checkCondition}, t_{Contract} \xrightarrow{contains} m_{Guard}$, $m_{Guard} \xrightarrow{returns} t_{Boolean}$
- $\exists p_{m_{Guard}} \in \mathcal{MP}$, $p_{m_{Guard}} \xrightarrow{isOf} t_{Variables}$, $m_{Guard} \xrightarrow{has} p_{m_{Guard}}$
- $\exists m_{Update} \in \mathcal{M}$, $name(m_{Update}) = \texttt{validate}$, $t_{Contract} \xrightarrow{contains} m_{Update}$, $m_{Update} \xrightarrow{returns} t_{Boolean}$
- $\exists p_{m_{Update}} \in \mathcal{MP}$, $p_{m_{Update}} \xrightarrow{isOf} t_{Variables}$, $m_{Update} \xrightarrow{has} p_{m_{Update}}$
- $\exists p'_{m_{Update}} \in \mathcal{MP}$, $p'_{m_{Update}} \xrightarrow{isOf} t_{Variables}$, $m_{Update} \xrightarrow{has} p'_{m_{Update}}$
- $\exists \mathcal{T}_{Contracts} \subset \mathcal{T} : \forall t \in \mathcal{T}_{Contracts} : t \xrightarrow{isOf} t_{Contract}$

The method `checkCondition` realizes the guard by taking one parameter $p_{m_{Guard}}$ of the variables type with the current variable assignment and evaluating it to a boolean value. The method `validate` realizes the update by taking two parameters $p_{m_{Update}}$ and $p'_{m_{Update}}$ of the variables type: One with the current variable assignment, and one with the variable assignment from the point in time before the transition fired. It can thus be evaluated to a boolean value by enabling comparisons between variable values.

These contracts are used in the meta data annotations of transition methods:

- $\exists p_{Contract} \in \mathcal{AP}$, $name(p_{Contract}) = \texttt{contract}$, $p_{Contract} \xrightarrow{references} t_{Contract} \in \mathcal{T}_{Contracts}$, $a_{Transition} \xrightarrow{has} p_{Contract}$

The expressions in guards und updates are either simple expressions or composite expressions.

Simple expressions in guards are a tuple $\langle \lambda, \omega, \rho \rangle$ with

- λ a variable identifier $m_v \in \mathcal{M}, t_{Variables} \xrightarrow{contains} m_v$,
- $\omega \in \{=, \neq, <, >, \leq, \geq\}$ an operator,
- ρ either a literal value or a variable identifier $m'_v \in \mathcal{M}, t_{Variables} \xrightarrow{contains} m'_v, m'_v \neq m_v$.

Simple expressions in updates are a tuple $\langle \lambda, \rho \rangle$ with

- the left-hand side λ a variable identifier so that $\lambda \xrightarrow{invokes} m_v \in \mathcal{M}$, $t_{Variables} \xrightarrow{contains} m_v$, $\lambda \xrightarrow{has} p_{m_{Update}}$,

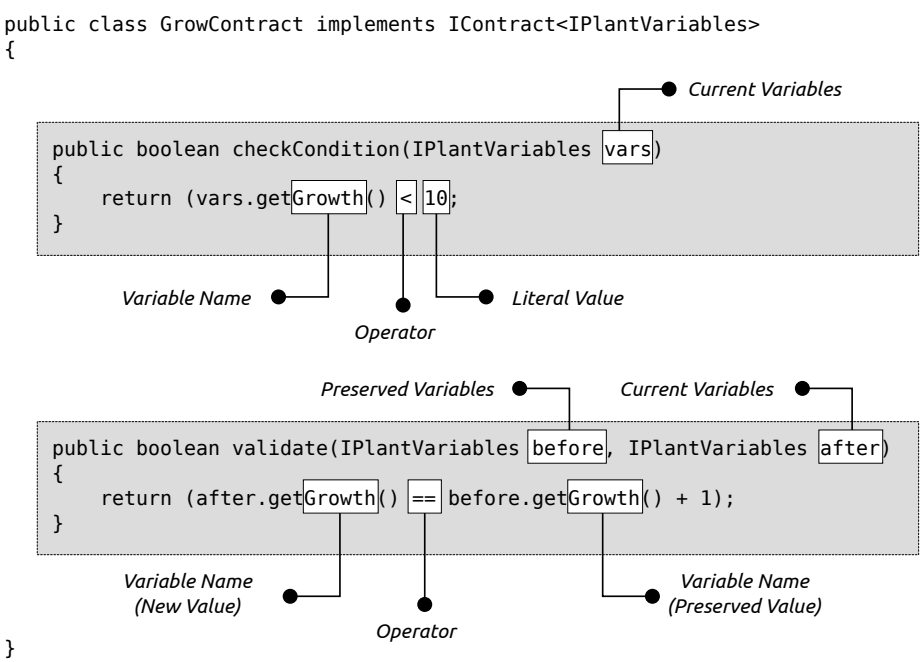


Figure 4.5: A contract class containing the guard method `checkCondition` and the update method `validate`. Both evaluate expressions by accessing the methods in the variables type, thus denoting the variable labels in use.

- the assignment operator $=$,
- the right-hand side ρ
  - either a literal value,
  - or a variable identifier ρ_1 connected to a literal value with an operator out of the set $\{+, -, \times, \div\}$ so that $\rho_1 \xrightarrow{invokes} m'_v \in \mathcal{M}, t_{Variables} \xrightarrow{contains} m'_v, \rho \xrightarrow{has} p'_{m_{Update}}$,
  - or a variable identifier ρ_1 connected to a variable identifier ρ_2 with an operator out of the set $\{+, -, \times, \div\}$ so that $\rho_1 \xrightarrow{invokes} m'_v \in \mathcal{M}, t_{Variables} \xrightarrow{contains} m'_v, \rho \xrightarrow{has} p'_{m_{Update}}$ and $\rho_2 \xrightarrow{invokes} m''_v \in \mathcal{M}, t_{Variables} \xrightarrow{contains} m''_v, \rho \xrightarrow{has} p'_{m_{Update}}$ with $m'_v \neq m''_v$.

A composite expression κ connects a set of simple expressions or composite expressions $\sigma_1 \ldots \sigma_n$ so that $\kappa = \sigma_1 \vee \ldots \vee \sigma_n$ or $\kappa = \sigma_1 \wedge \ldots \wedge \sigma_n$.

Figure 4.5 shows an examplary contract class for the transition with the action label `grow` introduced in the example in section 4.1.1.5. The variables are in this case provided by a the variables type `IPlantVariables`. The contract class is generic and parameterized with the variables type, which determines the type of the parameters. The guard for this transition requires that the variable `growth` is less than 10. Consequently, the program code consists of a method


```
public class EatChannel implements IChannel
{
    public ChannelSelection enable(Set<Object> senders, Set<Object> receivers)
    {
        // ...
    }
}
```

Figure 4.6: A channel class. The possible senders and receivers are passed to the method `enable` in sets. The method can then select a pair of sender and receiver that are enabled and return them in an instance of type `ChannelSelection`. While the channel definition is part of the model, the method body contains application logic that is not of interest to the state machine model.

call to the variable method `getGrowth()`, the operator `<`, and the literal value `10`. The update denotes that the variable `growth` is expected to be increased by 1. Thus, the update method contains a comparison between the variable for the points in time before and after the update. This is realized by an expression that invokes the method `getGrowth()` in its left-hand side on the update method parameter `after`. The right-hand side of the expression invokes the same method on the update method parameter `before` and adds the literal `1`.

4.1.2.4 Channels

Channels connect communicating state machines in a state machine system. From the perspective of the model introduced in section 4.1.1, they decide if state machines can send or receive based on the availability of a matching receiver or sender. When such models are embedded in complex software systems, as is the aim of embedded models, the related decisions may be based on the application logic of the underlying program. For this reason, channels are implemented as shown in illustration 4.6:

Each channel is represented by a class implementing the pre-defined interface `IChannel` that provides a method `enable`. This method supplies sets of state machine actors that try to send or receive as parameters. It returns a pair of actors selected by the channel implementation; the pair is represented by the type `ChannelSelection`. Inside the method, arbitrary application logic can be performed by the channel implementation to select the pair of state machines that are allowed to interact. For this purpose, the actor objects are accessible to represent the state machines and serve as an entry point for application logic. This leads to the following rules for channels:

- $\exists t_{Channel} \in \mathcal{T}, name(t_{Channel}) = \texttt{IChannel}$
- $\exists m_{enable} \in \mathcal{M}, name(m_{enable}) = \texttt{enable}, t_{Channel} \xrightarrow{contains} m_{enable}$
- $\exists p_{Senders} \in \mathcal{MP}, \; p_{Senders} \xrightarrow{isOf} any, \; num(p_{Senders}) \geq 0, \; m_{enable} \xrightarrow{has} p_{Senders}$

- $\exists p_{Receivers} \in \mathcal{MP},\ p_{Receivers} \xrightarrow{isOf} any,\ num(p_{Receivers}) \geq 0,\ m_{enable} \xrightarrow{has} p_{Receivers}$
- $m_{enable} \xrightarrow{returns} \langle s, r \rangle, s \in \mathcal{T}, s \xrightarrow{isOf} any, r \in \mathcal{T}, r \xrightarrow{isOf} any$

Channel classes are assigned to transition methods with the parameters `sends` or `receives` in the meta data annotation `@Transition`:

- $\exists p_{Sends} \in \mathcal{AP},\ name(p_{Sends}) = \texttt{sends},\ p_{Sends} \xrightarrow{references} t_{Sends} \in \mathcal{T},\ t_{Sends} \xrightarrow{isOf} t_{Channel}, a_{Transition} \xrightarrow{has} p_{Sends}$
- $\exists p_{Receives} \in \mathcal{AP}, name(p_{Receives}) = \texttt{receives}, p_{Receives} \xrightarrow{references} t_{Receives} \in \mathcal{T}, t_{Receives} \xrightarrow{isOf} t_{Channel}, a_{Transition} \xrightarrow{has} p_{Receives}$

4.1.3 Execution Semantics

As defined in section 3.1.4, each embedded model is accompanied by an execution framework that produces sequences of actions matching the execution semantics of the formal model. We will now outline the execution semantics of state machines that are of interest for this purpose and then introduce the execution algorithm necessary for embedded state machines.

4.1.3.1 Execution Algorithm

Since embedded state machines will be used to control the execution of application logic of surrounding applications, we consider state machine execution semantics in terms of sequences of actions [Peled et al., 2001]:

Definition 8 α is a sequence of actions $\alpha_0, \ldots, \alpha_n$ produced by a given state machine, iff α_0 is first action of a transition t where s_0 is the first component of t and the initial variable assignment is model for $F_{Guards}(t)$, and

- either α_m and α_{m+1} are succeeding actions in one transition
- or
  - α_m is the last action in a transition t,
  - and α_{m+1} is the first action in another transition t' where a state s' exists that is second component of t and first component of t',
  - and a variable assignment exists so that $F_{Updates}(t)$ and $F_{Guards}(t')$ evaluate to `true`,
  - and t' has no channels, or only channels that are enabled. □

This leads to the following algorithm for the execution of a single state machine:

1: $s := s_0$
2: **while** s.transitions $\neq \emptyset$ **do**
3: {Determine if the state machine has to wait for channels}
4: $wait := false$

```
5:    for all t in s.transitions do
6:      if (t.sends ∧ and ¬t.canSend) ∨ (t.receives ∧ ¬t.canReceive) then
7:        wait := true
8:      end if
9:    end for
10:   if wait = false then
11:     T := new empty set of transitions
12:     {Pre-select transitions with enabled channels first}
13:     for all t in s.transitions do
14:       if (t.sends ∧ and t.canSend) ∨ (t.receives ∧ t.canReceive) then
15:         T.add(t)
16:       end if
17:     end for
18:     {If no channels enabled, pre-select transitions without channels}
19:     if T = ∅ then
20:       for all t in s.transitions do
21:         if ¬t.sends ∧ ¬t.receives then
22:           T.add(t)
23:         end if
24:       end for
25:     end if
26:     {Select transition to fire}
27:     t_fire := null
28:     for all t in T do
29:       if t_fire = null then
30:         if invoke(t.guard) = true then
31:           t_fire = t
32:         end if
33:       end if
34:     end for
35:     {Fire transition}
36:     if t_fire ≠ null then
37:       invoke(t_fire)
38:       s := t_fire.target
39:       if invoke(t_fire.update ≠ true) then
40:         print Error: Update failed!
41:         exit
42:       end if
43:     else
44:       print Error: No viable transition!
45:       exit
46:     end if
47:   end if
48: end while
```

The execution starts in the initial state s_0 with the initial assignment. For each current state, the transition that will fire is selected in two steps.

First, transitions are considered that want to send or receive on channels. For each of those the execution framework checks if it is possible for them to send or receive. If yes, these transitions are pre-selected so that a temporary

set of transition candidates exists, denoted as T in the algorithm above. If no transitions are related to channels, all transitions without channels are added to T. By this means the execution algorithm ensures that transitions with channels are preferred, and disregards all transitions with non-enabled channels. If transitions have channels that are not enabled, the state machine waits and thus does not fire any transitions even if their guards are enabled.

Second, the transition to fire is selected from the set T by evaluating the guards. The execution framework invokes the guard for each transition and passes the variables to it. The first transition whose guard invocation evaluates to `true` is selected to fire. In the algorithm sketched above, it is denoted with the variable t_{fire}.

If t_{fire} exists, the related transition method is invoked. By this means the contained action labels are also invoked. Afterwards, the update of t_{fire} is invoked. Depending on the execution framework, detection of non-successful updates can lead to a warning or abortion of the execution. If the update is invoked successfully, the current state is set to the target of t_{fire}.

4.1.3.2 State Machine Interaction

The algorithm above describes the state machine execution from the perspective of a single state machine. The execution semantics provided by a framework must also accomplish execution of a state machine system. The execution of a state machine system is illustrated in figure 4.7 showing the interactions between different components. (Please note that, although this illustration is similar to UML sequence diagrams, only the interaction between components and not specific instances of components is depicted.)

The embedded state machine system works with so-called *steps*. In each step, all state machines in the system try to fire a transition. The beginning of a step is indicated with a call to the method `next` to the state machine system.

The state machine system first requests all state machines to register for channels. This means that each state machine considers the transitions in the active state. Those that want to send or receive on channel are registered as senders or receivers on the related channel. The channels are by this means supplied with information about possible senders and receivers so that they can allow state machines to interact later on.

This selection is initiated by the state machine system calling the command `enable` of each channel. The result is a pair of sender and receiver which is returned to the state machine system. After the state machine system has collected senders and receivers from all channels, it calls the command `checkAndFire` on the state machines of all senders and receivers and passes the transition to them. The state machines invoke the related guard. If it evaluates to `true`, the transition fires. If not, the state machine remains in the same state.

Afterwards the state machines that have not registered for channels are considered. The state machine system calls the command `selectAndFire` on each of them, leading to the evaluation of the guards of all transitions without channels and the firing of the first transition whose guard evaluates to `true`.

This perspective on the execution algorithm is slightly different than the algorithm introduced in the previous section since it is modularized with respect

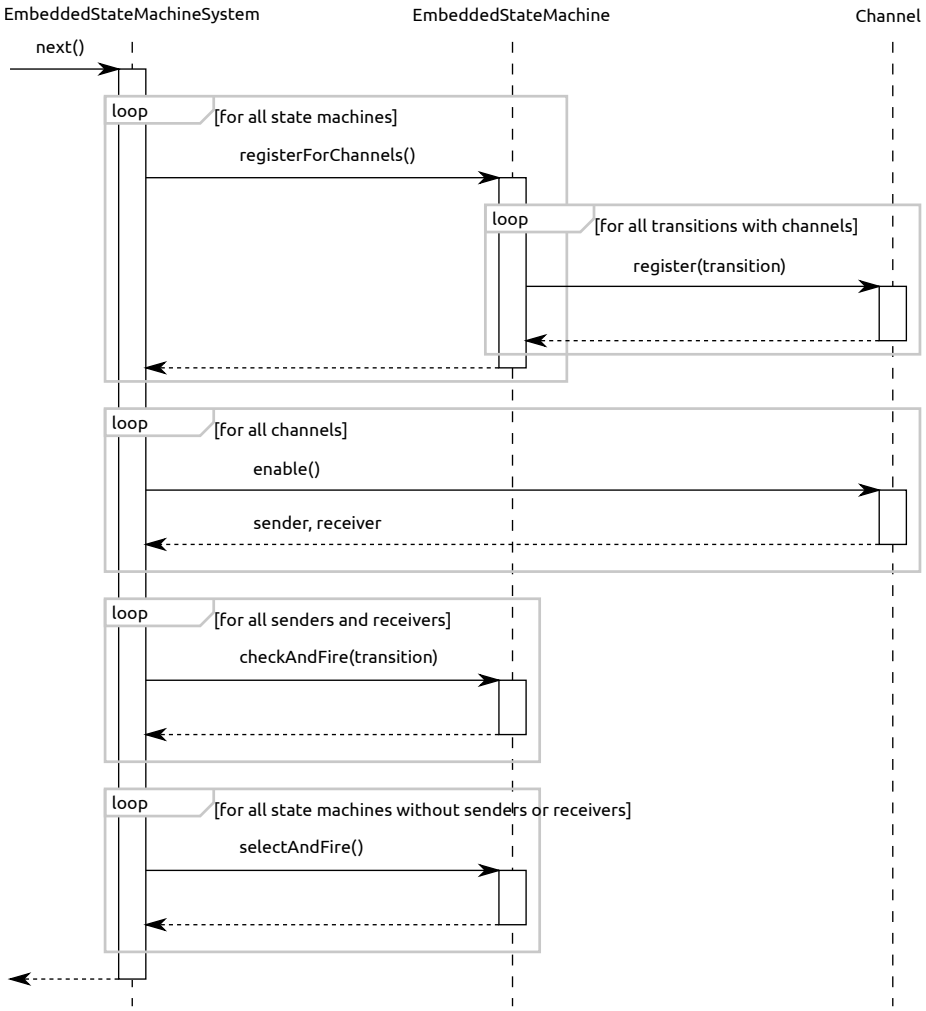


Figure 4.7: The execution sequence in an embedded state machine system.

to the participating components. However, the execution semantics are equivalent since the algorithm for single state machines abstracts from the interaction details, but performs the same actions.

4.1.3.3 Use of Program Code Fragments

The execution framework for embedded state machines works with the elements of the program code pattern.

For this purpose, the graph of states and transitions is *interpreted* by the execution framework. It reads the static structures of state classes marked with the interface `IState`, transition methods, and the transition meta data annotation `@Transition` with the parameter `target`. The framework also interprets the transition meta data parameter `contract` and the related contract classes that implement the interface `IContract`. For channels, the transition meta data parameters `sends` or `receives` are read together with the channel classes implementing the interface `IChannel`.

These static structures are interpreted by the execution framework to determine *invocations* of program code for making decisions or initiating the execution of actions. Following the execution semantics introduced above, this first concerns the method `enable` provided by the interface `IChannel` for the channel types. Before transitions fire, their guards are checked with the method `checkCondition` determined by the interface `IContract` for contract classes. When a transition is selected, its method is invoked so that the contained actor methods representing the action labels are invoked by this means, too. Afterwards, the method `validate` in the related contract is invoked. The locations of these four executable parts are determined by the static structures that are interpreted. The executable parts themselves are considered black boxes by the execution framework.

4.1.4 Interface Definitions

As described in section 3.1.5, interfaces are an important part of the program code patterns since they connect embedded models to arbitrary application logic of the programs the models are embedded in. In embedded state machines, three kinds of interfaces exist:

- The *actor* is a collection of labels. It is action-oriented since the methods provide entry points to the application logic, but do not return any data. Actors are accessed in transition methods and channel classes.

- The *variables* facade is a collection of labels. It is data-oriented since variable methods deliver variable values from the state space of the surrounding program, which are possibly aggregated. Variables are accessed in guard and update methods.

- *Channels* each represent one label. They are data-oriented, too, since they process application logic to determine which pair of sender or receiver will be enabled, and thus only return this pair. Channels are not accessed from inside the program code pattern, but only from the execution framework.

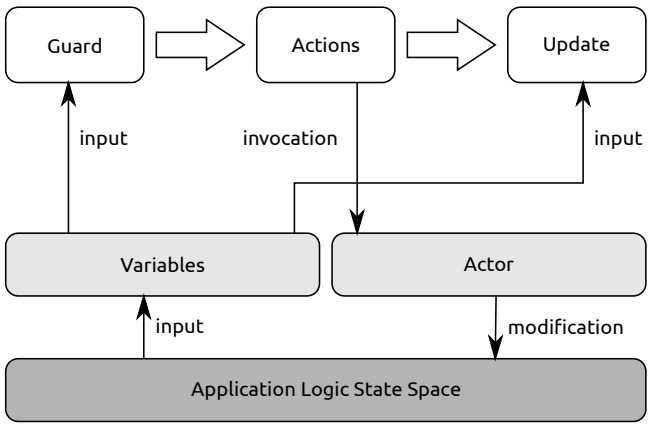


Figure 4.8: The role of interfaces in embedded state machines.

An embedded state machine is by this means connected to the surrounding program as illustrated in figure 4.8:

In each step, guards are evaluated first. They read variables as input which are extracted from the state space of the surrounding program. The state space is therefore expected to fulfill the requirements of a guard of at least one transition in the current state. However, the variable values cannot be controlled from inside the embedded model, and irregularities cannot be detected with model checking at the level of the state machine model. The application logic is therefore responsible for ensuring that the state machine variables are valid in the context of the model. At development time, this could be supported with static analysis and model checking of program code [Holzmann et al., 2008] since the requirements to the state space are for each state precisely given by the guards' expressions (cf. section 4.4.3.2). At run time, inconsistencies will be detected if the state machine takes other paths as expected or runs into a deadlock; detection and analysis of such errors can be supported by monitoring tools (cf. section 3.2.4).

When a guard has been evaluated successfully, the overall system is in a certain state as far as determined by the guard. The actions that are initiated at this point in time can modify the state space of the surrounding application. Since the actions themselves and the transition methods do not make any assumptions about these changes, they are black boxes to the embedded state machines so that the changes are not directly comprehensible.

After a transition has fired, the update reads the variables again. It validates if the overall program is in an expected state with respect to the set of variables used in the expression. Again, the variable values delivered to the update are extracted from the state space of the surrounding program. By this means the application logic is responsible for ensuring that modifications to the overall state space during actions are compatible to requirements made by the update. This connection between modifications in actions and variable values accessed for the related updates can also be supported by static code analysis and model checking. At run time, deviations can be detected with monitoring, too.

After an update has been evaluated and the target state has been reached, the evaluation of guards will start again. Therefore the update must leave the

system in a state that is compatible to the requirements of the next guard. Since this affects the variables of the state machine, it can be validated at the level of the model. However, for this validation to be meaningful, the variable methods are required not to change the overall state space themselves. Modifications to the state space should occur during actions only, which conforms to the model semantics. This requirement can also be validated with static code analysis.

The method `enable` in channels can access the application logic via actors to decide which pair of sender and receiver will be enabled. Since the principle that only actions should modify the state space applies here, too, channels should access actors only for reading. Since the `enable` methods are not part of the model, but represented in the model only with the channel's name, the interactions there are not comprehensible at the level of the model. Again, the application logic is responsible for working in a way that is compatible with the model semantics, which can be validated with static code analysis based on slicing (cf. section 3.1.5.2).

4.1.5 Transformations

To enable working at different abstraction levels, transformations are necessary for embedded state machines as described in section 3.1.6. For the domain of state machines, several views can exist for editing, verification, model checking, and monitoring. We will outline the default transformations here and then introduce a special transformation to the automata model checking tool UPPAAL.

4.1.5.1 Default Transformations

The default transformation extracting the model from source code considers the elements of the program code pattern as introduced in section 4.1.2:

- States with their names;
- transitions with their names, action labels, and references to targets, contracts, and channels;
- channels with their names;
- variables with their names and data types;
- actor methods with their names;
- guards and updates with their expressions.

The default transformation extracting the model from byte code works with the following elements:

- States with their names;
- transitions with their names, target, contract, and channels references;
- channels with their names;
- variables with their names and data types;

- actor methods with their names.

These transformations can be internal so that the elements can be represented in view-specific tools working with the state machine semantics. In this case no explicit target for the transformation is needed. However, the default transformations can also be extended to other external transformations with the definition of a target format.

4.1.5.2 UPPAAL

For state machines, verification tools exist that focus on some aspects of the models only, but provide thorough verification for them. We thus present a transformation from embedded state machines to the format of the timed automata model checker UPPAAL [Larsen et al., 1997]. Although timed automata themselves are not used in this embedded model, UPPAAL provides model checking features for general properties of state machines, like value ranges, reachability and deadlocks, and similar. However, it does for this purpose not consider actions. The transformation thus allows to focus on the aspects covered by UPPAAL.

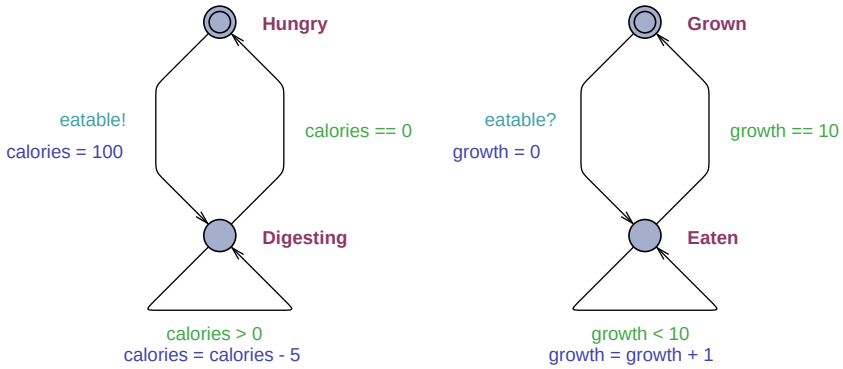
A state machine system in UPPAAL is denoted a *system*, too. It consists of global variable declarations, single state machines called *templates*, a definition of template instances to be used for verification, and channels identified by a name. The variable declarations each have a boolean or integer data type and a name.

Each template consists of local variable declarations, named states (called *locations*), and transitions between them (called *edges*). One location is the initial state. Edges have several properties: *Selections* declare variables with a value from a non-deterministic choice; *guards* are pre-conditions that work with global variables, template variables, and selections; *synchronisation* is specified with channel labels for receiving or sending; *updates* modify the system state after a transition has fired by modifying global variables and template variables.

The example introduced in section 4.1.1.5 consists of the following fragments if transformed to UPPAAL:

- The template for the animal, see figure 4.9(a).
- The variable declaration for the animal template: `int calories = 0;`
- The template for the plant, see figure 4.9(b).
- The variable declaration for the plant template: `int growth = 0;`
- The global declarations for the channel: `chan eatable;`
- The system declaration using one instance of each the animal and the plant templates (named `a` and `p`), see listing 4.1.

The data processed by UPPAAL is similar to the definition of the formal model for embedded state machines used here. However, the information about actions must be omitted. The transformation will thus consist of the following elements:



(a) The Animal template.

(b) The Plant template.

Figure 4.9: The UPPAAL templates for the state machine example (cf. figure 4.1).

```
// Place template instantiations here.
a = Animal();
p = Plant();

// List one or more processes to be composed into a system.
system a, p;
```

Listing 4.1: The system declaration for the state machine example in UPPAAL.

- For each state machine with its variables interface, a template in UPPAAL is created.
- Each channel class is transformed to a global channel declaration in UPPAAL.
- For each state class, a location in the respective UPPAAL template is created that has the name of the class.
- Each transition becomes an edge in UPPAAL. The name of the transition is not considered since edges in UPPAAL are not named.
- For each variable interface, all methods are transformed to local variable declarations in the related UPPAAL template. However, only integer and boolean data types are allowed.
- All guards and updates are transformed to guard and update expressions at the related edges in UPPAAL.
- The channel references in the transition meta data annotation are transformed to appropriate synchronization expressions attached to UPPAAL edges.

This special view can be used by the UPPAAL tool for three purposes: Visualization, simulation, and verification of the model. The visualization is shown in figure 4.9. Since the program code does not contain layout information of the model elements, the layout has to be set manually. Simulation of the model

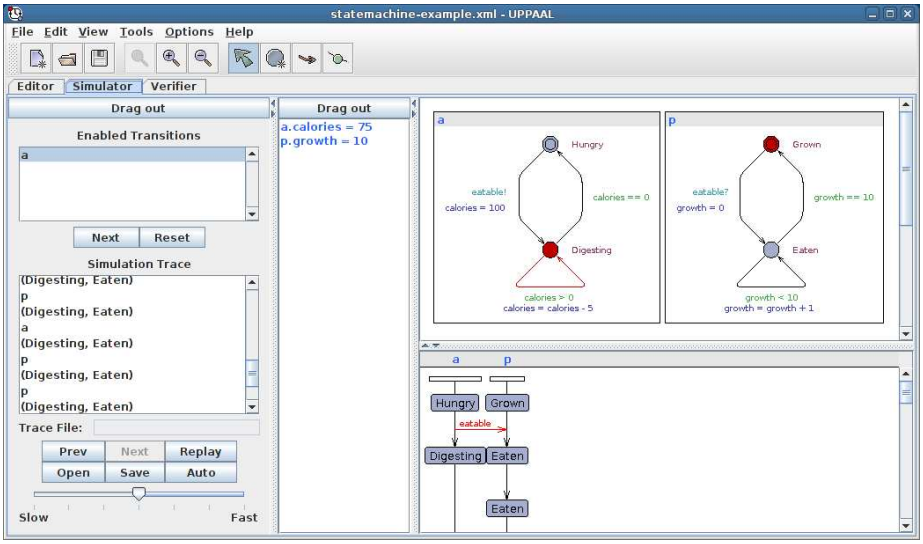


Figure 4.10: The example state machine in UPPAAL's simulator.

is possible as shown in figure 4.10: The current states of the single automata instances are shown including the state space. For each step taken by the state machine system, the states, transitions, and channels are visualized so that the behavior of the system is comprehensible.

In addition, the extracted model can be verified with respect to the state space and reachability properties as far as defined by UPPAAL. UPPAAL provides Computation Tree Logic expressions [Clarke et al., 1986] for verification. In the context of the example, the following exemplary queries to the UPPAAL verifier could be of interest:

- `A[] not deadlock` – the system is free of deadlocks.
- `A[] a.calories >= 0 && a.calories <= 100` and `A[] p.growth >= 0 && p.growth <= 10` – guards and updates are specified such that the variables are limited to their intended value ranges.
- `E<> a.Digesting` and `E<> p.Eaten` – the states that are not initial states are reachable. This means that variables and channels are designed in a way that the state machine system is functional.
- `E[] a.Digesting imply p.Eaten` – The fact that an animal is digesting corresponds with the fact that the plant has been eaten.

With this transformation, the automata view on embedded state machines can be instrumented to verify certain properties of the program thoroughly, although it does not cover all aspects of the model.

