Multi-Abstraction Model Based Software Development for Embedded Low-Cost Applications

Robert Heumüller

November 6, 2015
Master’s Thesis

**Multi-Abstraction Model Based Software Development for Embedded Low-Cost Applications**

Robert Heumüller

1. **Reviewer**  Prof. Frank Ortmeier  
Chair of Software Engineering  
OvGU Magdeburg

2. **Reviewer**  Ulrich Huggenberger  
XITASO GmbH Augsburg

**Supervisors**  Jan Güntner  
Dr. Michael Schackert

November 6, 2015
Robert Heumüller
Multi-Abstraction Model Based Software Development for Embedded Low-Cost Applications
Master's Thesis, November 6, 2015
Reviewers: Prof. Frank Ortmeier and Ulrich Huggenberger
Supervisors: Jan Güntner and Dr. Michael Schackert

Otto von Guericke University
Chair of Software Engineering (CSE)
Institute of Embedded Systems
Department of Computer Science
Universitätsplatz 1
39106 and Magdeburg
Abstract

This thesis proposes a novel approach of multi-abstraction model based software development, in particular for low-cost embedded systems. Instead of modeling systems by use of generic structural or behavioral formalisms, it enables the integration of heterogeneous, domain specific models under the control of a central state machine model. For this purpose a new state machine modeling language with formally defined operational semantics, as well as an associated compiler are introduced. Different target platforms are supported by the use of LLVM as hardware abstraction. The practical application of the modeling approach and workflow is demonstrated in two case studies, one of which runs on an actual ARM microcontroller and has already found a commercial application.
# Contents

1 Introduction ........................................ 1
   1.1 Motivation ...................................... 1
   1.2 Goals ........................................... 2
   1.3 Structure ....................................... 3

2 Related Work ........................................ 5
   2.1 Software and Systems Engineering Approaches .......... 5
   2.2 Synchronous Programming .......................... 6
   2.3 Niche ........................................... 7

3 Concept .............................................. 9
   3.1 Architecture .................................... 9
      3.1.1 Abstractions ................................. 10
      3.1.2 Common Driver Interface ..................... 11
      3.1.3 State Machine Executor ...................... 12
      3.1.4 LLVM-Based Hardware Abstraction ............ 12
   3.2 State Machine Formalism .......................... 13
      3.2.1 Extended Finite State Machines ............... 14
      3.2.2 Custom Hierarchically Modular State Machines .... 16

4 Modeling Language .................................. 17
   4.1 Modeling Language Syntax .......................... 17
      4.1.1 Graphical Notation ......................... 20
   4.2 Example Model ................................... 21
   4.3 Modeling Language Semantics ....................... 21
      4.3.1 Environment ................................. 23
      4.3.2 Structure ................................... 24
      4.3.3 Model Mechanics ............................. 24
      4.3.4 Transitions .................................. 25
      4.3.5 Operational Semantics ....................... 26

5 Compiler ............................................. 31
   5.1 Lexical and Syntactic Analysis ....................... 32
      5.1.1 Lexer ........................................ 32
      5.1.2 Parser ....................................... 33
      5.1.3 Linking ...................................... 37
   5.2 Code Generation .................................. 39
      5.2.1 Representation of the Dynamic Model State .... 39
      5.2.2 State Machine Updates ....................... 40
Introduction

1.1 Motivation

In today's highly digitized world microprocessors are ever-present. The establishment of mobile devices such as smartphones and tablets doubtlessly has enormous impact on the daily routines of many millions of people. Modern cars feature numerous dedicated processors in control-, driver-assistance- and multimedia-systems. Even trivial household devices like light bulbs are now sold with integrated wifi chips, to allow adjusting the color and intensity of light. Innumerable further examples could be listed and the number of devices with embedded processors is ever-growing. Therefore the development of software for embedded systems is of crucial significance.

With processors continuously becoming less expensive, the cost of the software development grows in relation to the overall cost of systems. The capability to efficiently create new software as well as to reuse elements of existing software is therefore highly desirable. In the domain of embedded systems, both of these aspects require the mastering of various challenges. These arise from domain-specific requirements and restrictions, as well as from the heterogeneity of the target hardware platforms. Model-based methods are one approach of dealing with several challenges of both categories.

Particularly in safety-critical (i.e. automotive- or aerospace-) applications, standards such as ISO 26262 demand formally verifying the correct operation of systems. Since complete verification on the software level is impossible or impractical, it is instead performed on abstract models. Provided with an appropriate system-abstraction, model-checking algorithms are able to formally verify different types of operational characteristics. However, even when the model has been successfully verified, there remains the issue of proving that the implemented software is operationally equivalent to the model. For this reason model based software development, puts emphasis on first modeling systems as completely as possible, before implementing the models in concrete software. A number of model based software development tools such as IBM’s Rational Rhapsody, Esterel’s Scade or Etas’ Ascet are able to partially or completely generate source code, depending on the completeness of the description provided by the models. Assuming the correctness of the code-generators\(^1\), this approach simplifies the verification or even dispenses with the need to prove the equivalence of model and implementation.

For non-safety-critical, small-scale and low-cost embedded systems, model based software development is also advantageous, since platform-independent models can be reused across different target architectures. Apart from the initial development phase, it also facilitates the

\(^1\) Ascet's and Scade's code generators are IEC 61508 certified
process of testing and maintaining the software. This reduces the cost and effort of targeting multiple platforms or of porting an existing software to new platforms. An example of this kind of situation is the case study in Chapter 6. A prototype of a microprocessor-based detective game had to be built without knowing exactly what the final hardware setup would be. Nevertheless, it was required that as much of the source code as possible could be reused for the real product. To achieve this, the model-based approach developed in this thesis uses LLVM\(^2\) as hardware abstraction to grant the developer flexibility in terms of target hardware.

A general purpose programming language can be used to construct arbitrarily complex software systems. The majority of technical systems is however better described on multiple points of view, addressing specific concerns. From the architectural point of view, this paradigm is supported by many model based engineering tools to some extent. Prominently, UML\(^3\) allows describing different aspects of software architectures in terms of structure and behavior through several types of diagrams. On the one hand UML is a very powerful tool as it can, when necessary, be extended or customized to fit almost any application. On the other hand this generality is also a disadvantage, because it complicates the definition of formal semantics. Due to this lack, there is no real guideline for the creation of UML tools.

In contrast to the existing approaches, this thesis proposes a technique of modeling systems by using multiple heterogeneous abstractions, instead of using only generic behavioral and structural formalisms. It suggests to arrange these abstractions around a central finite-state automaton, which describes the procedural aspects of system operation. For this purpose, a modeling language with formally defined semantics and a corresponding compiler are developed. A common driver call interface is introduced as general way of inter-model communication. This interface enables the integration of domain-specific models, which are more suited for solving the tasks at hand, than i.e. the generic behavioral and structural diagrams of the UML. As an example, the second case-study demonstrates how this interface is applied to integrate a model from control-theory. The latter is applied to describe how the system regulates the boiler temperature of an espresso machine.

### 1.2 Goals

This thesis aims at granting embedded developers larger flexibility in modeling and in the targeting of different hardware platforms. It proposes to achieve this goal, by creating a model-based software development approach, particularly for small-scale and low-cost embedded systems. Focus is put on providing not only the theoretical concepts, but also a modeling language and infrastructure, which can be successfully applied in real-world applications. Achieving these goals involves three steps:

- Creating a state machine based modeling language with formalized operational semantics
- Developing a technique for inter-model communication

\(^2\)Formerly Low Level Virtual Machine

\(^3\)The Unified Modeling Language
• Implementing a functional, LLVM based compiler and toolchain

The relevance of the approach is to be proven in two case-studies, which:

• Demonstrate the practical applicability by creating a product-prototype involving real hardware

• Demonstrate a method of testing models by means of cross-compilation to desktop architectures

1.3 Structure

Chapter 2 summarizes the state-of-the-art approaches and tools in the domain of model-based software development for embedded applications. The niche, which this thesis attempts to fill, is pointed out. Chapter 3 explains the concepts of the new approach. On this foundation, Chapter 4 introduces a new state machine modeling language and formally defines its operational semantics. Chapter 5 deals with the implementation of the LLVM based state machine compiler. The new modeling approach and infrastructure are applied in two case studies in Chapters 6 and 7. The conclusion in Chapter 8 summarizes the thesis, evaluates whether the stated goals have been achieved, and proposes starting points for further work.
Related Work

This section gives an overview of the existing tools and related approaches in the domain of model based software development and points out the niche in which this thesis resides.

2.1 Software and Systems Engineering Approaches

The Object Management Group’s (OMG) Unified Modeling Language (UML) dominates large areas of software engineering. It provides a large number of model views for the description of the static structural as well as the dynamic behavioral aspects of systems. Well-known examples of these categories of diagrams are the class- and package diagrams (static), as well as activity diagrams and statecharts (dynamic). UML can be extended and tailored to specific needs by using a profiling mechanism which is part of the language. The dynamic semantics of UML, i.e. statecharts, are only informally defined and partly under-specified. Even though the application of UML is not strictly limited to software engineering, its origin and focus on software systems is obvious [23, 15].

In order to extend the range of model based engineering with UML to general systems engineering, the OMG developed the SysML language. SysML is an extended subset of UML. It reuses some UML diagram types and extends it with popular systems engineering models such as the block diagram. SysML is indented to be applicable for the specification, analysis, design, verification, and validation of a broad range of system types [22, 35].

UML provides a set of diagrams for modeling many kinds of systems, but does not require the engineer to follow a specific development process. The OMG does however provide a comprehensive development process called Model Driven Architecture (MDA), which makes heavy use of UML. MDA relies on the concepts of different layers of models and incremental refinement through model transformations. The MDA process consists of the following phases (1) specification of requirements, (2) specification of platform independent models, (3) specification of platform models and (4) generation of platform specific models [21].

A variety of CASE\(^1\) tools exist to support UML, SysML and MDA based software development.

A prominent example of this category of tools is IBM’s Rational Rhapsody, which focuses on the development of embedded real-time applications. Its features include model-based simulation and testing, as well as code generation for Ada, C/C++ and Java. It is capable of targeting a variety of real-time operating systems [9, 28]. Since the UML specification does not formally specify its statecharts’ operational semantics, Harel and Kugler [12] describe

\(^1\)Computer Aided Software Engineering
Rhapsody’s implementation of the statecharts’ semantics, by comparing it to the semantics of Harel’s statecharts in Statemate [12, 13].

Latella et al. [17] approach a formalization of UML statecharts’ semantics by defining a mapping to Kripke structures. Thereby they aim at enabling the analysis of statecharts by means of model-checking algorithms.

Lettner et al. [19] evaluated the MDA approach using Rational Rhapsody in the development of applications for low-cost mobile phones. In the course of the case-study, they identified different categories of MDA related problems, i.e. portability- and team-coordination-issues, as well as advantages of the MDA approach.

2.2 Synchronous Programming

The synchronous paradigm and associated languages aim at facilitating the development of reactive systems, that is to say of all systems which repeatedly wait for outside input, to which they react in a deterministic way. All synchronous languages share the basic idea that time passes as a series of discrete steps. No difference is made between physical clocks, i.e. milliseconds, and logical impulses, such as the arrival of internal or external events[4].

The Esterel programming language is an imperative, synchronous and deterministic programming language. System behavior is expressed by means of a number of temporal statements, which implement different schemes of awaiting and emitting signals. Esterel can be compiled to general purpose languages such as C or Ada, but can also be translated into an automaton representation. This representation enables not only the analysis using formal methods, but also the direct implementation in programmable hardware [4, 5, 2].

Lustre is another synchronous programming language. In contrast to Esterel, it describes systems using declarative dataflow models, rather than imperative programs. Systems are modeled as networks of operators, similar to block-diagrams or electronic circuits. The computation is performed by the network of operators, which guide the flow and in passing transform the data. Similar to Esterel, Lustre also features code generators for C and Ada, as well as the capability of formal verification[25, 10, 11].

Esterel Technologies’ SCADE Suite is based on a graphical block-diagram editor for the synchronous dataflow language Lustre. In addition, SCADE was extended by a concept of hierarchical state machines. This allows, for example, selecting different flow diagrams based on the system state. The SCADE Suite puts emphasis on the formal specification and verification of critical systems. Its KGC code generator for the languages C and Ada is certified to be correct with regard to ISO26262 and a number of other standards [3, 29].

With the Matlab extensions Simulink and Stateflow, Mathworks provides popular tools supporting hybrid modeling with block-diagrams and statecharts. Compared to the SCADE Suite, Simulink focuses on simulation instead of on formal specification and verification. For this reason, Tripakis et al. [32] have developed a translation from Simulink to Lustre, because as they state, the semantics of Simulink are neither formally nor completely specified. This
way, the analysis and code-generation software for Lustre can also be applied to Simulink models [32, 30].

2.3 Niche

The approaches described in Section 2.1 focus on the architectural design of complex software systems. While efforts have been made to formalize statecharts' semantics, the UML based approaches still put less emphasis on the dynamic system behavior. In contrast, the synchronous programming approaches introduced in Section 2.2 aim at thoroughly modeling and verifying the reactive behavior of systems.

The common aspect of both approaches is that they attempt to describe arbitrary systems using sets of generic structural or behavioral formalisms. This strategy is an efficient course of action for many types of systems, as proven by the numerous successful applications.

However, embedded systems in particular often touch a variety of different disciplines, including but not limited to safety constraints, control theory, synchronization and scheduling of resources. These views can seldom be intuitively expressed in a generic structural or behavioral formalism. Instead they can be represented in a superior way, by using the models and formalisms native to their particular domain.

The approach of this thesis aims at describing embedded systems more efficiently, by providing a way of integrating heterogeneous types of models from domain specific disciplines. This way, for example, a declarative model could be used to represent safety constraints, or a Petri-Net could portray the synchronization of parallel processes. To coordinate the different abstractions, the thesis proposes the use of a central state machine model. This formalism was selected, because the behavior of a state machine which is periodically updated is very similar to the way many small-scale embedded systems, which do not make use of an operating system, are programmed. Often these systems run in an infinite loop, which receives input and then branches depending on the type of input and its accumulated understanding of the environment.
This chapter introduces the concepts of this thesis' multi-abstraction approach and toolchain.

As stated in the introduction, complex systems often combine different layers of operation. Consequently, it is preferable to portray such systems by composing multiple domain-specific abstractions, rather than expressing all details in one generic representation. To support this paradigm, engineering software must have the capability to integrate different types of models.

The architecture of the developed multi-abstraction MBSD approach is explained in detail in Section 3.1.

The use of a common state machine type for describing the procedural aspect of system operation is central to this approach. Therefore the state machine formalism is explained individually in Section 3.2.

3.1 Architecture

Figure 3.1 gives an overview of the multi-abstraction architecture, which is introduced in a top-down fashion in the following subsections.

The topmost two layers represent the different abstractions. Each abstraction consists of a model and an associated engine. Through the engines, the abstractions expose the functionality of their models to the common driver interface.

