Master's Thesis

Generating C++ out of UML state machines

Peter Huster
born May 22nd, 1984 in Zwickau

Course of studies Master Informatik

Westsächsische Hochschule Zwickau
Faculty Physikalische Technik/Informatik
Division Informatik

Supervisor, Institution:	Prof. Dr. Georg Beier, WH Zwickau
Due date:	November 11th, 2013

Abstract

This work analyzes and evaluates different approaches to translate UML state machines into C++ code. The first part of this thesis covers the ground of transforming information of a source language to a target language. It addresses the basics of language theory and different approaches of language transformation. The second part examines the properties and formalisms of state machines to value their characteristics for further reuse in the development cycle. The third part disassembles the programming language C++ with all its quirks and oddities. The last part puts all mentioned pieces together.

Beyond this approach the thesis tries to point out several concepts of language engineering to ease the use of software languages for the language user as well as the language engineer. It scrutinizes diverging solutions with the resulting consequences.

Preface

Acknowledgments

It has been a great pleasure working at the university of applied sciences Zwickau during my tenure as a master student. Although the idea of this thesis started out as something different, I have to thank Matthias Richter as well as Georg Beier and Wolfgang Golubski showing me a problem domain with such a huge potential for various solutions.

Today's development of embedded systems, especially in the automotive sector, requires skills in high level modeling and low level programming. It was seeing this approach in a real life application that convinced me to analyze and even find a more efficient method to tackle this problem. While working at a research project focusing this issue, I thank André Schwarzwald and Thomas Opp.

I would never have thought that working with languages and language concepts would be so much fun. Getting a lot of light bulb moments because of grammar and theory of language, I think some basics of linguistic should be part of everyone's life. I would like to thank Gwendolyn Lauterbach for long discussions about natural language, language acquisition and differences in languages because of history and culture, Sibylle Schwarz and the Kyrgyz students of my theoretical computer science course to grapple with the basics of software languages.

Working with a programming language that is even older than me is another story. So many bright minds formed C++ that it now contains various low and high level concepts. Yet it contains another Turing complete language with its cthuloid syntax horror. I thank Ronny Kubik for keeping me sane and explaining those nightmares to me.

A code generator is not something that gets a lot of attention because the potential target audience are software developers, and it is mostly a part of a tool chain to create the actual software. But principally without GeneSEZ and the team around GeneSEZ, this work would have never been conceived. Many thanks go to André Pflüger, Nico Herbig and Tobias Haubold for their enormous support, time, patience and criticism.

Although he was named before, very special thanks go to Tobias Haubold for sharing his wicked thoughts, ideas and mental acrobatics with me. Thanks to my parents for food and shelter, as well as thanks to my employer bartering work performance for money to get food and shelter. I would also like to thank the guy who had the inspiration to pick coffee beans, roast and grind them to create a brew, without this thesis would never have been completed.

Also thanks to the deity of the old testament, making this thesis possible and even more reasonable because of the babel incident[1] [1]. Especially no thanks to the concept of time and time itself.

Motivation

More and more parts of everyday life are getting computerized. Often people do not even realize they are interacting with a computer when they are using everyday objects. Those devices are called embedded systems.

> "An embedded system contains a computer as part of a larger system and does not exist primarily to provide standard computing services to a user. A desktop PC is not an embedded system, unless it is within a tomographical imaging scanner or some other device. A computerized microwave oven or a VCR is an embedded system because it does no standard computing. In both cases, the embedded computer is part of a larger system that provides some noncomputing feature to the user..."[2]

These computers contain only necessary components, and they are specified to do only the thing they were built for. As long as you can deploy a Java Virtual Machine (JVM), or something akin, on the device, it is possible to program it in a high level language like Java[2]. But a lot of embedded systems do not provide enough resources to execute a JVM. They use real time operating system, or they do not contain an operating system at all.

Most of these systems have to be programmed with languages like C or C++ not only because those languages compile programs that can run on plain vanilla components, but they are used for applications where the challenge is, "if something fails, people die"[3] to phrase Bjarne Stroustrup. C and C++ evolved for more than 20 years [4] and especially C++, as an extension of C, contains various language concepts. With its enormous flexibility it is possible to combine high level language features, like runtime and compile time polymorphism, with low level features, like pointers and memory management. The consequence of this feature richness is high complexity. In a language like Java it is not even possible to manage memory

[1] I do not know if it is a coincidence or not but it is part of the Book of Genesis ;-)

[2] Naturally it makes total sense to program a coffee maker in Java or CoffeeScript using the Hyper Text Coffee Pot Control Protocol specified in RFC 2324.