4.2 Process Models

State machines are appropriate for modeling systems that have a well-defined state space. As shown above, programs can be designed with embedded mod-

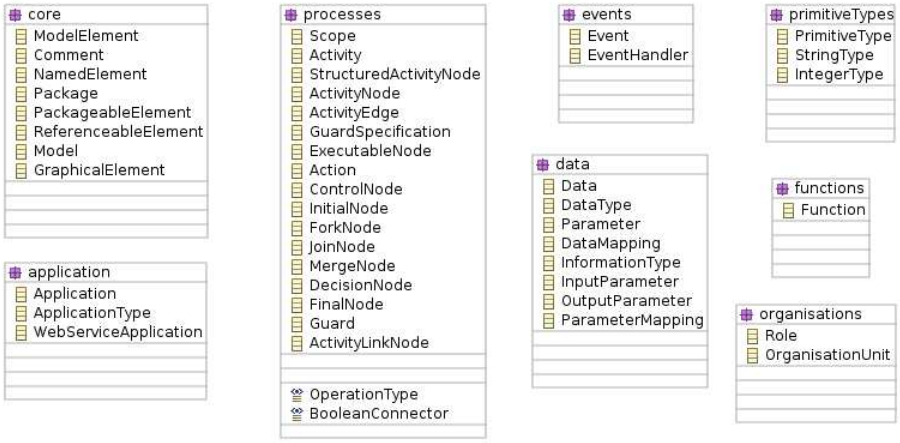


Figure 4.11: The Ecore meta model of JWT.

els and verification can be applied to the program code with respect to the state space and its semantics in the context of state machines. However, interaction of state machines with other parts of the program happens only implicitly and is not part of the model. For this reason we present another class of embedded models here for process models respectively workflow models. These are also based on the assumption that the program flow can be represented in certain states, actions, and transitions. But, different to state machines, they consider data exchange with programs (or parts of a program) explicitly. At the same time, the state space is not modeled completely. With the consideration of process models in addition to state machines we want to show that different aspects of behavioral systems can be expressed in appropriate embedded models.

4.2.1 Model Definition

Different approaches exist for modeling processes and workflows. The model we will use is based on the process meta model defined by the Java Workflow Toolkit [JWT], a set of plugins for the Eclipse IDE that support engineering workflow models independent from specific run time or server environments. JWT is for this purpose related to multiple other workflow modeling types [Lautenbacher, 2007] and appropriate transformations exist. The JWT meta model is defined as a model in *Ecore*, which is a language for formal model definition in the Eclipse Modeling Framework [Budinsky et al., 2009]. The meta model has certain packages that are shown in figure 4.11. We will now describe their contents and select a subset of features to create an embedded model. The resulting process model is defined as follows:

Definition 9 $A_{Process} = \langle L_{Activities}, S_{Nodes}, S_{Edges}, L_{Applications}, \Sigma_{Propositional}, F_{Guards}, F_{SubProcesses}, F_{Initial}, s_{start} \rangle$ is the abstract syntax derived from A_{Meta} for a model class of process models $M_{Process}$ with

- $L_{Activities}$ a finite set of activities,

- S_{Nodes} a finite set of nodes in activities,
- $S_{Edges} \subseteq S_{Nodes} \times S_{Nodes}$ a finite set of edges,
- $L_{Applications}$ a finite set of application labels,
- $\Sigma_{Propositional}$ the propositional logic, including a finite set of variables (called *data* in the context of JWT) L_{Data} and a finite set of data types for variables $L_{DataTypes}$, which are combined in a set of variable declarations $S_{Variables}$,
- $F_{SubProcesses}$ a function that assigns sub process invocations to some nodes,
- F_{Guards} a function that assigns logical formulas for guards to edges,
- $F_{Initial}$ a function that gives an initial assignment for all data defined in the logic,
- $s_{start} \in S_{Nodes}$ the initial node. □

4.2.1.1 Package `core`

This package contains common supertypes for model elements and the basic element `Model` that is the root object for all information in a JWT workflow model. It contains meta data for process models with the attributes `author`, `version`, `description`, and `fileversion`. These are not of interest for the embedded model itself. Considering the common supertypes in this package, the model element `NamedElement` is of interest since many other elements extend it. It provides the attribute `name` that assigns a name to elements.

Model Definition These common supertypes do not contribute to the definition of $M_{Process}$.

4.2.1.2 Packages `data` and `primitiveTypes`

The packages `data` and `primitiveTypes` contain descriptions for data used as input and output of actions. Two basic data types are pre-defined: `StringType` for character strings and `IntegerType` for integer numbers. They do not have any attributes since they are only used to identify data elements. Arbitrary types can be identified with model elements of type `DataType` that are described with a name. Data types are used by elements of type `Data` that have a name (attribute `name`), a value (attribute `value`), and a relation to the data type (attribute `dataType`). Each process model can thus have a set of data units that can be referenced by other model elements.

The package `data` also contains the model elements `InputParameter` and `OutputParameter` that define names and default values for parameters of actions.

Furthermore, the package `data` contains model elements that allow to map data types. This is not of interest for embedded models since the data definition is to be integrated in the program code of surrounding applications, so we will not consider this functionality.

Model Definition In $A_{Process}$, a set of variable labels L_{Data} and the set of their data types $L_{DataTypes}$ is defined as part of the propositional logic $\Sigma_{Propositional}$. They represent the `Data` and `DataType` elements. We say that a *variable declaration* v is a tuple $\langle d, t \rangle, d \in L_{Data}, t \in L_{DataTypes}$. $S_{Variables}$ is the set of all variable declarations. When the process is started, the function $F_{Initial}$ assigns an initial value to each variable. $\Sigma_{Propositional}$ itself is used to build expressions that access these variables in guards. In addition, the values can be read from elements out of L_{Data} as parameters to actions and be assigned to them as a result of actions.

4.2.1.3 Package `application`

The package `application` contains foremost the model element `Application` representing an application that can be invoked during actions. It has two attributes of interest: `javaClass` refers to a fully-qualified class name of a Java class that provides this application, and `method` is the name of the method inside the Java class that constitutes the application. Applications also have a list of input and output parameters (models elements `InputParameter` and `OutputParameter`). We define for our embedded model that each application does only have 0 or 1 output parameters; the reason will be explained in section 4.2.2.3 when the program code pattern is introduced.

In addition, this package contains model elements describing different types of applications. Since the embedded model will connect the processes to application code only, this is not of interest and will not be considered.

Model Definition For the embedded model, we assume that exactly one Java class provides methods as entry points to the application logic that are called from actions. Therefore, only a method name is needed to describe actions. $A_{Process}$ thus contains a set of application labels $L_{Applications}$. Input and output parameters will be defined by the method signatures and are thus not modeled explicitly.

4.2.1.4 Package `processes`

This package contains model elements representing the static structure of processes. In JWT, the graph of a workflow built from these elements is called *activity* and represented by the type `Activity`. Its contents can be roughly categorized in model elements for control nodes, executable nodes, and edges. Control nodes and executable nodes share a super type `ActivityNode` that has two attributes `in` and `out` with lists of incoming and outgoing edges.

Control Nodes Control nodes have a super type `ControlNode` that does not specify any attributes. The control nodes themselves do not have any attributes, too. `InitialNode` and `FinalNode` represent beginning and end of an activity. A `DecisionNode` has multiple outgoing edges that are equipped with guards to make decisions. Multiple paths started in decision nodes are brought together in nodes of type `MergeNode`. If parallel processing of tasks in activities is possible, a `ForkNode` represents the beginning of multiple threads. These different paths are brought together in nodes of type `JoinNode`. For our

purpose, we will not consider parallel processing and therefore neglect fork and join nodes in the following since they would have little influence on the program code pattern.

Executable Nodes The node type `Action` is of interest since it denotes execution of application logic. We consider the following attributes: `executedBy` refers to an application being executed in this action; `inputs` and `outputs` are sets of data used as input and output parameters. Actions contain in addition information about roles, mappings, and time constraints, but these will not be considered in the embedded model.

In JWT, activities can be nested. We consider for this purpose the executable node type `ActivityLinkNode`. It refers to another activity with its parameter `linksto`.

Edges Edges are represented by the model element `ActivityEdge`. Its attributes `source` and `target` refer to the nodes the edge connects. The attribute `guard` does optionally refer to a guard. The respective model element `Guard` contains as attributes a textual description of the guard expression (attribute `shortdescription`) and a data structure representing the guard in detail in a model element of type `GuardSpecification` (attribute `detailedSpecification`).

`GuardSpecifications` can contain two kinds of information. Either they define a simple expression with a data element (attribute `data`), an operation type (attribute `operation`), and a value (attribute `value`). The operation is in this case a model element `OperationType` out of the enumeration `Equals`, `Lower`, `LowerEquals`, `Greater`, `GreaterEquals`, or `UnEquals`. Or they define a list of sub specifications (attribute `subSpecification`) and connect them with a boolean operator (attribute `subSpecificationConnector`). The latter is a model element `BooleanConnector` out of the enumeration `AND` or `OR`.

Model Definition This package contributes to $A_{Process}$:

- $L_{Activities}$ is the set of activity names.
- S_{Nodes} represents all nodes in activities so that $S_{Nodes} = S_{InitialNodes} \cup S_{FinalNodes} \cup S_{Actions} \cup S_{DecisionNodes} \cup S_{MergeNodes} \cup S_{SubProcessNodes}$ with $num(S_{InitialNodes}) = 1 \wedge num(S_{FinalNodes}) = 1$. Each activity has exactly one initial node $s_i \in S_{InitialNodes}$ and one final node $s_f \in S_{FinalNodes}$.
- Each action $a \in S_{Actions}$ has an application label $application_a \in L_{Applications}$ for its attribute `executedBy`, zero or one output data label $output_a \in L_{Data}$, and zero or more input data labels $inputs_a \subseteq L_{Data}$.
- $F_{SubProcesses}$ assigns the name of an activity $a \in L_{Activities}$ to each $s \in S_{SubProcessNodes}$.
- $S_{Edges} \subseteq S_{Nodes} \times S_{Nodes}$ is the set of edges representing all elements of type `ActivityEdge`, so that each edge has a source node and a target node. Each node $n \in S_{Nodes}$ has a set of incoming edges In_n and a set of outgoing edges Out_n.

- Guards are assigned to transitions by F_{Guards}. Each `GuardSpecification` is either a simple or a complex boolean expression. Complex expressions connect simple expressions with operators of type `subSpecificationConnector`, i.e. out of $\{\&\&, ||\}$. Simple expressions have three components:
  - a left side which is always a variable identifier $d \in L_{Data}$,
  - an operator of `OperationType`, i.e. out of $\{==, <, <=, >, >=, != \}$,
  - a right side which is either a literal value or another variable identifier $d' \in L_{Data}$.

4.2.1.5 Unused packages

The package `events` describes types for events. This is not of interest since we only consider events with respect to program code invocations, so that they must not be modeled explicitly. The package `organisations` contains information about organisational aspects of processes. This is not of interest for the engineering of program code. The package `functions` describe application logic to be executed, which is not necessary in the context of an embedded model, since it is connected to program code for this purpose. These packages will therefore be neglected for the definition of our embedded model.

4.2.1.6 Example

A non-trivial use case modeled with JWT will be introduced in chapter 5. For now, consider the following simple example:

A load generator application for performance tests makes requests to a system under test with different worker threads. The number of threads is adjusted until the turnaround time of the requests is within a given range. The number of worker threads that produces turnaround times in this range is considered the optimum and returned as the result of the load generation process.

The load generation is thus controlled by the following variables:

- `numberOfWorkers`: The number of workers used for the current measurement.
- `turnaroundTime`: The mean turnaround time as determined by the last measurement.
- `lowerLimit` and `upperLimit`: The limits that define the acceptable range of turnaround times.

The measurement follows these steps:

1. The process is started with `numberOfWorkers` being the initial number of workers.
2. A measurement is performed with `numberOfWorkers` as input that returns the `turnaroundTime`.
3. If the `turnaroundTime` is less than the `lowerLimit`, the number of workers is increased and the process continues with step 2.

4. If the `turnaroundTime` is larger than the `upperLimit`, the number of workers is decreased and the process continues with step 2.
5. If the `turnaroundTime` is between `lowerLimit` and `upperLimit`, the `numberOfWorkers` is shown to the user and the process ends.

This can be modeled as a process as shown in figure 4.12 in the notation of JWT. The process consists of two activities, one for the load generation and one for the adjustment of the number of workers. The latter is included as a sub process in the first activity.

The load generation activity (figure 4.12(a)) has the necessary start node at the top that leads to a merge node (labeled with an exclamation mark) and then to the action node `MeasurementNode` where measurements are initiated. As denoted by a dashed so-called *reference edge* in the diagram, this node is executed by the application `performMeasurement`. The diagram also visualizes input data, in this case the `numberOfWorkers`, and output data, in this case the `turnaroundTime` that is the result of this measurement. After the measurement, the decision is to be made whether the number of workers will be adjusted or the measurement will be finished. This is realized by a decision node (labeled with a question mark) with two outgoing edges that are equipped by guards. The first edge applies if the `turnaroundTime` is within the upper and lower limits (guard `turnaroundTime >= lowerLimit && turnaroundTime <= upperLimit`). In this case, the results are printed in the action node `ShowResultsNode` and the process finishes. Otherwise (guard `turnaroundTime < lowerLimit || turnaroundTime > upperLimit`), the second edge leads to a sub process referenced in the `AdjustWorkersNode`. In this case, the process is continued at the merge node afterwards.

The sub process for the adjustment of workers (figure 4.12(b)) has also a start node. It leads to a decision node whose outgoing edges have guards: If the load has been too low to reach the defined limits (`turnaroundTime < lowerLimit`), the number of workers is increased in the `IncreaseWorkersNode`, which calls the appropriate application `increaseWorkers`. Accordingly, if the load has been too high, the number of workers is decreased in the `DecreaseWorkersNode` calling the application `decreaseWorkers`. Both actions return the data `numberOfWorkers`, i.e. the modified number of workers for the next measurement. After one of the actions has been performed, the process reaches a merge node and finishes.

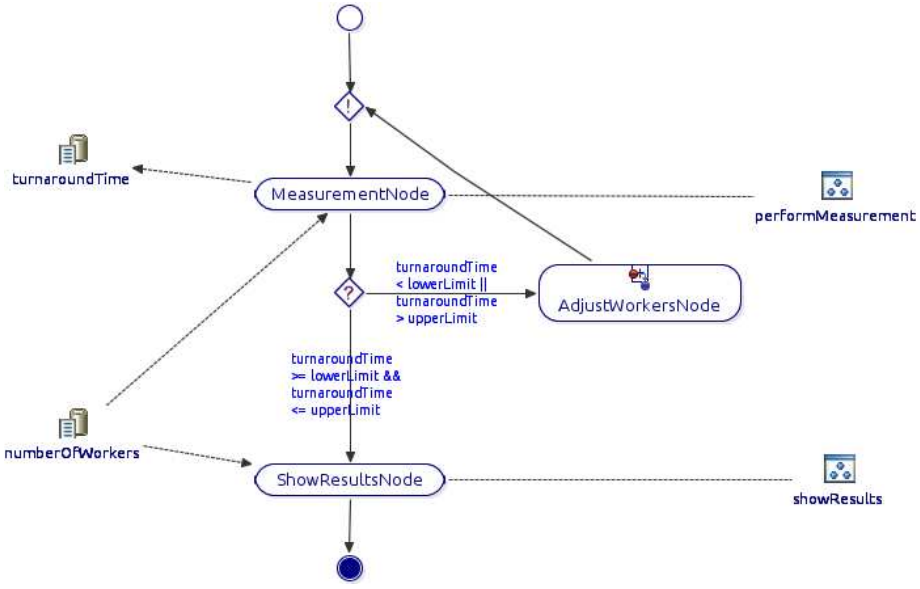
4.2.2 Program Code Pattern

The program code pattern considers the process model $M_{Process}$ defined above and defines program code structures that represent the abstract syntax of the processes and facilitate an appropriate behavior at run time. Based on our decision to use Java to realize the embedded model, it is defined as follows:

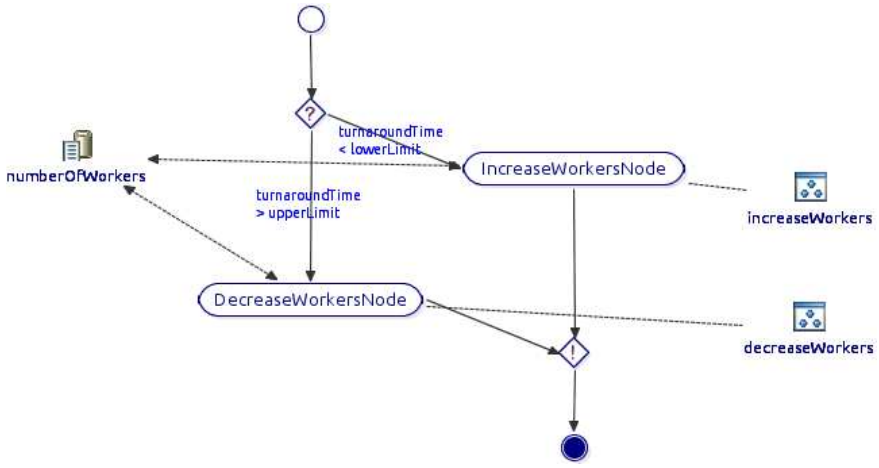
Definition 10 $E_{Process,Java}$ is a class of embedded models for $M_{Process}$ in the Java programming language. □

4.2.2.1 Applications

Applications are represented by methods in a facade type called *actor* that will be defined for each process model. For each application, one method exists in



(a) The load generation activity.



(b) The adjustment activity. It is a sub process contained in the node "AdjustWorkersNode" in figure 4.12(a).

Figure 4.12: The model for the process example in the graphical JWT notation.

this type. The method name in the facade type equals the method name of the related application; the class name of the application is always the class name of the facade class. This leads to the following rules:

- $\exists t_{Actor} \in \mathcal{T}$
- $\forall a \in L_{Applications} : \exists m \in \mathcal{M}, t_{Actor} \xrightarrow{has} m, name(m) = a$

Since the input and output parameters are not modeled explicitly in $M_{Process}$, which only references the method name, these rules for embedding the respective information are sufficient.

4.2.2.2 Data

Data items are represented by another facade type called *variables* that will be defined for each process model:

- $\exists t_{Variables} \in \mathcal{T}$

For each data item, two kinds of method can exist: A *get method* that takes no parameters and returns the data type, and a *set method* that takes a parameter of the data type and does not return anything (cf. figure 4.13). These methods are defined as follows:

Definition 11 get_v is a get method for the variable declaration v with

- $v = \langle d, t \rangle \in S_{Variables}$,
- $get_v \in \mathcal{M}$,
- $t_{Variables} \xrightarrow{contains} get_v$,
- $get_v \xrightarrow{returns} t$,
- $\nexists p \in \mathcal{MP}, get_v \xrightarrow{has} p$ □

Definition 12 set_v is a set method for the variable declaration v with

- $v = \langle d, t \rangle \in S_{Variables}$,
- $set_v \in \mathcal{M}$,
- $t_{Variables} \xrightarrow{contains} set_v$,
- $set_v \xrightarrow{returns} \epsilon$,
- $\exists p \in \mathcal{MP}, p \xrightarrow{isOf} t, set_v \xrightarrow{has} p$ □

From the perspective of the model, variables can be both read and written, so that both kinds of methods exist usually. However, developers can choose to implement only one of the methods in order to indicate that variables are only read or only written by the model. Since both kinds of methods allow to infer the variable name and type, they are independent from each other and thus separately sufficient for the definition of the variable:

- $\forall v \in S_{Variables} : (\exists get_v) \vee (\exists set_v)$

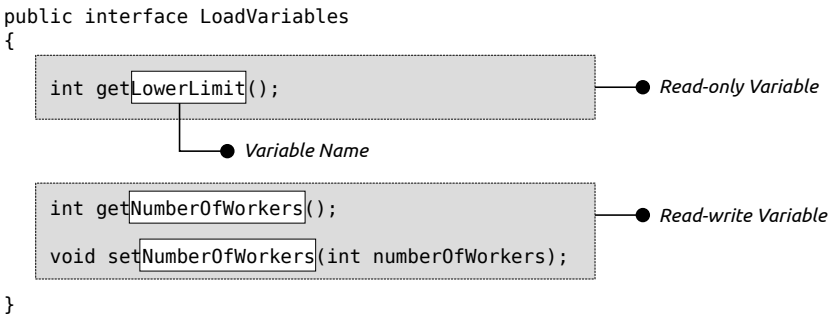


Figure 4.13: The definition of data items in a *variables* interface.

4.2.2.3 Activities

As described above, a process model contains one or more activities which are named. When they are embedded in program code, the activity names must be given and the respective code fragments will have to be structured so that their affiliation to one activity is represented. For this reason we define that a package exists for each activity, and that the name of the activity equals the name of the package:

- $\forall a \in L_{Activities} : \exists k_a \in \mathcal{K}_{Activities}, \mathcal{K}_{Activities} \subseteq \mathcal{K}, name(k_a) = a$

The foundation of process models is the activities' graph structure of process nodes and transitions between them. In the object-oriented program code fragments for the pattern, each process node is represented by a class definition which implements a pre-defined interface ("node class" in the following). These interfaces extend the marker interface `IProcessNode`. The name of the node equals the name of the node class. We distinguish the following kinds of nodes:

- Initial nodes with node classes in $\mathcal{T}_{InitialNodes} \subset \mathcal{T}$
- Final nodes with node classes in $\mathcal{T}_{FinalNodes} \subset \mathcal{T}$
- Decision nodes with node classes in $\mathcal{T}_{DecisionNodes} \subset \mathcal{T}$
- Merge nodes with node classes in $\mathcal{T}_{MergeNodes} \subset \mathcal{T}$
- Sub process nodes with node classes in $\mathcal{T}_{SubProcessNodes} \subset \mathcal{T}$
- Action nodes with node classes in $\mathcal{T}_{ActionNodes} \subset \mathcal{T}$

All nodes are in $\mathcal{T}_{Nodes} = \mathcal{T}_{InitialNodes} \cup \mathcal{T}_{FinalNodes} \cup \mathcal{T}_{DecisionNodes} \cup \mathcal{T}_{MergeNodes} \cup \mathcal{T}_{SubProcessNodes} \cup \mathcal{T}_{ActionNodes}$. All nodes except action nodes do not fulfill any other purpose than directing the process flow, so that they mainly implement a marker interface that defines no methods, but allows to distinguish nodes type-safely.

Initial and Final Nodes The marker interface for initial nodes is called `IInitialNode`, for final nodes `IFinalNode`. An example is shown in figure 4.14. In each activity package, exactly one inital and one final node must exist. Thus, the following rules apply:

- $\exists t_{InitialNode} \in \mathcal{T}, name(t) = \texttt{IInitialNode}, \nexists m \in \mathcal{M}, t_{InitialNode} \xrightarrow{contains} m$
- $\forall n \in S_{InitialNodes} : \exists t \in \mathcal{T}_{InitialNodes}, name(t) = name(n), t \xrightarrow{isOf} t_{InitialNode}$
- $\exists t_{FinalNode} \in \mathcal{T}, name(t) = \texttt{IFinalNode}, \nexists m \in \mathcal{M}, t_{FinalNode} \xrightarrow{contains} m$
- $\forall n \in S_{FinalNodes} : \exists t \in \mathcal{T}_{FinalNodes}, name(t) = name(n), t \xrightarrow{isOf} t_{FinalNode}$
- $\forall k \in \mathcal{K}_{Activities} : \exists n_i \in \mathcal{T}_{InitialNodes} \wedge \exists n_f \in \mathcal{T}_{FinalNodes}, k \xrightarrow{contains} n_i, k \xrightarrow{contains} n_f$.

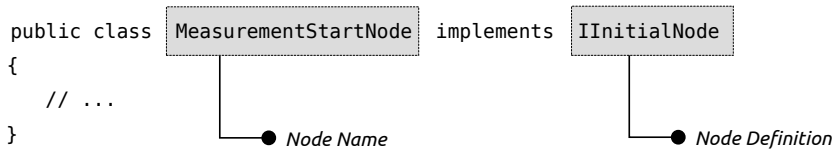


Figure 4.14: A simple process node represented in the design pattern. The marker interface determines the type of the node, in this case an initial node.

Decision and Merge Nodes Similarly, nodes for decisions and, afterwards, merges of different paths are marked with the interfaces `IDecisionNode` and `IMergeNode`. Since decisions are made in edges, these nodes are only placeholders. For each decision node, an appropriate merge node must exist. The following rules apply:

- $\exists t_{DecisionNode} \in \mathcal{T}, name(t) = \texttt{IDecisionNode}, \nexists m \in \mathcal{M}, t_{DecisionNode} \xrightarrow{contains} m$
- $\forall n \in S_{DecisionNodes} : \exists t \in \mathcal{T}_{DecisionNodes}, name(t) = name(n), t \xrightarrow{isOf} t_{DecisionNode}$
- $\exists t_{MergeNode} \in \mathcal{T}, name(t) = \texttt{IMergeNode}, \nexists m \in \mathcal{M}, t_{MergeNode} \xrightarrow{contains} m$
- $\forall n \in S_{MergeNodes} : \exists t \in \mathcal{T}_{MergeNodes}, name(t) = name(n), t \xrightarrow{isOf} t_{MergeNode}$
- $\forall n_d \in \mathcal{T}_{DecisionNodes}, k \in \mathcal{K}, k \xrightarrow{contains} n_d : \exists n_m \in \mathcal{T}_{MergeNodes}, k \xrightarrow{contains} n_m$

Sub Process Nodes Activity links are represented by node classes implementing the marker interface `ISubProcessNode`. To reference the actual sub process, they have an annotation `SubProcess` with a parameter of a class definition implementing `IStartNode`, thus referencing the entry point to the sub process. An example is shown in figure 4.15. The following rules apply:

- $\exists t_{SubProcessNode} \in \mathcal{T},\ name(t) = \texttt{ISubProcessNode},\ \nexists m \in \mathcal{M},\ t_{SubProcessNode} \xrightarrow{contains} m$
- $\forall n \in S_{SubProcessNodes} : \exists t \in \mathcal{T}_{SubProcessNodes}, name(t) = name(n), t \xrightarrow{isOf} t_{SubProcessNode}$
- $\exists a_{SubProcess} \in \mathcal{A}, name(a_{SubProcess}) = \texttt{SubProcess}$
- $\exists p_{Target} \in \mathcal{AP}, p_{Target} \xrightarrow{references} t_{Target} \in \mathcal{T}_{InitialNodes}, a_{SubProcess} \xrightarrow{has} p_{Target}$
- $\forall s \in \mathcal{T}_{SubProcessNodes} : s \xrightarrow{isOf} t_{SubProcessNode} \wedge s \xrightarrow{has} a_{SubProcess}$

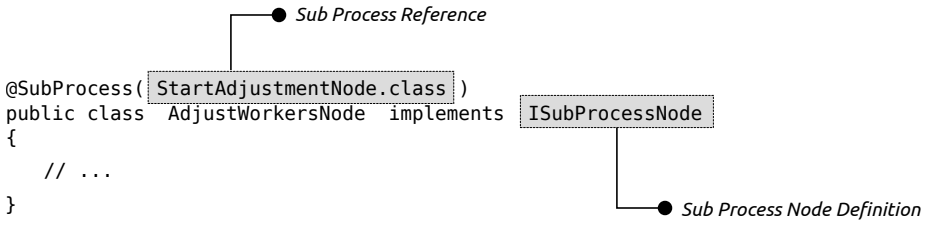


Figure 4.15: A sub process node in the program code pattern. The type of the node is determined by the marker interface. The annotation refers to the initial state of the sub process.

Actions Actions are represented by node classes implementing the interface `IActionNode`. It defines the method `action` whose method body in implementing node classes constitutes the actual action executed at run time. The method takes two parameters: The first is called `actor` and has the type of the actor facade. The second ist called `variables` and has the type of the variables facade:

- $\exists t_{ActionNode} \in \mathcal{T}, name(t_{ActionNode}) = \texttt{IActionNode}$
- $\exists m_{Action} \in \mathcal{M}, name(m_{Action}) = \texttt{action}, t_{ActionNode} \xrightarrow{has} m_{Action}$
- $\exists p_{Actor} \in \mathcal{MP}, name(p_{Actor}) = \texttt{actor}, p_{Actor} \xrightarrow{isOf} t_{Actor}, m_{Action} \xrightarrow{has} p_{Actor}$
- $\exists p_{Variables} \in \mathcal{MP}, name(p_{Variables}) = \texttt{variables}, p_{Variables} \xrightarrow{isOf} t_{Variables}, m_{Action} \xrightarrow{has} p_{Variables}$

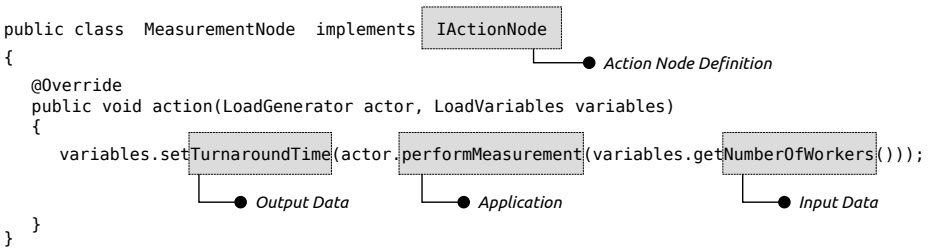


Figure 4.16: An action in the process program code pattern. The action is represented by the call to the actor method. The set method indicates output data, the get method parameters are input data (for a graphical representation of this node cf. figure 4.12(a)).

m_{Action} is implemented in all action node classes. Inside the method, a call to one of the actor methods is made in any case. If this method takes parameters, they must all be calls to get methods of the variables parameter. Thus they represent input data. Optionally, the result of the actor method is passed to a set method of the variables facade. This represents output data. In this case, the actor method invocation is a parameter to the set method. This is a simple way to represent output data, however, it limits the number of outgoing data in each action to 1. While this is certainly not sufficient in all contexts, it serves the purpose to demonstrate the model representation in program code in this thesis. An example can be seen in figure 4.16. The rules are the following:

- $\forall a \in S_{Actions} : \exists t \in \mathcal{T}_{ActionNodes}, name(t) = name(a), t \xrightarrow{isOf} t_{ActionNode}$
- $\forall t \in \mathcal{T}_{ActionNodes} : m_{Action} \xrightarrow{contains} s_{Action} \in \mathcal{S}, s_{Action} \xrightarrow{invokes} m_{Application} \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m_{Application}, name(m_{Application}) = application_a$
- $\forall i \in inputs_a : \exists p_i \in \mathcal{MP}, \exists v_i = \langle d_i, t_i \rangle \in S_{Variables}, name(d_i) = i, m_{Application} \xrightarrow{has} p_i, get_{v_i} \xrightarrow{constitutes} p_i$
- $output_a \neq \epsilon \Rightarrow \exists set_{v_a}, \exists v_a = \langle d_a, t_a \rangle \in S_{Variables}, name(d_a) = output_a, s_{Action} \xrightarrow{constitutes} p \in \mathcal{MP}, set_{v_a} \xrightarrow{has} p, m_{Action} \xrightarrow{invokes} set_{v_a}.$

4.2.2.4 Edges

Usually, each node has one incoming edge and one outgoing edge. Initial nodes have no incoming edge, final nodes no outgoing edge. Decision nodes have more than one outgoing edge, merge nodes have more than one incoming edge. In the program code, edges are represented by methods in the node class of their source. The method has an annotation `OutgoingEdge` with an attribute referring to the target node class, returns a boolean value, and optionally contains a guard expression iff the edge emanates from a decision node. Since guards can evaluate variables, a parameter of the variables facade type is given. The edge methods are thus defined as follows:

- $\exists a_{Edge} \in \mathcal{A}, name(a_{Edge}) =$ `OutgoingEdge`

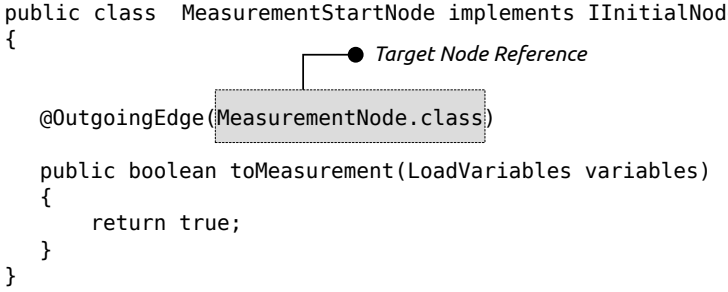


Figure 4.17: An edge without guard.