The state machine abstraction is of special importance to this approach, because it is in charge of controlling the domain-specific abstractions. For this reason it must be invoked periodically by the state machine executor.

A primary goal of this thesis is to create a MBSD toolchain capable of targeting different hardware platforms. The existing engineering software described in Chapter 2 achieve this,
by translating models to source code in general-purpose languages. Instead, this approach uses LLVM as common abstraction for heterogeneous target platforms.

3.1.1 Abstractions

This subsection explains the topmost two layers of the multi-abstraction architecture.

Models

The multi-abstraction approach aims at describing systems through a set of abstract models. Each model represents a specific view of the complete system using a task-specific formalism.

The state machine model is the central model in this approach. It is used to portray the general operation of systems. This is achieved by integrating it with the underlying hardware as well as the domain-specific abstractions. With their specific views of the system, these models provide input to guide the state machine's decision-making process. Since the state machine is present in all multi-abstraction models, a common formalism and language for all state machine models is presented in Section 3.2.

The purpose of the domain-specific models is to depict aspects of the system, which are not intuitively formulated in the general state machine formalism. For this purpose, many disciplines provide specific types of models. Since the domain-specific models vary for different multi-abstraction models, a common formalism cannot be used for their representation. The following subsection explains, how different model-types can nevertheless be integrated.

The hardware model is less tangible than the other model types, because it is not physically a part of the multi-abstraction model. The hardware model is the task-specific abstraction of the concrete hardware. For example, in order to perform a defined task, a system requires a set of sensors and actuators. These enable it to perceive and influence its environment, i.e. by recognizing a specific color or playing back an audio sample. The actual realization of these sensors and actuators is not part of the hardware model. It only describes which facilities exist and the tasks they fulfill.

Engines

Due to their nature, abstract models can neither be run directly on a processor, nor do they provide the driver interface required for the inter-model communication. Model engines bridge this gap by expressing the functionality of their models in a programming language, which supports the notion of function calls and is compatible with LLVM. Therefore the languages which qualify for engine development are the front-end languages and the intermediate representation of LLVM, which is explained in detail in Section 3.1.4.
The compiler developed in Chapter 5 is a tool for the generation of state machine engines. When the state machine model is expressed in the language introduced in Chapter 4, it automatically generates an appropriate engine, by transforming the input model into a set of functions, which simulate the state machine. All compiled state machine engines contribute at least one global update-function to the common driver interface. Calling this function triggers a discrete time-step in the simulated state machine.

The engines for domain-specific models need to be supplied manually. These can either take the form of model-interpreters or can be generated by custom compilers.

The hardware-drivers create the link between the abstract model of the hardware and its concrete realization. The signatures of the drivers reflect the hardware model. Therefore they can, to some extent, remain constant even when the specific hardware changes. In contrast the implementation of the drivers necessarily depends heavily on the specific hardware.

### 3.1.2 Common Driver Interface

This section motivates the use of a common driver interface as means of model-to-hardware communication as well as method of inter-model communication.

Most sensible embedded programs require a way of communicating with processor-internal and peripheral hardware to interact with the outside world. Input from sensors must be received, processed and transformed into control commands for a set of actuators. The processing step is usually performed in high-level application code. This code interacts with the underlying hardware through a layer of driver functions. These driver functions are modularized by the specific peripherals they interact with. To facilitate driver development, many vendors supply hardware-abstraction libraries which provide meaningful APIs to expose the processor-integrated hardware. The hardware-abstraction libraries communicate with the actual hardware by writing specific bytes to defined control registers in the processor. Figure 3.2 gives an overview of this layered system architecture.

Model based approaches aim at replacing the application code with an abstraction of the system’s operation, i.e. a state machine. In this setup the state machine is responsible for receiving input and controlling the actuators. This is achieved by linking the model’s input and output actions to specific driver functions. Figure 3.3 demonstrates how a state machine model interacts with the underlying hardware by mapping input and output actions to peripheral drivers.
This thesis proposes to use the common state machine formalism introduced in Section 3.2 to describe the general system-operation. Other aspects of the system are represented more naturally in domain-specific models. These models are integrated with the state machine through a common driver interface. The idea is that from the state machine’s point of view, communicating with domain-specific abstractions is essentially the same as communicating with the underlying hardware. Both provide the state machine with: (1) an input channel providing information to guide its decision-making process; (2) an output channel enabling the state machine to send control signals. Therefore the same concept of communication, namely driver calls, is also applied to integrate the domain-specific abstractions with the state machine. For this purpose all abstractions, including the state machine model, must expose their view of the system through a set of driver functions, which use compatible calling conventions.

### 3.1.3 State Machine Executor

As explained previously, the state machine compiler generates global update-functions for all state machine models. For the model to actually take control of the embedded system, these update functions need to be called periodically by the state machine executor. The state machine executor is a piece of code, which is integrated into the embedded system’s startup code. In the simplest case it consists of a loop, which calls the state machine’s update function once in every iteration. In more complex setups, the update function is called periodically by means of a timer interrupt or any other recurring signal.

### 3.1.4 LLVM-Based Hardware Abstraction

*LLVM* today is the literal name of an open-source compiler infrastructure. The project aims at providing a modular collection of tools and libraries to facilitate the development of static and just-in-time compilers.

LLVM implements a traditional three-phase compiler design[33] consisting of language front-ends, a common optimizer as well as platform-specific code generation back-ends. The front-ends are in charge of parsing the high-level input languages and translating the programs into a specified intermediate code format. This representation is the input-and output-format of the common optimizer. The optimized intermediate code is passed to the back-ends, which generate platform-specific machine code. The use of a common intermediate format makes this three phase design very efficient, because it enables the reuse of existing optimizers and code-generators. Every new front-end language can be automatically optimized and processed by all existing back-ends[6]. Figure 3.4 gives an overview of LLVM’s implementation of the three-phase compiler design.

Even though the advantages of the three-phase design are obvious, it is not consistently implemented by popular compilers. The established *GCC* also uses an intermediate code representation, which however is not the singular input required by the optimizers and back-ends. Due to the specific requirements of the different back-ends, GCC-based cross-compilers are commonly distributed separately. Example distributions are GCC-ARM and AVR-GCC for the ARM and Atmel AVR platforms.
In contrast LLVM stands out, because its intermediate representation (IR) is the complete and singular input used by the optimizers and the back-ends. It includes the main program logic as well as meta-information (i.e. function calling conventions). The appearance of LLVM-IR is that of an assembly language with several higher-language features. These include strong-typing, composite structures\(^1\) as well as support for exception handling. Due to the power of the intermediate representation, LLVM is a native cross-compiler. Provided with the correct target-triple, the unique identification of the target-platform, the LLVM back-ends directly generate machine code for a variety of different platforms. As of today, LLVM supports thirteen different platforms, including X86, X86-64, ARM and MIPS.

Due to the broad spectrum of supported target platforms, and the relative simplicity of creating new front ends, LLVM was chosen as the hardware abstraction for the MBSD toolchain developed in this thesis. The associated compiler front end parses models in the custom state machine language and generates equivalent intermediate code. The intermediate code can then be optimized and translated to the desired target platform's machine code by LLVM. The mechanics of the compiler and the necessary workflow are the subject matter of Chapter 5.

Model engines for the domain-specific abstractions can theoretically be developed in any of the programming languages supported by LLVM. The use of compatible function calling conventions constitutes the only restriction. When no other calling convention is explicitly specified, LLVM assumes the use of the C-language calling convention. Therefore engines written in the C-language will work without further adaptation. For other languages, the calling conventions can be changed in the generated intermediate code. The case-studies in Chapters 6 and 7 use C-based model engines.

### 3.2 State Machine Formalism

Finite state automata are a well-studied and versatile abstraction of sequential procedures. They describe the execution of concrete tasks through abstract machines, which process defined input to create a desired output. In general, a state machine is defined by a finite set of states, among those one initially active state, as well as a transitions relation. The transitions relation describes how the active state changes with time and input. Two basic categories of state machines are the Moore- and Mealy-Machines. These differ in the way they create output. For Moore-Machines, the output depends only on the current state. In

\(^1\)Comparable to structs in the C language
other words, output is produced when entering a state. For Mealy-Machines, the output depends on the current state and input, meaning the output is produced upon performing transitions[1].

Both types of traditional state machines share two significant limitations which make them impractical for the development of real-world applications. (1) They lack a concept of modularization. Real-world applications require very large numbers of states and transitions. Without any means of sub-structuring state machines by coherent blocks of functionality, models become difficult to understand, maintain and extend. (2) They lack a concept of state variables. Since there is no memory, the active state and current input provide the exclusive basis for the selection of the following state[1].

Figure 3.5 (a) shows an example procedure which cannot be directly expressed by Moore- or Mealy-Automata. The problem is that as soon as state C is entered, there is no memory of the previous state A or B. Therefore it is impossible to allow the correct, green paths A-C-D and B-C-E, without also allowing the forbidden, red paths A-C-E and B-C-D. In order to emulate the correct behavior in simple state machines, the critical crossing of paths in state C needs to be avoided. This can be achieved by duplicating the intersecting part of the paths as shown in Figure 3.5 (b). This solution has obvious drawbacks. The number of states is effectively doubled at every binary decision point. Apart from increasing the required amount of memory, the introduced redundancy can easily lead to inconsistencies.

### 3.2.1 Extended Finite State Machines

Since the concept of state machines has been around for a long time, variants have been developed to overcome their limitations. The extended finite state machine, in the following referred to as EFSM, is a formalism, which adds the missing concepts indicated in the previous section to the Mealy Machine.

Modularization is achieved through the refinement of the previously known simple states to composite states. Composite states are containers for sub-state machines which encapsulate specific sub-tasks. The incoming transitions of a composite state must lead directly to one of its sub-states. Outgoing transitions of a composite state can be traversed, when any one of its sub-states is active. This way common inputs can be handled across all of the contained sub-states. When composite states are nested, multiple transitions on different layers of
hierarchy could thus be applicable. In this case the transition originating from the innermost composite state is prioritized[1].

EFSMs may feature an arbitrary number of state-variables. The overall state of an EFSM is called *extended state*. It is defined by the machine's active state and the current valuation of its state-variables. Conditions on the transitions of EFSMs are boolean predicates on the variables as well as the current input. Output actions may modify variables as well as produce output. Variables enable EFSMs to solve the conditional independence problem of Figure 3.5 (a), by setting an appropriate flag-variable when entering states A or B. The flag-variable is then queried by the outgoing transitions of state C, to select the correct following state [1].

**UML Statecharts**

UML statecharts were developed as a visual EFSM notation². To increase their usefulness in the development of real-world applications, several additional features distinguish them from EFSMs. These include but are (by far) not limited to: (1) state-entry and exit actions, (2) orthogonal regions as well as (3) history-based composite state entry.

(1) State-entry- and exit-actions supplement the EFSM's transition-bound output actions with common, state-based output actions. Whenever a state is entered or exited, a specific action is triggered, independent of the active transition. (2) Orthogonal regions divide the behavior of states into multiple disjoint fragments, which may be executed in parallel. (3) History-based composite state entry enables composite states to pick up their execution in the state they were last in before exiting[23, 15].

In idealized systems, the processing of states and transitions happens instantly. For real-world applications, this is obviously not the case. The execution of a state or transition puts the system into an unstable state for a specific amount of time, before it stabilizes again. This amount of time depends primarily on the task to be performed and thus varies for different states or transitions. In UML statecharts, new input can only be processed when the system is in a stable state. Events which occur while the system is unstable, are queued until they can be handled. No action is taken before the previous one has been completed. This execution model is called *discrete-time run-to-completion semantics*[23].

The semantics of UML statecharts are not formally specified and leave many details open to the interpretation of the user or UML tool developer. Two examples for under-specified semantics in the statechart specification are the evaluation of transition guards and the execution policy for orthogonal regions. Statecharts allow multiple outgoing transitions of the same state to be applicable at the same time. Which transition to select in this case is left under-specified. A similar situation applies to the execution of orthogonal regions. The order of sequential execution or the semantics of parallel execution are not defined in detail [23]. Due to these ambiguities, different UML tools do not necessarily implement the

---

²Based on Harel Statecharts

---

3.2 State Machine Formalism
same semantics. In other words, the same model behaves differently depending on the used tool.

### 3.2.2 Custom Hierarchically Modular State Machines

The state machine modeling language developed for this thesis uses a custom variant of the EFSM. It employs the statechart’s *discrete-time run-to-completion semantics* as well as the entry- and exit-actions. A modified version of the composite states with history-based entry is applied for its method of modularization. In addition to the concepts derived from EFSMs and statecharts, it features a concept of probabilistic transitions. In contrast to UML statecharts, the semantics of the custom modeling language are formally specified. This provides an accurate guideline for the development of compilers such as the one in Chapter 5 and avoids misconceptions.

Similar to statecharts, this approach employs hierarchically organized, independent state machines. These are called modules and encapsulate common operations comparable to the functions in many programming languages. Exactly one module is active at any point in time. The active state of the active module is called the globally active state. Only the globally active state receives input, performs transitions and may signal other modules to become active. When this happens, the active module is suspended but its final state is preserved. The signaled module takes over. When the suspended module is signaled the next time, it picks up execution in the saved state. Apart from the current state and the current valuation of variables, transitions may depend on the active states of all modules. This way, inactive modules can also serve as memory, enabling a straight-forward solution to the conditional-independence problem. There is no need to resort to setting external flags. This is demonstrated in Figure 3.7.

Probabilistic transitions allow the specification of multiple targets for a single transition. Each target is associated with a probability value. When the probabilistic transition is activated, a proportional random selection based on the probabilities determines the target state.

The syntax and formal semantics of the custom modeling language are covered in detail in Chapter 4.
This chapter introduces the syntax and semantics of the custom state machine modeling language, which is a central element of the multi-abstraction modeling approach described in Chapter 3.

4.1 Modeling Language Syntax

In this section, the syntactic building blocks of the modeling language are introduced by example in a top-down fashion. Modules, states and transitions are covered in separate subsections.

Angle brackets (<> ) are used to mark placeholders for the concepts they surround. Alternatives are indicated by two pipe symbols (|| ). Regex-style repetitions are specified, by appending the plus (+) or the asterisk (*) symbols.

Module Definitions

Modules are the primary means of structuring models. Each model requires exactly one topmost main module. This module contains an arbitrary number of sub-modules. The latter can then extend the tree, by recursively containing further sub-modules. Exactly one module is active at every point in time. Initially this is the main module. All modules excepting the active one retain their state when the model is updated. The active module is changed by means of Signal and Suspend Statements, which are covered further on in this section. Modules in the same sub-tree perform their transitions sequentially depending on the order in which they are signaled. Each module contains an arbitrary, non-zero number of state definitions.

The syntax for the definition of modules is shown in Listing 4.1.