[3] Imagine a PATRIOT air defense missile system doing garbage collection while missing the opportunity to detect an incoming missile.

[4] To be frank, the idea of C was born 1969 by Dennis Ritchie and 1979 Bjarne Stroustrup started to develop C++. A lot of today's software languages vanish into oblivion after a year or two.

allocation and deallocation, thus resulting in a possible loss of performance. C++ instead is designed to pass any responsibilities to the developer to forward full control with all its consequences.

Unfortunately, it is not the developer who designs an embedded system but an engineer or a so called domain expert. He communicates in his domain language all requirements, specifications and wishes to the developer. One excellent syntax to represent this information is a state machine. Its graphical notation is easy to understand for the developer as well as the domain expert. State machines are not only easy to understand, but they are thoroughly researched and formally described. With all these properties it is self-evident to try to transform state machines in a low level language, like C++ , to deploy it on the embedded system. Even though a vast number of scientist, students and academics analyzed and researched this problem, the present thesis tries it again.

Prerequisites

To keep this thesis brief and to the point, it is necessary to presuppose basic issues which are essential for it. As the title conjectures, you should have a fundamental grasp of programming concepts as well as abstract data types[3] and the programming language C++[4].

State machines and the like will be thoroughly covered but the fundamentals will not be part of this thesis. Concomitant with the state machine fundamentals are the basics of language theory which are part of theoretical computer science. This knowledge is required and can not be delineated as good as by Hopcroft et al.[5].

This work builds upon basic patterns of language processing. Language recognition and transformation are discussed in depth by Aho et al.[6].

Contents

Chapter 1

Language transformation

1.1 Motivation

To generate something out of something else is a basic concept of general science. In computer sciences it is usually about code generation which means translating instances of a higher abstract language to instances of a lower abstract language. Programmers are working with code generators on a daily basis because a compiler is more or less a special code generator. A compiler creates bytecode or object code out of languages like Java or C++. "A generator just walks an internal data structure and emits output."[7]

This implies an internal data structure and an algorithm to walk this structure. The internal structure or representation needs to be deduced from the input. The algorithm or behavior to walk this representation needs to be specified. This sounds like a problem made for a computer because it consists of input, processing and output. Since the beginning of the computer age, scientists searched and found various solutions to it. The foundation is always a multistage pipeline of a language application, or plain vanilla language transformation, which will be thoroughly examined in this part of the thesis.

1.2 Introduction

The principle of language transformation or translation is straightforward. Providing a source language instance for the translation component results in a target language instance. In natural language processing this concept is necessary for communication. Without the concept of translation, people could only communicate with people which understand the same language. This verbal communication is the foundation of linguistics and translation theory. While the principle of translation is easy, the task itself is hard. It can be seen as a mapping problem[1] based on transformation rules. Unfortunately, not all languages contain the same concepts[2],

[1] Which is by definition a formal language.

[2] While a Jew immediately knows what a schmock is, it is difficult to explain this phrase to a goy.

but linguists found out that all languages contain at least a core of concepts that can be used to deduce other linguistic utterances [8].

Alas, the processing of natural languages cannot simply be mapped to computer or software languages. First of all, the communication partners could be man or machine. A machine requires correct and formal representations of linguistic utterances to process them. Another aspect is that natural languages[3] already exist, while software languages need to be created. Thus, it results in the difference between using a software language to design software and designing a software language [9].

1.3 Different views of software languages

Software development is usually about developing applications. The purpose of an application is to get specific tasks done more efficiently. The main purpose of the software developer is to create such a software. The application will be written in a software language. Using a software language to design software is an application centered process. Specifying a language, as well as creating the needed tooling to use the language, is a language centered process. Distinguishing between the two processes is as crucial as distinguishing between refactoring and developing [10].

If the software developer uses a language to create software for the end user, he will act as language user or application developer. If the Software developer creates a language, he will act as language engineer. The end user of the language engineer is the language user. It is even possible to use this approach recursively because a language engineer cannot create a language out of thin air[4]. He will use tools and formalisms to create a language.