- $\exists p_{EdgeTarget} \in \mathcal{AP},\ p_{EdgeTarget} \xrightarrow{references} t_{Target} \in \mathcal{T}_{Nodes}$, $a_{Edge} \xrightarrow{has} p_{EdgeTarget}$
- $\forall e \in S_{Edges} : \exists m_e \in \mathcal{M}_{Edges}, \mathcal{M}_{Edges} \subset \mathcal{M}, m_e \xrightarrow{has} a_{Edge}, m_e \xrightarrow{returns} t_{Boolean}, \exists t \in \mathcal{T}_{Nodes}, t \xrightarrow{contains} m_e$
- $\exists p_{Variables} \in \mathcal{MP}, name(p_{Variables}) = \texttt{variables}, p_{Variables} \xrightarrow{isOf} t_{Variables}$, $m_e \xrightarrow{has} p_{Variables}$
- $\forall m_e \in \mathcal{M}_{Edges}, t \in \mathcal{T}_{DecisionNodes}, t \xrightarrow{has} m_e : \exists s \in \mathcal{S}, m_e \xrightarrow{has} s$
- $\forall m_e \in \mathcal{M}_{Edges}, t \in (\mathcal{T}_{Nodes} \setminus \mathcal{T}_{DecisionNodes}), t \xrightarrow{has} m_e : \nexists s \in \mathcal{S}, m_e \xrightarrow{has} s$

In the Java program code pattern, the empty guard is denoted by the statement `return true;` which fulfills the method signature requirements and does not consider any variable values. The annotation parameter has no explicit name as is possible with annotations that have only one parameter. An example for such a simple edge is shown in figure 4.17.

If the node is a decision node, its edge methods must contain guards. In guards, we distinguish between simple and composite expressions:

Definition 13 $\sigma = \langle \lambda, \omega, \rho \rangle \in \mathcal{S}$ is a simple guard expression with

- a left hand side $\lambda \in \mathcal{S}, \lambda \xrightarrow{invokes} get_v, v \in S_{Variables}$,
- an operator $\omega \in \{==, <, <=, >, >=, !=\}$,
- a right hand side $(\rho \in \mathcal{S}, \rho \xrightarrow{invokes} get_{v'}, v' \in S_{Variables}, v \neq v')XOR(\kappa)$ with κ a literal from the data type of v. □

Definition 14 $\gamma = \langle \sigma_0 \star \ldots \star \sigma_m \star \gamma_{m+1} \star \ldots \star \gamma_n \rangle \in \mathcal{S}$ is a complex guard expression with

- a set of operands $(\sigma_0 \ldots \sigma_m \cup \gamma_{m+1} \ldots \gamma_n), n \geq 2$, with $\sigma_0 \ldots \sigma_m$ simple expressions and $\gamma_{m+1} \ldots \gamma_n$ complex expressions,

```
public class AdjustmentDecisionNode implements IDecisionNode
{
   @OutgoingEdge(IncreaseWorkersNode.class)
   public boolean increase(LoadVariables variables)
   {
       return variables.getTurnaroundTime() < variables.getLowerLimit();
   }
                              Variable      Operator      Variable

   @OutgoingEdge(DecreaseWorkersNode.class)
   public boolean decrease(LoadVariables variables)
   {
       return variables.getTurnaroundTime() > variables.getUpperLimit();
   }
}                             Variable      Operator      Variable
```

Figure 4.18: A guard with simple expressions comparing variables.

- a connecting operator $\star \in \{\&\&, ||\}$. □

For the program code pattern, the following rules apply:

- $\forall m_e \in \mathcal{M}, t \in \mathcal{T}_{DecisionNodes}, t \xrightarrow{contains} m_e : m_e \xrightarrow{has} \sigma \text{ XOR } m_e \xrightarrow{has} \gamma$

These rules are sufficient to define guards accessing the variables. An example can be seen in figure 4.18.

4.2.3 Execution Semantics

Similar to state machines, embedded process models need an execution framework to produce sequences of actions matching the execution semantics of the underlying model at run time. Thus we will now outline the execution semantics of the process model as introduced above and then introduce an appropriate execution algorithm.

4.2.3.1 Execution Algorithm

The essential purpose of the process model at run time as introduced here is to invoke application logic during actions. The execution semantics are thus, similar to state machines, considered in terms of sequences of actions, which are determined by the existence of edges and the current variable assignment influencing the evaluation of guards:

Definition 15 $n_1 \in S_{Nodes}$ and $n_2 \in S_{Nodes}$ are *currently connected* iff

- an edge $e \in S_{Edges}$ exists that connects n_1 and n_2,
- e has no guard *or* the guard is enabled with the current variable assignment. □

Based on this, transitive connections can be considered:

Definition 16 A *path* exists between $n_1 \in S_{Nodes}$ and $n_2 \in S_{Nodes}$ iff

- either n_1 and n_2 are currently connected

- or $\exists n'_1 \in S_{Nodes}$ so that n'_1 and n_2 are currently connected *and* a path exists between n_1 and n'_1. □

Thus we can define the execution semantics as follows:

Definition 17 α is a sequence of actions $\alpha_0, \ldots, \alpha_n$ produced by a given process model iff

- a path exists between $s_i \in S_{InitialNodes}$ and $\alpha_0 \in S_{Actions}$,
- a path exists between $\alpha_x \in S_{Actions}$ and $\alpha_{x+1} \in S_{Actions}$ with $0 < x < n$,
- a path exists between $\alpha_n \in S_{Actions}$ and $s_f \in S_{FinalNodes}$. □

This leads to the following algorithm for the execution of a process model:

```
1: s := s_i
2: while ¬s ∈ S_FinalNodes do
3:    if s ∈ S_DecisionNodes then
4:       {Select edge to follow}
5:       e_selected := null
6:       for all e in s.edges do
7:          if e_selected = null ∧ invoke(e.guard) = true then
8:             e_selected := e
9:          end if
10:      end for
11:      if e_selected = null then
12:         print Error:  No edge selected!
13:         return
14:      end if
15:      s := e_selected.target
16:   else
17:      if s ∈ S_ActionNodes then
18:         {Invoke action}
19:         invoke(s.action)
20:      else if s ∈ S_SubProcessNodes then
21:         {Invoke sub process}
22:         execute(s.target)
23:      end if
24:      {Only one edge available, select target}
25:      s := s.edge.target
26:   end if
27: end while
```

4.2.3.2 Use of Program Code Fragments

This execution algorithm and the related framework for embedded process models consider various elements of the program code pattern.

Most important are the marker interfaces of node classes which allow to interpret the graph of nodes and edges that constitute an activity. The entry point is marked with the interface `IInitialNode`. In the following, the node classes are distinguished by their type: `IActionNode` for action nodes,

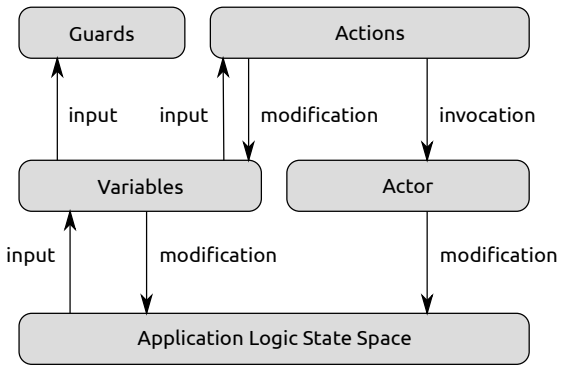


Figure 4.19: The role of interfaces in embedded process models.

`ISubProcessNode` for sub process nodes, `IDecisionNode` and `IMergeNode` for decision and merge nodes, and `IFinalNode` for final nodes. The graph of nodes and edges is thus *interpreted* by the execution framework considering the different node types, the annotation `@SubProcess` with its parameter `target` referring to a node of type `IInitialNode`, and edge methods with their meta data annotation `@OutgoingEdge` and its parameter `target`.

While processing the graph, the execution framework makes *invocations* in two situations: First, when edges are selected in decision nodes, their guards are evaluated by invoking the edge method. Second, the method `action` defined by the interface `IActionNode` of each action node is invoked when the node is active. Similar to the execution of state machines, the locations of these executable fragments are determined by the static structures that are interpreted and are considered black boxes by the execution framework.

4.2.4 Interface Definitions

Embedded process models fulfill the purpose to invoke application logic by means of interfaces as introduced in section 3.1.5. The following interfaces exist for the models introduced above:

- The *actor* facade is a collection of labels. It is action-oriented and data-oriented since the methods provide entry points to the application logic, can take parameters, and can return values. The actor is accessed in action nodes.

- The *variables* facade is a collection of labels which is data-oriented. Variable get methods read variable values from the state space of the application. The set methods take variable values that are the result of actions and pass them to the application logic. Variables are by this means not managed inside the model, but instead in the application logic. Thus, values read by the model may be aggregated. Variables are accessed in action and guard methods.

An embedded process model is by this means connected to the surrounding program as illustrated in figure 4.19:

In decision nodes, guards access variables which are extracted as input from the state space of the surrounding applications. The state space must thus fulfill the requirements of the guard of at least one edge in the current decision node. However, this cannot be controlled from inside the embedded model, and irregularities cannot be detected with a validation of the process model. The application logic is thus responsible for ensuring that the variables are valid in the context of the process model. At development time, this could be supported with static analysis and model checking of program code (cf. section 3.2.2) since the requirements to the state space are for each decision node precisely given by the guards' expressions. At run time, inconsistencies will be detected if the process takes other paths as expected or runs into a deadlock; detection and analysis of such errors can be supported by monitoring tools (cf. section 3.2.4).

During actions, the state space of the application can be modified with actor methods as well as set methods of variables. However, both kinds of methods are black boxes to the model, so that changes are not comprehensible. Only the fact *that* values are passed is defined in the model, but not their content or value range, since they are the result of action methods.

In summary, the state space is modeled only by the requirements of guards. While this is implicit, it is certainly a starting point for static analysis of the program code since method invocations in guards allow for slicing of the application's program code (cf. section 3.1.5.2).

4.2.5 Transformations

Since the embedded process model is to be considered at different abstraction levels, transformations will be used to provide abstract views on the program code. These transformations follow the rules defined in section 3.1.6. Since the model is entirely based on JWT, the transformation to the JWT format is considered the default transformation. The single elements of the pattern are transformed to JWT (and vice versa) based on the derivation of the pattern from the model as introduced above:

Applications As defined above, the mapping requires that all applications in a JWT model used for the embedded process model have the same implementing class. This class is an interface in the program code and can be identified since its fully-qualified name is given in the JWT model. Single applications have a name that equals the name of the related actor method.

Data Single data entries consist of a name and a data type which is defined by a name again. Both are directly mapped to the signature of variable methods. However, these methods are contained in a facade interface which is not part of the model. Thus the transformation must have a parameter denoting the location and class name of the interface.

Activities Activities are extracted from packages. For this reason, the transformation must know packages with activities inside; this information will have to be provided by tools that control the transformation. The simple name of the package is considered the activity's name. When program

code is generated from a JWT model, the single activities are mapped to packages again.

Nodes Nodes and their names can be identified in the source code by means of their marker interfaces. The names correspond to the class names. However, the transformation must consider that the program code pattern as introduced above does not support all node types defined by JWT.

Sub Processes Nodes with sub processes contain the reference information in the annotation `@SubProcess`. While the JWT model links to the name of the activity, the annotation in the code links to an initial node from which the package and thus the activity name can be inferred. This difference can be resolved unambiguously and is thus supported by the transformation.

Edges Edges are determined from edge methods in the program code pattern. When program code is generated from the model, the method names must be provided by a tool based on heuristics since edges are not named in JWT.

Guards Guards can be inferred directly from the source code. For this purpose, expressions are read and the variable method invocations are interpreted as data entries.

Actions Contents of action methods in action nodes can be transformed directly: The name of the method invoked in the actor class is the application name; its input parameters being calls to variable get methods are interpreted as input data labels; the result of the actor method being passed to a variable set method is considered an output data label. The sole limitation is that only one output data label can exist for each action, which is different in JWT (cf. section 4.2.2.3). Thus, JWT models used for embedded process models must conform to this limitation, which must be validated by the transformation.

4.3 Comparison of the Embedded Models

The two embedded model classes introduced in sections 4.1 and 4.2 are similar to a certain degree. However, considered together, state machines and process models cover many aspects of behavioral models. We will thus draw some general conclusions now.

4.3.1 Representation of Model Elements

The basic structure of behavioral models is usually some kind of graph denoting paths a program can take which are influenced by decisions depending on variable values. This is not only true for state machines and processes, but also for models like UML activity diagrams or state charts [OMG, 2010]. Common to both embedded models is the use of program code fragments to represent this structure:

Classes as the basic static structures in the program code can represent named nodes in such a graph. Edges in the graph can be represented by references between classes given in meta data annotations. Decisions that consist of expressions are realized in method contents. In addition, method contents can contain calls to interface methods, thus initiating arbitrary application logic. Although the specific location of such elements varies in both embedded models, the principles are the same. Considering the two embedded models, we are thus confident that program code patterns can be created with such building blocks for these kinds of behavioral models.

4.3.2 State Space

Embedded models are connected to other program code so that the state space may not be entirely under control of the model. This is different than in specific modeling environments like UPPAAL which control all variables except when random values are generated, but in this case even the random number generation is under control of the environment. In contrast, variables in both embedded models are managed by the application logic and are not directly visible from the model or from modeling tools. Considering the models introduced above, control over the state space by the model can be categorized as follows:

Complete Control The modeling environment manages the state space completely. Changes can be made only from inside the model. This is true e.g. for UPPAAL.

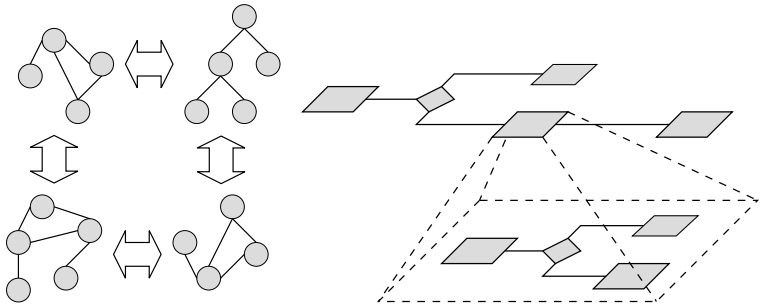
Emulated Updates The model defines changes to the state space. This allows to extract a complete model into a modeling environment considering the state space, e.g. embedded state machines to UPPAAL. It also enables model checking at an abstract level. However, at run time the variables are not managed by the execution framework, but instead influenced by the application logic, so that differences can occur.

Partial Updates The state space is mostly only read by the model. Updates may be defined in the model, but are not complete. Such models cannot be model-checked since the state space is never defined completely, not even at development time. This is the case for process models.

Thus, depending on the purpose of the model, the application logic can have different degrees of influence on the state space. While partial updates certainly provide higher flexibility, a more restrictive control over the state space allows for thorough verification and model checking (cf. section 3.1.5.2). When a program code pattern for an embedded model is defined, both objectives must be balanced.

4.3.3 Scalability

When considering abstract models, it is important how they can scale in order to represent larger or growing system descriptions. For embedded models, the question arises whether and how scaling models can be represented in the



(a) Scaling by communication (b) Scaling by hierarchy

Figure 4.20: Scaling mechanisms of embedded models introduced so far. Scaling by communication means that independent models are coupled among each other, in contrast to a hierarchy that results in a tree of models.

program code. With the models introduced above, two mechanisms for scaling have been introduced as sketched in figure 4.20:

Scaling by communication is used for state machines: Different state machines can be connected by means of channels. This means that the single models are of equal importance. In contrast, *scaling by hierarchy* defines a top-level model that is the entry point to the set of models and controls the overall execution. This is the case with process models. Specific fragments of such models contain references to other models residing at a lower level in the hierarchy.

In embedded models, it is possible that the scaling has no influence on the structure of the program code pattern. This is the case with the embedded state machine model where the model fragments reside in one or more packages which are not considered. Transformation tools can detect their interconnections and thus separate the single models from each other. In cases where a more explicit separation of single models is desirable, packages can be instrumented for scaling. In embedded process models, the package names identify different models and separate the related fragments. The two embedded models introduced above thus inidicate that scaling can be represented in the program code in structured and systematic ways. Scalability at the architectural level is possible in addition as discussed in section 3.3.

4.4 Tools

It is possible to work with embedded models at the level of the program code only. Similar to design patterns, the structures introduced above are easier to comprehend than “arbitrary” program code once the rules are known by the developers. However, model-based development is most beneficial when tools are used that realize abstract views on the software to develop for design, verification, model checking, etc. Thus we will now introduce tools for embedded state machines and process models. Based on the categorization given in section 3.2, the following tools will be presented in this section:

	State Machines	Process Models
Design & Implementation	Tool available (section 4.4.1)	Tool available (section 4.4.2)
Verification	Tool available (section 4.4.3)	
Execution	Tool available (section 4.4.4)	Tool available (section 4.4.5)
Monitoring	Tool available (section 4.4.6)	
Design Recovery	Tool available (section 4.4.7)	

For state machines, a tool suite exists that supports design, verification, execution, and monitoring [Balz et al., 2010] as explained in sections 4.4.1, 4.4.3.3, 4.4.4, and 4.4.6.5. The tool support for process models is only partially available since no tools for verification and monitoring exist for the model specifications used in this thesis.

4.4.1 Design Tool for State Machines

A visual design tool for embedded state machines that uses the default transformation (cf. section 4.1.5.1) has been developed in a master's thesis by Malte Goddemeier. It is implemented as a plugin for the Eclipse IDE [Eclipse] and uses its Java Development Tools API [The Eclipse Foundation, 2008] to access the source code [Goddemeier, 2009]. The user interface can be seen in figure 4.21 showing a non-trivial example.

The tool relies on an internal transformation, i.e., the program code is directly interpreted with the default transformation. To use the editor, a user selects an existing Java package. If the package already contains an embedded state machine, it is interpreted by the editor. If not, the package will be used to store models created in the editor. States and transitions can be created, modified, and removed in the graphical view. This implies creation, modification, or deletion of classes and methods in the source code. The names of states and transitions are defined in the graphical view and influence the names of the classes and methods. These structures can be mapped unambiguously except layout information, which is not included in the program code pattern. To apply a layout, the editor must rely on heuristics; for the future it is planned to save layout information externally, which would be acceptable because it affects only meta information and not the actual model specifications.

Guards and updates for transitions are edited as expressions containing variable names, operators, and literal values. They are mapped to expressions in the program code containing calls to the variables interface, Java operators, and literal values. This is realized with the JDT's Document Object Model (DOM) providing fine-grained access to the syntax tree of Java classes.

Action labels are assigned to transitions with a selection from available action labels and the definition of an invocation order by the user. This is realized by the tool by considering the method names of the actor class, which is either inferred from existing action labels or given by the user. Available channels can also be assigned to transitions for sending or receiving.

To achieve this functionality, the editor relies on technologies provided by

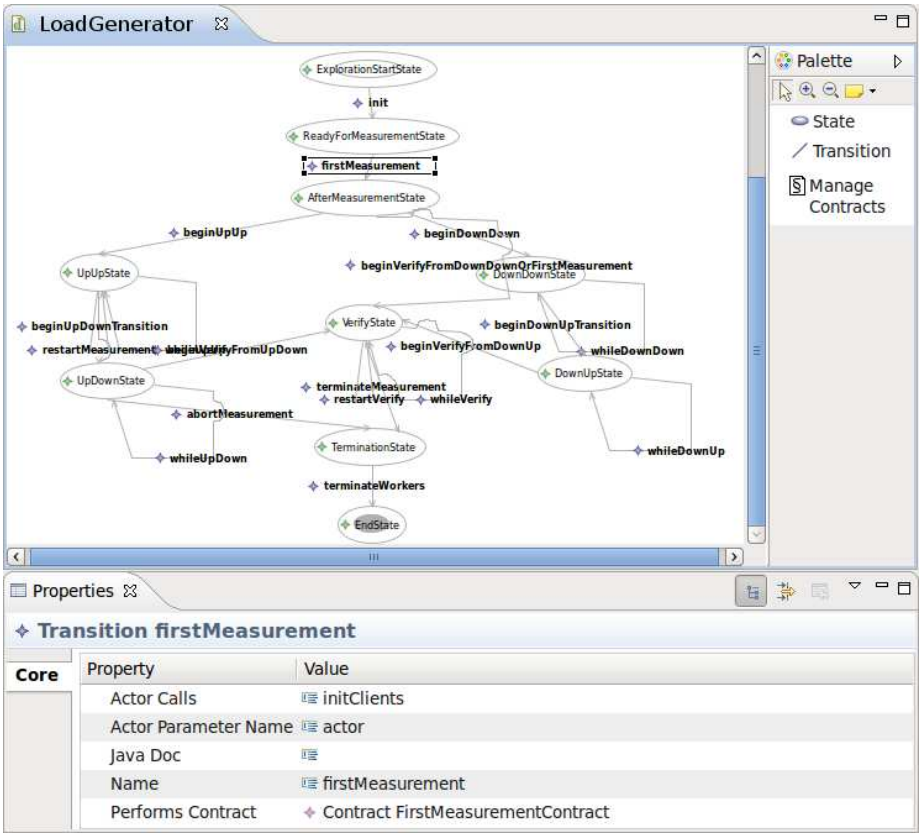


Figure 4.21: The design tool for state machines in the Eclipse IDE using an internal transformation. Nodes and transitions are designed graphically and can be supplied with additional information regarding names, guards, updates, and action labels.

Eclipse: The source code is read and manipulated with JDT. The abstract model is specified in EMF and used by graphical editing components provided by Eclipse. The EMF model is kept in memory during editing. When the model is saved by the user, the according source code is created instantly. When source code is modified by the user outside the design tool, the editor is notified by the Eclipse platform and adapts the graphical model representation accordingly.

4.4.2 Design Tool for Process Models

JWT already provides a visual design tool for process models as a plugin for the Eclipse IDE. It must be extended to be used with embedded process models. This is possible with the transformation between the program code pattern and JWT's file format. In contrast to the state machine editor, we use an external transformation here.

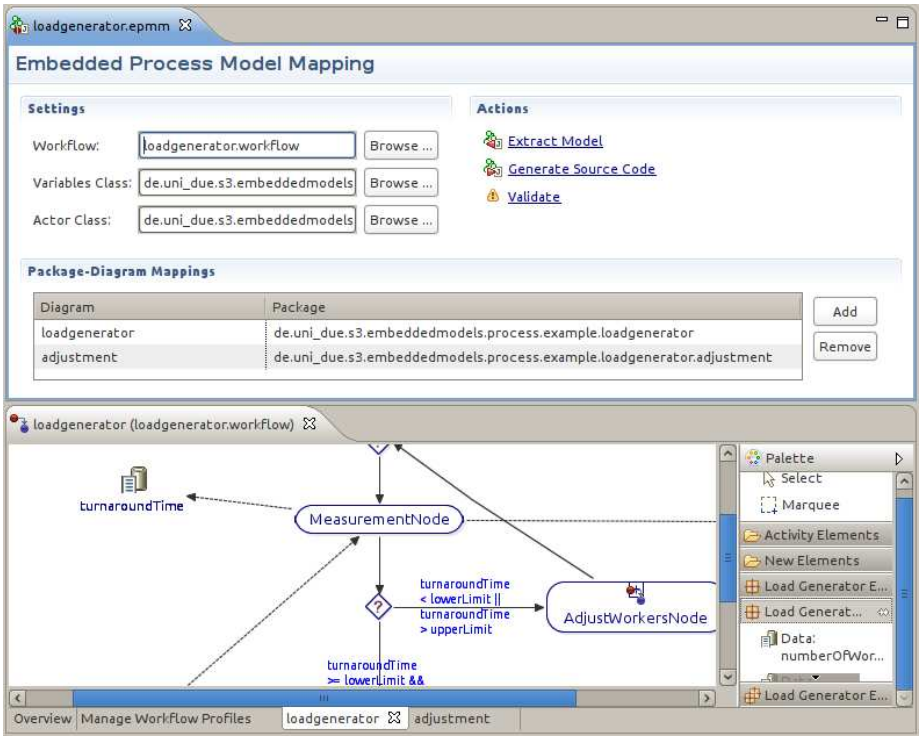


Figure 4.22: The design approach for embedded process models. The mapping shown at the top associates Java packages with diagrams in the notation of the JWT workflow editor for eclipse. The according file is then edited in JWT's visual editor shown at the bottom.

4.4.2.1 Approach

The transformation tool is also a plugin for Eclipse. It consists of an editor that allows the user to specify the mapping information as shown in figure 4.22. The foundation for the transformation is the selection of a "workflow" file containing the JWT model. The workflow file must already exist in any case, even if it contains no activites; the transformation tool will for model extraction as well as code generation only work with the existing file. The reason ist that certain meta data is only contained in the model, like author, version, and description. The user must then specify the facades as fully-qualified class names since they cannot be inferred from the model. The mapping between activities and packages must also be given by the user; each package-activity mapping consists of the activity name in the model and a fully-qualified package name. The definition of the embedded model is thus slightly enhanced by the tool so that activity names and package names must not be equal, but only serve as identifiers during the mapping, which allows for higher flexibility for the naming of activities in the model.

Technically, the transformation tool relies on two APIs: The code is accessed and modified with JDT; the model is accessed with Eclipse's Ecore tools since its notation is based on EMF. The JWT editor contains an appropriate API defin-

ing the EMF entities as Java interfaces, which is used for this purpose.

Three actions can be initiated in the mapping editor:

Validate The mapping information is validated: Existence of the workflow file; existence of the activities in the model; existence of actor and variables class.

Generate Source Code The model file is read and program code is create accordingly. If program code already exists in the packages of interest, the model information is merged.

Extract Model The modeling information is read from the source code and merged into the model file.

4.4.2.2 Code Generation

When code is generated from the model, the facade interfaces are adapted first: Get and set methods are created for each data item, and actor methods are created for each application. Information from the model can be merged into the interfaces unambiguously since elements can be identified by their names.

Afterwards, the program code for activities is generated or adapted. Nodes can be identified by their names and thus be added, kept, or removed, so that only necessary changes are made. This is not possible for edges, which cannot be identified since they have no names in JWT. In initial nodes, action nodes, merge nodes, and sub process nodes, exactly one outgoing edge exists, so that the single method can be adapted if necessary. In decision nodes, the transformation algorithm tries to identify edge methods by their target. When methods with a matching target are available, they are adapted. Otherwise, new methods are created whose names are guessed by the tool. Superfluous methods are removed. Inside methods, guard expressions are not merged, but simply replaced.

An example for the code generation can be seen in listing 4.2 showing code from a class in the transformation tool representing a data entry. Lines 1-13 constitute a constructor that reads modeling information from the JWT Ecore interface `Data` providing the name and data type. From the data name, the names of get and set method are inferred. Lines 17-28 show the code generation: The tool checks whether the methods already exist by identifying them with their name. If not, they are created in the object named `variablesType` which is of type `IType` as defined by JDT. The method itself is created by specifying its signature.

```
1  public MappedData(Data data, Map<String, MappedDataType> dataTypes) throws MappingException
2  {
3    this.name = data.getName();
4
5    MappedDataType type = dataTypes.get(data.getDataType().getName());
6    if (type == null)
7      throw new MappingException("Unknown␣data␣type␣'" + data.getDataType().getName() + "'␣for␣
           data␣'" + data.getName() + "'");
8    this.dataType = type;
9
10   // Method names
11   this.getMethodName = "get" + name.substring(0, 1).toUpperCase() + name.substring(1);
12   this.setMethodName = "set" + name.substring(0, 1).toUpperCase() + name.substring(1);
13 }
14
```

```
15 // ...
16
17 public void generateCode(IType variablesType, Map<String, IMethod> variableMethods) throws
       JavaModelException
18 {
19   // Create get method
20   IMethod getMethod = variableMethods.remove(getMethodName);
21   if (getMethod == null)
22     variablesType.createMethod(dataType.getName() + ' ' + getMethodName + "();", null, false,
           null);
23
24   // Create set method
25   IMethod setMethod = variableMethods.remove(setMethodName);
26   if (setMethod == null)
27     variablesType.createMethod("void " + setMethodName + '(' + dataType.getName() + ' ' +
         camelCodeName + ");", null, false, null);
28 }
```

Listing 4.2: Code generation out of JWT process models.

4.4.2.3 Model Extraction

The model extraction has a similar course of action: First, the method signatures in the variables facade are interpreted and the data names and data types are determined. The data types in the model are created or removed accordingly so that the model information from the code is merged. The same applies to action methods that become applications in the model.

Afterwards, the activities in the JWT model are generated or adapted. For this purpose, the JWT model API is used. Nodes are identified by their names and thus added, kept, or removed, so that only necessary changes are made. Edges are transformed directly since the method names are simply ignored. Inside edges, guard expressions are not merged, but simply replaced.

An example can be seen in listing 4.3 showing a method of a class in the transformation tool representing an action. In the constructor of the class, which is not visible here, a list of inputs, the application name, and an optional output have been extracted from the code. They are merged into the model in the method shown here. The variable `action` of the JWT model type `Action` is the target of the transformation: Its lists of inputs and outputs are cleared and the appropriate data items are set, which are in this case identified by their name and retrieved from maps in the transformation tool since they have been identified beforehand. The application name of the action is also overwritten.

4.4.3 Verification Tools for State Machines

Verification of embedded models can be performed at different abstraction levels: The *code structure* can be verified to conform to the rules of the program code pattern; the *code semantics* can be verified to conform to the semantics of the embedded model interfaces; the *model* itself can be verified when it is extracted from the code.

4.4.3.1 Code Structure Verification

The rules defined for the program code pattern are formalized and thus appropriate to be re-used in tools that verify the conformance of program code to the pattern definition. Such verification can be included in design tools, as is the

```
1  public void extractModel(Action action, Map<String, Application> applications, Map<String,
       Data> data)
2  {
3    // Output
4    action.getOutputs().clear();
5    if (this.output != null) {
6      Data output = data.get(this.output.getName());
7      action.getOutputs().add(output);
8    }
9
10   // Application
11   action.setExecutedBy(applications.get(this.application.getName()));
12
13   // Inputs
14   action.getInputs().clear();
15   for (MappedData d : this.inputs) {
16     Data input = data.get(d.getName());
17     action.getInputs().add(input);
18   }
19 }
```

Listing 4.3: JWT model extraction from program code.

case with the editor introduced above, but also be implemented with external code checking tools. Such tools often allow to specify rules in a descriptive way. We will now introduce an exemplary rule specification for the static code analyzer PMD [Copeland, 2005].