Listing 4.1: Module Definition Syntax

```plaintext
module <ModuleName> {
     <(StateDefinition || ModuleDefinition)+>
}
```

Names must be unique between all sibling modules in a sub-tree of the model.
State Definitions

Each state definition adds one state to its parent module. Exactly one state per model is active at any point in time. Therefore it is necessary to explicitly indicate the initial states for all modules. The bodies of state definitions are comprised of three parts: driver invocations, signal statements and transitions. This is also the order in which they are processed when a state is entered. A transitions relation, encoded into the states, describes how the active state changes with time and input.

The syntax for the definition of states is shown in Listing 4.2.

Listing 4.2: State Definition Syntax

```
init state <StateName> {  
   <DriverInvocation+>
   <SignalStatement+>
   <Transitions+>
}
```

The `init` keyword indicates the initial state and must thus be present on exactly one state per module.

Driver Invocations

Driver invocations constitute the low-level interface to the underlying hardware. The syntax for calling a specific driver is shown in Listing 4.3.

Listing 4.3: Driver Invocation Syntax

```
<DriverName>(<FirstParameter>, . . . , <LastParamter>);
```

The driver name is simply an identifier, which the compiler uses to link this specific invocation against an externally defined driver. Within the parenthesis, comma-separated parameters can be specified. These may be double-quoted strings, integer numbers, floating point numbers or nested driver invocations depending on the driver function's signature.

Signal Statements

Signal Statements are employed to change the active module. When the currently active module issues a signal to a sub-module, its execution is suspended. The sub-module takes its place until it explicitly suspends itself. Thereupon, the signaling module is reactivated and execution continues upon the next update with the directive following the processed signal. The syntax for signaling a sub-module is shown in Listing 4.4.

Listing 4.4: Signaling Sub-Modules Syntax

```
signal <SubmoduleName>;
```
Transitions

Transitions determine the possible following states of the currently active state. There are two types of transitions: continue and suspend. The difference between the two types is that suspend transitions suspend the active module as well as changing its state to target state. The execution flow then continues in the parent module. However, when the suspended module is signaled the next time, it continues in target state instead of its initial state. Continue transitions change only the active state to target state. Both types of transitions may feature a condition, indicating when it is to be applied. Conditions are boolean expressions on driver invocations and need to be mutually exclusive. When a transition has no condition, it is regarded as always applicable. It is possible that none of the conditions evaluate to true for the active state when the module is updated. Then the module remains in its current state for all proceeding updates, until a transition does become applicable. The basic syntax for defining a transition is shown in Listing 4.5.

Listing 4.5: Defining Continue and Suspend Transitions

transition (continue <TargetState>);
transition(<SampleDriverCall> == <SampleValue> -> continue <TargetState>);

transition (suspend <TargetState>);
transition(<SampleDriverCall> == <SampleValue> -> suspend <TargetState>);

This basic syntax does not yet allow for specifying drivers to call when taking a certain transition. Listing 4.6 shows how driver calls are attached to transitions.

Listing 4.6: Driver Calls on Transitions

transition({<SampleDriverCall1>;} continue <TargetState>);
transition(<Condition1> -> {<SampleDriverCall1>;<SampleDriverCall2>;} suspend <TargetState>);

Probabilistic Selection of Target States allows specifying several target states per transition. Each is assigned a specific probability, adding up to one. A fitness-proportional random selection is then used to determine the next active state. The syntax for defining probabilistic selections is demonstrated in Listing 4.7.

Listing 4.7: Defining Probabilistic Selection of Transition Sets

transition (<Condition1> -> probselect {
    <probability> : {<SampleDriverCalls>;} continue <TargetState1>;
    <1 - probability> : {<SampleDriverCalls>;} suspend <TargetState2>;
});
4.1.1 **Graphical Notation**

Graphical notations are a valuable aid in the process of understanding models. They allow the observer to first gain a structural overview of a system, before dealing with the details of implementation. This way the reader is given a conceptual understanding, which acts as guide when actually analyzing the source code. The graphical notation developed for the state machine modeling language is depicted in Figure 4.1. This visualization does not show driver invocations in states and on transitions. Therefore it is not equivalent to the source code representation.

Subfigure (a) demonstrates how modules and module-hierarchy are visualized. Each module is represented by a rectangle with rounded corners. Sub-modules are drawn within the boundaries of their parent module. States are visualized by circles as shown in Subfigure (b). A thick border is used to highlight the modules’ initial states.

In Subfigure (c), the graphical notation for sub-module signaling is presented. Dashed, red arrows from states to sub-modules are employed to visualize signal statements. When multiple signals are present in one state, the order of signaling is indicated by the integers in the captions of the arrows.

The two types of transitions are shown in Subfigure (d). Both are represented by arrows from source- to target states. If present, the arrows are labeled with conditions. Suspend transitions are distinguished from continue transitions by their dashed style.

Probabilistic transitions use branching arrows as shown in Subfigure (d). Conditions are common to all probabilistic selection options of one transition. Therefore they label the root part of the branching arrow, while the different probability values label the branches. Similar to normal transitions, the branches may be of the regular continue or the dashed suspend type.

**Fig. 4.1.:** Graphical Representation of Language Elements

(a) Modules

(b) States

(c) Signals

(d) Transitions

(e) Probabilistic Transitions
Section 4.1 introduced the syntactic elements available in the state machine modeling language. In this section, they are applied to build a first example model. The aim is to create a compact model, which gives a horizontal overview of all language features. This model is kept as simple as possible, as it will be used in Section 4.3 to illustrate the semantics definitions. Figure 4.3 shows the model in graphical notation and Listing 4.8 contains the complete source code.

The visual representation shows that the example features two modules, Main and Blink (ll. 1, 20), which are represented as rectangles with rounded corners. The graphical containment of Blink indicates that it is a sub-module of Main. The states inside each of the modules are visualized by blue circles (ll. 2, 11, 21, 25), a thick border highlighting the initial states A and Off (ll. 2, 21). Transitions are visualized by black arrows labeled with their conditions and/or probabilities (ll. 7, 8, 14, 22, 26). Regular arrows represent the continue transitions, while the suspend transitions use dashed arrows (ll. 22, 26). The dashed red arrow expresses the signal statement in state A to module Blink (ll. 5). The driver calls inside states and on transitions, while present in the source code, are not shown in the visualization (ll. 3, 12, 22, 26).

This example model portrays a simple technical system, which is always in one of four states. The system is initially in state A, before it branches to the sub-module Blink. Initially Off, Blink toggles its state and suspends itself every time it is signaled. State A is resumed and, depending on the Blink sub-module’s new state, branches either to B or reenters A. In state B, a probabilistic selection is performed. With a 70% chance, execution continues in A, otherwise B is reentered. Whenever A or B are entered, this is logged by the print driver. Whenever the Blink sub-module is toggled, the appropriate setLightOn or setLightOff driver is called.

### 4.3 Modeling Language Semantics

This section deals with the formalization of the modeling language's semantics. Therefore the following definitions constitute the foundation for the development of compilers such as the one which will be introduced in Chapter 5. At first, environments, structures, mechanics and transitions are defined in separate subsections. Each definition will be illustrated by demonstrating, how the example model from Section 4.2 is represented in terms of the definition. Using these definitions, the operational semantics of model-updates are then introduced.
Listing 4.8: Example Model Source Code

```java
module Main {
  init state A {
    print("Enter A");
    signal Blink;
    transition(Blink == On -> continue B);
    transition(Blink == Off -> continue A);
  }
  state B {
    print("Enter B");
    transition(probselect {
      0.7: continue A;
      0.3: continue B;
    });
  }
  module Blink {
    init state Off {
      transition({setLightOn();} suspend On);
    }
    state On {
      transition({setLightOff();} suspend Off);
    }
  }
}
```
4.3.1 Environment

Environments are contexts in which models are processed. They provide the link between the state machine model and the underlying system to control. Most importantly this involves providing a representation of the underlying system’s state and a means of sending control instructions. Changing the environment of a model modifies its operation, without touching its structure.

\[ \text{Env} = (\text{Dom}, \text{Var}, \text{Val}, D, C, R) \]

\[ \text{Dom} \neq \emptyset: \text{Domain} \]
\[ \text{Var} = \{v_0, \ldots, v_m\}: \text{Set of Variables} \]
\[ \text{Val} = \{\text{va}_0, \ldots, \text{va}_n|\text{va}_i: V \rightarrow \text{Dom}\}: \text{Set of Valuation Functions} \]
\[ D = \{d_0, \ldots, d_p|d_i: \text{Val} \rightarrow \text{Val}\}: \text{Set of Driver Functions} \]
\[ C = \{c_0, \ldots, c_q|c_i: \text{Val} \rightarrow \{\text{true, false}\}\}: \text{Set of Conditions} \]
\[ R \in \mathbb{R}^*_{[0,1]}: \text{Random Number Sequence} \]

An environment is a 6-tuple of the following elements: The Domain(1) is a non-empty set of symbols, which serve as values for the set of variables(2). These are used to track the system-state outside of the model. This grants the engineer the freedom to decide, which elements of state to handle explicitly within the model. Further, it provides a convenient way of incorporating environment state outside of the engineer’s direct control. Valuations(3) provide a mapping for every variable to a specific domain-value. Drivers(4) are functions which transfer one valuation into another. They are employed to interact with the execution-environment from inside of the model. Condition(5) functions are used on transitions and map valuations to truth values. Finally, a sequence of (pseudo-)random numbers(6) is necessary for performing probabilistic selections during model-updates.

Fig. 4.4.: Possible Environment Definition for Example Model

\[ \text{Dom} = \{0, 1\}, V = \{\text{outputPin}\}, \text{Val} = \{\{\text{outputPin, 0}\}, \{\text{outputPin, 1}\}\}, \]
\[ D = \{\text{print, setLightOn, setLightOff}\}, \]
\[ C = \{\text{alwaysTrue, blinkIsOn, blinkIsOff}\}, R = [0.95, 0.87, 0.73, 0.50, ...] \]

It is possible to come up with an infinite number of environments for any model. For the purpose of illustrating the definition on the example model, only one option is shown in Figure 4.4. This environment links the model to a hypothetical hardware setup, consisting of a microcontroller with an attached LED. As domain, the set containing only the numbers zero and one is chosen. A single variable outputPin corresponds to a hardware register, which sets the electrical state of the output pin connected to the LED.

In the example, the driver print is used for posting debug messages; setLightOn and setLightOff modify the value of outputPin to toggle the LED. The alwaysTrue condition is used for transitions which do not specify a condition. BlinkIsOn and blinkIsOff encapsulate the expressions located on A’s outgoing transitions, which evaluate the Button sub-module’s state. For the beginning of R, four pseudo-random real numbers were arbitrarily chosen.
4.3.2 Structure

Model-structures describe the layout models. For this purpose it is necessary to specify all modules and states as well as their hierarchy and containment.

\[ Mod = (m_0, M, M_H, S, S_H) \]

- \( m_0 \): Main Module
- \( M = \{m_1, \ldots, m_m\} \): Set of Sub-Modules
- \( M_H : M \to ((m_0) \cup M) \land \forall m_a : f(f(\ldots f(m_a) \ldots)) = m_a \): Module Hierarchy Function
- \( S = \{s_0, \ldots, s_n\} \): Set of States
- \( S_H : S \to ((m_0) \cup M) \): State Hierarchy Function

A model-structure is a 5-tuple of the following elements: Each model features at least the main module(1), which may span a tree of sub-modules(2). The structure of this tree is defined by means of the acyclic module-hierarchy function(3). The latter assigns exactly one parent-module to each sub-module. Modules are containers for structurally independent (sub-)state machines. In this capacity, they contain an arbitrary, non-zero number of states(4). The state-hierarchy function(5) portrays this by assigning containing modules to all states.

Fig. 4.5.: Example Model Structure Definition

\[ m_0 = \text{Main}, M = \{\text{Blink}\}, M_H = \{(\text{Blink}, \text{Main})\}, \]
\[ S = \{A, B, \text{Off}, \text{On}\}, \]
\[ S_H = \{(A, \text{Main}), (B, \text{Main}), (\text{Off}, \text{Blink}), (\text{On}, \text{Blink})\} \]

The example model contains only the main module Main and the sub-module Blink. The latter is located directly inside the main module, therefore the \( M_H \) contains only the tuple (Blink, Main). The four states are defined in \( S \). \( S_H \) is populated to reflect that A and B belong to the main module, while Off and On reside in the Blink module.

4.3.3 Model Mechanics

Model Mechanics describe the internal operation of states. This is achieved by specifying the drivers to invoke and the sub-modules to signal when each state is entered.

\[ ModMech = (S_D, M_S) \]

- \( S_D : S \to D^* \): State Drivers Function
- \( M_S : S \to M^* \): Module Signal Function

Model-mechanics are tuples consisting of the following elements: When a state is entered, a possibly empty sequence of drivers is invoked. Which drivers to execute in which state, is
defined by the state drivers function. Similarly, the sub-modules to signal as well as the order of signaling are determined by the module signal function.

Fig. 4.6: Example Model Mechanics Definition

\[ S_D = \{(A, 0, \text{print}("A")), (B, 0, \text{print}("B"))\} \]
\[ M_S = \{(A, 0, \text{Blink})\} \]

Figure 4.6 shows how the example is represented in terms of the model mechanics definition. Upon entering each of the states A or B, the print driver is called to log the new state. The signal statement in state A to the sub-module Button is encoded in \( M_S \).

4.3.4 Transitions

The possible following-states for each source-state are defined by means of the transitions relation. Conditions, probabilistic selection options and driver calls for all transitions need to be specified.

\[ T \subseteq (S, C, \mathbb{R}^{[0,1]}, \mathbb{R}^{[0,1]}) \times ([\text{CONTINUE, SUSPEND}], S, D^*) \]

This relation maps input tuples of the form \((source, condition, lowerbound, upperbound)\) to output tuples of the form \((type, destination, drivers)\).

The condition is a special function which determines whether the transition is applicable. Two constraints need to be respected by conditions. First, conditions use side-effect-free drivers. Otherwise the evaluation of a failing condition could influence the satisfiability of all other conditions, resulting in undefined behavior. Second, conditions in the same state must be mutually exclusive. This means that at most one of the concerned driver functions may return true at any point in time.

The interval \([lowerbound, upperbound]\)^1 defines an option for probabilistic selection^2. The options on one transition may not overlap and must cover the entire range from zero to one. The output tuple denotes whether the transition is of the continue or suspend type^3, as well as the target state. A driver sequence to execute when actually taking the transition is specified in the final element of the tuple. Otherwise the empty sequence is used.

For the example in Figure 4.7, the transitions relation contains one entry for each of the six transitions. In the example, there are two probabilistic selection options on the outgoing

---

^1Interval Notation \([\text{inclusive-lower-bound, exclusive-upper-bound}]\)
^2Deterministic transitions use the interval \([0, 1]\)
^3The difference between the transition types is explained in the Model-Update section.
Fig. 4.7.: Example Model’s Transitions Relation

\[
T = \{(4.1), (A, \text{blinkIsOn}, 0, 1, \text{CONTINUE}, B, []), (4.2), (A, \text{blinkIsOff}, 0, 1, \text{CONTINUE}, A, []), (4.3), (B, \text{alwaysTrue}, 0, 0.7, \text{CONTINUE}, A, []), (4.4), (B, \text{alwaysTrue}, 0.7, 1, \text{CONTINUE}, B, []), (4.5), (\text{Off}, \text{alwaysTrue}, 0, 1, \text{SUSPEND}, \text{On}, [\text{setLightOn}]), (4.6), (\text{On}, \text{alwaysTrue}, 0, 1, \text{SUSPEND}, \text{Off}, [\text{setLightOff}])\}
\]

transition of state B. 70% is represented as [0, 0.7) and 30% as [0.7, 1). The driver calls on the two transitions in the Blink module are encoded in the final two tuples. The other transitions invoke no drivers and thus use the empty sequence [].