Today's software developers use an Integrated Development Environment (IDE). An IDE is more or less a set of tools. These tools differ between language user and language engineer. The language user interacts with an editor to write code, transformers like a compiler, executors like a debugger, probably analyzers like a cost estimator, and additional support tools like version control and so on.

An IDE like Eclipse[5] renders the differences between the tools blurry. The language user won't even differ between the tools and the language he is using. For the language engineer those differences are very important because his objective is to provide the tools, as well as the language, to make creating software easier. To achieve this, the IDE has to support him in creating a language specification and the tools for the language user. Martin Fowler coined the phrase Language Workbench denominating those IDEs [11].

1.4 Purpose of a language

The common purpose of a language is communication. It should express the will of the user effective and efficient. Software languages, especially Domain Specific

[3] Except for constructed languages like Esperanto for example.
[4] The Object Management Group (OMG) would probably call him a meta language engineer.
[5] http://www.eclipse.org/

Languages, which will be explained in 1.10 on page 18, but also General Purpose Languages are using abstraction[6] to achieve this.

> "The abstraction level of a concept present in a software language is the amount of detail required to either represent (for data) or execute (for processes) this concept in terms of computer hardware."[9]

Based on the definition the expression "Java is more abstract than object code" is correct because the abstraction level of Java is higher. The terms low level and high level are not absolute but relative because the virtual level of abstraction grows by time[7]. Meaning that today a class or an object can be called high level, while an instruction could be called low level. But thinking in design patterns (see [12] for details) makes the class look low level. Considering language transformation as a pipeline to create an instance of a low level language out of an instance of a high level language, is the default approach to raise the abstraction level. E.g., the first edition of C++ has been transformed to C, and after that it had used the C pipeline to create executable code.

1.5 Characteristics of a language

Using a language is more or less like using a tool. Both have purposes and properties, rendering them useful in one area of application and useless in another one. E.g., a screw driver excels in the domain of screwing, can be misused as a can opener, and is completely useless as a paintbrush[8]. To quantify the value of a language for a specific domain, it is necessary to identify the possible features and their consequences of absence or presence.

1.5.1 Concrete representation

The concrete representation is an important feature. It can be divided in textual and graphical notation. A textual language, like the programming language C, can be edited with a common text editor, and it needs to be parsed for further transformation processes.

A graphical language usually requires a projectional editor or a complex recognition processes. E.g., Magic Draw[9] is a projectional editor for Unified Modeling Language (UML) diagrams. It is possible to draw those diagrams in Visio[10] or on

[6] Abstraction in the sense of leaving out irrelevant concepts from a certain point of view. The misunderstanding of leaving out details to gain higher level of abstraction results in an incomplete language.

[7] E.g., hardware is not virtual and interacting with hardware cannot be done with high level concepts, like classes, but with machine code.

[8] A language example would be Extensible Markup Language (XML) which excels in the domain of configuration, can be misused as a database, and is completely useless as a programming language.

[9] http://www.nomagic.com/products/magicdraw.html

[10] office.microsoft.com/en-us/visio

a napkin, but the recognition process needs to analyze the picture to create an internal representation for further transformation processes.

The consequence of a graphical language using a projectional editor is the sheer impossibility of creating syntactical incorrect instances. E.g., it is not possible to create an association without association ends.

On the contrary, a text editor accepts each input and the parser has to verify the syntactical correctness. As a consequence, it is possible to work with incorrect instances of textual representations, while it depends on the projectional editor whether the language user is allowed to do this or not.[9, 13]

1.5.2 Consistency

The consistency of a language is a feature that can be applied to other language features. It is about choosing a design and holding to it. E.g., an arithmetic language consisting of addition, subtraction and multiplication with their binary operators, will be inconsistent if the plus operator uses infix notation, the minus operator postfix notation, and times prefix notation.

The consistency of notations relates to the consistency of concepts. Related concepts should look and behave the same way, and they should be grouped together. Unrelated concepts should look and behave different, and they need to be separated. E.g., the concept of edges should not be mixed with the concept of nodes in a graph. A lot of software languages struggle with the consistency of the level of detail as well as abstraction.