As a first step, we must select rules from the pattern definition that the code must conform to. As an example, consider the rules for actions as explained in section 4.1.2.2:

- $\exists t_{Actor} \in \mathcal{T}$
- $\forall a \in L_{Actions} : \exists m_a \in \mathcal{M}, name(m_a) = a, t_{Actor} \xrightarrow{contains} m_a, \nexists p \in \mathcal{MP}, m_a \xrightarrow{has} p$
- $\forall m \in \mathcal{M}_{Transition} : \exists p_{Actor} \in \mathcal{MP}, p_{Actor} \xrightarrow{isOf} t_{Actor}, m \xrightarrow{has} p_{Actor}$
- $\forall m \in \mathcal{M}_{Transition} : \exists L_{Actions_m} \subseteq L_{Actions}, \forall a \in L_{Actions_m} : \exists s \in \mathcal{S}, m \xrightarrow{contains} s, s \xrightarrow{invokes} m_a \in \mathcal{M}, t_{Actor} \xrightarrow{has} m_a$
- $\forall m \in \mathcal{M}_{Transition} : \nexists s \in \mathcal{S}, m \xrightarrow{contains} s, \neg s \xrightarrow{invokes} m \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m$

These rules must be adapted to the level of the program code structures (i.e., not their semantics). Thus, references to the formal model must be removed. In this case, t_{Actor} is not known to the tool beforehand but is inferred from the context: $\exists s \in \mathcal{S}, m_t \in \mathcal{M}_{Transition}, m_t \xrightarrow{contains} s, s \xrightarrow{invokes} m_a \in \mathcal{M} \Rightarrow \exists t_{Actor} \in \mathcal{T}, t_{Actor} \xrightarrow{contains} m_a$. We thus assume that $\exists a \in L_{Actions}, \exists m_a \in \mathcal{M}, name(m_a) = a, t_{Actor} \xrightarrow{contains} m_a \Rightarrow \forall m_a \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m_a : \nexists p \in \mathcal{MP}, m_a \xrightarrow{has} p$ and $\exists L_{Actions_m} \Rightarrow \exists m \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m_a, m_t \xrightarrow{invokes} m_a$. This allows to derive the following rules for the source code level:

1. $\forall m \in \mathcal{M}_{Transition} : \exists p_{Actor} \in \mathcal{MP}, p_{Actor} \xrightarrow{isOf} t_{Actor} \in \mathcal{T}, m \xrightarrow{has} p_{Actor}$

2. $\forall s \in \mathcal{S}, m \in \mathcal{M}_{Transition}, m \xrightarrow{contains} s : s \xrightarrow{invokes} m_a \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m_a$

3. $\forall s \in \mathcal{S}, m \in \mathcal{M}_{Transition}, m \xrightarrow{contains} s, s \xrightarrow{invokes} m_a \in \mathcal{M} : \nexists p \in \mathcal{MP}, m_a \xrightarrow{has} p$

Since PMD verifies code in terms of so-called violations, the expressions must be inversed in order to specify invalid code to be detected:

1. $\exists m \in \mathcal{M}_{Transition} : \nexists p_{Actor} \in \mathcal{MP}, p_{Actor} \xrightarrow{isOf} t_{Actor} \in \mathcal{T}, m \xrightarrow{has} p_{Actor}$

2. $\exists m \in \mathcal{M}_{Transition} : \exists s \in \mathcal{S}, m \xrightarrow{contains} s, \neg s \xrightarrow{invokes} m_a \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m_a$

3. $\exists m \in \mathcal{M}_{Transition} : \exists s \in \mathcal{S}, m \xrightarrow{contains} s, s \xrightarrow{invokes} m_a \in \mathcal{M}, t_{Actor} \xrightarrow{contains} m_a, m_a \xrightarrow{has} p \in \mathcal{MP}$

Specifications 1 and 2 are represented in PMD rules as shown in listing 4.4 in XPath syntax [Copeland, 2005]: In lines 1-3 the rule defines that methods with the `Transition` annotation are invalid if they do not have exactly one parameter (function `count` for the element type `FormalParameter`). Subsequently, the rule in lines 5-7 defines that statements in transition methods must not assign any variables so that the method parameter cannot be changed. Based on this, the rule in lines 9-12 defines that method invocations cannot occur on any other variables than the method parameter; this is indicated by a `StatementExpression` whose `PrimaryPrefix` denotes the method invocation. The prefix is therefore not allowed to begin with anything other than the parameter's name and a dot.

```
1  //ClassOrInterfaceBodyDeclaration
2    [Annotation//Name[@Image='Transition']]
3    [count(MethodDeclaration//FormalParameter) != 1]
4
5  //ClassOrInterfaceBodyDeclaration
6    [Annotation//Name[@Image='Transition']]
7    [MethodDeclaration//AssignmentOperator]
8
9  //ClassOrInterfaceBodyDeclaration
10 [Annotation//Name[@Image='Transition']]
11 [MethodDeclaration//StatementExpression/PrimaryExpression/PrimaryPrefix/Name
12   [not(starts-with(@Image, concat(ancestor::MethodDeclaration//VariableDeclaratorId/@Image,
         '.')))]]
```

Listing 4.4: PMD rules for static validation of the rules for the actor parameter and its usage inside transition methods.

Specification 3 detects parameters of method invocations in transition methods, which are not allowed by the pattern. In PMD, method invocation parameters are contained in a node of type `ArgumentList`. The rule in listing 4.5 detects such declarations in transition methods.

```
1  //ClassOrInterfaceBodyDeclaration
2    [Annotation//Name[@Image='Transition']]
3    [MethodDeclaration//ArgumentList]
```

Listing 4.5: PMD rule detecting invalid parameters of action labels.

This approach can be applied to other program code pattern rules and also to other static code checking tools. By this means the syntactic requirements to program code following the pattern can be verified.

4.4.3.2 Code Semantics Verification

Since embedded models are interconnected with other program code and can thus be influenced by the state space of the surrounding application (cf. section 4.3.2), verification is desirable at the program code level with respect to the semantics of interfaces and data exchanged there. Such verification is based on the consideration of program code slices defined by the interfaces of embedded models (cf. 3.1.5.2). The assertions for the program code can be used by tools for static and dynamic analysis and model checking (cf. section 3.2.2).

In this context, the state machine model offers the following assertions:

- *Expected changes for updates*: Program code slices of actions must update the state space as specified in the update of the transition. This assertion must always be fulfilled.
- *Expected changes for guards*: In states, a limited set of guard expressions exists. In addition to expected changes for updates, actions must modify the state space so that at least one guard is enabled in the next state. This assertion must always be fulfilled.
- *Reproducible reading*: Since variables in guards are read-only, it may be desirable that they are not changed during read access. If this assertion makes sense depends on the current model. For the state machine example introduced in section 4.1.1.5, this is true for both variables `calories` and `growth` which are accessed for reading repeatedly.
- *Exclusive writing*: State machine models describe changes to the state space in updates. This may require exclusive writing when the variables are not influenced by other application logic. If this assertion makes sense depends on the current model. For the state machine example introduced in section 4.1.1.5, this is true for both variables `calories` and `growth` whose modifications are described by the model completely.

Since impartial criteria for the verification of slices exist, tool support for such verification is possible. However, this has not been implemented so far and must be considered future work. Tools must distinguish between assertions that can be derived from the model in every case, and those that depend on the current model instance and must thus be supplied by the user. E.g., exclusive writing may not be desirable if the state space is modified by complex application logic that cannot be expressed in terms of a state machine.

4.4.3.3 Model Verification

Model verification relies on the existence of a model and a specification the model is checked against. While the first is available in embedded models, the latter must be provided externally. For state machines, this is given with the tool UPPAAL, for which a transformation from the embedded models exists

(cf. section 4.1.5.2). A transformation tool based on graph transformations has been developed by Michael Striewe [Striewe, 2008; Striewe et al., 2010b] that extracts model information from the code to the UPPAAL notation.

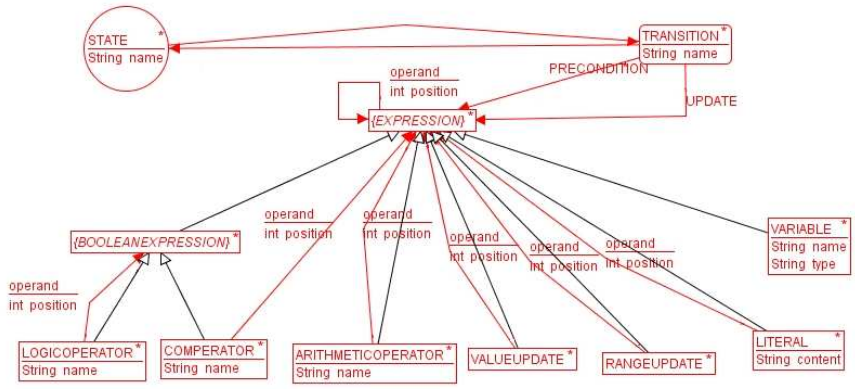


Figure 4.23: AGG type graph for transitions [Striewe, 2008].

The verification tool builds upon low-level transformation of Java code to the data format of the graph transformation tool AGG [AGG], during which the Java DOM is converted to a graph by the tool JAVA2GGX, e.g. with the addition of edges between elements [Striewe et al., 2010b]. Afterwards, triple graph grammars [Schürr, 1994] are used for the transformation: A so-called *correspondence graph* is defined that is independent from specific notations and described by a type graph shown in figure 4.23. The actual transformation between program code and UPPAAL is thus performed with the correspondence graph synchronizing the graphs resulting from specific notations. This is visualized with a small example in figure 4.24: At the left hand the graph representation of the program code pattern can be seen, in the middle the correspondence graph with the extracted information. At the right hand the resulting graph for the UPPAAL notation is shown. Direct mapping between elements is marked with yellow lines. All other elements are not part of the actual model, but necessary to complete the respective syntax [Striewe, 2008].

As a result of this mapping, the extracted model is fully decoupled from program code and thus available at an abstract level. It can be verified with respect to CTL formulas as well as to general automata properties like deadlocks as described in section 4.1.5.2. The information from the program code can therefore be validated with respect to abstract specifications. But, the significance must be considered critically since the abstraction entails that only a domain-specific view on the software is verified. For example, arbitrary program code in the state machine actor may change the application state in a different way than specified in UPPAAL updates. Even worse, deadlocks may occur at run time because of the channel's dependency on application logic, which is not part of the automata view. In both cases, additional program code verification is necessary as described in the previous section.

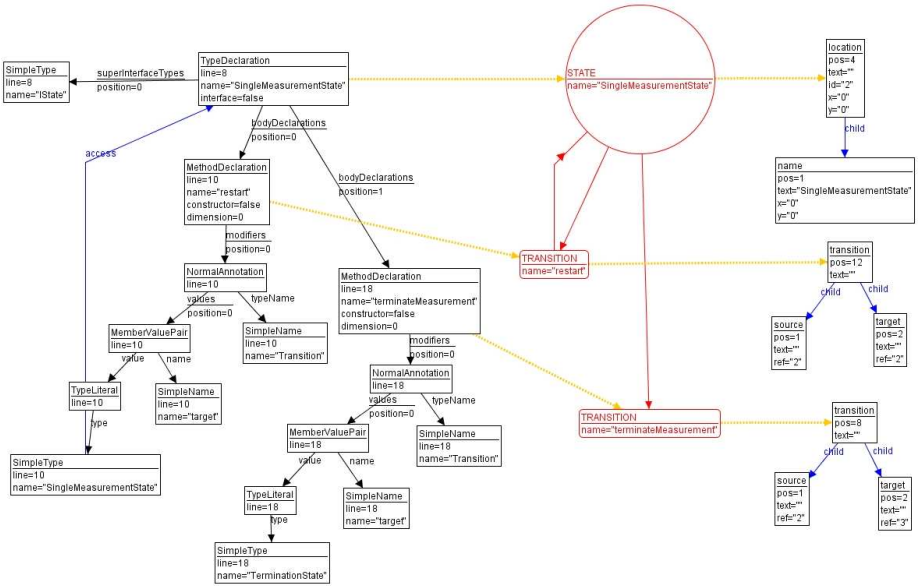


Figure 4.24: The graph transformation between program code and UPPAAL. The left-hand side is the graph as extracted from the code. In the middle the abstract information of interest is shown. The right hand constitutes the same information in the UPPAAL syntax [Striewe, 2008].

4.4.4 Execution of State Machines

The execution algorithm introduced in section 4.1.3 has been implemented in Java. The execution framework takes the following parameters:

- `Class<? extends IState> initialState`: The class definition for the initial state. Since state classes have the marker interface `IState`, the parameter type can be limited to such class definitions; i.e., no invalid classes can be passed since the compiler would not allow compilation of such code.

- `Object actor`: The facade object defining the action labels. At this point in time, no further assumptions about this object are made.

- `V variables`: The facade object for variable labels. Its type is not given explicitly, but only with the generic placeholder named `V`, which is reused in the following parameter definition.

- `Class<V> variablesInterface` A class definition for the interface providing the variable methods. This is necessary since the execution framework will have to store variable values before a transition fires and provide them for updates, so that not only the object implementing the variables facade is required, but also the facade definition itself. With the generic placeholder `V`, the method signature type-safely defines that the `variables` object has to implement the class definition of the `variablesInterface`.

First, the graph of node classes and transition methods is traversed and all nodes are initialized as shown in listing 4.6: The state instances are stored in a map with the class definition being the identifier (line 1). A recursive method checks if instances exist (line 8), creates new instances if necessary, and stores them (line 9). Afterwards, transition methods are identified (line 11) and the method is called for the target node recursively (line 12).

```
1  Map<Class<IState>, IState> states = new HashMap<Class<IState>, IState>();
2  createStateDefinition(initialState);
3
4  // ...
5
6  void createStateDefinition(Class<? extends IState> s)
7  {
8    if (!states.containsKey(s)) {
9      states.put(s, s.newInstance());
10     for (Method m : s.getMethods())
11       if (m.isAnnotationPresent(Transition.class))
12         createState(m.getAnnotation(Transition.class).target());
13   }
14 }
```

Listing 4.6: Node traversal and instantiation in the state machine execution framework.

The variable storage for updates is realized as shown in listing 4.7: In line 1, an object of type `VariableWrapper` is instantiated. In the following line, a so-called *dynamic proxy* [Forman and Forman, 2004] is created. Dynamic proxies in Java are objects that provide a certain interface, but are not instance of a specific type implementing the interface. Instead, they are notified about method invocations and handle them. Thus, the proxy is of type `V` and can be passed to update methods alongside the variables facade.

The object `variables` will handle method calls in the proxy. It is of type `VariableWrapper` that is shown in lines 6-24: The wrapper takes the variables interface and the variables facade as parameters. It has a map named `values` that assigns Java objects to method definitions. They are filled when the method `read` is invoked, which happens before a transition fires: All methods in the variables facade are invoked and the current values retrieved from them are stored in the map and identified by the method (lines 20-21). This is helpful when the methods provided by the proxy are accessed in updates: According to the definition of dynamic proxies, invocations are forwarded to the method `invoke` defined by the interface `InvocationHandler`, which is implemented by the wrapper. The Java platform supplies in this method information about the method that was called, e.g. `getCalories()`. The wrapper object simply returns the value that was stored for this method beforehand (line 14) and thus realizes the provision of cached values for updates.

```
1  VariableWrapper<V> wrapper = new VariableWrapper<V>(variablesInterface, variables);
2  V proxy = (V)(Proxy.newProxyInstance(variablesInterface.getClassLoader(), new Class<?>[]{
       variablesInterface }, wrapper));
3
4  // ...
5
6  public class VariableWrapper<V> implements InvocationHandler
7  {
8    private final Map<Method, Object> values = new HashMap<Method, Object>();
9
10   // ...
11
12   public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
13   {
```

```
14      return values.get(method);
15    }
16
17    public void read() throws EmbeddedStateMachineException
18    {
19      for (Method m : variablesInterface.getMethods()) {
20        Object value = m.invoke(instance);
21        values.put(m, value);
22      }
23    }
24  }
```

Listing 4.7: Creation of a dynamic proxy storing variables.

The selection and invocation of transitions is shown in listing 4.8: For all transition methods in the current state, the contract is instantiated (line 5). Its guard method is invoked (line 6), and if it returns `true`, the transition is selected. In this case, the current variable values are stored (line 7); the transition method is invoked and the actor facade is passed to it (line 8); the update method of the contract is called with the stored and the current variable values (line 9); and, finally, the next state is selected (line 11).

```
1   void next()
2   {
3     for (Method m : currentState.getClass().getMethods()) {
4       if (m.isAnnotationPresent(Transition.class)) {
5         IContract<V> c = (IContract<V>)m.getAnnotation(Transition.class).contract().newInstance
              ();
6         if (c.checkCondition(vars)) {
7           wrapper.read();
8           m.invoke(currentState, actor);
9           if (!c.validate(varsProxy, vars))
10            System.out.println("Validation␣failed␣in␣transition␣" + m.getName());
11          currentState = states.get(m.getAnnotation(Transition.class).target());
12          return;
13        }
14      }
15    }
16  }
```

Listing 4.8: Selection and invocation of transitions.

With these simple mechanisms relying on Java's reflection capabilities, the execution framework is realized. The source code shown above does not consider the more complex situation that channels exist (cf. section 4.1.3.2), but the principles of accessing and interpreting the program code structures are essentially the same.

4.4.5 Execution of Process Models

Although embedded process models introduce more functionality regarding variables and more types of nodes than state machines, their execution is even simpler. The reasons are

- a simpler scaling – process hierarchies are less complex to execute than communicating state machines;
- less state space management – variables are not considered by the execution framework since they are only passed to and retrieved from the application logic, in contrast to the management of stored variables for state machines;

- a shift of logic into program code expressions – the expressions in process actions are more complex and handle variables themselves, which makes them more complicated to consider at development time, but easier to invoke at run time.

Thus, execution of process models can be implemented with as little source code as shown in listing 4.9:

```
1  public void execute(Class<? extends IInitialNode> initialNodeClass, Object variables, Object
       actor) throws Exception
2  {
3    Map<Class<? extends IProcessNode>, IProcessNode> nodes = new HashMap<Class<? extends
       IProcessNode>, IProcessNode>();
4
5    // Initial node
6    IProcessNode currentNode = initialNodeClass.newInstance();
7    nodes.put(initialNodeClass, currentNode);
8
9    while (!(currentNode instanceof IFinalNode)) {
10     // Interpret current node
11     if (currentNode instanceof IActionNode) {
12       IActionNode.class.getMethods()[0].invoke(currentNode, variables, actor);
13     } else if (currentNode instanceof ISubProcessNode) {
14       Class<IInitialNode> sub = currentNode.getClass().getAnnotation(SubProcess.class).value
           ();
15       execute(sub, variables, actor);
16     }
17
18     // Get next node
19     Class<? extends IProcessNode> target = null;
20     for (Method m : currentNode.getClass().getMethods())
21       if (target == null && m.isAnnotationPresent(OutgoingEdge.class) && Boolean.TRUE.equals(
           m.invoke(currentNode, variables)))
22       target = m.getAnnotation(OutgoingEdge.class).value();
23     if (target == null)
24       throw new Exception("No␣enabled␣edge␣in␣node␣" + currentNode.getClass().getName());
25     currentNode = nodes.get(target);
26     if (currentNode == null) {
27       currentNode = target.newInstance();
28       nodes.put(target, currentNode);
29     }
30   }
31 }
```

Listing 4.9: A simple execution framework for process models.

The method `execute` takes as parameters the class definition of the initial class and the variables and actor facades. First, a map for the node instances is created and the initial node is stored therein (lines 3-7). Then the process is executed as long as the current node is no final node (line 9).

When the current node is an action node, the action method is invoked and the facades are passed as parameters (line 12). Since the expression in the source code will invoke the action, retrieve variables, and set the result, no further effort by the execution framework is necessary. When the current node is a sub process node, the `SubProcess` annotation is read to determine the initial node of the sub process (line 14) and the method `execute` is called recursively to execute the sub process (line 15).

Afterwards, the next node is to be determined. We can expect two situations here: (1) The current node is a decision node so that all edges must be considered with their guards; (2) the current node is no decision node and has thus only one edge method containing the statement `return true`. In both cases we can simply chose the first edge returning `true` when invoked, which is done in line 21. Afterwards, the target node class definition is read in line 22,

a possible deadlock handled in lines 23-24, and the target class is instantiated in lines 25-29.

This simple execution framework is fully functional. It relies on the assumption that all rules of the program code pattern are obeyed (e.g., transition methods in non-decision nodes always returning `true`). A more sophisticated execution framework would certainly provide more validation facilities, but the principle of the execution remains the same.

4.4.6 Monitoring Tool for State Machines

The pattern code fragments that are considered by tools at development time are interpreted and invoked at run time by the execution framework. Thus, the model specifications are available at run time to a certain degree, so that the program can be monitored with respect to abstract models.

4.4.6.1 Concept

Considering embedded models for state machines, the following steps are under control of the execution framework and desirable for monitoring:

- Initialization and start of a state machine. This includes information about all states, transitions, and variables as extracted from the Java code via reflection. States are uniquely identified by their fully qualified class names.
- Activation of states. This indicates that guard evaluation and transition selection in this state will happen subsequently.
- Selection of transitions. This indicates that program control will be handed over to the application logic in this transition.
- Validation of updates after a transition. The variable values are updated in this event. Additionally, the cached variable values from the point in time before the transition fired are also supplied to allow for comparisons. Additional information is supplied if the validation failed. When this event is fired, program control has been taken over by the state machine execution framework again.

A monitoring tool must thus work with the program code fragments defined by the pattern. Since this must happen at run time, the degree of detail varies as the bytecode has partly different structures than the source code (cf. section 3.1.3.3). Several ways exist to obtain information at run time, however, they have different advantages and implications. The following criteria are of interest:

- *Integration*: First, it is of interest if and how a technology can be integrated in existing environments, in this case the Java platform.
- *Accessibility*: The most important question will be which information about the state machine semantics is generally available at run time and at which points in time it is accessible.

- *Intrusiveness*: We will consider what changes to source code or byte code of the modeled system or the execution framework are indispensable to facilitate monitoring.
- *Self-monitoring*: Use cases are imaginable that systems being monitored gain access to the monitoring information, for example for self-inspection, validation, or dynamic adaption.
- *Performance*: Possible performance issues related to each approach will be considered.
- *Tools*: Regarding tool support it is of interest how the monitoring can be controlled and how far it can be integrated into development tools.

We now introduce three approaches in sections 4.4.6.2 to 4.4.6.4 that allow to access the required information in different ways and evaluate them briefly with respect to these criteria.

4.4.6.2 Listener Approach

Since all information about the running system and the embedded state machine semantics is available to the state machine execution framework, the easiest way for monitoring is to extend this framework in order to emit information of interest for active monitoring (cf. section 3.2.4). The execution framework is based on structural reflection and accesses and interprets a considerable part of the program code structures constituting the pattern. Besides setting listeners programmatically, module-based platforms (like OSGi [OSGi Alliance, 2005a] in the context of Java) allow for a loose coupling of execution framework and components receiving information about the execution. In the case of state machines, listeners can be notified about each step performed on the embedded model by the execution framework (cf. section 4.4.6.1). With respect to the criteria defined above, listeners are appropriate for monitoring embedded models as follows:

Integration Listener concepts can be realized in the programming language and the related platform without using any additional technologies or tools, since they are only an addition to the execution framework. They are thus easy to integrate.

Accessibility Considering the accessibility of elements of the program code pattern, the listener approach is clearly limited since expressions inside methods are not available by means of reflection (cf. section 3.1.3.3). Thus, operations inside guards and updates are not visible, but only their results after the related method was invoked by the framework. They must thus be considered black boxes.

Intrusiveness The approach is intrusive to the source code of the execution framework since it must be extended for the listener implementation. Since listeners rely on information extracted by reflection, they are completely non-intrusive to the application being monitored.

Self-monitoring Self-monitoring of applications is possible since the surrounding application invokes the execution framework and can thus register listeners, too. By this means the application can gain information about its own execution inside the state machine. This is possible without concurrency problems since the framework passes control over the program flow to the listeners during notifications, so that the program flow is strictly sequential.

Performance Because of the sequential access, the listeners can have a serious performance impact on the execution of the overall system, depending on the program logic executed during notifications. Since the listener service can be accessed by arbitrary components, the performance impact is not predictable. Developers of an application must in this case rely on reasonable listener implementations. Apart from that, the technical performance impact caused by the execution framework's notification functionality can be considered minimal since, technically, only additional methods are invoked.

Tools Because of the component-based realization, the listener approach can be easily integrated in different kinds of tools. This is definitely true for tools running in the same JVM, e.g. coupled by a module system. To use external tools, for example development environments that monitor the running system, listeners must send data over JVM boundaries. While this is possible, it could cause performance problems due to the communication.

4.4.6.3 Aspect-oriented Approach

Aspect-oriented programming (AOP) [Pawlak et al., 2005] aims at separating cross-cutting concerns from application logic. Monitoring and tracing are often-mentioned examples for AOP usage [Mahrenholz et al., 2002; Chen et al., 2004]: Emission of monitoring information is formulated as aspects that are woven into program code. To monitor state machine execution, the code structures of interest are accessed by pointcuts. Appropriate advice written in AspectJ [Colyer et al., 2004] are shown in listing 4.10. The first and the third pointcut wrap around guard and update methods, invoke them, and read the result. Afterwards a listener, named `monitor`, is notified about the contract class and the current result. The second pointcut is invoked before a transition method is executed, i.e., any method in a class implementing the `IState` interface. It notifies the monitor about the related state class and transition method name.

The main advantage of AOP in this context is that this passive monitoring (cf. section 3.2.4) can be applied without the need to modify the execution framework. With load-time weaving, monitoring capabilities can even be supplemented in systems after the program code has been compiled. This allows for flexible mechanisms that can be applied depending on the context. The pattern elements of embedded models are well-known and obligatory so that aspects can identify them and advice and pointcuts can address program code elements related to model elements. With respect to the criteria defined above, AOP is appropriate for monitoring embedded models as follows:

```
// Wrap guard method invocation and notify about the result
boolean around(Object vars) : execution(* IContract.checkCondition(..)) && args(vars) {
  boolean result = proceed(vars);
  monitor.notifyGuard(thisJoinPointStaticPart.getSignature().getDeclaringType(), result);
  return result;
}

// Notify about forthcoming transition method invocation
before() : execution(* *.*(..)) && target(IState) {
  monitor.notifyTransition(thisJoinPointStaticPart.getSignature().getDeclaringType(),
    thisJoinPointStaticPart.getSignature().getName());
}

// Wrap update method invocation and notify about the result
boolean around(Object before, Object after) :
       execution(* IContract.validate(..)) && args(before, after) {
  boolean result = proceed(vars);
  monitor.notifyUpdate(thisJoinPointStaticPart.getSignature().getDeclaringType(), result);
  return result;
}
```

Listing 4.10: The AspectJ monitoring aspect. All points of interest in the program code pattern are clearly identifiable by simple rules regarding their class and method names, so that pointcuts can be defined unambiguously.

Integration AOP technologies are available for many platforms, including Java. However, their use can be restricted by frameworks or technologies in use, e.g. module systems that control the class loading and thus may prevent the weaving. When weaving is possible, AOP monitoring can be applied without support by the execution framework and thus provides a certain flexibility.

Accessibility The information that can be accessed this way is the same as in the listener concept, i.e. with the limitation of black box methods in Java byte code. In addition, due to the limitations of pointcuts, the monitoring tool cannot observe the whole process of transition selection:

Pointcuts can handle information regarding the location of program code in which they are executed (keyword `thisJoinPointStaticPart`). But, they do not gain access to information in terms of sequences of pointcuts: In each state, a certain number of guards is evaluated. Afterwards, one transition method is invoked. While pointcuts are informed about the single actions, they cannot determine which guard belongs to the transition being executed; this information has to be guessed or supplemented by interpreting the program code afterwards.

The aspect-oriented approach is thus only capable of explaining decisions after the transition method has been selected and is about to be called.

Intrusiveness The usage of AOP does not require modifications in the execution framework architecture. Instead, modifications are applied to the compiled Java bytecode by the aspect weaver at compile or load time and thus transparent to the developer. However, which modifications are applied to which parts of the byte code is controlled by the weavers and thus out of control of the developer. When the correctness of the code is of interest in detail, further examinations would have to be undertaken to determine the changes and possible impacts when errors occur.

Self-monitoring Since the monitoring aspect code will run in the same JVM as the state machine, the same consequences to self-monitoring apply as with listeners. With compile-time weaving, the self-monitoring must be prepared during development. With load-time weaving, a self-monitoring would even be possible ad-hoc; however, this requires appropriate support by the platform and module frameworks, so that the aspects can be deployed on demand alongside the pattern code being monitored.

Performance Since the advice being used notify the monitor explicitly, performance is decreased by the cost for this additional communication. As with any AOP-based system, the changes introduced by weaving mechanisms in general must be considered, which are hard to predict since they are strongly influenced by the weaving implementations. This applies to both compile time and load time weaving, however, the effect will be stronger with load time weaving because the modifications are applied when the system is running.

Tools The AOP-based solution allows to communicate information gained from pointcuts to be transferred to tools for giving feedback to the developer. However, the limitations in accessibility are relevant, so that not all information is available in the tools or must be inferred by tools. This applies for example for information in guards: The tool is informed about the evaluations in guards, but it has to know the transition method being invoked afterwards to relate the guard to the transition. From this moment on, the tool can present the correct information to the developer. Apart from that, the same possibilities and limitations as with listeners apply.

4.4.6.4 Debugging Approach

The debugging approach delegates low-level observation of the program state to the executing platform. The related Java Platform Debugger Architecture (JPDA) is based on events that are triggered in certain situations, e.g. when a class is loaded or a field value is changed. Only some events are of interest for monitoring and have to be filtered from the event queue. By this means passive monitoring can be achieved. One way to achieve this is to use the *event request manager* that controls which events are passed from the application being debugged to the debugger. Another way is to perform the necessary filtering in the debugger by analyzing the contents of the event object. This saves the effort for configuring the manager but implies more communication overhead for irrelevant events.

State activation and transition selection are monitored by observing certain fields, using both ways sketched above. First, all instances of `ClassPrepareEvent` are considered to gain class loading information. If the class for state machine execution is loaded, which can be clearly identified by its name, the event request manager is configured to provide `ModificationWatchpointEvents` for the class attributes pointing to the current state and transition instance inside the execution framework. When they are modified, the events will afterwards provide the current value of the variable and the value to be set.

Of special interest within the debugging approach is the possibility to monitor operations inside guard and update methods. To achieve this, events are considered that are triggered when a related method is entered. The method parameters pointing to the variables facade instance for current or cached model variable values are available as local variable values from the stack frame of the state machine execution thread. However, model variables are represented by methods inside this variables type and neither their names nor their values are thereby directly visible to the debugger. Hence, `ClassPrepareEvents` are also used to detect when an implementation of this interface is loaded and fetch a list of all methods. The debugger is then able to iterate over the list of methods and invoke each method to retrieve the return value and thus the value of the respective model variable. This is sufficient for non-intrusive monitoring in terms of unchanged source code, but causes method invocations that may modify the application's state space. For this reason `MethodExitEvents` are useful which are triggered after all code of a method is executed, but before the method is left. At this point in time the return value of variable methods is accessible without additional method invocations. Since only expressions are used inside guards and updates, the evaluation is fully comprehensible afterwards by inspection of the values of local variables. The return value of the method and thus the result of the evaluation is accessible with `MethodExitEvents` applied to the guard and update methods themselves.

With respect to the criteria defined above, debugging is appropriate for monitoring embedded models as follows:

Integration The JPDA is available as part of the Java platform. But, debuggers always need two running JVM instances: The application being debugged and the debugger itself that controls execution. While an integration is thus possible at the technical level, production environments cannot fulfill the requirements of debuggers. Monitoring with debuggers is for this reason only appropriate for testing purposes.

Accessibility A debugger can access all elements of the program code pattern in model implementations as well as all local variables in the execution framework. Different to the listener and AOP approaches, this allows for monitoring guard and update method contents. Since all details of expressions are available, the evaluation of guards and updates can be recorded and presented to the developer for each step. The debugging approach is therefore the only one able to access all elements of the program code pattern. Access to variables and method invocation results is possible without additional effort when they are accessed by the application itself. For the state machine model this is sufficient since the variables are of interest only when they are evaluated in guards. A debugger would also allow to invoke methods at any time. This could be of interest for variable methods to determine their current value.

Intrusiveness Debugging is not intrusive to the execution framework and also not to the bytecode, which is not modified. However, it is intrusive at the level of the Java platform: With debugging interfaces and tools, developers can invoke methods during breakpoints. Thus, the sequence of method

invocations can be different than defined by the model, so that side-effects can occur.

Self-monitoring Information from the JVM being debugged can only be received by the debugging JVM. This prevents self-monitoring since the latter is only a development tool and not available in production environments.

Performance Debuggers in general have a strong impact on performance, so that a monitoring of production systems is currently not desirable with this approach.

Tools Since debuggers themselves are tools, appropriate support is given. Most Java IDEs support this and provide inference to source code, variable monitoring, and evaluation of statements and expressions by the developer. Such tools could be extended for embedded state machines: The model structure as read from the code could be visualized and provide entry points to the program code. The state space and the decisions in guards could be monitored in detail. When methods in the facade types are invoked, the tool could switch to the usual debugging view so that developers can analyse the program code that is not part of the model.