4.3.5 Operational Semantics

In this section, the operational-semantics of the modeling language are specified. This is achieved by defining how the dynamic model state changes with every timestep. Therefore, the dynamic model state must first be defined. By applying the previous definitions, the next state mechanism is then introduced. It relies on three sub-steps which transfer a dynamic source state into its following-state \(state'\). These steps driver calls, signal statements and transitions, will be covered individually. They are then utilized to define the next-state relation.

To simplify the otherwise redundant equations, these conventions are used: \(S_{GA} := A_S(A_M)\):

- Globally Active State
- \(Relation(p_1, p_2, \ldots, p_n) \iff (p_1, p_2, \ldots, p_n) \in Relation\): Tuple is Element of Relation
- \(Relation(p_1, p_2, \ldots, p_n) \not\iff (p_1, p_2, \ldots, p_n) \not\in Relation\): Tuple is not Element of Relation
- \(\oplus\): Exclusive Or Operator

Dynamic Model State

The operational semantics of the modeling language are specified by defining how following states for any source state are computed. Dynamic states represent the specific configurations, models take at discrete time-steps during execution.

\[
ModSt = (A_M, A_S, A_P, A_V, R_I)
\]

- \(A_M \in (\{m_0\} \cup M)\): Active Module
- \(A_S : (\{m_0\} \cup M) \to S\): Active States Function
- \(A_P : (S \to (\{\alpha, \omega\} \cup N_0))\): Active Phase Function
- \(A_V \in Val\): Active Valuation
The state of a model is characterized by a 5-tuple consisting of the following components: The active module(1) is the module to process during the next model-update. Every module has exactly one active state at every point in time. This active states function(2) specifies the states to process when their containing modules are updated. The active state of the active module is called the globally active state. The execution of a state is always in one of three phases according to the active phase relation(3): \( \alpha \), when the state is entered and the state drivers have not yet been executed; an integer number \( i \in \mathbb{N} \) indicating which signal to process next; or \( \omega \) when the drivers and signals sections have been completed and state-transitions are pending. Finally, the active valuation(4) provides values for all variables and the random index(5) tracks the current position in the random number sequence.

Initialization Every model needs to be in a defined state at the time of initialization. Primarily active modules and states, as well as a configuration of variables, need to be specified.

\[
\text{Init-ModSt} = (m_0, I, \{(s_i, \alpha) | s_i \in S\}, v_{a0}, 0)
\]

\( m_0 \): Main module
\( I : (\{m_0\} \cup M) \rightarrow S \): Initial States Function
\( v_{a0} \): Initial Valuation

The initially active module is always the topmost main module, \( m_0 \). An initial states function must be specified as starting point for the modules’ active states. Concerning the syntax described in Section 4.1 this function is easily derived from the init keywords, which mark the initial states of all modules. All states start out in the \( \alpha \) phase, to ensure their execution starts with the driver invocations section. An initial valuation is provided, to describe the underlying system’s state at the time of initialization. The random index is set to the value zero. The initial state of the example model is shown in Figure 4.8.

Fig. 4.8.: Example Model Initial State

\[
\begin{align*}
A_M &= \text{Main} \\
A_S &= \{(\text{Main, A}), (\text{Button, Off})\} \\
A_P &= \{(A, \alpha), (B, \alpha), (C, \alpha), (\text{Off, } \alpha), (\text{On, } \alpha)\} \\
A_V &= \{(\text{outputPin, 0})\}, \\
R_I &= 0
\end{align*}
\]
**Functional Overview of the Next-State Relation**

Figure 4.9 gives an overview of how the next-state relation is defined by combining the state driver calls, the sub-module signals and the transitions relations. It also illustrates, that exactly one child-relation is selected based on the globally active state’s phase.

**Fig. 4.9.:** Functional Overview of Operational Semantics

\[
\begin{align*}
AP(A_S) = \alpha \rightarrow & \text{state}’ = \\
\begin{cases} 
A’_M = A_M \\
A’_S = A_S \\
(A’_P, A’_V) = \text{drivers(state)} \\
R’_I = R_I
\end{cases} \\
\}\text{Next-State(state)} = \\
AP(A_S) \in \mathbb{N} \rightarrow & \text{state}’ = \\
\begin{cases} 
A’_S = A_S \\
(A’_M, A’_P) = \text{signals(state)} \\
A’_V = A_V \\
R’_I = R_I
\end{cases} \\
\end{align*}
\]

Drivers: \(\text{ModSt} \rightarrow AP \times \text{Val} \quad \text{state} \mapsto (A’_P, A’_V)\)

Signals: \(\text{ModSt} \rightarrow AM \times AP \quad \text{state} \mapsto (A’_M, A’_P)\)

Transitions: \(\text{ModSt} \rightarrow \text{ModSt} \quad \text{state} \mapsto \text{state}’\)

**State Driver Calls**

State Driver Calls are processed as first action when entering a new state. Concerning the model-state, the active state’s phase and the current valuation are transitioned into the state’s next phase \(A’_p\), and the new valuation \(A’_V\) (3.1).

\[
\begin{align*}
\text{Drivers}(\text{ModSt}, A’_P, A’_V) & \iff \\
[(SD(SGA, \emptyset) \land A’_V = A_V)] & \quad (4.1) \\
\oplus(SD(SGA, [d_0, \ldots, d_n]) \land va_0 = d_0(V) \land va_1 = d_1(va_0) \land \cdots \land A’_V = d_n(va_{n-1})) & \quad (4.2) \\
\land[(MS(SGA, \_)) \land A’_P = (AP\{\{SGA, \alpha\}\} \cup \{SGA, 0\})) & \quad (4.3) \\
\oplus(MS(SGA, 0, \_)) \land A’_P = (AP\{\{SGA, \alpha\}\} \cup \{SGA, \omega\})] & \quad (4.4)
\end{align*}
\]

States are not required to have driver calls. If the driver sequence is empty, the current valuation is not changed (3.2). If drivers are present, they are transitively applied to the current valuation, producing temporary valuations \(va_0\) to \(va_{n-1}\). The final driver emits the new valuation \(A’_V\) (3.3). Finally, the state’s phase is set either to the first sub-module signal (3.4) or to \(\omega\) when there are no signal statements (3.5).
Sub-Module Signals

Sub-Module Signals advance a state’s current phase and change the active module (3.6).

\[ \text{Signals} (\text{ModSt}, A'_M, A'_P) \iff \]
\[ i = A_P(A) \land [(M_S(S_{GA}, \ldots, m_i = A'_M, \ldots)] \land \]
\[ [(M_S(S_{GA}, \ldots, m_{i+1}, \ldots)] \land A'_P = (A_P \setminus \{(S_{GA}, i)\}) \cup \{(S_{GA}, i + 1)\}] \]
\[ \oplus ((M_S(S_{GA}, \ldots, m_{i+1}, \ldots)] \land A'_P = (A_P \setminus \{(S_{GA}, i)\}) \cup \{(S_{GA}, \omega)\})] \]
\[ \oplus (M_S(S_{GA}, \ldots, m_{i+1}, \ldots)] \land A'_M = A_M \land A'_P = (A_P \setminus \{(S_{GA}, i)\}) \cup \{(S_{GA}, \omega)\})] \]

(4.6)
(4.7)
(4.8)
(4.9)
(4.10)

To process the next signal of a state, the signal’s index \( i \in \mathbb{N}_0 \) is retrieved from the active phase relation (3.7). The index is then used to look up the next active module \( A'_M \) in the module signals relation (3.7). If a corresponding entry is present, the state’s phase is advanced either to \( i + 1 \) (3.8) or to \( \omega \) (3.9), depending on the existence of an \((i + 1)\)th signal. Otherwise, the active module remains unchanged and the state’s phase is advanced directly to \( \omega \) (3.10).

Transitions

A constraint regarding transitions’ conditions is employed. Without loss of generality, only conditions consisting of single, atomic driver calls are allowed. This is legitimate, because any complex expression can be easily encapsulated into an atomic driver call. The constraint avoids the definition of a complete expression arithmetic and thus constitutes a significant facilitation.

\[ \text{Transitions} (\text{ModSt}, A'_M, A'_S, A'_P, A'_V, R'_T) \iff \]
\[ \text{Opts} = \{(c, r_0, r_1, t, s, [d_0, \ldots, d_n]) | (S_{GA}, c, r_0, r_1, \ldots) \in T \land c(V) = \text{true} \} \]
\[ \land \{(\text{Opts} = \emptyset \land A'_M = A_M, A'_S = A_S, A'_P = A_P, A'_V = A_V) \}
\[ \lor (R(R_t, P) \land R'_T = R_t + 1) \]
\[ \lor \{(C_X, R_t, X_T, [d_0, \ldots, d_n]) \}
\[ \land (\text{Opts}(d_p, r_0, r_1, t, s, [d_0, \ldots, d_n]) \land r_0 \leq P < r_1) \}
\[ \land (A'_P = (A_P \setminus \{(S_{GA}, \omega)\}) \cup \{(S_{GA}, \omega)\}) \land A'_S = (A_S \setminus \{(A_M, S_{GA})\}) \cup \{(A_M, S_X)\}) \}
\[ \land \{(T_X = \text{CONTINUE} \land A'_M = A_M) \lor (T_X = \text{SUSPEND} \land A'_M = M_H(A_M))) \}
\]

(4.11)
(4.12)
(4.13)
(4.14)
(4.15)
(4.16)
(4.17)
(4.18)
(4.19)

Transitions modify all components of the model-state (3.11). First, the active transition in the globally active state is determined by evaluating the available transitions’ condition drivers. Only those which return \textit{true} are kept (3.12). All conditions in the same state must be mutually exclusive. Therefore, the resulting set is either empty\(^4\), or corresponds to the probabilistic selection options (\text{Opts}) of the active transition.

Then, if no transition is active, the model-state remains unchanged (3.13). Otherwise, one of the available options is probabilistically selected as follows: The next random number, \( P \), is taken from the environment’s sequence \( R \), and the index in the sequence is incremented (3.14). The options are then filtered to include only those with

\(^4\)This is the case when no transition is active

43 Modeling Language Semantics 29
an interval which includes $P$. Since intervals of different probabilistic selection options cannot overlap, a single option is left (3.15/3.16). The transition’s drivers are then retrieved and processed (3.17), similar to state driver calls, resulting in the new valuation $V'_{A}$. The now completed state’s phase is then reset to $\alpha$, and the module’s active state is set to the transitions target state $S_{X}$ (3.18). Finally, depending on the transition’s type, the active module is changed (3.19). For suspend, transitions it is set to the currently active module’s parent. In case of continue transitions, it remains unchanged.

**Next State**

\[
\text{Next-State}(\text{ModSt}, (A'_{M}, A'_{S}, A'_{P}, V'_{A}, R'_{I})) \iff \left(\begin{array}{c}
(A_{P}(S_{GA}) = \alpha \land \text{Drivers}(\text{ModSt}, A'_{P}, V'_{A}) \land A'_{M} = A_{M} \land A'_{S} = A_{S} \land R'_{I} = R_{I}) \\
\oplus (A_{P}(S_{GA}, I) \land \text{Signals}(\text{ModSt}, A'_{M}, A'_{P}) \land A'_{S} = A_{S} \land V'_{A} = V_{A} \land R'_{I} = R_{I}) \\
\oplus (A_{P}(S_{GA}, \omega) \land \text{Transitions}(\text{ModSt}, A'_{M}, A'_{S}, A'_{P}, V'_{A}, R'_{I})))
\end{array}\right) (4.20)
\]

The next-state relation defines how to determine the following-state for any source state by combining the equations for state driver calls, sub-module signals and transitions. This is achieved by multiplexing to the correct relation, depending on the globally active state’s phase: $\alpha$ multiplexes to state driver calls (3.22), integers $i \in \mathbb{N}_{0}$ to sub-module signals (3.23) and $\omega$ to transitions (3.24).
This chapter introduces the compiler for the state machine modeling language introduced in Chapter 4. The compiler’s job is to transform state machine models into state machine engines. For this purpose, a LLVM front-end for the state machine language is developed. This front-end generates state machine engines in the LLVM intermediate representation, which can be cross-compiled for a variety of target platforms by the LLVM compiler.

Traditional compiler front-ends use a six step design consisting of lexical- and syntactical analysis, abstract syntax tree generation, linking, semantic validation and intermediate code-generation. The compiler developed in the following section implements this classic approach, but leaves out the semantic validation step. This means, that the compiler will not detect when it is provided with a syntactically valid yet erroneous state machine model.

Figure 5.1 gives an architectural overview of the compiler. Section 5.1 explains the first four steps of the compilation process. These steps are consecutively applied to the input state machine’s source code in order to transform it into an abstract syntax tree (in the following referred to as AST). The AST is the logical representation of the state machine model, which serves as input for the code generation step.

In order to simplify writing state machine models, one of the semantics definitions’ constraints is relaxed in the compiler implementation. The compiler does not enforce the mutual exclusivity of the conditions on the outgoing transitions of states. When multiple transitions inside one state are applicable at the same point in time, the transition, which appears first in the source code, is prioritized.
5.1 Lexical and Syntactic Analysis

The following subsections explain the steps necessary to transform a state machine model into an AST which is ready for the code generation.

5.1.1 Lexer

The lexer is the first component in the compiler architecture. Its task is to group the sequence of characters which represent the state machine model into defined sub sequences called tokens. The lexer for the state machine language generates unique tokens for keywords, identifiers, numbers, strings, arithmetic operators and special characters. Figure 5.2 illustrates the result of the lexing process for the example of a transition.

In order to classify a sequence of characters as a specific token, lexers usually employ regular expressions. A lexer grammar is therefore a sequence of rules consisting of a target token and an associated regular expression. Figure 5.3 shows five example rules used in the state machine lexer. The rules for the tokens `TRANSITION`, `EQ` and `SEMIC` match the characters in the right column literally. More complex regular expressions define character classes within brackets and use the plus\(^1\) and asterisk\(^2\) operators to define repetitions. Therefore the `INT` rule matches sequences of the digits zero to nine. The `IDENTIFIER` rule matches sequences of characters, which must begin with a lower or upper case letter and may be followed by further letters, digits or underscores.

---

\(^1\)At least one repetition
\(^2\)Zero or more repetitions
The rules are consecutively applied to the head of the character stream, until a match is found. When this happens, the associated token is produced and the matched characters are consumed. This process is repeated until the end of the stream is reached.