An example is C++, making it possible to combine pointer arithmetic and multiple inheritance. Experienced language users know that misusing the language this way is a bad idea, but from the consistency view it seems plausible. Another example are good libraries and frameworks, providing the user only with necessary information about how to use them, but hide the details of the implementation.

Mixing level of details and abstractions results in a highly flexible language that tends to be misused accidentally [9]. This leads to minimalism as an intentional language feature.

1.5.3 Minimalism

Minimalism is about exposing only what is necessary for the domain and the user to accomplish his task. E.g., a financial broker wants to express a trading process, like placing an order, as seen in listing 1.1 from [14].

```
newOrder.to.buy(100.shares.of('IBM')) {
  limitPrice   300
  allOrNone    true
  valueAs      {qty,unitPrice -> qty * unitPrice - 500}
}
```

Listing 1.1: Example of a financial brokerage system DSL

Providing only the proper interface and hiding the implementation details will prevent invalid client code because of necessary evolution of the abstraction.

Minimalism should go along with distillation. Distillation is about getting rid of nonessential details [14]. Keeping the implementation dense and pure minimizes accidental complexity [15]. It is like using smart pointers in C++ to get rid of memory management details.

1.5.4 Simplicity

Another language feature is simplicity. Keeping a language simple makes it easy to learn and easy to use. Simplicity concerns minimalism because it is about omitting needless artifacts. Modern programming languages contain alternative notations for expressions like `a[i]` as a synonym for `*(a+i)`. These additional artifacts are called syntactic sugar. They make it easier to express and read linguistic utterances [16]. If anything, it should be applied judiciously, and it requires consulting the language users [14].

1.6 Language components

For natural languages it is possible to create an unlimited amount of instances. To represent a language in a dense but complete form, linguists use grammars [17].

A grammar consist of rules and primitives. The same applies for software languages. A language specification has several representation. It consists of one or more concrete forms and one abstract form. The abstract form is usually an abstract syntax tree or abstract syntax graph. The abstract syntax is used by the tools. The concrete syntax is used by the language user. To transform a concrete syntax into an abstract syntax a transformation, the so called syntax mapping is needed. Next to the syntax is the meaning. A description of the meaning is the so called semantics (see [11, 9, 6] for details).

1.6.1 Formalisms of language specification

To process linguistic utterances of a language, it is necessary to specify the language formally. According to [9], those formalisms can be distinguished as follows:

Context-free grammars are type-2 grammars of the Chomsky hierarchy. Those grammars have long been the standard to define programming languages. They are thoroughly researched, thus providing the language engineer with many tools to support him. Using the Backus-Naur Form as well as the extended Backus-Naur Form as formal syntax, it is possible to define sets of production rules to specify the concrete syntax of a language.

The recognition process can be represented by a pushdown automaton. The result of the recognition process is a derivation tree which is the underlying structure of a context-free grammar. With an algorithm like in [18] it is

possible to transform the derivation tree into an abstract syntax tree. While it is usually not possible to reference other nodes than parent or child nodes, compiler constructors use symbol tables to eradicate this flaw (see [6, 7] for details). Combining a symbol table with its respective abstract syntax tree results in an artificial representation of a directed graph. Only a subset of context-free grammars are reasonable for further processing. These are so called LL and LR grammars[11] [19].

That reduces the amount of grammars and dictates the structure of the production rules. Context-free grammars cannot be used to specify graphical languages. That can be interpreted as a major drawback for present-day language specification. For more details about context-free grammars see [5, 6].

Attributed grammars are context-free grammars with additional attributes associated to their vocabulary. Furthermore, the grammars are expanded by semantic rules[12] and conditions [20].

The attributes support the compile process and can have any structure. Combined with symbol relation grammars, it is possible to specify visual languages [21].

A significant difference between this and the metamodeling approach is the containment of notions like boxes and arrows.

Graph grammars use graphs to represent entities and relations. The fundamental approach is based on instance graphs[13], which are typed over a type graph[14], to represent the relation between concept and occurrences. In addition to that, rules and transformations are necessary for the recognition and graph rewriting process [22]. Further reading about graph grammars and even more complex notations can be found in [23, 24].