4.4.6.5 Monitoring Tool

The approaches to monitoring provide different means to monitor state machines in different degree of detail. Considering them together, we can state that monitoring of embedded models is possible in detail, although limitations can apply during run time when platforms or frameworks do not fulfill the requirements of the approaches.

A monitoring tool has been implemented that relies on listeners. It provides a graphical view and is connected to the execution framework by means of OSGi, thus realizing a loose coupling.

User Interface The user interface of our monitoring tool provides three views on the monitored state machine as can be seen in figure 4.25. In the center it shows a graphical representation of the state machine, highlighting active states and transitions. A spring layouter arranges the states in an appropriate way, but they can easily be rearranged by the user. The chosen layout can then be saved and is automatically reloaded when the same state machine is monitored the next time. A view at the right hand side lists all variables defining the state space of the monitored machine. It shows the current values according to the last update as well as the previous values before this update. Another view can be shown by selecting "Validation" that lists the updates which could not be validated successfully. Since the execution framework is in full control of the program flow, the views are accompanied by buttons to pause and resume state machine execution that can be operated when an event is handled.

These views and controls offer several ways to analyze a state machine at run time. The simplest way is to just watch the machine running and check manually whether the machine is always in the state that is expected from the outward behavior and whether it is performing the expected transitions. This

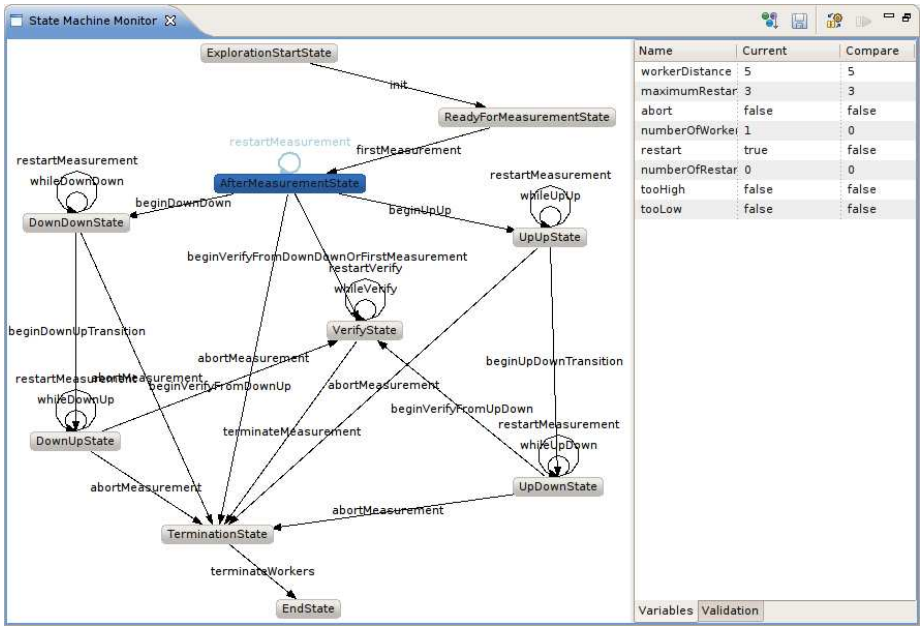


Figure 4.25: The monitoring tool showing the load generator example at run time. The information is extracted from the execution framework with structural reflection. We can monitor the selection of states and transitions as well as the variable values and the evaluation of guards and updates.

would answer the question what the state machine and thus the application is doing at all without further inspection of details.

A more detailed way is to stop the machine after performing a transition and to have a look at the variable values defining the state space. The comparison between new and old values after validations allows to analyze the state space in a more detailed way and to understand the reasons why certain transition have fired. This can explain in detail the chosen path the state machine walks through. In addition, this information can be used to predict what has to happen in order to force the state machine to reach a certain state.

The user interface also shows the lists of variables and of failed contract validations. If the execution component detects a failed update, this is a strong hint that the implementation of the application logic does not conform to the intended model. Since the embedded model can always be extracted and checked with validation tools, it is then possible to track down whether the failure origins from a wrong implementation of the state machine or from errors in the application logic code.

Architecture We use the OSGi platform's service registry [OSGi Alliance, 2005b] for the tool. Listeners like the monitoring user interface shown above are hence OSGi bundles being deployed alongside, but independent from application logic. The listener is registered as a named OSGi service that is detected by the execution framework. The related loosely-coupled architecture as sketched in figure 4.26 allows to use almost arbitrary tools to

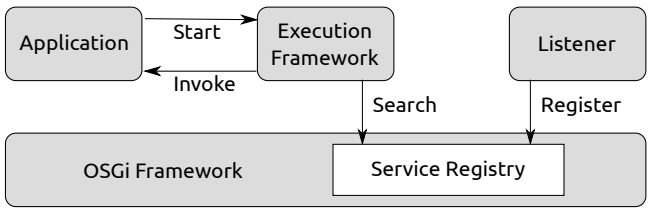


Figure 4.26: Component architecture for the monitoring listener. Application logic may be composed of components using the execution framework as well as of components using the OSGi platform. The listener component for monitoring is optional in any case and hence only coupled via the service registry.

be notified about events: The execution framework discovers listeners using the platform. Monitoring components may also transfer data to external tools, for example debuggers in development environments.

4.4.7 Design Recovery Tool

Embedded models enable a tight coupling between specification and implementation. This is especially true when the code is accessed by model transformations. One area of application for such transformations is that of design recovery: With embedded models, implementation and documentation cannot be inconsistent, since design information is contained in the code. This enables automated design recovery and an automated transformation of the pattern code to other structures. As a prototype for such transformations, a tool has been developed by Michael Striewe that transforms embedded state machines to embedded process models by means of graph transformation [Goedicke et al., 2009; Striewe et al., 2010b].

4.4.7.1 Concept

As stated in section 4.3, state machines and process models share common elements and structures. The major exceptions are interaction of state machines, which cannot always be mapped to process hierarchies unambiguously, and updates in state machines, so that both are not considered here. The transformation consists of the following steps:

1. All states of the state machine are converted to decision nodes in the process model since each state has potentially multiple outgoing transitions with guards.
2. All transitions in the state machine are converted to action nodes properly connected to decision nodes in the process model.
3. Each action node that contains more than one action label is split up into a sequence of action nodes. This step can be performed here or at any later point in time.
4. Each decision node with exactly one incoming and one outgoing edge is discarded by connecting the nodes of the incoming and outgoing transition directly.

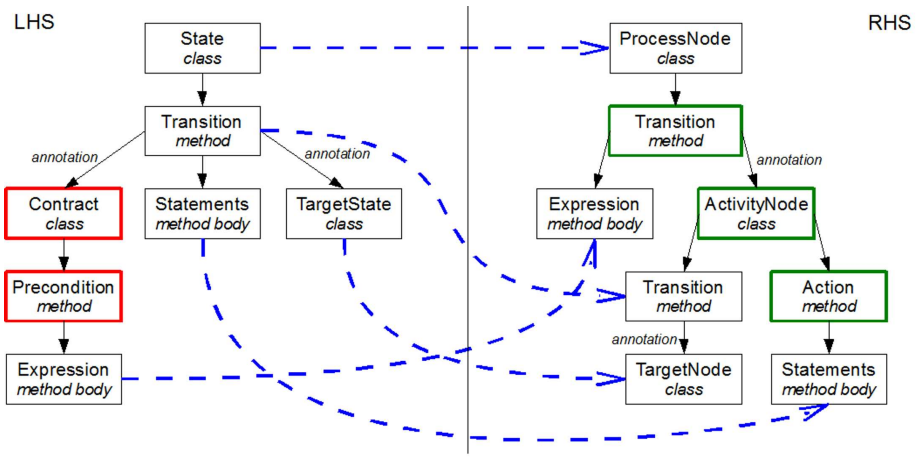


Figure 4.27: Simplified graph transformation rule for transforming states into process nodes. Nodes deleted from the syntax graph are marked in red while newly-created nodes are marked in green. Some of the preserved nodes are renamed during transformation. Information from the state class is moved to a newly-created activity node class, while content from the contract class is integrated in node classes [Goedicke et al., 2009].

5. Each decision node without incoming transitions is changed to a start node.

6. Each decision node without outgoing transitions is changed to an end node.

7. Each descision node with multiple incoming transitions and only one outgoing transition is changed to a merge node.

While this list of steps applies to the model transformation itself, using embedded models requires additional steps because formal aspects of program code (e.g. import statements) have to be taken into account.

4.4.7.2 Implementation

The steps of the transformation are implemented as graph transformation rules acting on a graph generated from source code (cf. section 4.4.3.3). For this purpose, 21 rules are needed. First, two rules convert states to decision nodes and transitions to activity nodes. One of these rules that changes states to process nodes and creates activity nodes is shown in figure 4.27 in a simplified manner. According to the rules of the program code pattern for embedded state machines, elements to be moved can easily be identified by their annotations on the left-hand side of the rule and thus reassembled on the right-hand side. Similarities between state machines and process models allow to reuse larger parts of existing program code, e.g. complete method bodies for guards.

After nodes have been converted or created, a set of six rules applies changes for cleaning up the graph, like reordering imports or removing

unnecessary annotations. These rules are not strictly necessary, but desirable for a clean implementation. Two additional rules remove the contract classes which are not needed in the process model as well as useless decision nodes that have been introduced by rules applied earlier. Afterwards, the set of nodes is complete, so three rules can be applied for marking start nodes, end nodes, and merge nodes. An additional rule splits action nodes with more than one action label; this is necessary since state machine transitions can have multiple action labels, while action nodes will have exactly one action label. Another set of seven rules is finally concerned with some adjustments to the code.

4.5 Chapter Summary

In this chapter we introduced implementations of the embedded models approach from chapter 3.

The first implementation is for the modeling domain of state machines in section 4.1. After a definition of the formal model, the program code pattern was defined and its interfaces, execution semantics, and transformations were explained.

The second implementation in section 4.2 is based on process models, in this case the meta model of JWT. A program code pattern was derived and the interfaces to other program code were defined. The definition was completed with execution semantics and transformations.

Both embedded models were compared in section 4.3 with respect to their representation of model elements, the management of the state space, and their particular scaling mechanisms.

Based on the embedded model definitions, tools supporting development at different levels of abstraction were introduced in section 4.4 for design, verification, execution, monitoring, and design recovery. For state machines, a tool suite exists that supports all these stages of the development process. For process models, design and execution tools exist. Design recovery is realized by a transformation from state machines to process models.

In the next chapter we will evaluate the approach with two case studies that use embedded state machines and process models.

Chapter 5

Evaluation

So far, the approach for embedded models and specific implementations for the domains of state machines and process models have been presented in chapters 3 and 4. In this chapter we will evaluate the approach based on criteria that will be introduced in section 5.1. The evaluation is performed by considering two case studies:

First, the approach is applied to a load generator for performance tests, called "SyLaGen", in section 5.2. Essential algorithms for controlling the load generation are modeled with both embedded state machines and process models. These models are integrated in the larger context of the actual application. This will demonstrate the applicability of the approach for non-trivial real-world applications.

The second case study in section 5.3 is small, but focuses on different aspects. Embedded state machines are here integrated in a visual programming learning environment. Based on this, small games are developed, whose actors are communicating state machines. The software has been used for teaching formal methods in a university master's course. In this context, a small user study has been performed.

An overall evaluation of the approach that considers the partial evaluation results from both case studies will finally be given in section 5.4.

5.1 Criteria

Based on the objectives from chapters 1 and 3, embedded models have the goal to apply the advantages of models to program code patterns that are connected to application logic. An evaluation of the approach must thus focus on advantages of models and validate their usefulness in the case studies. We will use as criteria what is called by Selic the *quality of models* [Selic, 2003]:

Abstraction "A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for a given viewpoint, it lets us understand the essence more easily." [Selic, 2003] In our context, it is of interest if a meaningful abstraction from program code details is possible. This is to be considered especially careful for the interfaces where other application logic is connected to the model.

Understandability "A good model provides a shortcut by reducing the amount of intellectual effort required for understanding." [Selic, 2003] For embedded models, we do not evaluate the formal models themselves. Instead, two aspects of the understandability are of interest: First, we must evaluate how the entirety of program code with embedded models is comprehensible. Second, we must evaluate the understandability of model instances that are not only driven by domain requirements, but also by the consideration of embedded model conventions and interfaces to application logic.

Accuracy "A model must provide a true-to-life representation of the modeled system's features of interest." [Selic, 2003] Since embedded models are not considered stand-alone, but always in connection with interfaces to application logic, the models may be influenced. We will consider how accurate the models in use will reflect the domain requirements under such circumstances.

Predictiveness "You should be able to use a model to correctly predict the modeled system's interesting but nonobvious properties [...]." [Selic, 2003] While this depends heavily on the formal model in use, we will evaluate here how models that are influenced by interface considerations are still eligible for model checking and analysis.

Inexpensiveness "A model [...] must be significantly cheaper to construct and analyze than the modeled system." [Selic, 2003] In the context of embedded models, two kinds of overhead have to be considered: First, the program code is likely to consist of more static fragments than manually-written algorithms. Second, models considering rules of embedded models and interface definitions may be more complex than pure abstract models would have been. For both, we will evaluate the tool support for working at different abstraction levels.

With these criteria in mind, we introduce the case studies now. Each case study will be evaluated separately in sections 5.2.4 and 5.3.5. Afterwards, an overall evaluation will be given in section 5.4.

5.2 Case Study "Load Generator"

The first case study focuses on the use of embedded models in a non-trivial, real-world application. We describe SyLaGen first in section 5.2.1. The implementation of embedded state machines is described in section 5.2.2 and the implementation of embedded process models in section 5.2.3. Afterwards the evaluation criteria are applied to this case study in section 5.2.4.

5.2.1 Description of "SyLaGen"

SyLaGen is a load generator for performance tests [Striewe et al., 2010a; Bordewisch et al., 2003] that has been under development since 1999 by the working group *Specification of Software Systems* at University of Duisburg-Essen. The overall architecture has two distributed components:

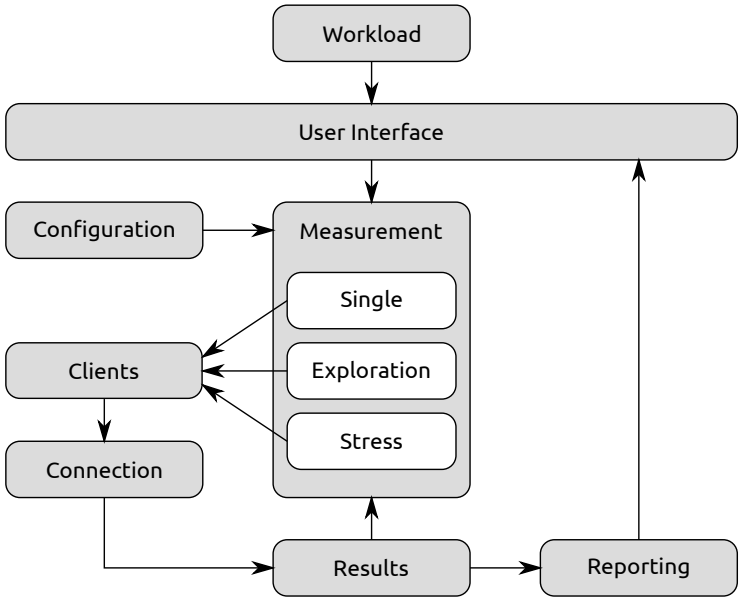


Figure 5.1: The SyLaGen master modules. In the center, the measurement module can be seen with its different load strategies. It coordinates the actual load generation.

The so-called *master* controls a measurement by directing *clients* that run on different physical machines. Each client has a certain number of threads called *workers* that generate the actual load.

5.2.1.1 Modules

The master has a high complexity and is for this reason split into different modules as sketched in figure 5.1:

The *workload* module manages, stores, and validates load descriptions. These are edited by the user in the *user interface*. When a performance test is started, control is passed to the *measurement* module. It reads general configuration from the *configuration* module. The measurement module itself contains different strategies for generating load: The *single* strategy performs only one measurement with load that is specified for a certain time frame. The *stress* strategy performs also one measurement, but generates as much load as possible. The *exploration* strategy is the most complex since it performs a sequence of measurements by evaluating the results after each single measurement and adapting the number of workers in use in order to find the optimum.

The measurement module and its strategies use the *client* module that manages clients connected to the master. It also distributes commands given by the measurement module to all clients that are used in the current measurement. The actual communication with clients occurs in the *connection* module, which also receives data from the clients after a single measurement has finished. These results are passed to the *result* module which receives them from sin-

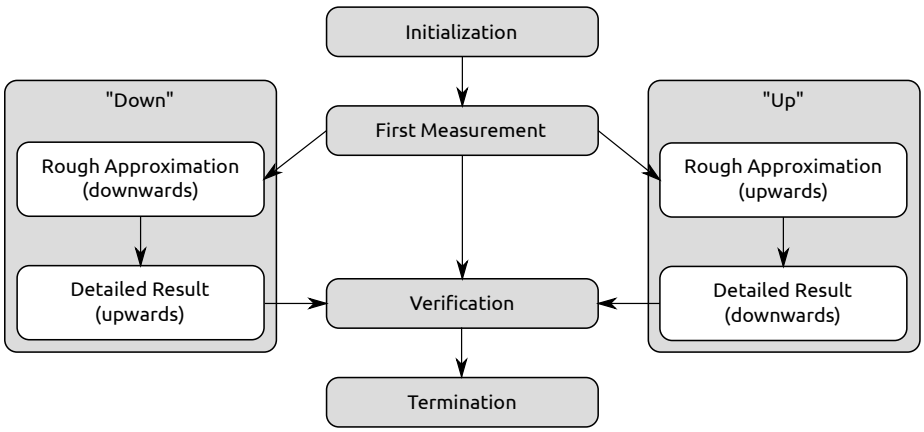


Figure 5.2: The measurement process of the "exploration" strategy in SyLaGen. After the first measurement, the exploration can continue downwards at the left hand, upwards at the right hand, or simply finish when the system under test is already overloaded.

gle clients and merges them into an overall result, e.g. with average response times measured by each client.

The results are then either analyzed by the exploration strategy, which may decide to perform additional measurements, or are passed to the *reporting* module that processes them for graphical presentations. Then the results are passed to the user interface.

We can see that a measurement follows a certain process through the modules which has been presented here in a simplified way. The measurement module is essential since it controls the actual measurement and directs other modules for this purpose. The actual load generation strategies have different well-defined states. Since they are exchangeable, they are realized as plug-ins in the measurement module and thus interact with their environment by means of well-defined interfaces. These are good conditions for implementing them as embedded models, which can facilitate documentation and program comprehension and at the same time use the interfaces to integrate in the context of SyLaGen.

5.2.1.2 Exploration Measurement

The process of the exploration measurement is roughly sketched in figure 5.2: First the environment is initialized, i.e. connections to the clients are established, etc. Then the first measurement is performed with a number of threads defined by the user in the workload. Depending on the response time of the system under test, called *turnaround time* in this context, three possible directions can be taken:

- When the number of workers is 1 and the system under test is already overloaded, i.e. does not fulfill the requirements for the turnaround time defined in the workload, the measurement is not continued.

- When the number of workers is >1 and the system under test is overloaded, the exploration measurement is continued *downwards*: The number of workers is decreased in larger steps (also defined by the user, e.g. 5 workers) until the system is no longer overloaded. After this rough approximation, the number of clients is increased again by 1 until the system is overloaded again, so that the exact limit is known.
- When the system under test is not overloaded, the exploration measurement is continued *upwards*: The number of workers is increased in larger steps until the system is overloaded. After this rough approximation, the number of clients is decreased again by 1 until the system is no longer overloaded, so that the exact limit is known.

Whether the system is overloaded is decided as follows. Two constraints are given by the user: The *expected turnaround time* that should not be exceeded on average and the *maximum turnaround time* that is not allowed to be exceeded at all. The system is overloaded if either the mean turnaround time is greater than the expected turnaround time or if one of the workers encountered a turnaround time that is greater than the maximum turnaround time. This decision is non-trivial as it requires access to each single result of all workers, so that it is hard to integrate in abstract models and must be abstracted from.

After all of the three steps, a last measurement is performed called *verification*: The number of workers is decreased again in larger steps and measurements are performed until the last measurement is performed with 1 worker. This is done in order to verify that the system under test is fully functional and the previous measurement results have not been corrupted by errors. Afterwards, the environment is terminated, i.e. the clients are instructed to shut down the workers, the results are stored and passed to the reporting module, etc.

In the implementation, this process is more complicated as more decisions are possible after each measurement. The user can abort the measurement so that after each measurement the termination can occur. When errors are detected, the measurement is restarted a certain number of times as defined by the user; this can lead to repetitions and also to abortion of the measurement.

We will now describe the implementation of this algorithm with two alternative embedded model classes, state machines (section 5.2.2) and process models (section 5.2.3).

5.2.2 State Machine

The exploration strategies have been documented for a certain time with UPPAAL, but we made the experience that documentation and implementation were out of sync quite often. Thus the decision was made four years ago to apply embedded state machines to SyLaGen based on the requirements described in the previous section. Since then, the strategies are maintained with the related program code pattern.

5.2.2.1 Structure

The exploration measurement as described in section 5.2.1.2 is implemented in terms of states and transitions as shown in figure 5.3:

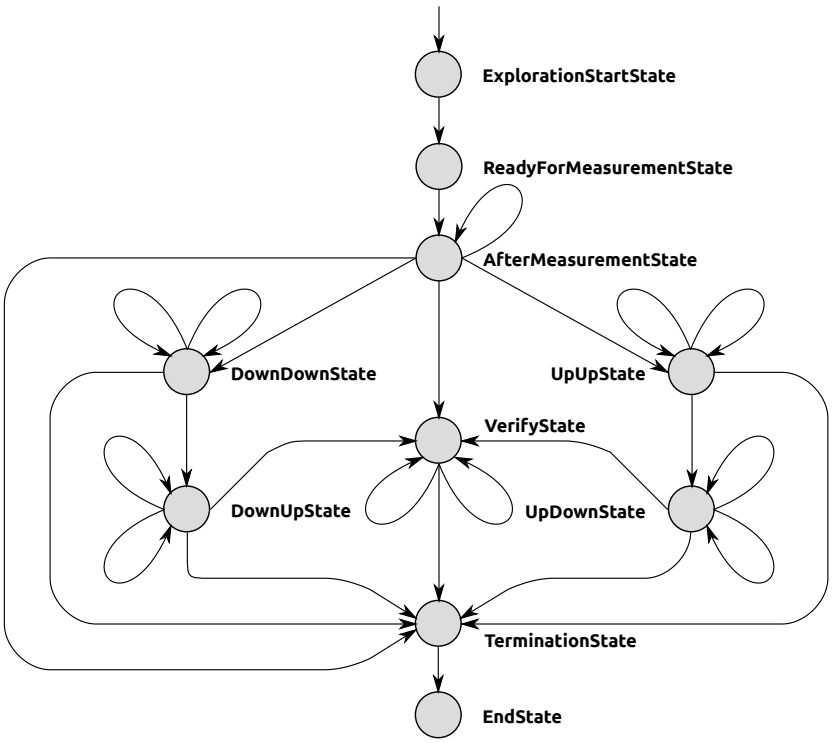


Figure 5.3: The states of the exploration strategy.

At the beginning, the environment is initialized in the states *ExplorationStartState* and *ReadyForMeasurementState*. In the following transition to *AfterMeasurementState*, the first measurement is performed. When errors have occured, the measurement is repeated. This is indicated with the "loop" transition that leads back to *AfterMeasurementState*. When the number of allowed restarts is exceeded or the measurement is aborted by the user, the transition to *TerminationState* is chosen. When the system under test is overloaded with the number of workers being 1, the transition to *VerifyState* will fire.

When the measurement is continued downwards, the number of clients is reduced and a measurement is performed in the transition leading to *DownDownState*. As long as the system under test is overloaded, the two loop transitions fire: Either no errors occur and the number of workers is decreased, or the last measurement is repeated if errors were encountered. In the case of too many errors or abortion by the user, the transition to *TerminationState* will fire. When the system under test is no longer overloaded, the transition to *DownUpState* is selected. Here, the number of workers is increased by 1 with a loop transition until the limit is reached and the transition to *VerifyState* fires. In the case of errors and user abortion, the same rules apply as within *DownDownState*.

The upwards measurement follows the same principles: *UpUpState* increases workers in large steps, *UpDownState* decreases workers afterwards by 1 to find the exact limit. In both states, repetition and user abortion are

handled with appropriate transitions.

During verification, the number of workers is decreased stepwise with a loop transition. A second loop transition handles repetition of measurements after errors. The only transition from this state leads to *TerminationState* where the measurement environment is shut down in the transition to *EndState*.

With these states and transitions, the exploration measurement is described completely. In the next sections, the variables controlling the flow through the state machine and the actions invoked in transitions will be introduced.

5.2.2.2 Variables

Eight variables control the exploration measurement. They can be assigned to four groups:

- Workers:
  - *NumberOfWorkers* (integer): The number of workers currently in use. The initial value is defined by the user in the workload. This variable is not aggregated, but directly used by the state machine.
  - *WorkerDistance* (integer): The step size used to increase or decrease the number of workers for the approximation. This variable is not aggregated and does not change during measurement.
- Turnaround times:
  - *TooHigh* (boolean): Indicates that the turnaround time requirements have not been fulfilled by the system under test and thus that the number of workers is too high. This value is strongly aggregated by the application logic of SyLaGen since it is extracted from statistical calculations based on the results from multiple workers.
  - *TooLow* (boolean): Indicates that the turnaround time limits have not been reached in the last measurement and thus that the number of workers is too low. This value is also aggregated from the results.
- Restarts:
  - *Restart* (boolean): Indicates that an error occured during the last measurement. This value is aggregated as it is the summary of errors detected in the results.
  - *NumberOfRestarts* (integer): The current number of restarts that were necessary in the measurement. This variable is not aggregated, but directly used by the state machine.
  - *MaximumRestarts* (integer): The maximum number of restarts allowed as defined by the user in the workload. This variable is not aggregated and does not change during measurement.
- Abortion:
  - *Abort* (boolean): Indicates that the user aborted the measurement. This value is aggregated as it represents actions performed by the user.

The turnaround times influence the decisions during the measurement. The number of workers is then used and adapted between single measurement steps. Errors and restarts are considered for each measurement, and abortion by the user is considered in each transition. The guards and updates thus result in a model shown in figure 5.4 as extracted to UPPAAL:

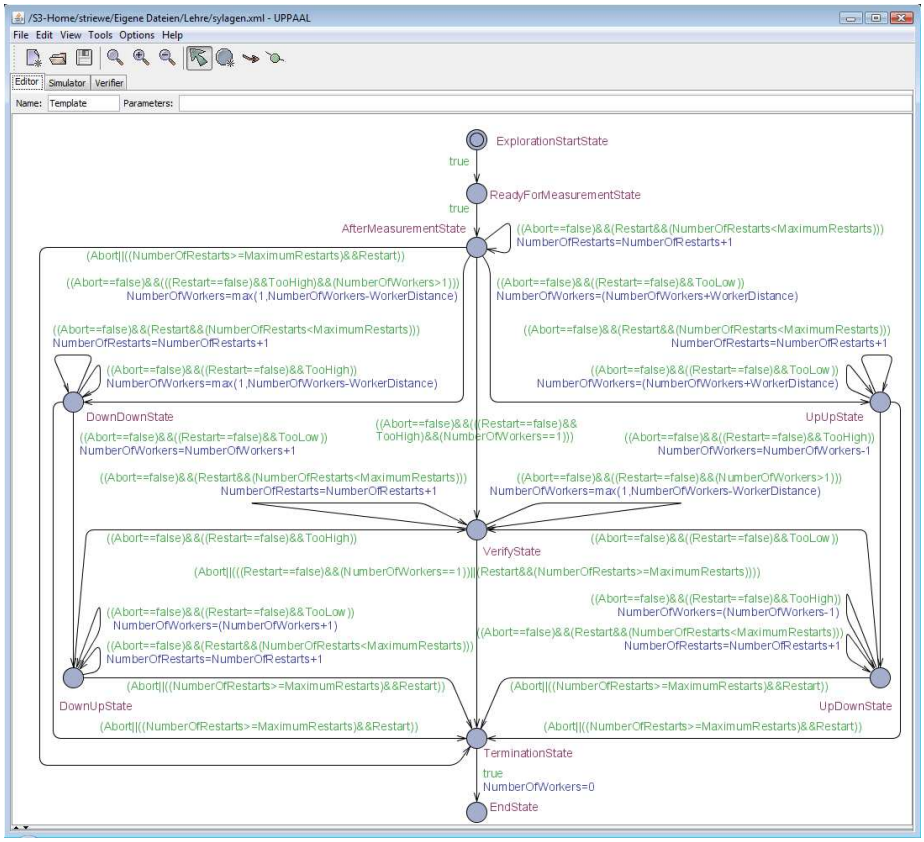


Figure 5.4: The exploration strategy as a state machine in UPPAAL.

As can be seen in this figure, all variables are used in guards and thus cover the whole exploration strategy. However, the aggregated variables are not under control of the state machine. Therefore only a few variables are used in updates: Values are assigned to *NumberOfWorkers* and *NumberOfRestarts*.

5.2.2.3 Actions

The actor provides the following actions:

- Environment:
  - *initMeasurement*: Initialize the measurement environment.
  - *terminateMeasurement*: Terminate the measurement environment.
  - *terminateWorkers*: Termine workers on clients.

- Measurement:
  - *initClients*: Adjust the number of workers on clients for a specific measurement.
  - *doMeasure*: Perform a single measurement.
  - *doRestartMeasure*: Perform a single measurement and note in the result documentation that this is a repetition of a measurement that had errors before.
  - *doVerifyMeasure*: Perform a single measurement and note in the result documentation that this is done for verification.
- Adjustment of the number of workers:
  - *beginDownDown* and *whileDownDown*: Decrease the number of workers for downwards measurement in large steps.
  - *beginDownUp* and *whileDownUp*: Increase the number of workers by 1 for detailed measurement.
  - *beginUpUp* and *whileUpUp*: Increase the number of workers for upwards measurement in large steps.
  - *beginUpDown* and *whileUpDown*: Decrease the number of workers by 1 for detailed measurement.
  - *whileVerify*: Decrease the number of workers for verification measurement in large steps.
- Restarts:
  - *increaseNumberOfRestarts*: Increase the counter for the number of restarts.
  - *resetRestarts*: Reset the counter for the number of restarts.

These actions are all used in the exploration state machine. It is important to note that multiple actions can be invoked in one transition. The environment actions are invoked only at the beginning and the end of the exploration strategy. The measurement actions are re-used for different transitions that perform actual measurements. The adjustment actions are each invoked in only one transition before the actual measurement is performed. Finally, *increaseNumberOfRestarts* is only invoked when errors have occured.