Since the task of programming lexers is very mechanical, lexer-generators have been developed to automate the process. Provided with a list of regular expressions and associated tokens, they generate lexer source code in different target languages. Prominent representatives of lexer generators for the C programming language are the proprietary tool Lex and its open-source alternative Flex. The latter was used to generate the lexer for the state machine modeling language. The lexer rules used for this purpose are included in Appendix A.1.1.

5.1.2 Parser

The parser performs the syntactic analysis on the stream of tokens produced by the lexer. This is achieved by verifying the state machine source code's conformity to a context free grammar of the modeling language.

A context free grammar is a tuple \( G = (\Sigma, N, P, S) \), consisting of a set \( \Sigma \) of terminal symbols, a set \( N \) of non-terminal symbols, a set \( P \subseteq N \times (\Sigma \cup N)^* \) of production rules\(^3\) and a start symbol \( S \in N \). Context free grammars are a generative formalism for the definition of languages. Beginning with the start symbol \( S \), the production rules are repeatedly applied until only terminal symbols remain. This way, all syntactically valid constructs of a language can be generated [34].

An example from the state machine modeling language is used to illustrate how context free grammars generate languages. The following grammar generates sequences of zero or more signal statements of the type which appears in state definitions:

\[
\begin{align*}
\Sigma &= \{ \text{SIGNAL, IDENTIFIER, SEMIC, } \epsilon \} \\
N &= \{ \text{Signal\_Seq} \} \\
P &= \{ \text{Signal\_Seq} \rightarrow \epsilon, \text{Signal\_Seq} \rightarrow \text{SIGNAL IDENTIFIER SEMIC Signal\_Seq} \} \\
S &= \text{Signal\_Seq}
\end{align*}
\]

The set of terminal symbols \( \Sigma \) contains the lexer tokens produced for the signal keyword, the identifiers and the semicolon character. Additionally it contains the empty word \( \epsilon \) to allow the termination of the sequence. The set of non-terminals \( N \) contains only the symbol \( \text{Signal\_Seq} \), which must therefore also be the start symbol \( S \). The set of production rules \( P \) contains two rules. Starting with \( \text{Signal\_Seq} \), the two production rules are repeatedly

\(^3\)The asterisk denotes the Kleene Star. \((\Sigma \cup N)^* \) is therefore the set of words over the combined terminal and non-terminal alphabets.
applied by replacing an occasion of the non-terminal on the left side of the rule, with the
symbols on the right side of the rule:

\[
\begin{align*}
\text{Signal}_\text{Seq} & \quad // \text{0 x Rule 2} \\
\underline{\text{SIGNAL IDENTIFIER SEMIC}} & \quad \text{Signal}_\text{Seq} \quad // \text{1 x Rule 2} \\
\text{One Statement} \\
\underline{\text{SIGNAL IDENTIFIER SEMIC SIGNAL IDENTIFIER SEMIC}} & \quad \text{Signal}_\text{Seq} \quad // \text{2 x Rule 2} \\
\text{Two Statements} \\
\underline{\text{SIGNAL IDENTIFIER SEMIC . . . SIGNAL IDENTIFIER SEMIC}} & \quad \text{Signal}_\text{Seq} \quad // \text{N x Rule 2} \\
\text{N Statements} \\
\text{// Termination using Rule 1} \\
\underline{\text{SIGNAL IDENTIFIER SEMIC . . . SIGNAL IDENTIFIER SEMIC}} & \\
\text{N Statements}
\end{align*}
\]

Each application of the second rule appends one more signal statement to the sequence. The
application of the first rule terminates the sequence, because afterwards no non-terminals
remain to be replaced.

A parser is a program which essentially performs the inverse of generating a language
using a context free grammar. Given a sequence of terminal symbols, a parser attempts
to discover whether this sequence is part of a target language and in which manner the
target grammar’s production rules can be applied to generate the sequence. A scheme of
rule applications resulting in a specific sequence of terminal symbols is called a derivation
of that sequence[34].

There are several types of parser algorithms which differ in the way they attempt to compute
derivations. Two important categories of algorithms are \textit{LL} and \textit{LR} parsers. The former
attempt to compute the derivation using a top-down-approach, the latter using a bottom-up-
approach. Parsers can be written manually. However the process is very error-prone. For
this reason generator tools were developed to facilitate the construction of parsers. Given
a context free grammar in the correct syntax, parser generators produce source code for
parsing algorithms recognizing the target language. Established representatives of generators
producing parser source code in the C programming language are \textit{Yacc} and \textit{Bison}. Both
produce a variant of the \textit{LR} parser which is called \textit{LALR} parser [33].

For the development of the state machine parser a less prominent tool called \textit{Lemon} was
used. It also generates \textit{LALR} parsers in the C language. \textit{Lemon} is part of the popular \textit{SQLite}
database project and features a more intuitive syntax and method of error-handling than the
much-older \textit{Yacc} and \textit{Bison} programs. The following example demonstrates \textit{Lemon}’s grammar
notation using the now familiar example of the sequence of signal statements:

1 signal_sequence ::= signal_sequence module_signal.
2 signal_sequence ::= .
3 module_signal ::= SIGNAL IDENTIFIER SEMIC.

Each of the above lines defines one production rule. To the left of the ::= operator, the
non-terminal to replace is defined. On the rule’s right side, the replacement is specified. A
full stop marks the end of each rule. In order for \textit{Lemon} to distinguish terminal symbols, they
must be written in capital letters. The above example differs from the original context free grammar in two ways. (1) It uses left-recursion instead of right recursion, because Lemon is optimized to handle left-recursion. (2) It uses a third rule to encapsulate the terminal symbols comprising one signal statement. This third rule is not necessary, but increases the readability of the grammar.

The complete parser grammar for the state machine modeling language is defined using this syntax. It is included in Appendix A.1.2.

**AST Generation**

Provided with the grammar, Lemon automatically generates the source code of an LALR parser for the recognition of the state machine modeling language. By itself, however, this parser is of limited value since it is only able to decide whether a state machine model is syntactically valid. In order to be of further use, the parser must pass on its understanding of the model's structure, specifically the computed derivation, to the following computation steps. To visualize and preserve computed derivations parse-trees are used.

The inner nodes of a parse-tree represent non-terminal symbols and the leaves represent terminal symbols. In the root of a parse-tree resides the grammar's starting symbol. The tree is constructed by recursively adding new child nodes to each non-terminal, which represent its substitution as indicated by the derivation. Non-terminal nodes which are substituted with the empty word $\varepsilon$ are deleted from the parse-tree [34]. Figure 5.4 shows the parse-tree of the familiar signal sequence for three elements.

Parse-trees contain nodes for all non-terminal and terminal symbols present in the derivation. Therefore the original source code can be completely restored from the tree by iteration of its terminal nodes. However, parse-trees also contain structural information which is often no longer relevant after the parsing process has been completed. For example, the parse-tree in Figure 5.4 shows three hierarchically organized signal_sequence non-terminal nodes. The lower two of these nodes are relics of the parse-process and can be removed. Their child nodes can instead be appended to the remaining signal_sequence node resulting in the abstract syntax tree shown in Figure 5.5. However, this tree exhibits a pattern which can be exploited to abstract it even further: Each non-terminal signal node sprouts an IDENTIFIER node, which is surrounded by SIGNAL and SEMIC terminal nodes. Considering these terminal nodes, only the IDENTIFIER nodes could possibly be traced back to different strings in the original source code. Therefore the other two nodes can also be removed, resulting in the final AST shown in Figure 5.6. Compared to the original parse-tree of the example, the final AST has less than half the number of nodes.

The parse-trees for the complete state machine language would feature 29 different types of non-terminal and terminal nodes. By using ASTs instead of parse-trees, this number is reduced to 18 different types of nodes. Naturally, this decrease in the number of nodes significantly speeds up further compilation steps.
In contrast to parser generators like ANTLR 4, Lemon does not by itself generate parse trees or ASTs [24]. Instead it provides a syntax for associating parser rules to arbitrary C code fragments. These fragments are automatically executed when the associated rule is successfully reduced by the parser[@14]. Since Lemon generates bottom-up parsers, the code fragments can be used for the bottom-up construction of ASTs. Listing 5.1 demonstrates how the AST is created for the signal sequence example using C code fragments.
Listing 5.1: AST Generation with Lemon

```c
1 signal_sequence (NODE) ::= signal_sequence (SEQ) module_signal (MS).
2 {
3     if (SEQ == NULL){
4         NODE = alloc_signal_sequence();
5     } else {
6         NODE = SEQ;
7     }
8     NODE = append_child (NODE, MS);
9 }
10 signal_sequence (NODE) ::= .
11 {
12     NODE = NULL;
13 }
14 module_signal (NODE) ::= SIGNAL IDENTIFIER (I) SEMIC.
15 {
16     NODE = alloc_module_signal (I);
17 }
```

As shown in this example, arbitrary C code fragments can be associated to parser rules by inserting them, surrounded by braces, behind the corresponding rule (ll. 2-0, 11-13, 15-17). Additionally, the syntax allows specifying custom, parenthesized names for all terminal and non-terminal symbols occurring in a rule. These names are used to identify the corresponding node within the code fragment. Since the AST is constructed in a bottom-up fashion, the `module_signal` rule is the first to be reduced. Its code fragment calls the `alloc_module_signal` function, which allocates a new AST node (called `NODE`) of the appropriate type and stores the token text of the identifier `I` for future use. The second alternative of the `signal_sequence` rule, which produces the empty word (l. 10), allocates no node and instead returns `NULL` (l. 12). The first alternative is in charge of collecting the previously allocated `module_signal` nodes, which are accessed using the name `MS` (l. 1). Only a single `signal_sequence` node is ever allocated, which happens when the `NULL` value passed in by the second alternative is received as `SEQ` (l. 4). This node then passed upwards as `SEQ` and is eventually appended all `module_signal` nodes (l. 8). This method generates no superfluous nodes and therefore the resulting AST is of the type shown in Figure 5.6.

5.1.3 Linking

In the state machine compiler architecture, the linking step is the final step required to prepare the AST for the code generation. Its duty is to resolve the cross-references, which appear frequently in state machine models. This is necessary, because the code generator will later have to know to which declaring node a cross-reference points, in order to determine the correct memory address.

In the state machine modeling language cross-references always lead from a qualifier node to either a module or a state node. The target node of a cross-reference is identified by its
name. However, identically named states or modules may appear in a model, as long as the names are unique among sibling nodes. This way a full-qualifier, consisting of the names of all containing modules, as well as the name of the target node, can be used for the unique identification of each module or state.

The source code and AST of an example model involving cross-references is shown in Listing 5.2 and Figure 5.7.

**Listing 5.2:** Model Containing Cross-References

```plaintext
1 module Main {
2     state A {
3         transition (Main.Sub == Main.Sub.Active
4             -> continue B);
5     }
6     module Sub {
7         state Active { ... }
8     }
9 }
```

**Fig. 5.7:** Cross-References in the AST

The linking step resolves the cross-references and stores pointers to the target nodes directly within the qualifiers. To simplify the linking step, all module nodes contain special hash-maps called scopes. These are populated in the parsing step and contain a mapping between the names and nodes of all directly contained states and sub-modules. The scope hash-maps greatly simplify the resolution of qualifiers. The first segment of a qualifier must always point to the root module. Therefore the target of the second qualifier segment can be looked up in the root module's scope. If the qualifier has only two segments, the discovered node is
final target of the qualifier. Otherwise, the final target is found, by consecutively searching through the scopes using the remaining segments.

5.2 Code Generation

The code generation is the final step in the LLVM frontend for the state machine modeling language. In this step, the code generator performs a pre-order traversal of the AST, which was generated using the methods introduced in Section 5.1. During the traversal it emits different fragments of LLVM intermediate code⁴ for each visited node. In combination, the generated IR fragments form an engine which simulates the execution of the compiled state machine.

LLVM intermediate code can be specified in three forms: An assembler-like textual representation, an in-memory data structure, and a compact bytecode format. The three forms are fully equivalent and LLVM provides tools to convert from one representation into another[6]. For this reason any of the three representations could be selected as the output format for the compiler. Excluding the cryptic bytecode format, the textual- and data-structure-representations remain as veritable options. Programming the code-generator to output the assembler-like textual representation is an intuitive strategy. However, complex string processing without the use of external libraries is not a feature the C programming language excels at⁵. Therefore the in-memory data-structure representation of the IR was chosen. Conveniently, LLVM provides a C-API, which enables building these data-structures by calling a set of specific function. For debugging purposes, the generated data-structures can easily be output in the more comprehensive textual representation[@26].

The compiled state machine engine must be capable of (1) storing the dynamic model state, (2) performing updates of the state machine (3) invoking external drivers. The following subsections explain how the compiler generates LLVM IR to realize these tasks.

5.2.1 Representation of the Dynamic Model State

According to the definition in Section 4.3.5, the dynamic model state consists of the following five elements of state.

- The active module
- The active states of all modules
- The active phases of all states
- The active valuation of variables
- The current position in the random number sequence

Of this list, the first three elements are handled explicitly by the state machine engine. The last two elements of the dynamic model state are handled in the hardware drivers and in the domain-specific abstractions' engines.

⁴See Section 3.1.4 for a general introduction on LLVM and the role of its intermediate representation
⁵Opinion of the author
In order to store the active module, active states and active phases, the compiler generates a global structure for each module and for each state. Similar to structs in the C programming language, LLVM structures aggregate multiple variables to form a composite datatype. Instances of the `module_type` and `state_type` structures are generated for this purpose.

**The module_type** structure consists of two 8-bit integer fields. The first field is used for the identification of the active module. Its value is either zero, or the one-based index of a potentially active sub-module. The active module of the state machine hierarchy is determined, by traversing the tree of modules along the edges specified by the active sub-module indices. The active module is found, when a value of zero is encountered. All active sub-module indices are initially set to zero.

The second field is the zero-based index of the active state. All active state indices are initially set to the indices of the corresponding modules’ initial states.

**The state_type** structure consists of two 8-bit integer fields. The first field holds the associated state’s phase. The second field stores the zero-based index of the next signal statement to be executed. A phase value of zero represents the $\alpha$ phase, indicating that the execution should start with the driver invocations section. A phase value of one means that the state execution should be continued with the signal statement identified by the second field. A phase value of two represents the $\omega$ phase. In this phase the transitions are evaluated to determine the next state.

Listing 5.3 shows the textual LLVM IR structures which the compiler generates for the model from Listing 5.2.

```
Listing 5.3: LLVM IR Defining the Module and State Structures
1 %module_type = type <{ i8 , i8 }>
2 %state_type = type <{ i8 , i8 }>
3
4 @glob_module_Main = global %module_type zeroinitializer
5 @glob_module_Sub = global %module_type zeroinitializer
6 @glob_state_A = global %state_type zeroinitializer
7 @glob_state_Active = global %state_type zeroinitializer
```

### 5.2.2 State Machine Updates

According to the update semantics defined Section 4.3.5, the next dynamic state of a model is computed based on the current globally active state\(^6\). To implement this behavior, the compiler generates update functions for each module and for each state.