The underlying structure is a derivation graph. Derivations in graph grammars are not necessarily unambiguous which makes the recognition and processing part rather complex and hard to handle efficiently. On the contrary, graph grammars are very expressive and represent type-1 grammars.

UML profiling is a so called lightweight extension mechanism to customize UML models [25, 26, 27].

A profile is a collection of elements, like stereotypes, which can be applied to specific model elements like classes. Generally, UML profiles represent issues of a specific domain, and they can be used to address issues that cannot be represented with common UML elements.

The UML specification provides the concrete syntax and the abstract syntax mapping to create a profile [28, 29]. The language engineer creates an abstract syntax model with its own classes and associations. Furthermore, semantics are undefined and need to be specified.

[11] The first character is the abbreviation for 'gets parsed from left to right'. The second character means 'constructing a left respectively right most derivation'.

[12] In the context of this work it would be more correct to call them attribute rules.

[13] Which could be represented by object diagrams.

[14] Which could be represented by a class diagram.

Metamodeling is the process of defining models which define models. In the context of this work the common interpretation of a model is an abstraction of the real world[15]. More precisely: "A model is a combination of a type graph and a set of constraints of various types."[9]
Thus, metamodeling is a graph based approach and its underlying structure is a graph.

Metamodels are generally illustrated by class diagrams. While the representation and the structure does not differ significantly from graph grammars, metamodels are less complex and more intelligible [30].

Usually a metamodel is a model of the abstract syntax. The abstract syntax of UML, for example, is defined by metamodels. Further details of the relevance of metamodels for model-driven software development and model-driven architecture can be found in [31, 32, 33, 27].

Compared to context-free grammars, metamodels are more expressive. That means by implication that it is possible to transform a context-free grammar into a metamodel [34].

1.6.2 Abstract syntax

In the subject of linguistics the abstract syntax is the hidden, underlying, unifying structure [17]. E.g., the sentences "the state machine extension references gcore" and "gcore is referenced by the state machine extension" render the same fact with different concrete representations. Yet the abstract syntax is the same. This applies also to UML diagrams and their XMI serialization [9].

The abstract syntax is fundamental for languages with more than one concrete representation for a linguistic utterance. It is the in-memory representation, in which the concrete representation gets transformed in, to process it further. Therefore, rendering the abstract syntax as the intermediator between concrete syntax and semantical interpretation.

The abstract syntax is the representation of internal concepts. In other words, it is a concept model consisting of the meaning of concepts and their relationships. With its features, the abstract syntax should be considered as input of the transformation process.

1.6.3 Concrete syntax

The concrete syntax is the actual interface of the language for the language user. It consists of an alphabet and transformation rules to form the derivation tree. Additionally, it can include abstraction rules for concrete elements with no abstract counterpart, and binding rules for concrete elements representing the same abstract element. Tools represent models or programs to interact with. Textual languages need a recognition process, like scanning and parsing, to derive the abstract form [6].

[15] The term model is derived from the Latin word 'modulus' which means standard.

Visual languages, especially modeling languages like UML, are projected by the tool[16]. A projected view and projected editing provides almost direct access to the abstract representation. The consequence is a separate data format and restricted interactivity. A free format editor, like emacs[17] or Visual Studio[18], accepts any input, and allows the language user to save invalid language instances.

A structure editor, like Papyrus[19] or MagicDraw, prohibits this[20]. Once the language user got used to projected editing, this aspect renders obsolete.

The main purpose of the concrete syntax and its tool for representation is the part as an interaction device. Thus, it should especially be designed for humans with respect to the content of section 1.5 on page 3 [9].

1.6.4 Semantics

The synopsis of syntax is rendered by the following quotation: "...everything on paper or the screen is a syntactic representation. This is also true of the machine's internal representation, the so-called abstract syntax or metamodel."[35]

The syntax is an infinite set of legal elements. But it requires semantics to understand and thus process these elements. The meaning of the elements is the semantic domain. "... it [the semantic domain] serves as an abstraction of reality, capturing decisions about the kinds of things the language should express."[35]

The mediator of syntax and semantic domain is the semantic mapping. It associates syntactic elements with its specific meaning. To understand the meaning of meaning it is best to refer to the linguistic concept of the meaning triangle [36].