A part of the exploration state machine is shown in figure 5.5 with action labels: After the first measurement, a restart can occur in the loop transition. In this case the restart counter is increased, the clients are initialized, and a restart measurement is performed. When this is not the case, the restart counter is reset, the number of workers is adjusted for the upwards measurement with *beginUpUp*, the clients are initialized, and a normal measurement is performed before *UpUpState* becomes the active state. The first loop transition here performs measurements after a *whileUpUp* adjustment, the second handles restarts. One transition leading to *TerminationState* (not shown in the figure) and calling the action *terminateMeasurement* is chosen when the user aborts the measurement. Another transition leads to *UpDownState* if the measurement is continued. It adjusts the number of workers with *beginUpDown*,

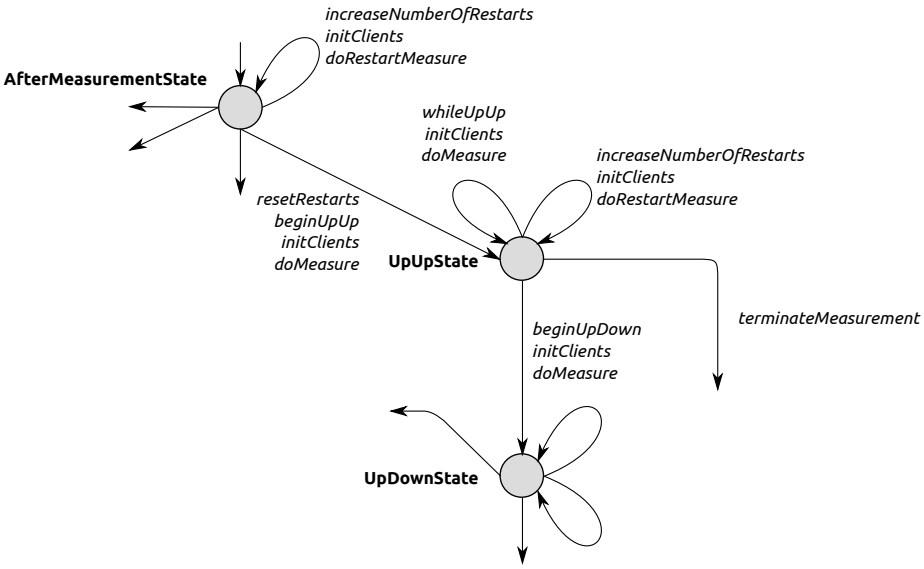


Figure 5.5: A part of the exploration state machine with action labels.

initializes the clients, and performs a measurement. While this is only a part of the whole state machine, the same principles with respect to actions apply to other transitions, too.

5.2.2.4 Implementation

With the states, transitions, variables, and actions defined above, the state machine model is complete. It respects the facades to the application logic and can thus be implemented straightforwardly.

The variables interface is shown in listing 5.1. It follows the rules for variable facades of embedded state machines, i.e. it contains only get methods with simple data types.

```
1  public interface IMeasurementVariables
2  {
3    int getNumberOfWorkers();
4    int getWorkerDistance();
5
6    boolean getTooHigh();
7    boolean getTooLow();
8
9    boolean getRestart();
10   int getNumberOfRestarts();
11   int getMaximumRestarts();
12
13   boolean getAbort();
14 }
```

Listing 5.1: The variables facade interface for SyLaGen measurement strategies with get methods for the embedded state machine model.

As a representative example for all states of the model, the class `AfterMeasurementState` can be seen in listing 5.2. Its transition methods contain the actions describes above.

```
1  public class AfterMeasurementState implements IState
2  {
3    @Transition(target = DownDownState.class, contract = BeginDownDownContract.class)
4    public void beginDownDown(MeasurementModule actor) throws MeasurementAbortedException
5    {
6      actor.resetRestarts();
7      actor.beginDownDown();
8      actor.initClients();
9      actor.doMeasure();
10   }
11
12   @Transition(target = UpUpState.class, contract = BeginUpUpContract.class)
13   public void beginUpUp(MeasurementModule actor) throws MeasurementAbortedException
14   {
15     actor.resetRestarts();
16     actor.beginUpUp();
17     actor.initClients();
18     actor.doMeasure();
19   }
20
21   @Transition(target = VerifyState.class, contract =
         BeginVerifyFromDownDownOrFirstMeasurementContract.class)
22   public void beginVerifyFromDownDownOrFirstMeasurement(MeasurementModule actor) throws
         MeasurementAbortedException
23   {
24     actor.resetRestarts();
25     actor.initClients();
26     actor.doVerifyMeasure();
27   }
28
29   @Transition(target = AfterMeasurementState.class, contract = RestartMeasurementContract.
         class)
30   public void restartMeasurement(MeasurementModule actor) throws MeasurementAbortedException
31   {
32     actor.increaseNumberOfRestarts();
33     actor.initClients();
34     actor.doRestartMeasure();
35   }
36
37   @Transition(target = TerminationState.class, contract = AbortMeasurementContract.class)
38   public void abortMeasurement(MeasurementModule actor)
39   {
40     actor.terminateMeasurement();
41   }
42 }
```

Listing 5.2: The node class `AfterMeasurementState`.

The contract for the transition *beginUpUp* is shown in listing 5.3. The guard method checks that the two default conditions apply, i.e. that the measurement was not aborted and is not to be restarted; in both cases, other transitions have to be chosen. The third condition controls the actual measurement and checks whether the variable *TooLow* is true, i.e. whether the number of workers is to be increased. In this case the transition from *AfterMeasurementState* to *UpUpState* will fire. As defined above, the number of workers will in this case be increased by *WorkerDistance*. This is validated in the update method where the *NumberOfWorkers* is compared with the stored value from the point in time before the transition fired.

```
1  public class BeginUpUpContract implements IContract<IMeasurementVariables>
2  {
3    @Override
4    public boolean checkCondition(IMeasurementVariables vars)
5    {
6      return (!vars.getAbort() && !vars.getRestart() && vars.getTooLow());
7    }
8
9    @Override
10   public boolean validate(IMeasurementVariables before, IMeasurementVariables after)
11   {
12     return (after.getNumberOfWorkers() == (before.getNumberOfWorkers() + before.
           getWorkerDistance()));
13   }
14 }
```

Listing 5.3: A contract class in the embedded exploration state machine.

This is only a small part of the implementation, but all other classes follow the same rules. The embedded model instance for the exploration state machine is by this means complete.

5.2.3 Process Model

The exploration strategy will now be modeled with JWT and an embedded process model.

5.2.3.1 Structure

The most influential differences between embedded state machines and process models regarding the structure are that only one action can be called in each action node and that hierarchies are available in processes. While the first introduces higher complexity in the structure, the latter will help to manage the complexity. We will thus not create a single model, but a hierarchy of models. The top level of this hierarchy is sketched in figure 5.6: It resembles the structural view of the state machine. However, most of the nodes are sub process nodes (cf. section 4.2.1) that refer to underlying processes. The decision which functionality is aggregated in a sub process is mostly given by the JWT model: Since sub processes cannot influence decisions at higher levels, sub processes must terminate before decisions.

5.2.3.2 Variables

The variable definitions are similar to the state machine model. However, our goal here is to use the fact that variable values can be set as the result of actions

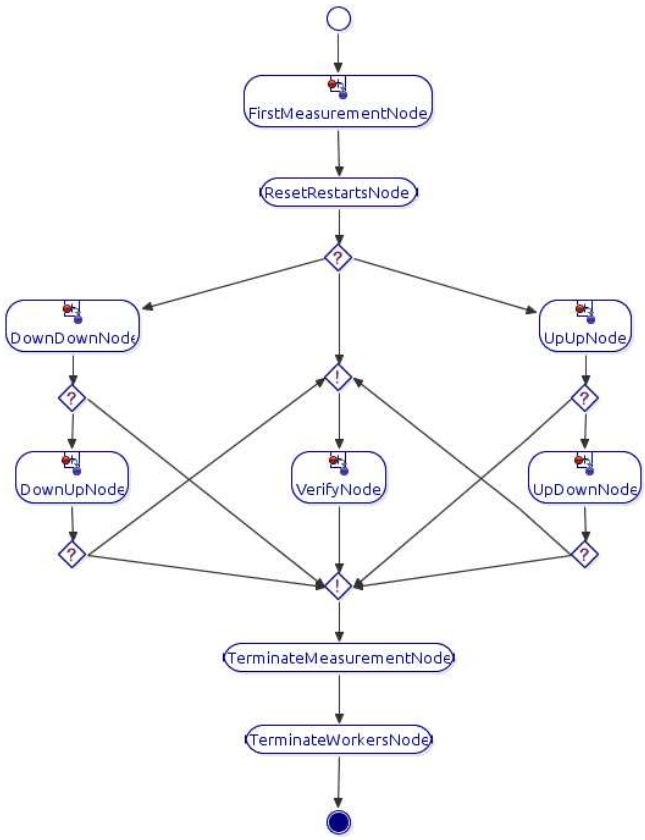


Figure 5.6: An overview of the process structure in SyLaGen. The top level in the hierarchy mainly defines sub processes which will invoke actions.

in process models. Thus more logic will be shifted into the model, which affects the variable *numberOfWorkers* that will be read *and* written by the process model. The definition of the other variables is the same as in section 5.2.2.2. Note that the JWT naming convention prescribes names starting with lowercase letters, which we follow here. The use of the variables in the top-level model can be seen in figure 5.7: The variables *tooHigh* and *tooLow* decide about the general direction to be taken for load generation. Apart from that, the abort criteria – user abortion and number of restarts – are considered between the single sub processes to determine if the measurement must be terminated.

5.2.3.3 Actions

While the goal to shift logic into the model had only little influence on the variables, it has more impact on the actor methods. Most of the actions are the same as described in section 5.2.2.3, but the adjustment actions are completely replaced by methods that only calculate the changes to be made and return the new value:

- Adjustment of the number of workers:

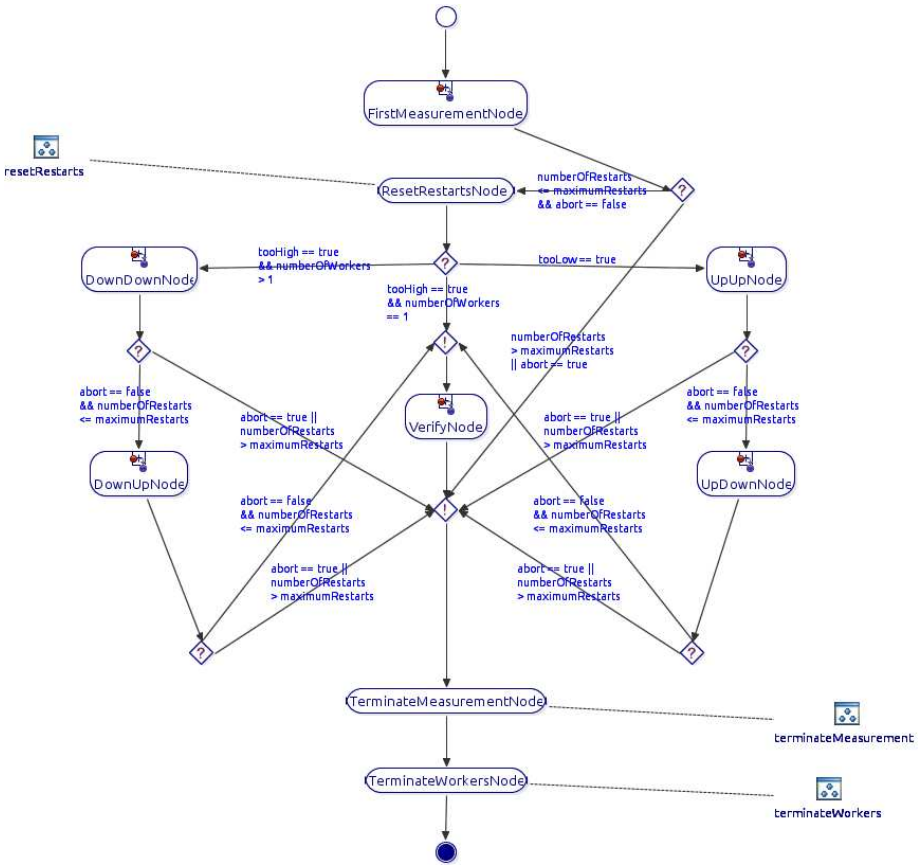


Figure 5.7: The use of variables in the top-level activity in SyLaGen. The guards decide if load is to be increased or decreased after the first measurement and monitor abort criteria.

- *increaseWorkers*: This method takes as parameter the current number of workers and returns a value increased by 1.
- *increaseWorkersByDistance*: This method takes as parameters the current number of workers and the distance and returns the sum of both.
- *decreaseWorkers*: This method takes as parameter the current number of workers and returns a value decreased by 1.
- *decreaseWorkersByDistance*: This method takes as parameters the current number of workers and the distance and returns the difference.

This results in structures for the sub processes as shown in figure 5.8: Each action is executed by an action node. The number of workers is increased by a separate action. The normal measurement can be seen in the left-hand column of actions. Two decisions are made inside this sub process: First, the measurement is repeated with an increased number of workers as long as the system under test is not overloaded, as denoted by the variable *tooLow*. Second, restarts are handled with separate actions shown in the right-hand column; the

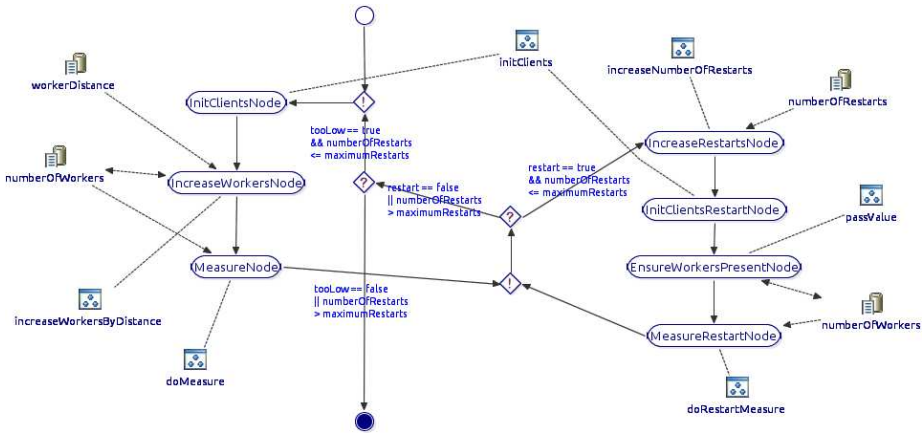


Figure 5.8: The upwards measurement (sub process of *UpUpNode*, cf. figure 5.7) in detail. In contrast to the state machine (cf. figure 5.5), actions require a separate node and are not invoked during transitions.

need for restarts is inferred from the variable *restart* and the number of restarts is validated by comparing *numberOfRestarts* to *maximumRestarts*.

In *IncreaseWorkersNode*, the action is connected to variables: *numberOfWorkers* is an input parameter and output parameter of this action as denoted by the bidirectional arrow. The *distance* is only an input parameter.

5.2.3.4 Implementation

According to the definitions given above, the variables interface is realized as shown in listing 5.4 with get and set methods for the variable *numberOfWorkers*.

```
1  public interface IMeasurementVariables
2  {
3    int getNumberOfWorkers();
4    void setNumberOfWorkers(int numberOfWorkers);
5    public int getWorkerDistance();
6
7    public boolean getTooHigh();
8    public boolean getTooLow();
9
10   public boolean getRestart();
11   public int getNumberOfRestarts();
12   public int getMaximumRestarts();
13
14   public boolean getAbort();
15 }
```

Listing 5.4: The variables facade interface for SyLaGen measurement strategies with get and set methods for the embedded process model.

The node *IncreaseWorkersNode* shown in figure 5.8 contains all features defined for action nodes, i.e. action labels, input parameters, and an output parameter. It is shown in listing 5.5: The class implements the interface `IActionNode` and its method `action`. The action *increaseNumberOfWorkers* is represented by the corresponding method call to the method parameter `actor`. The input parameters of the action are represented by calls to variable get methods whose results are passed as method parameters to the actor

method. The output parameter is represented by a call to the variable set method that takes as parameter the result of the actor method. Since this class is no decision node, it contains only one edge method annotated with `OutgoingEdge` that has no guard, which is represented by the statement `return true`.

```
1  public class IncreaseWorkersNode implements IActionNode<IMeasurementVariables,
       MeasurementModule>
2  {
3    @Override
4    public void action(IMeasurementVariables variables, MeasurementModule actor)
5    {
6      variables.setNumberOfWorkers(actor.increaseNumberOfWorkers(variables.getNumberOfWorkers()
           , variables.getWorkerDistance()));
7    }
8
9    @OutgoingEdge(MeasureNode.class)
10   public void toMeasureNode(IMeasurementVariables variables)
11   {
12     return true;
13   }
14 }
```

Listing 5.5: An action node represented in the program code with action method invocation, get method invocation for input parameters, and set method invocation for the output parameter.

A decision node from the same sub process is shown in listing 5.6: It implements the marker interface `IDecisionNode`. The methods in this decision node are edge methods with guards. The guards access the variables *tooLow*, *numberOfRestarts*, and *maximumRestarts* as get methods in the variables facade. The annotation `OutgoingEdge` refers to the target node of the edge.

```
1  public class UpUpDecisionNode implements IDecisionNode
2  {
3    @OutgoingEdge(UpUpMergeNode.class)
4    public void toMergeNode(IMeasurementVariables variables)
5    {
6      return (variables.tooLow() == true && variables.getNumberOfRestarts <= variables.
           getMaximumRestarts());
7    }
8
9    @OutgoingEdge(UpUpFinalNode.class)
10   public void toMeasureNode(IMeasurementVariables variables)
11   {
12     return (variables.tooLow() == false || variables.getNumberOfRestarts > variables.
           getMaximumRestarts());
13   }
14 }
```

Listing 5.6: Guards in edge methods of the embedded process model.

5.2.4 Evaluation

With this substantial information about the implementation of two embedded models for SyLaGen, we will now evaluate the approach for this case study with respect to the criteria defined in section 5.1.

5.2.4.1 Abstraction

The SyLaGen master consists of more than 230 Java classes. The measurement module that is the actor for the state machine has more than 600 lines of code

that mostly delegate tasks to other (more complex) modules. While it would be possible to write an algorithm that invokes the methods in the same way as the embedded models, it would not be comprehensible without much effort. This naive approach was pursued in prior versions of SyLaGen, but quickly abandoned in favor of a model-based approach. In contrast, both embedded models provide a meaningful abstraction with explicit states in the program and connections between them. This can be seen in the structural overviews for both models in figures 5.3 and 5.7. Even when equipped with information about variables and actions, the abstraction is certainly given since only variables and actions of interest are considered by the model. The application logic does therefore not influence the model.

It remains to discuss whether the abstraction level realized here is ideal. The variables *tooHigh* and *tooLow* aggregate complex information that cannot be part of abstract behavioral models. In theory it would be desirable to consider turnaround times directly in the models, but the aggregation is necessary to enable the modeling of complex application logic with the modeling domains chosen here. We can thus state that in such situations a compromise is to be made. However, this compromise is caused by the capabilities of the modeling domains of state machines and process models and is thus not under control of the approach presented in this thesis.

5.2.4.2 Understandability

The load strategies are not only some of the most complex algorithms in SyLaGen, but also direct many other modules und algorithms and control the execution over long periods of time during measurements. Thus, the abstract models clearly reduce the complexity since the application logic is partitioned in smaller portions that are invoked during actions or with variables. The relationships between the small portions are comprehensible by the static structures of the model. From our personal experience of working with SyLaGen, this increases the understandability of the whole application and eases maintenance, e.g. when the location of errors is to be detected. However, no user study has been performed that evaluates this systematically.

5.2.4.3 Accuracy

When a compromise is to be made between features of the abstract model and the requirements of application logic connected to the model, one can assume that this goes at the expense of the model's accuracy. However, we argue on the contrary here: Selic requires that accuracy is a "true-to-life representation of the modeled system's features of interest" [Selic, 2003]. This is clearly given with embedded models: While the abstract model may be influenced by the fact that the model is embedded in program code, this leads to a tight connection between model and implementation. Thus an embedded model is more accurate in terms of Selic's definition than a more abstract model that has only indirect connections to any implementations.

Thus, SyLaGen's embedded models represent the application as far as possible; the variables *tooHigh* and *tooLow* hide functionality of interest from the model, but are an accurate way to represent the complex internal state of the application to models like state machines and processes.

5.2.4.4 Predictiveness

The state machine for the exploration strategy is partly eligible for model checking. Basic validation, e.g. that no deadlocks occur, is possible. However, the state space is heavily influenced by the application logic and thus not completely under control of the model. For the process model this cannot be evaluated since no appropriate functionality exists in JWT.

This case study thus gives a hint that predictiveness is possible to a certain degree, but this cannot be generalized here.

5.2.4.5 Inexpensiveness

Considering the fragments created during modeling, the development of both embedded models is certainly not inexpensive compared to manual development: The state classes, contract classes, and node classes of both models are more complex than a compact algorithm in a method body would have been. In the models, the embedded model definitions require e.g. that all nodes have names, which is often neglected when a model is created at an abstract level only.

But, maintenance of a system is even more important than its initial creation: Our personal experience with SyLaGen's embedded models for almost four years now is that program comprehension and error detection are much easier than they would be in a large method body. This is even true without tools since the program code structures following the pattern already provide an entry point for navigating in the program code. In addition, embedded models are to be supported by appropriate tools. These can also simplify the creation of software containing embedded models since redundant information can be transformed between code and models.

We thus think that software with embedded models is slightly more expensive to create. However, maintenance and analysis of the resulting software is less expensive, especially with appropriate tools like transformations to UPPAAL. This conforms to our personal experience with developing SyLaGen, but more detailed studies considering the creation and maintenance of non-trivial systems from the beginning will be necessary to evaluate this.

5.3 Case Study "Game Design with Greenfoot"

In this case study we describe how embedded state machines are integrated in the game-oriented IDE Greenfoot [Henriksen and Kölling, 2004] which allows to create small visual simulations with little effort for learning purposes. The requirements of the simulation to consider are introduced in section 5.3.1 and the realization with embedded models in section 5.3.2. Based on this, we describe model verification with this approach in section 5.3.3. A user study is described in section 5.3.4 before we evaluate the case study in section 5.3.5.

5.3.1 Requirements

Here we introduce the requirements for a sample Greenfoot simulation and the communicating entities therein. For each entity type contained in the Greenfoot simulation we describe the rules and variables that control its behavior

and also the application logic that is specific to Greenfoot, mainly concerning graphical appearance and interaction with the environment.

The design of Greenfoot can roughly be summarized as follows: A so-called "world" provides the background for the simulation. This includes the size and graphical appearance of the visual background, and allows to access all entities currently available in this world. Single entities – called "actors" – can move, change their appearance, and gather information about their vicinity in the world. Apart from that, they can implement almost arbitrary application logic. Actions in Greenfoot are performed in so-called cycles. In each cycle, a pre-defined method named `act` of each entity and the world is called. In this method, the objects must determine their current state and decide which actions to take. If and how often these cycles are called is controlled by the user and thus not influenced by the objects running in the simulation.

5.3.1.1 Concept for "Stateful Wombats"

The Greenfoot distribution is equipped with several examples explaining the functionality. The default example is a world with two kinds of actors, wombats and leafs, which interact in the way that wombats eat leafs. For this purpose, wombats move through the world randomly. Whenever they encounter a leaf at their current position, they "eat" it by removing it from the world.

For our purpose we modify this example to emphasize the fact that actors are stateful and communicate with each other in well-defined situations: Wombats have the desire to eat leafs until they are saturated; in this case their state changes and they digest everything they have eaten. Leafs are, when eaten, not completely removed, but are considered to be "not grown". They recreate afterwards and grow again, but need a certain amount of time for this. The concept is shown in figure 5.9: The wombat at the top is currently hungry and thus walking upright. In contrast, the wombat at the left is saturated and thus lying on its back to digest. The large leafs are fully grown and can thus be eaten. The small leafs are half-grown, and some leafs are currently not visible at all because they have been eaten a short time ago.

These requirements are constituting a simple game and are as such not complicated. However, they define different states for the actors and assign different visual appearance and possible (inter)actions to these states. This reveals the need for appropriate models to represent the requirements. In this case, state machines can be used to represent the behavior of the actors:

- A state machine corresponds to a single entity in the simulation. The fact that the entities must know about their current state complies to explicit states of a state machine.
- The decisions that are made based upon a certain state can be represented by outgoing transitions.
- The actions that modify the system state in transitions correspond to action labels.
- A system of communicating state machines corresponds to a world containing different entities that can interact.

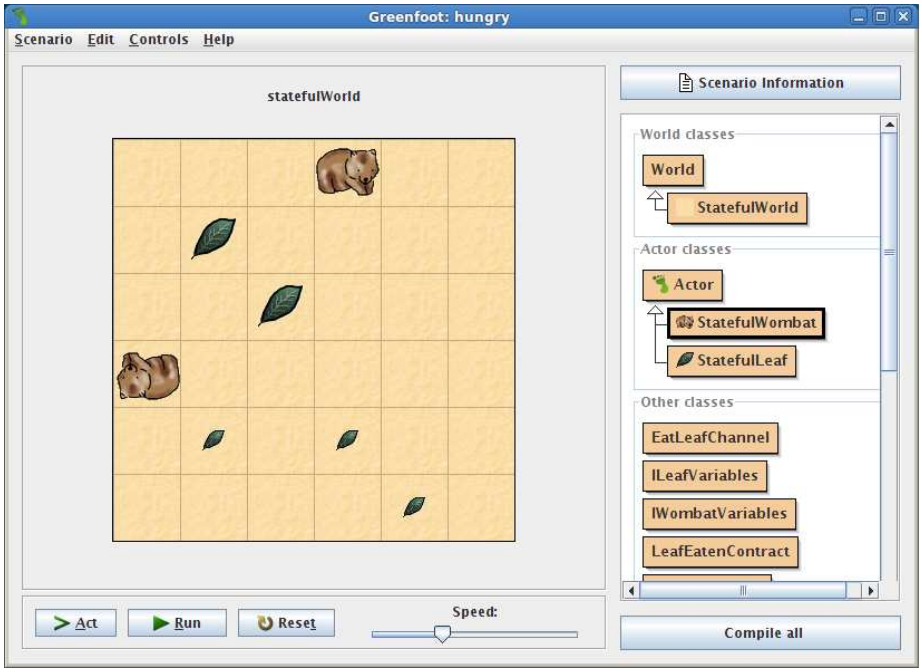


Figure 5.9: The Greenfoot actors in the use case. The wombat at the top is currently in the "hungry" state. The large leafs are in the state "grown", the small leafs in the state "halfgrown". Some leafs are currently not visible at all because they are in the state "eaten".

Based on this assumption, we will now define appropriate state machines for the actors in detail in sections 5.3.1.2 and 5.3.1.3.

5.3.1.2 Wombat Requirements

Wombats act based on the following rules:

- They determine if they are hungry by the number of calories they have eaten.
- They are hungry with less than 100 calories. When wombats are hungry, they appear walking randomly through the world.
- When a wombat eats a leaf, it gains 20 calories.
- When wombats are saturated, i.e. the number of calories equals 100, they stop eating and digest until the number of calories reaches 0.
- When saturated, a wombat digests 5 calories in every cycle. As long as a wombat digests, it is laying on its back and does not move.

Thinking in state machines, this behavior can be represented with two states, *Hungry* and *Saturated*. The behavior, especially the change between states, is controlled by a variable *calories*. When wombats are hungry, they try

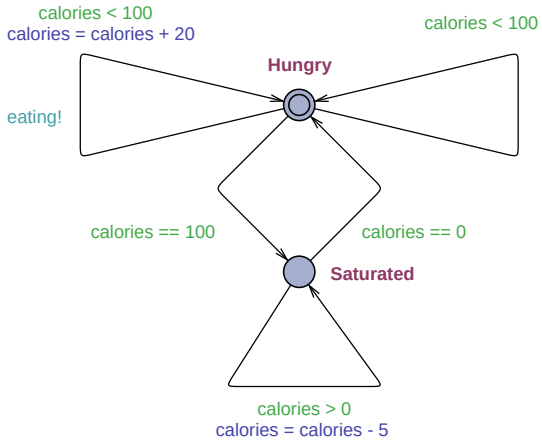


Figure 5.10: The state machine controlling the wombat's behavior.

to communicate with available leaf state machines. This is represented by a channel named *eating*, on which the wombat state machine is sending. The resulting state machine is shown in figure 5.10 and defines the following behavior:

- The initial state of a wombat is *Hungry*.
- When the wombat is hungry and able to send on the channel, it gains 20 calories. The guard for this is `calories < 100` and the update is `calories = calories + 20`. Sending on the channel is indicated with `eating!`.
- When the wombat is hungry and not able to send, the same guard applies, but the number of calories is not changed, so that no update occurs. In this case, application logic must be called that modifies the wombat's position.
- When the wombat has gained 100 calories (guard `calories == 100`), its state changes to *Saturated*. During this transition, the image of the actor must be changed by appropriate Greenfoot-specific application logic.
- As long as the wombat is saturated and has more than 0 calories (guard `calories > 0`), it digests and loses by this means 5 calories (update `calories = calories - 5`).
- When the number of calories has reached 0 (guard `calories == 0`), the state is changed to *Hungry*. Again, the image of the actor must be changed.

5.3.1.3 Leaf Requirements

The behavior of the Leaf actors is controlled by the following rules:

- The growth of a leaf is specified by an abstract value with the range $[0, 100]$.

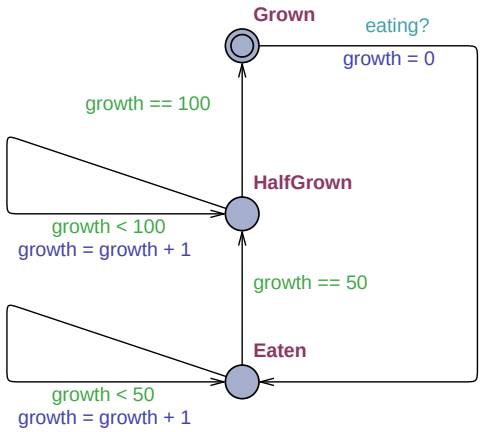


Figure 5.11: The state machine controlling the leaf's behavior.

- When a leaf is added to the world, it is fully grown, i.e. has a growth of 100. When a leaf is grown, it appears in the simulation in its normal size.
- When a leaf is eaten, its growth is set to 0. From now on, leafs start to grow by 1 in every cycle. They do not appear visually at all.
- When the growth reaches 50, the leaf is considered half-grown. It appears in a smaller size in the world again, but continues to grow until its growth reaches 100 again.

In the resulting state machine, three states exist: *Grown*, *HalfGrown*, and *Eaten*. Behavior and change between states are controlled by the variable *growth*. When leafs are grown, they are open for communication with wombats by receiving on the channel *eating*. The behavior is modeled with the state machine shown in figure 5.11:

- The initial state of a leaf is *Grown*.
- This state is changed to *Eaten* when the leaf receives on the channel as specified with `eating?`. In this case, the growth is changed with the update `growth = 0`. During this transition, the application logic is required to make the leaf invisible.
- When the leaf is eaten, it grows by 1 in each cycle until the growth reaches 50. This is realized by a transition with the guard `growth < 50` and the update `growth = growth + 1`.
- When in the same state the guard `growth == 50` is applicable, the state is changed to *HalfGrown*. The image of the actor must now show a small leaf.
- When the leaf is half grown, it continues to grow as long as the growth is less than 100 (guard `growth < 100`, update `growth = growth + 1`).
- When the guard `growth == 100` is applicable, the leaf changes to the state *Grown* and shows its image in full size.

5.3.2 Implementation

These domain specifications can be implemented with embedded state machines directly. First, the execution mechanism is integrated in the Greenfoot simulation. We use the world for this purpose since it has access to all actors in the simulation and has also an `act` method that is called in each cycle. The world is shown in listing 5.7: In the constructor, the world size is initialized with a specific size in line 7. In line 8, the class `EmbeddedStateMachineSystem` from the execution framework is instantiated. It provides an execution environment for communicating state machines as described in section 4.1.3.2. The channel is then added to the system in line 9. In each cycle, the method `next` of the execution framework is invoked which will select and fire the next transition for all state machines.

```
1  public class StatefulWorld extends World
2  {
3      private final EmbeddedStateMachineSystem system;
4
5      public StatefulWorld()
6      {
7          super(6, 6, 60);
8          system = new EmbeddedStateMachineSystem();
9          system.add(new EatLeafChannel());
10     }
11
12     public void act()
13     {
14         system.next();
15     }
16
17     // ...
18 }
```

Listing 5.7: The source code of the Greenfoot world. The state machine system controls the actors in the world that use embedded state machines.