**The module update functions** are in charge of recursively determining and executing the globally active state. For this purpose the update functions branch depending on the

\(^6\)This is the active state of the active module as defined in Section 3
active sub-module and active state values in the associated global structure. If the active sub-module value is equal to zero, the update function of the state indicated by the active state value is called. Otherwise, the update function of the module indicated by the active sub-module value is recursively invoked.

Started at the root module's update function, this recursion searches out and executes the globally active state.

**The state update functions** are in charge of executing their states. Since the manner in which a state is executed depends on its current phase, the compiler generates functions, which branch depending on the active phase value in the associated global structure.

If the state is in the $\alpha$ phase, its driver invocations are consecutively processed. Depending on the presence of signal statements, the state's active phase value is then updated. When the state contains signal statements, the value is set to one. The next call of the state's update function will therefore branch to the signals section. When the state contains no signal statements, the value is set to two. In this case the next update of the state will branch to the transitions section.

If the state is in the signals phase, the state's active signal value determines the next sub-module to be activated. In this case the currently active module's active sub-module value is set to the index of the signaled sub-module. Afterwards the active signal value is incremented if further signal statements exist. When this is not the case, the active signal value is reset to zero and the state's phase-value is set to two, indicating the $\omega$ phase.

In the $\omega$ phase, the available transitions are evaluated in the order of their appearance in the state definition. The first transition whose condition evaluates to true is selected. At this point different code is generated for deterministic and probabilistic transition targets. If none of the conditions evaluate to true, the state's update function returns without performing any modification.

Probabilistic transition targets first generate a random number in the range from zero to one. This is done by calling a special function called $smc\_random$. This function must be defined externally to grant the developer the ability to select an appropriate algorithm. The random number is used to perform a fitness-proportional selection of one of the available transition targets. This target is then handled in the same way as deterministic targets.

If present, deterministic transition targets first process the sequence of associated driver invocations. Next, the current module's active state value is set to the index of the transition's target state. In case of continue transitions the function returns at this point. In case of suspend transitions, the active sub-module value of the current module's parent is reset to zero. This way the sub-module is suspended. The next state machine update resumes the execution in the state which originally issued the signal in the parent module.
5.2.3 Declaration of External Drivers

In order to generate code which invokes externally defined driver functions, the compiler needs to know the signatures of these functions. In the C programming language external functions’ signatures are declared by including header files. In contrast, the state machine compiler parses the signatures directly from bytecode modules. Therefore the driver’s LLVM compiled bytecode modules must be passed to the compiler on the commandline.
The first case study is a children’s detective board-game involving an embedded microcontroller. The game is developed in cooperation with two partners from industry. Although the details of the game presently underlie the terms of a non-disclosure agreement, the development of a real-world product highlights the practical applicability of the concepts. In the game the player tracks down a criminal by interacting with a number of suspects and locations. Hereby clues and testimonies are given which finally allow the correct identification of the villain. Inputs are made through buttons and a color sensor. When the player receives new information, it is presented acoustically by means of an integrated loud-speaker. There are multiple levels of difficulty and new cases are presented each time the game is played.

Figure 6.1 gives an overview of how the concept introduced in Chapter 3 is applied to implement the detective game. Section 6.1 explains the abstractions used to describe the game. A state machine is used to represent the procedural aspects of gameplay. Beside of the state machine, a declarative model implemented in Prolog describes the structure and creation of playable cases. To prove the feasibility of the multi-abstraction concept for this particular application, a prototype was built using an ARM microcontroller. The prototype demonstrates a few major use-cases of gameplay. Its hardware realization and required software workflow are described in Section 6.2. Section 6.3 describes how the detective game can be extended with little effort by identifying and applying recurring patterns.

6.1 Model Abstractions

The detective-game case study is described on two levels of abstraction. The state and procedures of gameplay are modeled as the central state machine. For this purpose the modeling language from Chapter 4 is used. The criminal cases to solve by the player, hereafter referred to as game-instances, are a kind of logic puzzles and can be expressed naturally in a declarative model such as a Prolog program. Therefore a declarative model is used as second abstraction. As explained in Chapter 3, all abstractions must provide engines which export the functionality of their models via a driver call interface. This is necessary for the state machine to be able to query the declarative abstraction for information about
the current game-instance. The following subsections deal with both abstractions and their appropriate engines individually.

6.1.1 State Machine Abstraction

Most board games are described by a fixed number of rules, which determine the possible moves and their consequences. Additionally, games and moves may usually be subdivided into phases. Whether or not a rule is applicable at a certain point in time depends on the current phases and overall situation on the board. In the same way the consequences of an action may differ. The situation on the board and the active phases therefore constitute the game's state. The set of rules describes how the state can change depending on the players' actions. In this regard, the case-study's detective game is very similar. For this reason the courses of action available at any point in time can be intuitively represented in a state machine abstraction.

Listing 6.3 shows a simplified extract of the detective game's state machine source code in the language presented in Chapter 4. The example shows the use-cases of visiting a location and searching for clues or of questioning an employee. The structural overview presented in Figure 6.2 shows that each of these use-cases is encapsulated in its own module. When the game is started the Start state is initially active (l. 2). The clear_buttons driver call resets a flag which is used to track the state of the input buttons (l.3). This flag is preemptively set via an interrupt whenever the players press a button. It remains set until it is explicitly cleared. For this reason, the clear_buttons driver needs to be called first before waiting for further input. When this is detected by the any_button driver, the active state progresses to Visit (ll. 4-5). There the update_location driver is called to determine the new location using the color sensor (l. 8). The announce_location driver then plays an audio sample associated with the new location (l. 9). Having concluded the driver calls, the Action module is signaled (l. 10). Depending on the state of the button flag, it directly suspends itself to either the Ask or the Search state (ll. 25-26). Back in the Visit state, a transition is performed depending on the current state of the Action module (ll. 11-14). In the DoSearch state the button flag is cleared (l. 17) before signaling the SearchLocation module (l. 18). This module encapsulates the use-case of searching for clues. Its transitions interact with the declarative abstraction by querying the current game-instance using the location_has_clue driver (ll. 35, 37). This way, it checks whether a clue is available at the current location. When this isn't the case, the failure of the search is announced by calling the announce_no_clue driver (l. 36). Otherwise, a probabilistic selection determines whether the clue is actually found by the player. If yes, the module transitions to the FindClue state, which is not detailed here. If no, the failure is again announced by the announce_no_clue driver (ll. 37-40). In any case, the module eventually suspends itself and the execution continues in the DoSearch state. The game remains in this state until another button press occurs. Thereupon the loop is closed by continuing to the Visit state (l. 19). The procedure for questioning a suspect is essentially identical. The DoAsk state would also branch to its corresponding module, which would check the declarative abstraction for the correct testimony. The latter would be presented and afterwards the module would suspend itself.
Fig. 6.2.: Extract of Detective Game: State Machine Structure

module Game

module SearchLocation

module Action

module QuestionSuspect

...
module Game {
    init state Start {
        clear_buttons();
        transition(any_button() == 1
                   -> continue Visit);
    }
    state Visit {
        update_location();
        announce_location();
        signal Action;
        transition(Action == Action.Search
                   -> continue DoSearch);
        transition(Action == Action.Ask
                   -> continue DoAsk);
    }
    state DoSearch {
        clear_buttons();
        signal SearchLocation;
        transition(any_button() == 1
                   -> continue Visit);
    }
    state DoAsk { ... }
}

module Action {
    init state Ask {
        transition(button(0) == 1
                   -> suspend Ask);
        transition(button(1) == 1
                   -> suspend Search);
    }
    state Search {
        transition(button(0) == 1
                   -> suspend Ask);
        transition(button(1) == 1
                   -> suspend Search);
    }
}

module SearchLocation {
    init state Search {
        transition(location_has_clue() == 0
                   -> {announce_no_clue();} suspend Search);
        transition(location_has_clue() == 1
                   -> probselect {
            0.5: continue FindClue;
            0.5: {announce_no_clue();} suspend Search;
        });
    }
    state FindClue { ... }
}

Chapter 6  Case Study I: Detective Game
State Machine Engine and Driver Interface

The state machine model is compiled to LLVM bytecode using the compiler from Chapter 5. The generated module-update function constitutes the state machine abstraction's part of the common driver interface. The complete workflow of transferring the state machine- and other abstractions to the hardware is covered in Subsection 6.2.2.

6.1.2 Declarative Abstraction

During the game the players repeatedly visit a fixed number of locations. It is known that one of the location's employees is a villain. To discover the identity of this person, the players search for clues and question suspects. The discovered evidence incrementally reveals the identity of the villain. The different game-instances, with which the players are presented are therefore instances of a type logic of logic puzzle. Similar to traditional logic puzzles and Sudokus, the player starts out only with the knowledge of the rules and some fixed information. By means of deduction the number of possible solutions is narrowed down until only the correct solution remains. This category of games is intuitively described in a declarative way by stating facts and providing deduction-rules which generate further facts.

The logical programming language Prolog is designed to tackle exactly this category of problems. The programmer provides Prolog with a database of facts, a set of deduction rules and a query. The backend then employs the method of resolution to enumerate all fulfilling solutions to the query[8]. By providing a game-specific set of facts and rules, the backend is therefore able to enumerate all possible game-instances of the specified pattern. Examples for facts are the suspects, locations and attributes of either category. Most importantly evidence must be given in such a way, that the players are able to correctly identify exactly one villain.

Figure 6.4 shows the Prolog syntax for defining facts and deduction rules. In Prolog lower-case variables generally refer to constant literals. In the example these constants are personA, personB, tall and blonde. Variables, in contrast, begin with upper-case letters. Therefore Suspect is a variable. Subfigure (a) shows the syntax for the definition of facts. A predicate name (i.e. has_attr) is followed by a parenthesized list of arguments ending with a full stop. The example database is interpreted as the definition of two people. PersonA is tall and blonde, while personB is only blonde. Subfigure (b) shows how rules are defined. Rules consist of two parts which are separated by a stylized implication arrow "\(\rightarrow\)". The first part is a predicate with a parenthesized list of arguments, similar to facts. The second part is a list of constraints, needed to be fulfilled in order to satisfy the predicate. The comma between the constraints

\[
\begin{align*}
\text{(a) Facts} \\
person(personA). \\
person(personB). \\
attribute(tall). \\
attribute(blond). \\
has\_attr(personA, tall). \\
has\_attr(personA, blond). \\
has\_attr(personB, blond).
\end{align*}
\]

\[
\begin{align*}
\text{(b) Rules} \\
villain(Suspect) :\rightarrow
\end{align*}
\]

\[
\begin{align*}
\text{person(Suspect), has\_attr(Suspect, tall), has\_attr(Suspect, blond).}
\end{align*}
\]
is interpreted as logical and operator. Therefore the villain relation is satisfied by all people who are tall as well as blonde. Since Prolog uses a closed-world-assumption, this relation is solely satisfied by personA[7].

**Declarative Engine and Driver Interface**

As stated in Chapter 3, domain-specific abstractions, including the declarative one used in this case study, are treated as control input by the state machine. Therefore an engine is needed, which exposes the declarative model to the state machine via the common driver interface. The straight-forward approach would be to embed a Prolog interpreter into the application. SWI Prolog for example provides an application programming interface for the C language. Using this API it could be integrated with little effort. However, common Prolog interpreters require too much system memory to be practical for embedded environments with very limited resources. This problem was worked around by computing and sampling a fixed number of game instances ahead of time on a suitable machine. The precomputed instances are serialized into a compact binary representation. Due to the small footprint, many of these instances can be fit into the hardware prototype’s flash memory. By providing appropriate driver functions, the game instances can then easily be accessed by the state machine in order to make decisions. One of these drivers is the location_has_clue driver, used in the extract of the state machine model shown in the previous subsection.

### 6.2 Prototype

Creating a functional prototype of the detective game involved two major steps. The first subsection covers the hardware and low-level software aspects of designing a microcontroller setup capable of running the detective game in the intended way. The second subsection outlines the workflow of transforming and running the case-study model to run directly on the microprocessor.

#### 6.2.1 Hardware and Drivers

The selection of a microcontroller to use for the prototype was made based on a number of functional and non-functional requirements. The main functional requirements are the abilities of playing back audio, interfacing with a color sensor and processing input from a number of buttons. In terms of non-functional requirements the controller needs to fit dimensional, power-consumption and cost constraints.

![Fig. 6.6: Hardware Prototype Based on the STM32F0-Discovery Board](image-url)
Several ARM processors from a variety of manufacturers fulfill these requirements. Due to its integrated DAC\(^1\) and the compatibility with open source tool chains, the STM32F051 Cortex M0 microcontroller was selected. This 32-bit microcontroller clocks a 48Mhz and features 8Kb of RAM as well as 64Kb of internal flash memory. The manufacturer ST provides an evaluation board called STM32F0DISCOVERY for this processor, which greatly facilitates breadboard prototyping. This board was used as base for the hardware prototype. A custom PCB\(^2\) was created to accommodate additional flash memory as well as a small audio amp and loudspeaker. The external flash memory is necessary, because the internal 64Kb of flash can only hold a few seconds of raw 8-bit audio at a sample rate of 8kHz. As shown in Figure 6.6, the PCB was designed to be directly mountable on top of the evaluation board. The flash memory and the color-sensor are connected to the microcontroller’s SPI\(^3\) and \(^2\)C\(^4\) buses.

On the software side the microcontroller can be programmed in ARM’s Thumb\(^5\) assembler or in C using either a proprietary compiler or the GCC port for ARM architectures. Low-level interfacing with the integrated peripherals is done by writing specific values to a variety of control registers. Additionally, ST provides a Standard Peripherals Library which enables higher level control through an API for the C language. Using this library greatly facilitates programming the STM32F051.

The implementation of the detective game prototype required drivers for controlling the color-sensor, for reading from the external flash memory and for playing audio using the digital to analog converter. The color-sensor is a breakout of the TCS34725 provided by Adafruit. It features an \(^2\)C and thus reduces the driver to simple usage of the appropriate API from the Standard Peripherals Library.

For the external memory an Atmel Dataflash SPI flash chip was selected. Access to the external flash is more complicated, because it is closely linked with the audio playback. The microcontroller does not have an interface capable of mapping the external memory into its internal address-space. Therefore the DAC cannot be used to play audio directly from the external flash. Instead, one of the microcontroller’s DMA\(^6\) channels was used to link the DAC to a fixed-size double buffer in RAM. A second DMA channel links the double buffer to the external SPI flash. This way, audio can be played from the one half of the buffer, while the second half is prefilled via DMA. Whenever the end of a part is reached, the buffers are swapped. Figure 6.7 illustrates the two buffer configurations. When reading from the first buffer, the dashed switch setting is active and the second buffer is filled from flash. When reading from the second buffer, it is the other way around. The audio playback driver uses a hardware timer to trigger DAC conversions at a rate of 8kHz. Thereby it pulls bytes from the double-buffer

---

\(^1\)Digital to Analog Converter  
\(^2\)Printed Circuit Board  
\(^3\)Serial-Peripheral-Interface  
\(^4\)Inter-Integrated-Circuit Bus  
\(^5\)Thumb-1 and a subset of Thumb-2 instructions are available  
\(^6\)Direct Memory Access
through direct memory access. When the end of its current buffer is reached, it is in charge of swapping the buffers.