The wombat implementation is shown in listing 5.8 with the variables interface and the actor class. It fulfills the following purposes: (1) Instantiating a state machine and adding it to the state machine system (method `addedToWorld` defined by Greenfoot and called from the framework after instantiation of the object), (2) providing the variable method implementation (`getCalories`), (3) providing actor methods that are called during transitions (`eat`, `search`, `digest`, `beginDigest`, `becomeHungry`), (4) providing a method that is called from the channel (`canEat`).

```
1  public interface IWombatVariables
2  {
3      int getCalories();
4  }
5
6  public class StatefulWombat extends Actor implements IWombatVariables
7  {
8      private int calories = 0;
9
10     protected void addedToWorld(World world) {
11         EmbeddedStateMachineSystem s = ((StatefulWorld)getWorld()).getSystem();
12         s.add(WombatHungryState.class, this, this, IWombatVariables.class);
13     }
14
15     public void eat() {
16     { calories += 20; }
17
18     public void search()
19     { /* Random selection of new location occurs here */ }
```

```
20
21      public void digest()
22      { calories -= 5; }
23
24      public int getCalories()
25      { return calories; }
26
27      public void beginDigest()
28      { setRotation(180); }
29
30      public void becomeHungry()
31      { setRotation(0); }
32
33      public boolean canEat(StatefulLeaf l)
34      { return (getIntersectingObjects(StatefulLeaf.class).contains(l)); }
35  }
```

Listing 5.8: The classes representing the wombat variables and actor.

The method `search` contains application logic specific to Greenfoot for moving inside the world. The methods `beginDigest` and `becomeHungry` also contain Greenfoot-specific application logic for rotating the appearance of the actor. The method `canEat` is provided by the wombat class to be invoked by the channel. The reason for this is that the channel is created in the world and valid for all actors, however, it must determine if certain wombat actors are at the same position as a leaf. The functionality for this is provided in the Greenfoot API of actors only. The method `canEat` invokes in line 32 the method `getIntersectingObjects` which is inherited from the superclass belonging to Greenfoot. It takes as parameter the class definition of objects that are of interest, in this case leafs, and returns a list of objects at the current position.

The leaf actor contains application logic related to its visual appearance. Apart from that, it only manages the variable `growth`. The states and transitions are implemented following the requirements given in section 5.3.1 and the approach introduced in the load generator case study. But, these structures are complemented by interaction mechanisms in this case study: The channel controls interaction between state machines and must for this purpose be able to invoke application logic. So, the abstract model definition must also be supplemented with a connection to application logic that can access and interpret the specific Greenfoot environment. In UPPAAL, the channel does not contain any additional semantics; it is always activated when a sending wombat and a receiving leaf exist. However, in the Greenfoot world, the channel fulfills a certain purpose: It must determine if a wombat can eat a leaf by testing if two appropriate actors are at the same location. For this purpose it must contain application logic that is able to access the actors of interest.

The channel implementation is shown in listing 5.9: The senders and receivers are passed to the method `enable` by the execution framework. This method identifies wombats and leafs and invokes application logic on the actors with the method `canEat` to determine if they intersect. When a matching pair is found, it is returned in an object of type `ChannelSelection`.

```
1   public class EatLeafChannel implements IChannel
2   {
3       public ChannelSelection enable(Set<Object> senders, Set<Object> receivers)
4       {
5           for (Object s : senders) {
6               if (s instanceof StatefulWombat) {
7                   StatefulWombat wombat = (StatefulWombat)s;
8                   for (Object r : receivers) {
```

```
9                   if (r instanceof StatefulLeaf) {
10                      StatefulLeaf leaf = (StatefulLeaf)r;
11                      if (wombat.canEat(leaf))
12                          return new ChannelSelection(s, r);
13                  }
14              }
15          }
16      }
17      return null;
18  }
19  }
```

Listing 5.9: The source code of the channel realizing the interaction between wombats and leafs.

5.3.3 Verification

The models used in Greenfoot control the state space completely. None of the values used in guards and updates is aggregated. Thus, meaningful analysis of the models is possible with appropriate tools. We consider verification of the model first and then verification of program code connected to the embedded model.

5.3.3.1 Model Verification

UPPAAL provides a simulator that visualizes each step of a running state machine and shows current states, activation of transitions, and current variable values. Since the values are not aggregated, the simulation is significant with respect to the embedded model. In addition, the verification feature of UPPAAL can be used for thorough model checking of the state machines and their communication.

UPPAAL's verifier is shown in figure 5.12 with examples for the state machine of this case study. The examples include model checking of the following aspects:

Reachability The basic verification is that all states are reachable and that no deadlock occurs. The invariant query `A[] not deadlock` is appropriate for this purpose, with `A[]` denoting that the invariant must hold for all states. This allows to detect guards causing deadlocks, e.g. `calories != 0` instead of `calories == 0`.

Value Ranges The example introduced above relies on the two variables `calories` and `growth` for wombats and leafs, which have well-defined value ranges. The verifier can thus, for example, be used to ensure that both variables never exceed the range $[0, 100]$. This is achieved with the invariant query `A[] w.calories >= 0 && w.calories <= 100`. Errors detected with this query are e.g. usage of wrong operators in guards and updates, for example with the expression `growth <= 100` instead of `growth < 100`.

Liveness It is desirable to verify that some conditions will hold eventually, i.e. at some point in time. For example, the wombat must be able to reach the state *Saturated*. This would not happen if the update `calories = calories + 20` was missing. The query `E<> w.Saturated` is appropriate to detect such errors.

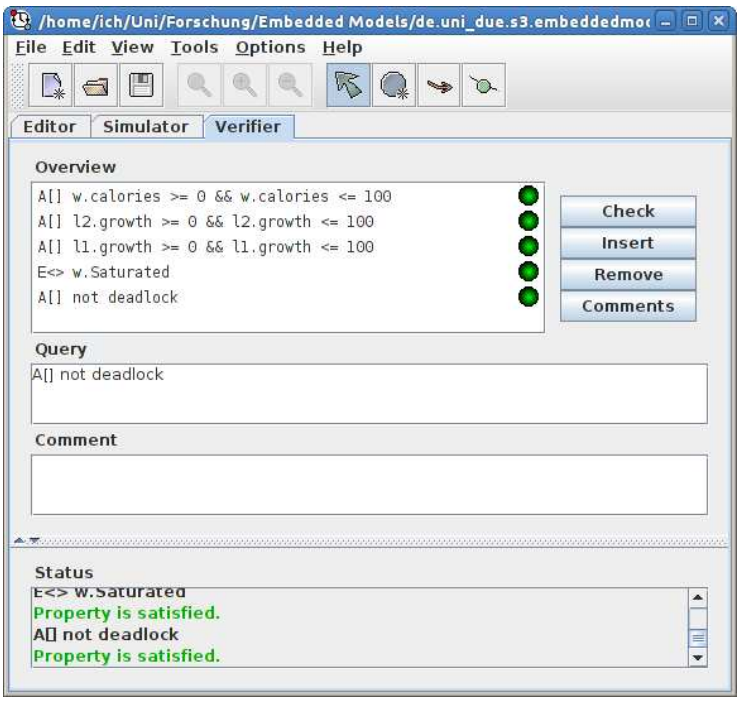


Figure 5.12: The UPPAAL verifier used to ensure the range for the variables `calories` and `growth`.

We can thus state that this case study is appropriate for thorough analysis of the models representing the Greenfoot simulation, while at the same time an integration in the specific context of Greenfoot and an execution in the visual environment is possible.

5.3.3.2 Code Semantics Verification

With respect to the entry points for verification of application logic connected to the embedded state machines (cf. sections 3.1.5.2 and 4.4.3.2), we now give an outlook which aspects of the application logic can be verified within this case study. In this case, the source code structures are appropriate for static analysis that can prove the assertions. This applies to the following classes of fragments:

Expected changes for updates The UPPAAL models defines two updates for the wombat's variable `calories` (cf. figure 5.10). They apply to the methods `eat` and `digest` (cf. listing 5.8). The following slices are concerned (cf. figure 5.13):

1. The variable method `getCalories` returns the member variable `calories` of the class `StatefulWombat`. As determined by the semantics of the program code pattern for embedded state machines, this member variable influences the state space when it is accessed by the variable method.

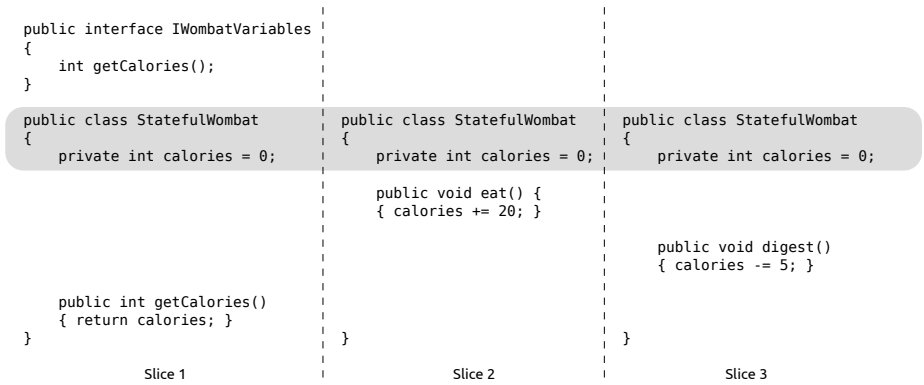


Figure 5.13: The slices for code semantics verification in the Greenfoot example (cf. listing 5.8). The grey bar marks an overlapping part considered by all slices.

2. According to the model, the actor method `eat` must increase the value of the state machine variable `calories` by 20. Based on the first slice, we can state that the member variable `calories` must be increased by 20. In this case, the statements in the method body can be verified to fulfill this requirement.

3. The same rules as for the second slice apply to the method `digest` decreasing the value of the variable `calories` by 5.

The expected changes for updates can thus be verified.

Reproducible reading The variable `calories` is read from the program code by the implementation of the variables interface method `getCalories`. The wombat's state machine model requires that the variable be changed during actions, but not during read access. Static program code analysis considering the control flow can be applied to a slice with the method `getCalories` being the entry point. Considering the implementation in listing 5.8, it can be verified that the variable is not modified here and that the read access is thus reproducible.

Exclusive writing The updates defined for the wombat require certain modifications to the variable `calories`. The wombat's model does not expect any other changes to the variable. Thus it makes sense to verify that it is not modified outside the slices for actions. Program code analysis must for this purpose detect invalid writing access to the variable `calories` outside the methods `eat` and `digest`.

Summary The Greenfoot case study is eligible to exhibit the entry points for verification of application logic code connected to embedded models. In case of the wombat, the required modifications to the state space can be ensured and invalid modifications can be detected. However, appropriate tool support for this purpose must be considered future work.

5.3.4 User Study

The approach of embedded state machines in Greenfoot has been used experimentally in a master's course of an information science program at University of Duisburg-Essen. One day of the block course "Application of Formal Methods" (8 hours) was dedicated to working with embedded state machines in Greenfoot. 11 students participated who had already gained experience with UPPAAL in other courses. They had to perform three exercises which all allow for validation of the embedded models approach.

5.3.4.1 First Exercise: Design Recovery

After a short introduction to Greenfoot, the students were given a Greenfoot program that provides exactly the same functionality as the Stateful Wombats, but which was written manually. They had to analyse the program and create a documentation in terms of UPPAAL models. With this task we wanted to sensibilize them for the connection between models and (Greenfoot) program code. The students were allowed to exchange questions and partial solutions among each other to stimulate discussions.

After 2 hours, 7 of 11 students had extracted models that contained a large part of states and transitions as well as the variables for both state machines. 3 more students achieved this for only one of the state machines. A short reflection on the exercise at the end revealed that design recovery with models was more complicated than students had imagined since practical experience with this topic was missing, which was what we expected.

5.3.4.2 Second Exercise: Embedded State Machine Implementation

In the second exercise, the concept of embedded state machines was explained. The students were given the complete UPPAAL model of Stateful Wombats as described in section 5.3.1 and a comprehensive list of embedded state machine pattern fragments with examples. In addition, they were provided with the transformation tool to UPPAAL (cf. section 4.4.3), so that they could validate their models.

Surprisingly, the concept was understood almost immediately by the students. The creation of the complete model was hindered by the fact that all state and contract classes had to be written manually, which is not supported in a usable way in Greenfoot compared to more sophisticated IDEs like Eclipse. Thus it was more time-consuming than expected. The creation of the program code was strongly supported with the UPPAAL transformations that were used by the students frequently so that the implementation approach of most students was very focused. After 3 hours, only 2 students had problems at the conceptual level. The other implementations were not complete due to the difficulties with the Greenfoot tool, but were partly working including states, transitions, variables, actions, and Greenfoot-specific application logic.

5.3.4.3 Third Exercise: Model Verification

So far the students had gained experience in analyzing program code and implementing embedded state machines. Now they were given a second Greenfoot simulation implemented with embedded models. The simulation "Space"

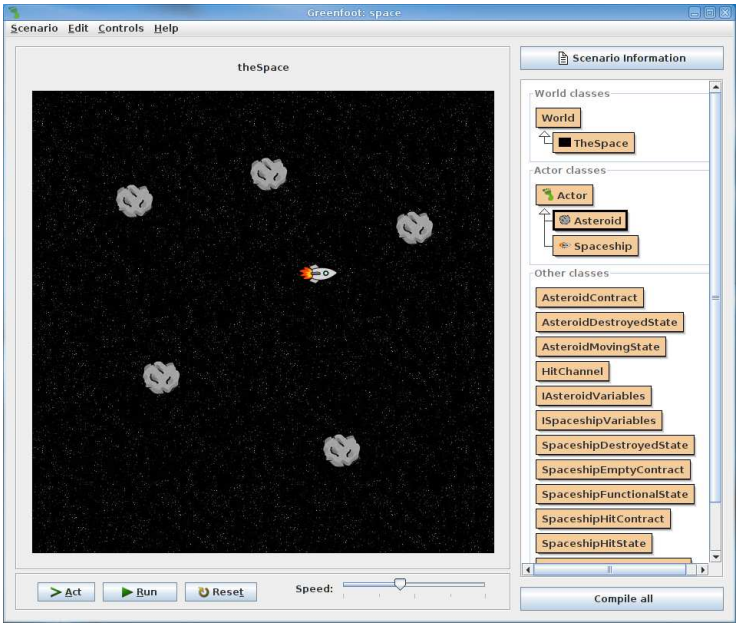


Figure 5.14: The "Space" simulation in Greenfoot. This application was given to the students with the task to detect an error at the model level.

is shown in figure 5.14: A space ship moves through the world controlled by the user pressing arrow keys. When the ship is hit by an asteroid, it cannot move for a certain time frame for repairing. When it is hit again in this time, it is destroyed. The corresponding UPPAAL model can be seen in figure 5.15.

The task given to students was to find an error in this simulation by using the verifier: The guards `energy < 100` and `energy == 100` had been swapped. Thus, collisions had no effect at all. This can be discovered with the UPPAAL query `A[] s.SpaceshipFunctionalState imply s.energy == 100` that makes an assertion for a single state. However, to our surprise, most students discovered the error in a few minutes just by considering the visual UPPAAL model after extracting it from the source code.

5.3.5 Evaluation

The embedded models in this case study are different than those in SyLaGen: They are less complex, but control the state space completely and are thus more appropriate for verification. This influences the evaluation with respect to the criteria defined in section 5.1.

5.3.5.1 Abstraction

The nature of UPPAAL applications – actors performing tasks based on their state and communicating with other actors in a delimited world – matches the semantics of communicating state machines to a certain degree. Some of the aspects of Greenfoot actors can thus be represented in such models. By this means, a reasonable abstraction is achieved: The state machine models focus

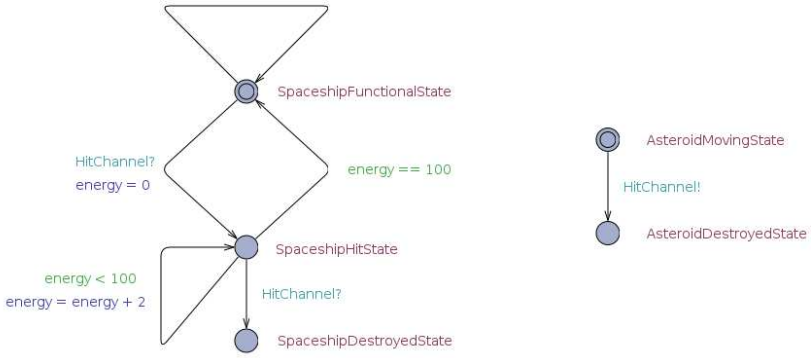


Figure 5.15: The "Space" simulation modeled in UPPAAL. In the embedded model given to the students, the guards `energy < 100` and `energy == 100` were swapped.

on states and variables and enable meaningful verification of them. The action labels are not considered in the UPPAAL model and are thus abstracted from. Greenfoot simulations are thus appropriate for modeling with state machines and embedded models connect them to the Greenfoot environment with well-defined interfaces.

5.3.5.2 Understandability

The understandability of embedded state machines in Greenfoot can be evaluated by considering the user study. First, we can state that the students were able to comprehend the rules for embedded models quite fast and apply them to create simulations as described in section 5.3.4.2. Second, the connection between the executable simulation in Greenfoot and an abstract model was understood by the students fast as described in section 5.3.4.3: The deduction from the error in the "Space" simulation to the errors in the guards was drawn in minutes by most of them. The user study thus confirms that understandability is given in this context.

5.3.5.3 Accuracy

In this case study, the variable values were not aggregated. Instead, they were used in guards and updates directly so that the state space of the model can be considered a subset of the state space of the application. The behavior of the Greenfoot actors with respect to states and variables is thus reflected in the formal model very accurately.

5.3.5.4 Predictiveness

Since the state space is in this case study completely under control of the model, the verification concept described in section 5.3.3 shows that the models can be analysed and checked soundly with respect to states and variables and without distracting influence from the interfaces to Greenfoot. This is supported by the results of the user study confirming that the students were able to identify

errors. Thus, a high degree of predictiveness is given with embedded state machines in this context.

5.3.5.5 Inexpensiveness

Reflecting the results from the user study, we can state that the expensiveness of developing with embedded model correlates with the availability of appropriate tools. The implementation of the model was not supported by a design tool and thus cumbersome for the students. But, the availability of a tool for visual representation focussed the development. We expect that the problems can be resolved completely with the integration of a visual embedded state machine editor in Greenfoot since no conceptual issues occured. The error detection by students described in section 5.3.4.3 was supported by tools, i.e. the transformation tools and UPPAAL itself, so that design recovery and model analysis were performed by the students very fast. This indicates that maintenance of software is less expensive if embedded models are used with appropriate tools.

5.4 Overall Evaluation

Considering the results of the evaluation of both case studies, we can summarize the results as follows.

5.4.1 Abstraction

In both case studies, the embedded models allow for abstraction from the detailed application logic. In SyLaGen, the degree of aggregation had to be high since state machines are not appropriate for representing the complex examination of measurement results. In Greenfoot, more details of the application were considered in the model due to the strong correlation of simulations and state machine systems. While the SyLaGen state machines are mainly used for documentation, the Greenfoot state machines are appropriate for sound verification. The degree of abstraction thus depends on the modeling domain and the system to model; the embedded models approach supports the different objectives of both case studies.

5.4.2 Understandability

The objective to achieve better understandability for applications containing embedded models is reached. This is not only our personal experience with SyLaGen, but also confirmed by the user study: The application itself as well as the relations between the different abstraction levels were clearly comprehensible with less effort.

5.4.3 Accuracy

As we argued for SyLaGen, accuracy requires that models reflect the characteristics of the actual software and not only that of an abstract model itself. While it is open for discussion if the abstract models and their domains are the best

choice for modeling the applications, the embedded models introduced here do not compromise the accuracy: In both case studies, the limits of modeling were caused at the model level; the interfaces of the embedded models realize these limits and connect application logic to it.

5.4.4 Predictiveness

Embedded models support predictiveness depending on the capabilities of the modeling domains regarding the software to develop. While variable values in SyLaGen must be aggregated and thus have limited significance, Greenfoot models can be verified soundly. In both case studies, no influence of the interfaces on predictiveness can be observed.

5.4.5 Inexpensiveness

In the evaluation of the SyLaGen and Greenfoot case studies we reached the same conclusion: The costs of developing with embedded models depend on the tool support. This is especially true for the creation of applications where the necessary fragments have to be created according to the pattern definition. The difference between situations with and without tools were observable in the user study. However, even in this case the students were able to grasp the concept at the source code level and create valid implementations, so that tools are not mandatory.

5.5 Chapter Summary

In this chapter the embedded models approach was evaluated with two case studies. The SyLaGen case study in section 5.2 introduced a situation where embedded models are used in a non-trivial software product with complex application logic that was only partly representable by the abstract models. In this case, two different modeling domains were used that are similar, but focus on different aspects. The Greenfoot case study (cf. section 5.3) put emphasis on smaller projects that can be modeled and thus verified in detail.

The evaluation was performed with criteria defined in section 5.1. *Abstraction* and *predictiveness* are given with both models, but only influenced by the abstract models and not the approach presented in this thesis. *Inexpensiveness* strongly depends on tool support. The evaluation result for these criteria is true for almost all model-based approaches in software development; the important conclusion for this thesis is that no impact of embedded models could be detected that impairs one of them in the case studies.

In contrast, *accuracy* in terms of the "true-to-life representation" is in our opinion strongly supported by embedded models since the overall application is tightly connected to the model specifications. Based on this, *understandability* could be verified in the user study to be given across different abstraction levels during software development. By this means, the criteria introduced in section 5.1 are satisfied.

Chapter 6

Impact and Future Work

In this chapter we give a short outlook to future work and possible impact of the embedded models approach.

First, starting points for additional evaluation are introduced in section 6.1. So far, only behavioral models have been considered; additional modeling domains of interest are stated in section 6.2. When different modeling domains are of interest, interaction of different embedded model domains must be considered as will be sketched in section 6.3. This leads to thoughts about meta models for embedded models in section 6.4. Finally, possible consequences for future software description languages are discussed in section 6.5.

6.1 Evaluation

The case studies described in chapter 5 already provide evaluation results that are promising. However, they are both specific: SyLaGen is a non-trivial real-world application, but is only partly adequate for verification. The Greenfoot-based applications can be verified, but are too small to have significance for the development of large applications.

Thus, additional evaluation makes sense with an application that is large *and* suited for modeling with state machines in a way that the state space is represented in the model in more detail. Such evaluation should be performed using the tool suite introduced in section 4.4, thus reducing the complexity for developers.

In further case studies, understandability will be of interest. To evaluate this more thoroughly, the evaluation must consider existing approaches for *program comprehension* [Storey, 2005]. We propose to apply the *integrated code comprehension model* [Mayrhauser and Vans, 1995] to this situation. The approach for this will be sketched now.

6.1.1 Program Comprehension Approach

In the integrated code comprehension model, a *domain model* describes the situation when programmers have knowledge about a certain application domain. Programmers act in this case on certain assumptions or hypotheses to reach

specific objectives. They can thus analyse a system based on their expectations and enhance their knowledge with specific information. This strategy is classified as *opportunistic*, i.e. not based on a structured approach to understanding of code. Support can be given by abstraction utilities like visualization tools, documentation, and formal models. This leads to a process where general conditions are determined at abstract layers and then gradually supplemented with program-specific information.

The *program model* applies when programmers have no domain knowledge. In this case they build their own mental representations and abstractions which are technical and based on control flow in the first place and do not consider program semantics immediately. The overall picture is developed over time when single well-understood pieces are aggregated and connections are found. Tools of value may be debuggers or slicers. The familiarization process is in this situation considered *systematic* (in contrast to opportunistic) because missing domain knowledge prevents the programmer from making assumptions beforehand. When a mental representation is built from the program model, programmers elaborate it into a *situation model*. This elaboration leads to interconnection of understood pieces and may be systematic or opportunistic. This bears the risk of misunderstandings: While single programming constructs are easy to understand, their side effects and impact on whole applications are hard to determine by programmers with little experience regarding a specific application.

6.1.2 Program Comprehension with Embedded Models

We assume that embedded models can at development time support the program comprehension process as defined in the domain model component for two reasons: First, the model definition that is embedded can be the domain knowledge a programmer already has. In this case he can easily gain an overview of the program structure, as was indicated in the user study. Second, the well-defined interfaces provide structured entry points into the arbitrary program code. The programmer can thus navigate in the well-defined source code structures and examine arbitrary application logic step-by-step.

For the program and situation model components, the value of embedded models is not that easy to define because knowledge regarding model semantics cannot be assumed. However, the program comprehension theories suggest that program understanding is likely to happen in a more systematic way because of the programmer's uncertainty. Embedded models can here be of use if we can consider knowledge about the concept of embedded models (not of specific embedded model domains) to be part of the basic knowledge of development techniques that programmers are supposed to have. In this case, the systematic familiarization could include the search for possible embedded models. This would require appropriate documentation inside the source code since a multitude of embedded models is imaginable that feature completely different structures.

By this means, starting points for a systematic evaluation of embedded models with respect to understandability and program comprehension are given. Appropriate studies must consider the relation between domain knowledge and familiarization with specific program code. When correlations

are detected, the effects of program comprehension as described in the integrated code comprehension model are supported by embedded models.

6.2 Additional Modeling Domains

In this thesis, two classes of behavioral models have been considered. This was intentional since many modeling approaches focus on static models like class diagrams, which are easy to represent in program code, but neglect semantical connections to application logic. However, the embedded models approach is not limited to behavioral models. We will sketch an approach for the application to component models now.

The basic features of component models are the representation of modules, their interfaces, interaction, hierarchies, and assembly [Müller et al., 2010]. For such features, the elements of an embedded model definition (cf. section 3.1) can be defined as follows:

- *Model Specifications*: The structural elements of such component models can be directly represented with the meta model described in section 3.1.2.
- *Program Code Pattern*: The definition of program code patterns has to be extended: While interfaces can be described with the program code pattern elements introduced so far, modules are not completely representable at the level of classes in a programming language. In addition, module description files and library files (e.g. Java's JAR files) must be included [Müller et al., 2010]. The meta model in section 3.1.3 has to be extended for this purpose. The result is that not only program code is considered, but also other configuration files which are used by the platform and describe the program code.
- *Execution Semantics*: The main difference to behavioral models concerns the execution: Component models usually control the life cycle of modules, e.g. in OSGi [OSGi Alliance, 2005a], but do not access their content in detail. The "execution" of component models is thus limited to the life cycle.
- *Interfaces*: When components provide services, their description can be considered by the component model. In this case, the interface description serves as an abstraction to the contents of the component, which are considered a black box. This corresponds to the semantics of interfaces in the embedded models definition.
- *Transformations*: When the program code pattern is defined completely, the model can in general be transformed to component modeling tools. As described for the embedded model implementations so far, appropriate tools must be selected or developed according to the focus of the model.

The implementation of this concept with precise model specifications and an extended program code pattern will be an interesting extension of the approach in future work.

6.3 Interaction of Embedded Models

The case studies introduced in chapter 5 use only one modeling domain at the same time. However, as sketched in section 3.3, complex software systems can consist of different embedded models. Due to the nature of embedded models, an interaction between the models is *always* possible – the integration of the models in the program code does not require a common meta model, but allows developers to faciliate interaction with appropriate algorithms. However, a consideration of different embedded models at higher levels of abstraction is desirable, too. When different models are to be coupled, model-based approaches for interaction will be developed as future work. However, this will never be a general approach, but only a pair-wise coupling of models, as long as no common meta model is used (cf. section 6.4).

6.4 Meta Modeling

Many model specifications are based on meta models that cover multiple modeling domains. This is true e.g. for UML and EMF models. Instead of creating embedded model definitions for each modeling domain separately, it will be interesting future work to evaluate if meta models can be embedded in program code, too. This seems reasonable since they are covered by our meta model (cf. section 3.1.2). However, two issues will be non-trivial to handle:

- *Efficiency*: The embedded models described in this thesis focus on efficient implementation. Embedded meta models may be inappropriate for such optimization so that the number of fragments to create for implementations may be considerably larger.
- *Execution*: When a meta model is embedded in program code and different specific modeling domains are covered by it, execution semantics will be necessary that cover all modeling domains, too. However, no common execution semantics exist for meta models like UML and EMF – instead, they are defined in different ways for different purposes [Mellor et al., 2004; Soden and Eichler, 2009]. This is not a problem to be solved at the level of embedded models since it concerns the abstract model specifications. Thus, a general solution would have to be found – either with separate execution semantics being coupled, or with execution semantics that cover different modeling domains.

6.5 Software Description Languages

The embedded models approach closes a gap between different notations used for different purposes in software engineering (cf. section 3.2). However, one can argue that the program code is misused for other purposes than it was invented for. While this is not unusual (cf. section 2.2.1), future work may consider more elegant solutions for this problem. One possible solution can be a removal of the semantic gap between model notations and program code by creating a common language for both purposes, whose principles will be sketched here.

Embedded models couple models and program code by the following means:

1. Model elements represented in the code can be connected to arbitrary implementations at the level of the programming language.
2. Program code can reference code fragments that represent model elements.
3. Some parts of the program code follow certains rules so that they can be interpreted with respect to models.

This leads to the following requirements for a common language:

Common Type Concept The language needs a type concept that applies to models and code. Model types must be given with their properties as e.g. in EMF, while code types must follow object-oriented principles. This is not exclusive, so that types may be model types and at the same time object-oriented classes.

Common Reflection Concept The reflection functionality of such a language must consider all types. This is necessary e.g. to let execution frameworks interpret model elements.

Common Schema Concept Schemata are used to define valid structures. This applies to models e.g. in the way that XML schemata can be used to validate XML-based modeling notations. In addition, this must apply to program code, too: By this means a mechanism is introduced that makes pattern definitions a core concept of the language, so that no external tools (cf. section 4.4.3.1) have to be used.

Development with this language can be supported by tools that enable different views at different abstraction levels on software described in one consistent notation. The different views are realized based on appropriate schemata. The tools can thus be specific and adapted even for different types of users. For example, a process design tool based on a schema that excludes algorithmic details can be used by non-programmers, but work on the same code base as programmers.

6.6 Chapter Summary

In this chapter we gave an outlook on future work regarding embedded models. This concerns further evaluation of the approach first. Then, different aspects have been considered for extending the approach: Additional modeling domains, interaction of embedded models, and meta models. Finally, we gave an outlook on possible consequences for the development of new software description languages that may facilitate the tight connection between specifications and program code even further than embedded models.

Chapter 7

Conclusion

In this chapter we summarize the thesis and present conclusions for the embedded models approach. The approach itself is summarized in section 7.1. The implementations for specific modeling domains are reflected in section 7.2. The applicability in the case studies is subsumed in section 7.3, before we conclude in section 7.4.

7.1 Embedded Models Approach

The aim of embedded models is to close the semantic gap between modeling notations and program code constituting the actual implementations. For this purpose, we propose to write program code according to certain rules so that it reflects the model specifications unambiguously. This results in an approach as sketched in figure 7.1 and introduced in chapter 3: *Model specifications* for a certain modeling domain are defined clearly and a *program code pattern* is developed that represents the abstract syntax of the models in static program code structures. Appropriate *execution semantics* enable frameworks to invoke the program code at run time so that the models are executable. Their execution can be part of other program code since well-defined *interfaces* exist that allow for data exchange and program code invocation. A set of *transformations* enables considering the program code with embedded models at different levels of abstraction. This is embraced by tools that can e.g. visualize, verify, and monitor the model specifications. The semantic gap is by this means narrowed since the program code is a valid notation for the model specifications.

7.2 Embedded State Machines and Process Models

In chapter 4, two implementations for the domains of state machines and process models were introduced and appropriate tools were presented.