### 6.2.2 Software Workflow

Creating the binary program, which can actually be run on the hardware is a process involving five steps. The overall input to this workflow consists of four elements: (1) the game-instances generated as solutions by Prolog, (2) the state machine model, (3) the driver sources and (4) the state machine executor sources. An overview of the complete workflow is shown in Figure 6.9.

The first step is to process the game-instances generated by Prolog into a C header file. Prolog answers queries by printing all fulfilling solutions in a fixed format. Considering the query `has_attr(X, Y)` on the database from Subsection 6.1.2, it responds with the three solutions shown in Figure 6.8. Each of these assigns values to both of the variables `X` and `Y`. To make the solutions available to drivers, they are converted to C structs. Due to the simplicity of Prolog's output format, this is achieved by regular expression based string processing. Since memory on the microcontroller is very limited, the data is binary-encoded using bit-fields. This way, each of the prototype's game-instances requires only around twenty bytes of memory. The compact structs are then written to a header file, which can be included by drivers when necessary.

The second step is to process all the C source files, including the generated game-instances header, to LLVM bytecode. This is achieved by compiling them with LLVM’s C frontent Clang, using the `emit-llvm` option.

In the third step, the state machine model is also compiled to LLVM bytecode. For this purpose the state machine compiler, SMC, is invoked. As input it requires the model source code and the signatures of the drivers called by the model. When compilation is successful an LLVM bytecode module is produced.

Having successfully converted all inputs to LLVM bytecode, the fourth step involves compiling each platform-independent bytecode file to platform-specific ARM assembler using the LLVM compiler LLC. LLC needs to be called with flags to configure the right output assembler dialect for the target hardware. The commandline for targeting the STM32F051 is:

```
llc -mcpu=cortex-m0 -mtriple=thumb-none-eabi -float-abi=soft bytecode_file.bc
```

As a result one assembler file is produced for each bytecode module. All of the assembler files are passed to the GCC-ARM compiler in the fifth and final step. GCC-ARM assembles and links the assembler sources. Its output is a binary .elf file, which can be flashed to the microcontroller.
6.3 Extending the Detective Game

The specific section of code chosen for the example in Subsection 6.1.1 was selected, because it gives a general idea of how the game is implemented, as well as demonstrating two recurring patterns of module usage: (1) Using modules for encapsulating coherent blocks of functionality as in SearchLocation. Such modules process a sequence of functionally related states. Since these vary depending on the evaluation of the conditions, the structure of the module is similar to the control flow graph of a function. (2) Using modules to store decision-relevant flags inside of the model (i.e. Action). This pattern avoids hiding relevant information from the state machine-abstraction, by providing an alternative to the use of external variables as flags. These patterns proved to be very useful in adding further features to the prototype.

The first pattern was used in the implementation of all of the actions available to the players. Further use-cases, i.e. solving the case at the police office, are implemented using the same method as SearchLocation and QuestionSuspect. First the Action module is extended with a state to indicate the new use-case. Then a module is created to encapsulated the set of states which will represent the feature. This module is signaled in the Visit state, when the Action module is in the appropriate state. The feature is then implemented by incrementally by refining the new module with further states.

The division of the game into a number of phases was implemented using the second pattern. A module called GamePhase containing a number of states corresponding to the phases was created. The passing of time is simulated by dynamically incrementing an external variable upon taking specific actions. A set of threshold values determine the boundaries between the game’s phases. Whenever time passes the GamePhase module is signaled, possibly changing its state when the next threshold value is exceeded. The state of the GamePhase module is used to modify the game dynamics, by exploiting it to select different probability distributions or target states for transitions.
In this second case-study, the state machine modeling language, compiler and multi-abstraction paradigm are applied to build the model of an espresso machine. A traditional espresso machine uses pressurized hot water to extract a concentrated shot of espresso from a compressed puck of ground coffee. There are many variables which influence the taste of the result, the most obvious being the quality of the coffee beans. Assuming decent coffee is used, there are still many variables which influence the extraction process. The amount, temperature and pressure of the water, the amount and particle size of the coffee powder and the extraction time are important factors among several others. Consequently pulling good espresso shots requires altering a subset of the variables while keeping the remainder as constant as possible. A good espresso machine needs to control the factors it influences as precisely as possible. To control the brewing temperature of the water, espresso machines with PID-controlled boilers are available for consumer as well as professional use. For this reason the model of such a machine serves as real-world example for this case study.

Figure 7.2 gives an overview of how the concept introduced in Chapter 3 is implemented in this case-study. Section 7.1 deals with the two abstractions employed to build the espresso machine model. The first abstraction is the state machine, which describes the machine’s general operation. The second abstraction focuses on the aspect of temperature-control, by modeling a PID-controller. Instead of actually building an espresso machine in hardware, Section 7.2 explains how the architecture is modified to unit-test the model on a desktop machine. For this purpose the state machine executor is replaced with a testing framework.

---

1Not-PID controlled; Image [www.gaggia.com](http://www.gaggia.com)
7.1 Model Abstractions

This section introduces the abstractions of the second case study. Two models are used to portray the espresso machine. These are covered together with their corresponding engines in the following subsections. The machine's general operation, represented as a state machine, is covered in Subsection 7.1.1. The second abstraction, the model of a PID controller introduced in 7.1.2, focuses on the aspect of temperature control.

7.1.1 State Machine Abstraction

Similar to many common espresso machines the case-study model implements the following three use-cases: (1) Pulling a shot of espresso, (2) frothing milk using the steam rod or (3) drawing water from a steam rod. A typical interface, such as the one shown in Figure 7.3, therefore requires only two control buttons apart from the main power switch. The first button sets the boiler temperature either to the lower coffee- or to the higher steam-temperature. The second button toggles the pump, which moves water from the reservoir into the boiler. A 3-way solenoid valve connects either the boiler to the brew-group, or the brew-group to a disposal container. The setting of the solenoid valve changes depending on the current states of boiler and pump.

To brew espresso, the solenoid valve connects the boiler to the brew-group, to enable hot water to flow from the boiler, through the sieve with the ground coffee, into the cup. When the brewing process is completed, the solenoid valve is toggled to relieve the residual pressurized water from brew-group into the disposal container.

To froth milk, the boiler is set to the higher steam temperature. The solenoid valve seals the increasingly hot water in the boiler. The water begins to evaporate causing the internal pressure to rise until an equilibrium is reached. By opening a manual valve, positioned at the top of the boiler, steam is released through the steam rod.

To draw water from the steam rod, the boiler is also set to the steam temperature. However, the manual valve is opened to prevent internal pressure to build up. Next, the pump is turned on. This time, the solenoid valve continues to seal the boiler. Therefore the water level in the boiler rises and eventually flows through the manual valve out of the steam rod.

The following table summarizes the use-cases and their corresponding configurations of the buttons and the internal machine components.

---

2Image www.gaggia.com
Figure 7.4 gives a structural overview and Listing 7.1 contains the source of the state machine, which implements the above use-cases. The Main module contains three states (Idle, ToggleSteam, TogglePump) (ll. 2, 9, 14) and three submodules, which represent the machine’s internal components: Heater, Pump and Solenoid (ll. 19, 27, 35). Each of the modules contains two states and their initial configuration represent the machine’s idle use-case. The hardware drivers steamButton and brewButton query the hardware to discover if one of the two buttons were pressed. When the touch of a button is detected, the state machine transitions to the appropriate ToggleSteam or TogglePump state. Since the conditions of the Idle state’s outgoing transitions are not mutually exclusive, the steam button is prioritized because its transition appears first in the source code. Upon entering either of the toggle-states, the corresponding submodule is signaled in order to update the associated boiler’s or pump’s state (ll. 11, 16). Invisible in the structural overview, the source code shows that the transitions inside of the Boiler and Pump modules call the hardware drivers setBoilerState and setPumpState with either the parameter zero or one depending on the target state (ll. 21, 24, 29, 32). These drivers set the boiler temperature or activate the pump, based on the value of the integer parameter. Since the boiler is regulated by the PID controller introduced in the following subsection, the setBoilerState function is part of the driver interface exposed by the PID abstraction’s engine. Having performed the transition, the active Boiler or Pump module is suspended and the execution is resumed in the previous toggle-state. Both of these states next signal the Solenoid module, in order to set the solenoid valve to the correct position (ll. 11, 16). The Solenoid module performs a transition based on the active states of the Boiler and Pump modules (ll. 37, 40, 43, 46). Congruent with the table of use-cases, its Brew state is only activated, when the Boiler module is in the Coffee state and the Pump module is in the On state. When the active state of the Solenoid module changes, the setSolenoidState hardware driver is invoked by the active transition, in order to update the hardware solenoid valve’s position. Afterwards the Solenoid module is suspended and the execution is again resumed in the appropriate toggle-state. Having completed the submodule signaling, the state machine returns to the initial Idle state. The transitions leading back to the Idle state call the clearButtons hardware driver. This is required to reset the internal flags, which buffered the button-press event until it could be processed by the state machine.

State Machine Engine and Driver Interface

Similar to the state machine model in the first case-study, the state machine engine is again generated from the source-code by invoking the compiler from Chapter 5.
Fig. 7.4.: State Machine Structure

module Main
steamButton() == 1 brewButton() == 1

Toggle-Steam Idle Toggle-Pump

module Boiler
Coffee Steam

module Solenoid
Disp. Boiler == Coffee & & Pump == Off Brew

module Pump
Off On

Fig. 7.5.: PID Controller Model

setTemp - I iVal
P pVal
D dVal

System control

Chapter 7 Case Study II: Espresso Machine
Listing 7.1: State Machine Source Code

```java
module Main {
    init state Idle {
        print("Idle");
        transition(steamButton() == 1
            -> continue ToggleSteam);
        transition(brewButton() == 1
            -> continue TogglePump);
    }

    state ToggleSteam {
        print("ToggleSteam");
        signal Boiler; signal Solenoid;
        transition({clearButtons();} continue Idle);
    }

    state TogglePump {
        print("TogglePump");
        signal Pump; signal Solenoid;
        transition({clearButtons();} continue Idle);
    }

    module Boiler {
        init state Coffee {
            transition({setBoilerTemp(1);} suspend Steam);
        }
        state Steam {
            transition({setBoilerTemp(0);} suspend Coffee);
        }
    }

    module Pump {
        init state Off {
            transition({setPumpState(1);} suspend On);
        }
        state On {
            transition({setPumpState(0);} suspend Off);
        }
    }

    module Solenoid {
        init state Dispose {
            transition(Main.Boiler == Main.Boiler.Coffee
                && Main.Pump == Main.Pump.On
                -> {setSolenoidState(1);} suspend Brew);
            transition(suspend Dispose);
        }
        state Brew {
            transition(Main.Boiler == Main.Boiler.Steam
                || Main.Pump == Main.Pump.Off
                -> {setSolenoidState(0);} suspend Dispose);
            transition(suspend Brew);
        }
    }
}
```

7.1 Model Abstractions
7.1.2 **Controller Abstraction**

Simple espresso-machines employ thermostats to drive the boiler at its maximum, as long as the measured temperature is below a fixed threshold value. When the latter is exceeded, the boiler is turned off completely until the temperature falls back below the threshold value. In control-theory this mechanism is known as on-off-control and produces an unstable output. The system’s inertia leads to the target value being alternatingly over- and undershot[31].

Superior temperature stability is achieved by using more complex control schemes such as the discrete PID controller, which is shown in Figure 7.5. These regulate an actuator in a feedback loop, by setting its output power according to the sum of three terms. The proportional term is based on the current error, the difference between the actual value and the target value. Therefore, the proportional term deals with large current errors. The integral term is calculated based on the sum of previous errors. Therefore, the integral term deals with longer-enduring errors. The derivative term is based on the observed rate of change. It extrapolates this time series to counterbalance an anticipated over- or undershooting. By taking advantage of these techniques, correctly configured PID controllers achieve a much higher stability, and the desired target value can be held when there is no outside influence[20].

**Controller Engine and Driver Interface**

Tuning the gain values for the three terms of a PID-controller is often a complex, time-consuming task. To facilitate this job, tool suites such as Matlab provide software assistance for simulating and tuning controllers. These can be exported to platform-independent C functions. The generated source code serves as the controller-engine. Its driver interface consists of a method to set the target value, as well as an update function, which needs to be called periodically. In this case, the method for setting the target value is the setBoilerState value, which is called from within the Boiler module of the state machine. The task of repeatedly calling the controllers’ update-function is identical to the state machine-executors task of invoking the state machine-engine’s update function. Therefore controller-execution code is placed in the same loop.

7.2 **Testing the Espresso Machine**

This section explains how the LLVM based hardware-abstraction can be exploited to unit-test the espresso machine model on a desktop computer. Since common desktop computers possess neither boilers nor pumps or solenoid valves, the missing hardware can only be simulated. Subsection 7.2.1 explains how mocks of the hardware drivers are used for this purpose. In subsection 7.2.2 the design of the actual unit tests is described. The test oracles are derived from the component configurations of the machine's use-cases. Subsection 7.2.3 introduces the workflow of compiling and running the tests.
7.2.1 Mocked Drivers

The purpose of the mocked driver functions is to provide an interface identical to that of the real hardware drivers. This way the driver invocation in the state machine and in the domain-specific abstractions can be associated to the mocked drivers, without having to change the model and thereby diminish the meaningfulness of the test results.

For the sake of simplicity, calls to drivers which send control instructions to hardware peripherals are instead made to set or unset a global variable depending on the state the hardware should take. Drivers which acquire input from the peripheral hardware operate similarly, by returning the values of other global variables. Externally accessing these global variables provides the testing framework with a convenient way of analyzing the machine’s current state and of simulating user input. In order to provide immediate feedback of the simulated operation, a message is logged to the console when an output-driver is called.

The complete source code of the mocked drivers is located in Appendix A.2.1. Five global variables, \((\text{boiler\_state}, \text{pump\_state}, \text{solenoid\_state}, \text{steam\_button}, \text{brew\_button})\) are used to emulate the states of the machine’s internal components and interface buttons.

7.2.2 Testing Framework and Unit-Tests

Unit-testing the espresso machine model means to analyze if the model reacts correctly to input actions. To achieve this, the state machine executor, which would usually be placed in the embedded processor’s main loop, must be replaced with a testing framework. The framework has to supply the model with input, call the state machine’s update function a specific number of times, and check whether the model has performed the expected actions. The unit-tests were written using a modified version of the Minunit\(^4\) testing framework. The source code contains three exemplary tests and is located in Appendix A.2.2.

In order to grant the testing framework access to the state machine’s update function and to the mocked drivers’ global variables their signature needs to be declared in the test’s source file (ll. 7-9).

The test called \text{test\_toggle\_steam} verifies that the model reacts correctly to a pressing of the machine’s steam button (l. 27). Before the first input is made, the test asserts that all components are currently in their initial state, by checking if their global variables equal zero (l.28). Next, the value of the global \text{steam\_button} variable is set to one, to simulate the activation of the steam button (l.31). After an input has been made the model requires six processing steps until it returns to the \text{Idle} state. Correspondingly the state machine’s update function is called six times by invoking the \text{wait\_cycle} function (l.31). At this point the boiler temperature should have been set to the higher steam temperature. This is checked by asserting that the global \text{boiler\_state} variable now equals one (l.32). If this check succeeds another activation of the steam button is simulated. Six update function calls later (l.36), the boiler temperature should have been reset to the lower coffee temperature (l.37).

---

\(^4\)Minunit is a MIT licensed minimal unit-testing framework for the C programming language. It consists of a collection of macros which are distributed a single header file
The test of the brewing function (l.41) is not discussed in detail, because it is almost equivalent to `test_toggle_steam`. The major difference is that there is an additional assert concerning the position of the solenoid valve during the brewing process (l.48) as well as checking that the pump is turned on (l.46).

The test called `test_toggle_steam_brew` deals with the sequential activation of both input buttons. For this purpose the steam- and brew-buttons are activated consecutively (ll. 62-64). At this point the boiler should be set to the steam temperature, the pump should be turned on, but the solenoid valve should be in the `dispose` position (ll. 65-70). Next the steam button is activated once more (l. 73). In consequence the boiler should afterwards be set to the coffee temperature, the pump should still be running, but the solenoid valve should be set to the `brew` position (ll. 74-79). Finally the brew button is also activated once more (l.82), resulting in the machine’s components returning to their idle states (l. 83).

### 7.2.3 Compiling and Running

Three steps are required to compile the tests to a native binary. In the first step, the source file containing the mocked drivers is compiled to bytecode using LLVM’s C frontend `clang`. The second step is to generate the state machine engine by invoking the `smc` compiler from Chapter 5 on the state machine source code. As additional input the compiler is handed the bytecode file containing the compiled drivers, from which it extracts the necessary signatures. In the third and final step the generated engine and drivers bytecode, as well as the test sources are compiled and linked using `clang`. In result a binary file called `tests` is produces.

The following listing summarizes the commands required for the compilation process:

```bash
clang -emit-llvm -o drivers.bc -c drivers.c
smc -i coffee.sm -d drivers.bc -o coffee.bc
clang -o tests coffee.bc tests.c drivers.bc
```

Execution of the tests discovers no failures and produces the report located in Appendix A.2.3. The successful execution of the unit-tests proves that the model behaves correctly in the analyzed situations.
Conclusion

This thesis describes a novel approach of model-based software development particularly for small-scale and low-cost embedded systems. The technique proposes to describe systems through a common state machine formalism in combination with a set of domain-specific abstraction. In contrast to existing tools, it aims at integrating not only generic structural and behavioral models, but also specific model-types from the involved disciplines. This integration is achieved through a common driver interface, through which abstractions are required to mutually expose the functionality of their models. This enables the central state machine to interact with the domain-specific abstractions in the same way it interacts with the underlying hardware.

The thesis specifically aims at providing not only the theory, but also a functional toolchain supporting its technique of multi-abstraction MBSD. For this purpose a state machine formalism, a modeling-language, an inter-model communication method and a compiler were developed. The semantics of the modeling language are formally specified in order to provide an unambiguous foundation for the development of compilers. In order to grant the developer maximum flexibility in terms of the target-hardware and the choice of programming languages, LLVM was selected as backend for the compiler.

Two case-studies were conducted to prove the applicability of the approach in different real-world systems.

The first case-study, a microcontroller-based children’s detective game, was developed in cooperation with two partners from industry. The developed system heavily profits from the multi-abstraction approach, because it allows the separation of the different game aspects. The operational aspect of gameplay can intuitively be expressed in the custom state machine modeling language. The structure of the criminal cases, which the players need to solve, however, cannot be sensibly formulated within the state machine. They are more meaningfully represented in a declarative model. This is achieved by using the common driver interface to connect the state machine with a Prolog model. The hardware side of the prototype is based on an ARM Cortex-M0 microcontroller surrounded by a color-sensor, a loud-speaker for audio playback and various buttons. The targeting of the ARM microcontroller was enabled by the toolchain’s LLVM backend. The functional hardware prototype, which demonstrates major aspects of gameplay, was actually built and approved by the industry partners.

The second case-study is a temperature-controlled espresso machine. It makes use of the multi-abstraction approach by formulating the control aspect of the system in the terms of control-theory. The model of a PID temperature controller is integrated alongside the state machine abstraction, which portrays the general system-operation. Instead of building a hardware prototype, the second case-study demonstrates how the toolchain can easily be
applied to unit-test the multi-abstraction model on a desktop computer. Since the hardware-drivers are completely separated from the models, the embedded system can be tested outside of the actual target-hardware without having to modify the models.

In summary, the goals stated in the introduction were achieved. A new model based software development technique and supporting toolchain were created. Their expressive power and practical applicability for small-scale and low-cost embedded applications were demonstrated.

8.1 Outlook

The concepts and toolchain developed in this thesis can be extended in multiple ways.

This thesis proposes to use a state machine as the central element of a multi-abstraction model. In the case-studies, this state machine is updated periodically by an executor in the system's main loop. This is an intuitive approach for many real-world systems, however, it is not the only option. An interrupt driven multiplexer could, for example, be used to trigger appropriate abstractions based on the type of the received input.

The model engines used in the case-studies use the C programming language. As stated in Chapter 3, it should be possible to use any front-end language of LLVM for this purpose, modifying the function-calling conventions when necessary.

The compiler developed in Chapter 5 could be further improved with a semantic validation step. Semantic validation is required to find issues in the model, which cannot be detected by the parser. An example of such an issue are the probabilistic transitions. Currently, the compiler does not enforce that the sum of the probabilities on a transition equals one. Issues of this type could be detected by inserting a semantic validation step between the AST- and code-generation steps.

An incomplete graphical notation for the custom state machine modeling language was already proposed in Chapter 4. The use of a graphical notation greatly simplifies model-comprehension, particularly for non-specialists. Therefore a graphical frontend for the state machine language would be of additional value.
Appendix

A.1 Compiler

A.1.1 Lexer Grammar (Flex)

1  "component"  return COMPONENT;
2  "module"  return MODULE;
3  "state"  return STATE;
4  "init"  return INIT;
5  "signal"  return SIGNAL;
6  "transition"  return TRANSITION;
7  "continue"  return CONTINUE;
8  "suspend"  return SUSPEND;
9  "probselect"  return PROBSELECT;
10 \"(\:\|\:\|\:\|\:\|^\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\
A.1.2 Parser Grammar (Lemon)

1  translation_unit ::= module_sequence.
2
3  module_sequence ::= module_sequence module.
4  module_sequence ::= module.
5
6  module ::= MODULE IDENTIFIER LBRACE module_content RBRACE.
7
8  module_content ::= module_content state.
9  module_content ::= module_content module.
10  module_content ::= state.
11  module_content ::= module.
12
13  state ::= INIT STATE IDENTIFIER LBRACE
14       driver_sequence signal_sequence transition_sequence
15       RBRACE.
16  state ::= STATE IDENTIFIER LBRACE
17       driver_sequence signal_sequence transition_sequence
18       RBRACE.
19
20  driver_sequence ::= driver_sequence driver_call.
21  driver_sequence ::= .
22  driver_call ::= driver_invocation SEMIC.
23
24  driver_invocation ::= IDENTIFIER LPAREN expression_sequence RPAREN.
25
26  signal_sequence ::= signal_sequence module_signal.
27  signal_sequence ::= .
28  module_signal ::= SIGNAL IDENTIFIER SEMIC.
29
30  transition_sequence ::= transition_sequence transition.
31  transition_sequence ::= transition.
32  transition ::= TRANSITION LPAREN condition RARROW probselect RPAREN SEMIC.
33  transition ::= TRANSITION LPAREN probselect RPAREN SEMIC.
34  condition ::= expression.
35  probselect ::= PROBSELECT LBRACE option_sequence RBRACE.
36  probselect ::= transition_target.
37  option_sequence ::= option_sequence probselect_option.
38  option_sequence ::= probselect_option.
39  probselect_option ::= expression COLON transition_target SEMIC.
40  transition_target ::= LBRACE driver_sequence RBRACE
41       CONTINUE IDENTIFIER.
42  transition_target ::= LBRACE driver_sequence RBRACE
43       SUSPEND IDENTIFIER.
44  transition_target ::= CONTINUE IDENTIFIER.
45  transition_target ::= SUSPEND IDENTIFIER.
expression_sequence ::= expression_sequence COMMA expression.
expression_sequence ::= expression.
expression_sequence ::= .

// Expression Grammar
expression ::= disj_expression.
disj_expression ::= disj_expression OR conj_expression.
disj_expression ::= conj_expression.
conj_expression ::= conj_expression AND eq_expression.
conj_expression ::= eq_expression.
eq_expression ::= eq_expression EQ rel_expression.
eq_expression ::= eq_expression NEQ rel_expression.
eq_expression ::= rel_expression.
rel_expression ::= rel_expression GT add_expression.
rel_expression ::= rel_expression GE add_expression.
rel_expression ::= rel_expression LT add_expression.
rel_expression ::= rel_expression LE add_expression.
rel_expression ::= add_expression.
add_expression ::= add_expression ADD mult_expression.
add_expression ::= add_expression SUB mult_expression.
add_expression ::= mult_expression.
mult_expression ::= mult_expression MULT negation.
mult_expression ::= mult_expression DIV negation.
mult_expression ::= negation.
negation ::= SUB negation.
negation ::= primary_expression.
primary_expression ::= LPAREN expression RPAREN.
primary_expression ::= atomic.

// Atomic Expression
atomic ::= driver_invocation.
atomic ::= qualifier.
atomic ::= DOUBLE.
atomic ::= INT.
atomic ::= STRING.
qualifier ::= qualifier DOT IDENTIFIER.
qualifier ::= IDENTIFIER.
A.2 Case Study II

A.2.1 Mocked Driver Sources

```c
#include <stdio.h>

int boiler_state = 0, solenoid_state = 0, pump_state = 0;
int steam_button = 0, brew_button = 0;

void print(char *msg){
    printf("PRINT: %s\n", msg);
}

void clearButtons()
{
    steam_button = 0;
    brew_button = 0;
}

int steamButton()
{
    return steam_button;
}

int brewButton()
{
    return brew_button;
}
```
void setBoilerTemp(int val){
    if (val == 0){
        print("Set boiler to coffee temperature");
    } else {
        print("Set boiler to steam temperature");
    }
    boiler_state = val;
}

void setPumpState(int val){
    if (val == 0){
        print("Turn pump off");
    } else {
        print("Turn pump on");
    }
    pump_state = val;
}

void setSolenoidState(int val){
    if (val == 0){
        print("Set solenoid valve to dispose position");
    } else {
        print("Set solenoid valve to brew position");
    }
    solenoid_state = val;
}
A.2.2 Test Sources

```c
#include <stdio.h>
#include "minunit.h"

// The testing framework is the modified version of minunit introduced by Zed Shaw in the online version of his book "Learn C The Hard Way". It is available at http://c.learncodethehardway.org/book/ex30.html

void update_module_main();
extern int boiler_state, solenoid_state, pump_state;
extern int steam_button, brew_button;

char *assert_reset(){
    mu_assert(boiler_state == 0,
        "Boiler not coffee.");
    mu_assert(pump_state == 0,
        "Pump not off.");
    mu_assert(solenoid_state == 0,
        "Solenoid not dispose.");
    return NULL;
}

void wait_cycle(){
    for (int i = 0; i < 6; i++){
        update_module_main();
    }
}

char *test_toggle_steam(){
    assert_reset();
    // Set boiler to steam
    steam_button=1; wait_cycle();
    mu_assert(boiler_state == 1,
        "Boiler not steam.");

    // Set boiler to coffee
    steam_button=1; wait_cycle();
    assert_reset();
    return NULL;
}
```
41 char *test_toggle_brew(){
    assert_reset();

    // Set pump to on
    brew_button=1; wait_cycle();
    mu_assert(pump_state == 1,
               "Pump not on.");
    mu_assert(solenoid_state == 1,
               "Solenoid not brew.");

    // Set pump to off
    brew_button=1; wait_cycle();
    assert_reset();

    return NULL;
}

58 char *test_toggle_steam_brew(){
    assert_reset();

    // Set boiler to steam
    steam_button=1; wait_cycle();

    // Set pump to on
    brew_button=1; wait_cycle();
    mu_assert(boiler_state == 1,
               "Boiler not steam.");
    mu_assert(pump_state == 1,
               "Pump not on.");
    mu_assert(solenoid_state == 0,
               "Solenoid not dispose.");

    // Set boiler to coffee
    steam_button=1; wait_cycle();
    mu_assert(boiler_state == 0,
               "Boiler not coffee.");
    mu_assert(pump_state == 1,
               "Pump not on.");
    mu_assert(solenoid_state == 1,
               "Solenoid not brew.");

    // Set pump to off
    brew_button=1; wait_cycle();
    assert_reset();

    return NULL;
}
```c
char *all_tests() {
    mu_suite_start();
    mu_run_test(test_toggle_steam);
    mu_run_test(test_toggle_brew);
    mu_run_test(test_toggle_steam_brew);
    return NULL;
}
RUN_TESTS(all_tests);
```
A.2.3 Test Output

1 DEBUG tests.c:100: -------- RUNNING: ./tests
2 --------
3 RUNNING: ./tests
4 DEBUG tests.c:93:
5 -------- test_toggle_steam
6 PRINT: Idle
7 PRINT: ToggleSteam
8 PRINT: Set boiler to steam temperature
9 PRINT: Idle
10 PRINT: ToggleSteam
11 PRINT: Set boiler to coffee temperature
12 DEBUG tests.c:94:
13 -------- test_toggle_brew
14 PRINT: Idle
15 PRINT: TogglePump
16 PRINT: Turn pump on
17 PRINT: Set solenoid valve to brew position
18 PRINT: Idle
19 PRINT: TogglePump
20 PRINT: Turn pump off
21 PRINT: Set solenoid valve to dispose position
22 DEBUG tests.c:95:
23 -------- test_toggle_steam_brew
24 PRINT: Idle
25 PRINT: ToggleSteam
26 PRINT: Set boiler to steam temperature
27 PRINT: Idle
28 PRINT: TogglePump
29 PRINT: Turn pump on
30 PRINT: Idle
31 PRINT: ToggleSteam
32 PRINT: Set boiler to coffee temperature
33 PRINT: Set solenoid valve to brew position
34 PRINT: Idle
35 PRINT: TogglePump
36 PRINT: Turn pump off
37 PRINT: Set solenoid valve to dispose position
38 ALL TESTS PASSED
39 Tests run: 3
Bibliography


Online Resources


Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, 6. November 2015

______________________________
Robert Heumüller