State machine models focus on the structure of states and transitions as well as the management of the state space. The state space is constituted by variables that are defined in interfaces and extracted from the application logic of the surrounding program. Transitions are equipped with action labels that denote application program code to be invoked in interfaces. Different state

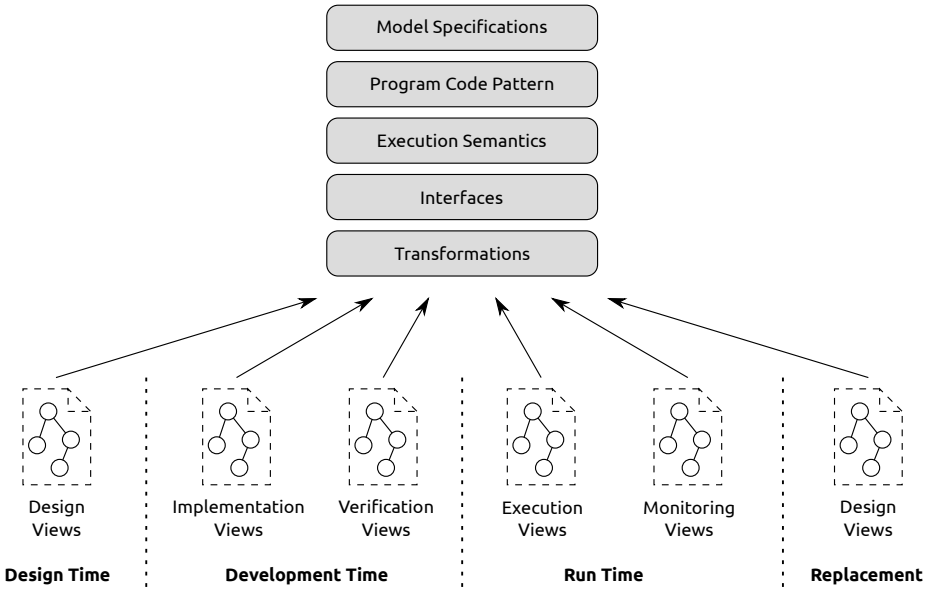


Figure 7.1: The structure of the conceptual thesis part.

machines can interact with channels. An abstract view is e.g. realized with the automata model checker UPPAAL that neglects action labels, but allows for verification of the state space.

Process models put emphasis on actions, decisions, and input and output of variables. Variables and actions are realized as interfaces to other program code. Scaling of models is realized with sub processes so that a hierarchy of models can be created. The abstract model used is based on the Java Workflow Tooling for Eclipse. An appropriate bidirectional transformation allows to use the program code in the visual editor of JWT.

Both model types are completely and unambiguously embedded in the program code. The connection to other program code requires design decisions regarding the semantics of interfaces, e.g. with respect to the management of the state space, that influence the ability of the models for formal verification. The different views sketched in figure 7.1 have been realized by appropriate tools. For state machines, a tool suite covers design, implementation, execution, monitoring, and design recovery. For process models, design and execution are supported. Thus, comprehensive tooling for embedded models has proven to be possible.

7.3 Applicability of the Approach

The approach has been applied to two case studies as explained in chapter 5.

In a load generator for performance tests, “SyLaGen”, measurement strategies have been implemented with embedded models for state machines and process models. Both model classes are appropriate since the models mainly fulfill documentation purposes here. This is due to the fact that the state space is too complex to be fully represented in the models, so that a high degree of

abstraction is necessary and formal verification is not applicable. However, the models are represented in the program code and allow for a maintenance of embedded models over longer time so that the approach fulfills the expectations.

In contrast, the second case study concerned the development of small visual simulations in the learner's IDE "Greenfoot" with embedded state machines. The state space is in this case completely represented in the model so that verification and model checking in UPPAAL are possible. The approach was applied in university teaching for formal methods. In this context, a user study was performed showing that the connection between formal models and implementations is clarified with embedded models; however, the productivity of the use of embedded models strongly depends on the availability of appropriate tools.

7.4 Conclusions

In section 1.3, objectives for the approach were stated. Considering the overall results of the thesis, the objectives are fulfilled as follows:

- *Consistency between model semantics and source code* is given since the program code patterns introduced so far enable bidirectional and unambiguous transformations.
- *Integration of arbitrary source code* is possible since the interfaces proposed in this thesis allow for data exchange and program code invocation in a manner that reflects the model specifications.
- *Consistency between model semantics and executing program code* is realized by the execution semantics and execution frameworks providing appropriate sequences of actions.
- *Consistency between model semantics and monitoring* has proven to be possible since the embedded model specifications can be accessed at run time so that tools provide this functionality.

The objectives are by this means fulfilled completely. Thus the approach in this thesis is novel and a relevant contribution to model-based software development has been made.

Appendix A

CD-ROM Contents

The CD-ROM enclosed with this thesis contains the software artifacts developed during creation of the embedded models approach. On the one hand, the program code of the patterns, the related tools, and the case studies is given. On the other hand, a virtual machine is provided that contains the tools and the case studies in an executable way.

A.1 Program Code

The folder `code` contains the program code developed in this thesis. Most of the folders contain Eclipse projects.

The projects for embedded state machines are:

- `de.uni_due.s3.embeddedmodels.statemachine.definitions`

  The pre-defined program code of the embedded state machine pattern.

- `de.uni_due.s3.embeddedmodels.statemachine.execution`

  The execution framework for embedded state machines.

- `de.uni_due.s3.embeddedmodels.statemachine.execution.osgi`

  The OSGi integration for the listeners of the execution framework for embedded state machines.

- `de.uni_due.s3.embeddedmodels.statemachine.greenfoot.runtime`

  The execution framework for embedded state machines in Greenfoot.

- `de.uni_due.s3.embeddedmodels.statemachine.monitor.eclipse`

  The Eclipse-based monitoring tool for embedded state machines.

- `de.uni_due.s3.embeddedmodels.statemachine.monitor.eclipse.listener`

  The OSGi listener integration for the Eclipse-based monitoring tool for embedded state machines.

- `de.uni_due.s3.StateMachineEditor`,

  `de.uni_due.s3.StateMachineEditor.CodeEditorInterface`,

`de.uni_due.s3.StateMachineEditor.diagram`,

`de.uni_due.s3.StateMachineEditor.edit`, and

`de.uni_due.s3.StateMachineEditor.editor`

The state machine editor as developed by Malte Goddemeier.

- `rules_verification`

  The graph transformation artifacts for verification (i.e., transformation to UPPAAL). These files were contributed by Michael Striewe.

- `rules_designrecovery`

  The graph transformation artifacts for model transformation from embedded state machines to process models. These files were contributed by Michael Striewe.

The projects for embedded process models are:

- `de.uni_due.s3.embeddedmodels.process.definition`

  The pre-defined program code of the embedded process model pattern.

- `de.uni_due.s3.embeddedmodels.process.execution`

  The execution framework for embedded process models.

- `de.uni_due.s3.embeddedmodels.process.jdt_jwt`

  The design tool for embedded process models.

The projects for case studies are:

- `de.uni_due.s3.sylagen.master_statemachine`

  The packages of interest for the SyLaGen master with the embedded state machine. Since SyLaGen is proprietary, the application logic is replaced by dummy classes.

- `de.uni_due.s3.sylagen.master_process`

  The packages of interest for the SyLaGen master with the embedded process model. Since SyLaGen is proprietary, the application logic is replaced by dummy classes.

- `hungry`

  The *hungry* scenario with the embedded state machine.

- `hungry_manual`

  The *hungry* scenario developed manually.

- `space`

  The *space* scenario with the embedded state machine.

A.2 Executable Tools

The folder `executable` contains a virtual machine that is executable with VirtualBox[1]. The operating system is Ubuntu 11.04. User name and password are both "`embeddedmodels`". The following executable programs are available:

- *Eclipse for embedded state machines*:

  A ready-to-use Eclipse installation for embedded state machines is provided in the folder `eclipse_statemachine` on the desktop. All necessary tools are deployed as plugins in the `dropins` folder.

  The workspace contains the packages of interest for the SyLaGen case study. Since SyLaGen is proprietary, the application logic is replaced by dummy classes. The embedded state machine is used in the project `de.uni_due.s3.sylagen.master` in three packages:

  - `de.uni_due.s3.sylagen.master.measurement.impl.single`
  - `de.uni_due.s3.sylagen.master.measurement.impl.stress`
  - `de.uni_due.s3.sylagen.master.measurement.impl.exploration`

  The context menu for these packages has a category *Embedded State Machine* with the entries *Edit* and *Verify*. *Edit* opens the visual editor for the embedded state machine in the current package. To use *Verify*, the class `IMeasurementVariables.java` in the parent package must also be selected. The entry then extracts the model and opens UPPAAL. UPPAAL must be downloaded[2] and installed in the virtual machine for this purpose. The path to the UPPAAL binary must be given in the Eclipse preferences in category "Embedded State Machine".

- *Eclipse for embedded process models*:

  A ready-to-use Eclipse installation for embedded process models is provided in the folder `eclipse_process` on the desktop. All necessary tools are deployed as plugins in the `dropins` folder.

  The workspace contains the packages of interest for the SyLaGen case study. Since SyLaGen is proprietary, the application logic is replaced by dummy classes. The embedded process model is used in the project `de.uni_due.s3.sylagen.master` in the package `de.uni_due.s3.sylagen.master.measurement.impl.exploration`.

  The models are stored in the project folder `models`. The file `sylagen-exploration.workflow` contains the JWT model. The file `sylagen-exploration.epmm` is the mapping between model and code.

- *Greenfoot with embedded state machines*:

  Greenfoot is provided in the folder `Greenfoot` on the desktop. The following scenarios are contained in the folder `scenarios`:

  - `hungry`
    The *hungry* scenario with the embedded state machine.

[1] `http://www.virtualbox.org/`
[2] `http://www.uppaal.com`

  - `hungry_manual`
    The *hungry* scenario developed manually.
  - `space`
    The *space* scenario with the embedded state machine.

  After opening a scenario, it can be used as any other Greenfoot program.

- *GGX Toolbox*:

  The GGX Toolbox[3] is provided in the folder `ggxToolbox` on the desktop. It realizes graph transformations for embedded models, in this case for verification with UPPAAL and model transformation from embedded state machines to process models.

  For verification, the Java files of interest must be selected, e.g. all Java files from one of the Greenfoot scenarios. Then an output file must be given, e.g. `/home/embeddedmodels/output.xml`, and the *UPPAAL* button must be selected. The output file can be opened with UPPAAL later on.

  The model transformation takes the same Java files as input. In addition, the rule files and the control script must be selected from the folder `rules_designrecovery` (see above) according to the GGX Toolbox documentation. To start the transformation, the *Run* button must be selected.

A.3 Videos

The folder `videos` contains three screen casts illustrating the basic usage of tools for embedded models with SyLaGen:

- `StateMachine Editor.ogv`: Parallel editing of the exploration state machine in source code and visual editor.
- `StateMachine Monitoring.ogv`: Monitoring of the exploration state machine while SyLaGen is executed.
- `Process Editor.ogv`: Parallel editing of the exploration process model in source code and visual editor.

[3] developed by Michael Striewe, `http://www.s3.uni-due.de/research/ggx-toolbox.html`

Bibliography

Ritu Agarwal and Atish P. Sinha. Object-Oriented Modeling with UML: A Study of Developers' Perceptions. *Communications of the ACM*, 46(9):248–256, 2003. ISSN 0001-0782.

AGG. AGG website. `http://tfs.cs.tu-berlin.de/agg/`.

Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling Design Patterns: Application to Pattern Detection and Code Synthesis. In *Proceedings of the First ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 166–173, 2001.

Michał Antkiewicz. Round-Trip Engineering Using Framework-Specific Modeling Languages. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 927–928, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-865-7.

Michał Antkiewicz and Krzysztof Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In Nierstrasz et al. [2006], pages 692–706. ISBN 3-540-45772-0.

Michał Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Automatic Extraction of Framework-Specific Models From Framework-Based Application Code. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321664.

Michał Antkiewicz, Thiago T. Bartolomei, and Krzysztof Czarnecki. Fast Extraction of High-Quality Framework-Specific Models from Application Code. *Automated Software Engineering*, 16(1):101–144, 2009. ISSN 0928-8910. doi: http://dx.doi.org/10.1007/s10515-008-0040-x.

Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231149.

Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand and Williams [2005], pages 476–491. ISBN 3-540-29010-9.

Moritz Balz, Michael Striewe, and Michael Goedicke. Tool Support for Continuous Maintenance of State Machine Models in Program Code. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Inc., 2007.

Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In *WOSP '02: Proceedings of the 3rd International Workshop on Software and Performance*, pages 35–45, New York, NY, USA, 2002. ACM.

Martin Bichler and Kwei-Jay Lin. Service-Oriented Computing. *IEEE Computer*, 39(3):99–101, 2006.

Eric Bodden. The Design and Implementation of Formal Monitoring Techniques. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 939–940. ACM, 2007. ISBN 978-1-59593-865-7.

Reinhard Bordewisch, Bärbel Schwärmer, Michael Goedicke, and Peter Tröpfner. Lastsimulation für Anwendungsumgebungen in vernetzten IT-Architekturen. *Mitteilungen der GI-Fachgruppe MMB Nr. 43*, 2003.

Steen Brahe. BPM on Top of SOA: Experiences from the Financial Industry. In *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, volume 4714 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2007. ISBN 978-3-540-75182-3.

Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 365–383, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-9.

Martin Bravenboer, René de Groot, and Eelco Visser. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. Technical report, Department Information and Computing Sciences, Utrecht University, 2006.

Lionel C. Briand and Clay Williams, editors. *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-29010-9.

Alan W. Brown, Sridhar Iyengar, and Simon Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.

Frank Budinsky, David Steinberg, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, 2009. ISBN 978-0-321-33188-5.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 2007.

Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 273, Washington, DC, USA, 2001. IEEE Computer Society.

Thomas Büchner and Florian Matthes. Introspective Model-Driven Development. In *Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, 2006*, volume 4344 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2006.

Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 386–387, London, UK, 1998. Springer-Verlag.

Feng Chen, Marcelo D'Amorim, and Grigore Roşu. A Formal Monitoring-based Framework for Software Development and Analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 357–373. Springer-Verlag, 2004.

Peter P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, April 1986. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/5397.5399.

Adrian Colyer, Andy Clement, and George Harley. *Eclipse AspectJ*. Addison-Wesley, 2004.

Tom Copeland. *PMD applied*. Centennial Books, 2005.

James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. doi: http://doi.acm.org/10.1145/337180.337234.

Iain Craig. *The Interpretation of Object-Oriented Programming Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001. ISBN 1-85233-547-2.

Michaelle L. Crane and Jürgen Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In Briand and Williams [2005], pages 97–112. ISBN 3-540-29010-9.

Jesus Sanchez Cuadrado and Jesus Garcia Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. *IEEE Transactions on Software Engineering*, 35(6):825–840, Nov.-Dec. 2009. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2009.14.

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective – GRACE Meeting Notes, State of the Art, and Outlook. In *ICMT2009 - International Conference on Model Transformation, Proceedings*, LNCS. Springer, 2009.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. A Two Phase Approach to Design Pattern Recovery. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 297–306, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2802-3. doi: http://dx.doi.org/10.1109/CSMR.2007.10.

François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a Short Comparative Study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why Unified is not Universal? UML Shortcomings for Coping with Round-trip Engineering. In Robert B. France and Bernhard Rumpe, editors, *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*, pages 630–644. Springer, 1999.

Brian Dobing and Jeffrey Parsons. How UML is used. *Communications of the ACM*, 49(5):109–113, 2006. ISSN 0001-0782.

Jing Dong, Yajing Zhao, and Tu Peng. Architecture and Design Pattern Discovery Techniques – A Review. In *Proceedings of the International Workshop on System/Software Architectures (IWSSA) 2007*, pages 621–627, 2007.

Stéphane Ducasse and Tudor Gîrba. Using Smalltalk as a Reflective Executable Meta-language. In Nierstrasz et al. [2006], pages 604–618. ISBN 3-540-45772-0.

Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM Algorithm and its Application to Software Behavior Model Induction. *Applied Artificial Intelligence*, 22(1-2):77–115, 2008. ISSN 0883-9514. doi: http://dx.doi.org/10.1080/08839510701853200.

Eclipse. Eclipse website. `http://www.eclipse.org/`.

ECMA International. Standard ECMA-334 – C# Language Specification, 2006a.

ECMA International. Standard ECMA-335 – Common Language Infrastructure (CLI), 2006b.

Amnon H. Eden, Amiram Yehudai, and Joseph Gil. Precise Specification and Automatic Application of Design Patterns. In *ASE '97: Proceedings of the 12th International Conference on Automated Software Engineering*, pages 143–152, Nov 1997a. doi: 10.1109/ASE.1997.632834.

Amnon H. Eden, Amiram Yehudai, Yoram Hirshfeld, and Joseph Gil. Towards a Mathematical foundation for Design Patterns. Technical report, Department of Information Technology, Uppsala University, 1997b.

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformations*. Springer, 2006.

Ghizlane El Boussaidi and Hafedh Mili. A Model-driven Framework for Representing and Applying Design Patterns. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 97–100, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2870-8. doi: http://dx.doi.org/10.1109/COMPSAC.2007.31.

Robert Englander. *Developing Java Beans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. ISBN 1-56592-289-1.

Jacky Estublier, Germán Vega, and Anca Daniela Ionita. Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. In Briand and Williams [2005], pages 69–83. ISBN 3-540-29010-9.

Andy S. Evans. Reasoning with UML Class Diagrams. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 102, Washington, DC, USA, 1998. IEEE Computer Society.

Jean-Marie Favre. Towards a Basic Theory to Model Driven Engineering. In *3rd Workshop in Software Model Engineering (WiSME 2004)*, 2004.

Mohamed Fayad and Douglas C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/262793.262798.

Joao M. Fernandes, Simon Tjell, Jens Bæk Jørgensen, and Oscar Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. In *SCESM '07: Proceedings of the Sixth International Workshop on Scenarios and State Machines*, page 2, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2958-5.

Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications Co., Greenwich, CT, USA, 2004. ISBN 1-9323-9418-4.

Martin Fowler. PlatformIndependentMalapropism, 2003a. `http://martinfowler.com/bliki/PlatformIndependentMalapropism.html`.

Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003b.

Martin Fowler. DomainSpecificLanguage, 2006a. `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`.

Martin Fowler. InternalDslStyle, 2006b. `http://www.martinfowler.com/bliki/InternalDslStyle.html`.

David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. OMG Press, 2003.

Steve Freeman and Nat Pryce. Evolving an Embedded Domain-Specific Language in Java. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 855–865, New York, NY, USA, 2006. ACM.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

Malte Goddemeier. Entwicklung eines grafischen Editors für Zustandsautomaten im Quellcode. Diploma thesis, Specification of Software Systems, University of Duisburg-Essen, September 2009.

Michael Goedicke, Torsten Meyer, and Gabriele Taentzer. ViewPoint-Oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *RE '99: Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 92–99, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0188-5.

Michael Goedicke, Michael Striewe, and Moritz Balz. Support for Evolution of Software Systems using Embedded Models. In *Design for Future – Langlebige Softwaresysteme*, 2009.

Martin Gogolla and Francesco Parisi Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations - or Diagrams are nice, but graphs are worth their price. In *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*, pages 55–72, 1998.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification, 3rd Edition*. Addison-Wesley Professional, 2005. ISBN 978-0-3212-4678-3.

Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise Modeling of Design Patterns. In *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, pages 482–496, 2000.

Vadim S. Gurov, Maxim A. Mazin, Andrey S. Narvsky, and Anatoly A. Shalyto. UniMod: Method and Tool for Development of Reactive Object-Oriented Programs with Explicit States Emphasis. In *Proceedings of St. Petersburg IEEE Chapters*, pages 106–110, 2005.

Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

Atzmon Hen-Tov, David H. Lorenz, and Lior Schachter. ModelTalk: A Framework for Developing Domain Specific Executable Models. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, 2008.

Poul Henriksen and Michael Kölling. greenfoot: Combining Object Visualisation with Interaction. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 73–82, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: http://doi.acm.org/10.1145/1028664.1028701.

C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580, October 1969. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/363235.363259.

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic Embedding of DSLs. In *GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 137–148, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: http://doi.acm.org/10.1145/1449913.1449935.

Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. ISBN 0-321-22862-6.

Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Model driven code checking. *Automated Software Engineering*, 15(3-4):283–297, 2008. ISSN 0928-8910. doi: http://dx.doi.org/10.1007/s10515-008-0033-9.

Paul Hudak. Modular Domain Specific Languages and Tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8377-5.

Ke Jiang, Lei Zhang, and Shigeru Miyake. OCL4X: An Action Semantics Language for UML Model Execution. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 633–636, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2870-8. doi: http://dx.doi.org/10.1109/COMPSAC.2007.158.

Ke Jiang, Lei Zhang, and Shigeru Miyake. Using OCL in Executable UML. *ECEASST*, 9, 2008.

JWT. JWT website. `http://www.eclipse.org/jwt/`.

Jan Jürjens. A UML statecharts semantics with message-passing. In *SAC '02: Proceedings of the 2002 ACM Symposium on Applied Computing*, pages 1009–1013, New York, NY, USA, 2002. ACM. ISBN 1-58113-445-2.

Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. A UML-Based Metamodeling Language to Specify Design Patterns. In *Workshop in Software Model Engineering (WisME) 2003*, 2003.

Steven H. Kim. *Knowledge Systems Through Prolog: An Introduction*. Oxford University Press, Inc., New York, NY, USA, 1991. ISBN 0-19-507241-3.

Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

Florian Lautenbacher. Comparison of Business Process Metamodels. Technical report, Programming Distributed Systems Lab, University of Augsburg, 2007.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-43294-3.

Felix Lindlar and Armin Zimmermann. A Code Generation Tool for Embedded Automotive Systems Based on Finite State Machines. In *Proceedings of the 6th IEEE International Conference on Industrial Informatics (INDIN), 2008*, pages 1539–1544, July 2008.

LIQUidFORM. LIQUidFORM Website. `http://code.google.com/p/liquidform/`.

Miguel Pinto Luz and Alberto Rodrigues da Silva. Executing UML Models. In *3rd Workshop in Software Model Engineering (WiSME 2004)*, 2004.

Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 249–256. IEEE Computer Society, 2002.

Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise Modeling of Design Patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

Shahar Maoz. Using Model-Based Traces as Runtime Models. *Computer*, 42 (10):28–36, 2009. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2009.336.

Anneliese von Mayrhauser and A. Marie Vans. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10(5): 171–182, 1995.

William E. McUmber and Betty H. C. Cheng. A General Framework for Formalizing UML with Formal Languages. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

Klaus Meffert. Supporting Design Patterns with Annotations. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 437–445, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2546-6. doi: http://dx.doi.org/10.1109/ECBS.2006.67.

Theo Dirk Meijler, Serge Demeyer, and Robert Engel. Making Design Patterns Explicit in FACE. *ACM SIGSOFT Software Engineering Notes*, 22(6):94–110, 1997. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/267896.267905.

Stephan J. Mellor and Marc J. Balcer. *Executable UML*. Addison-Wesley, 2002.

Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe Leblanc. An Action Language for UML: Proposal for a Precise Execution Semantics. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, volume 1618 of *LNCS*, pages 514–514. Springer Berlin / Heidelberg, 2004. doi: http://dx.doi.org/10.1007/978-3-540-48480-6_24.

Tommi Mikkonen. Formalizing Design Patterns. In *ICSE '98: Proceedings of the 20th International Conference on Software Engineering*, pages 115–124, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8368-6.

Hafedh Mili and Ghizlane El-Boussaidi. Representing and Applying Design Patterns: What Is the Problem? In Briand and Williams [2005], pages 186–200. ISBN 3-540-29010-9.

Jishnu Mukerji and Joaquin Miller. Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1, 2003.

Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713, pages 264–278. Springer, 2005.

Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.917525.

Marco Müller, Moritz Balz, and Michael Goedicke. Representing Formal Component Models in OSGi. In *Proceedings of Software Engineering 2010, Paderborn, Germany*, pages 45–56, 2010.

George C. Necula. Proof-Carrying Code. Design and Implementation. In *Proof and System Reliability*, pages 261–288, 2002.

James Newkirk and Alexei A. Vorontsov. How .NET's Custom Attributes Affect Design. *Software, IEEE*, 19(5):18–20, Sep/Oct 2002. ISSN 0740-7459. doi: 10.1109/MS.2002.1032846.

Iftikhar Azim Niaz and Jiro Tanaka. Code Generation from UML Statecharts. In *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003), Marina Del Rey, USA*, pages 315–321, 2003.

Ulrich A. Nickel, Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Roundtrip Engineering with FUJABA. In *Proceedings of the 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*, 2000.

Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Recovering UML Diagrams from Java Code using Patterns. In *Proceedings of the 2nd Workshop on Soft Computing Applied to Software Engineering, Enschede, The Netherlands (J.H. Jahnke and C. Ryan, eds.)*, 2001.

Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards Pattern-Based Design Recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM. doi: http://doi.acm.org/10.1145/581339.581382.

Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-45772-0.

Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michaele Lanza, and Horst Bunke. Example-Driven Reconstruction of Software Models. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR) 2007*, pages 275–286, 2007.

Bashar Nuseibeh. To Be and Not to Be: On Managing Inconsistency in Software Development. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 164, Washington, DC, USA, 1996. IEEE Computer Society.

Antoni Olivé. Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005. ISBN 3-540-26095-1.

OMG. Action Semantics for the UML, 2001.

OMG. Meta Object Facility (MOF) Core Specification, 2006.

OMG. MOF 2.0 / XML Metadata Interchange (XMI), v2.1.1 specification, 2007.

OMG. OMG Unified Modeling Language^{TM}(OMG UML), Superstructure, Version 2.3. Technical report, Object Management Group (OMG), May 2010.

OSGi Alliance. *OSGi Service Platform, Core Specification, Release 4, Version 4.1*. IOS Press, Inc., 2005a. ISBN 978-90-79350-01-8.

OSGi Alliance. *OSGi Service Platform, Service Compendium, Release 4, Version 4.1*. IOS Press, Inc., 2005b. ISBN 978-90-79350-02-5.

Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 978-3-540-71867-3.

Renaud Pawlak, Lionel Seinturier, and Jean-Philippe Retaille. *Foundations of AOP for J2EE Development*. Apress, 2005.

Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001. ISBN 0-387-95106-7.

Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software and Systems Modeling*, 4(1):55–70, February 2005. doi: http://dx.doi.org/10.1007/s10270-004-0059-9.

Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML*. Cambridge University Press, New York, NY, USA, 2004.

Jonathan Riehl. Assimilating MetaBorg: Embedding language tools in languages. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 21–28, New York, NY, USA, 2006. ACM. ISBN 1-59593-237-2. doi: http://doi.acm.org/10.1145/1173706.1173710.

John Anil Saldhana and Sol M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *International Conference on Software Engineering and Knowledge Engineering*, pages 103–110, 2000.

Reinhard Schauer and Rudolf K. Keller. Pattern Visualization for Software Comprehension. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 4, Washington, DC, USA, 1998. IEEE Computer Society.

Don Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. *ONJava.com*, June 2004. `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, 1994.

Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20 (5):19–25, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231146.

Shane Sendall and Jochen Küster. Taming Model Round-Trip Engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, 2004.

Giovanna Di Marzo Serugendo and Michel Deriaz. Specification-Carrying Code for Self-Managed Systems. In *IFIP/IEEE International Workshop on Self-Managed Systems and Services*, 2005.

Nija Shi. *Reverse Engineering of Design Patterns from Java Source Code*. PhD thesis, University of California, Davis, 2007.

Nija Shi and Ronald A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: http://dx.doi.org/10.1109/ASE.2006.57.

Forrest Shull, Walcélio L. Melo, and Victor R. Basili. An Inductive Method for Discovering Design Patterns from Object-oriented Software Systems. Technical report, University of Maryland, Computer Science Department, College Park, MD, 1996.

Herbert A. Simon. *The Sciences of the Artificial*. The MIT Press, 3rd edition, 1996.

Michael Soden and Hajo Eichler. Towards a Model Execution Framework for Eclipse. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 4:1–4:7, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-503-1. doi: http://doi.acm.org/10.1145/1555852.1555856.

Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 666–675, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

Squill. Squill Website. `https://squill.dev.java.net/`.

Leon Starr. *Executable UML: How to Build Class Models*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0-130-67479-6.

Margaret-Anne Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2254-8.

Michael Striewe. Using a Triple Graph Grammar for State Machine Implementations. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations (ICGT) 2008, Leicester*, volume 5214 of *LNCS*, pages 514–516, 2008.

Michael Striewe, Moritz Balz, and Michael Goedicke. SyLaGen – An Extendable Tool Environment for Generating Load. In Bruno Müller-Clostermann, Klaus Echtle, and Erwin Rathgeb, editors, *Proceedings of "Measurement, Modelling and Evaluation of Computing Systems" and "Dependability and Fault Tolerance" 2010, March 15 - 17, Essen, Germany*, volume 5987 of *LNCS*, pages 307–310. Springer, 2010a.

Michael Striewe, Moritz Balz, and Michael Goedicke. Enabling Graph Transformations on Program Code. In *Proceedings of the 4th International Workshop on Graph Based Tools, Enschede, The Netherlands, 2010*, 2010b.

Sun Microsystems, Inc. Java™Platform Debugging Architecture API, a. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

Sun Microsystems, Inc. JSR 220: Enterprise JavaBeans™, Version 3.0 - Java Persistence API, b. `http://jcp.org/en/jsr/detail?id=220`.

Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

Sun Microsystems, Inc. JSR 318: Enterprise JavaBeans™3.1 - Proposed Final Draft, March 2008. `http://jcp.org/en/jsr/detail?id=318`.

Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML Action Semantics for Model Execution and Transformation. *Information Systems*, 27(6):445–457, 2002. ISSN 0306-4379. doi: http://dx.doi.org/10.1016/S0306-4379(02)00014-5.

Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-74572-0.

Toufik Taibi and Taieb Mkadmi. Generating Java Code from Design Patterns Formalized in BPSL. In *Innovations in Information Technology, 2006*, 2006.

Toufik Taibi and David Chek Ling Ngo. Formal specification of design patterns – a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

The Eclipse Foundation. Eclipse Java Development Tools, 2008. `http://www.eclipse.org/jdt/`.

Matthias Tichy and Holger Giese. Seamless UML Support for Service-based Software Architectures. In N Guefi, E Artesiano, and G Reggio, editors, *Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003, Luxembourg*, volume 2952 of *Lecture Notes in Computer Science*, pages 128–138. Springer-Verlag, November 2003.

Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000. ISSN 0362-1340.

Dániel Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 378–392, London, UK, 2002. Springer-Verlag.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), 2003.

John Vlissides. Generation Gap. *C++ Report*, 8(10):12, 14–18, 1996.

Marek Vokáč and Jens M. Glattetre. Using a Domain-Specific Language and Custom Tools to Model a Multi-tier Service-Oriented Application – Experiences and Challenges. In Briand and Williams [2005], pages 492–506. ISBN 3-540-29010-9.

Nic Volanschi. A Portable Compiler-Integrated Approach to Permanent Checking. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: http://dx.doi.org/10.1109/ASE.2006.8.

Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reclipse – A Reverse Engineering Tool Suite. Technical report, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2010.

Hiroshi Wada and Junichi Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Briand and Williams [2005], pages 584–600. ISBN 3-540-29010-9.

Dennis Wagelaar and Viviane Jonckers. Explicit Platform Models for MDA. In Briand and Williams [2005], pages 367–381. ISBN 3-540-29010-9.

Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3034-6. doi: http://dx.doi.org/10.1109/WCRE.2007.45.

Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

Lothar Wendehals and Alessandro Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In *Proceedings of the 4th ICSE 2006 Workshop on Dynamic Analysis (WODA), Shanghai, China*, 2006.

Lothar Wendehals, Matthias Meyer, and Andreas Elsner. Selective Tracing of Java Programs. In *Proceedings of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, 2004.

XOCL. eXecutable OCL webpage, 2008. `http://xocl.sourceforge.net/`.

Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 470–479, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.