The meaning triangle consists of the components concept, linguistic symbol and real world. A linguistic symbol, like a book, is associated to a real world copy of 'War and Peace' and vice versa through a mental concept of the person. Figure 1.1 on the following page illustrates the example. Those concepts are based on each other to understand the world, and they represent the unique knowledge of a person. Thus, semantics are subjective and render the process of communication rather difficult.

[16] Additional research has been done to recognize raster graphics but this is out of scope of this thesis.

[17] http://www.gnu.org/software/emacs/

[18] http://www.microsoft.com/visualstudio

[19] http://www.eclipse.org/papyrus/

[20] Depending on the interaction the tool spoon-feeds the user, limiting his creativity, on the contrary the tool guides and supports the user to avoid adverse consequences.

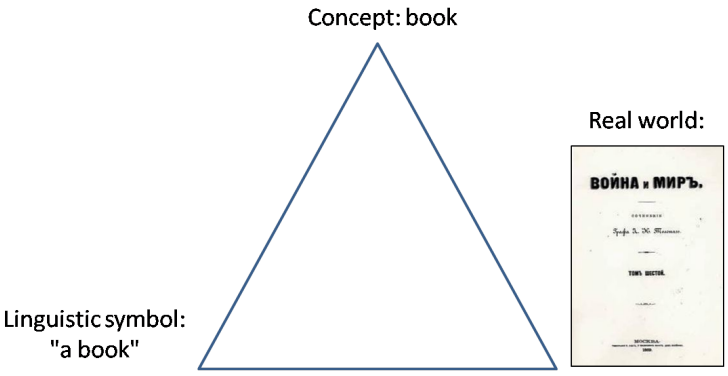

Figure 1.1: A possible meaning triangle of the concept book. The picture of the book is from http://en.wikipedia.org/wiki/War_and_Peace/.

E.g., a computer scientist would have a computer error in mind when someone raises the subject of a bug. Then again, a biologist would associate it with an insect. Thus, this issue requires a semantic description: "A description of the semantics of a language L is a means to communicate a subjective understanding of the linguistic utterances of L to another person or persons."[9]

The semantic description targets the audience of the language and not the computer because a computer cannot construct own mental concepts. The semantic description contains the semantic mapping and the semantic domain. E.g., the semantic description of the UML is informal and consists of semantic variation points. Thus, it is ambiguous and renders proper processing impossible. Hence, it is necessary to create a formal semantic description.

The semantic domain can be expressed by formal languages like logic, algebraic specification languages or standard mathematics. According to [35, 9], the semantic mapping can be expressed by:

Denotational semantics Based on pure mathematical notation, this approach represents a rather complex form to describe the semantic mapping.

Pragmatic semantics Based on specific examples, this approach is simple but less formal because it uses a reference implementation. The examples are the input of the execution process. By comparing the output to the input it is possible to deduce the behavior and thus the semantics.

Translational semantics Based on another language known to the audience, this approach is also known in linguistics. To learn a new language, humans reference the linguistic utterance of their known language of a concept to the unknown linguistic utterance by translation. The concept of soup, for example, is known as 'Suppe' in German and 'polévka' in Czech.

Translational semantics reuse the semantic mapping of the known language. Consequently, the target language needs equivalent constructs for the concepts. Especially in computer sciences, this problem is nontrivial. Even general purpose languages, like C and Java, are hard to translate interchangeable because their programming concepts differ significantly.

In computer sciences translational semantics are represented by transformations[21]. They will be used in a bootstrapping fashion if the direct translation renders too complex.

Operational semantics While translational semantics are based on a known language, operational semantics render the concept itself, next to the semantic mapping. It is like explaining an issue as a sequence in real life. To formalize such a sequence, a chronology of snapshots is sophisticated. The formal representation is a state transition system, using graph transformations as realization.

1.7 Principles of language transformation

The most fundamental and best-known model of computer sciences is the Input-Process-Output model[22] to grasp the big picture of software or hardware. Even language transformation, as well as so called language applications like compilers, can be explained by this model. The input is an instance of the source language that gets processed. If that performs flawlessly, an instance of the target language will be the output.

In the context of language processing, it is also known as a multistage pipeline illustrated in figure 1.2. It is called multistage because it consists of several phases, and it is a pipeline because these phases will be executed in a predefined order. The consequence of using discrete phases is replaceability and reuseability. E.g., the first C++ compiler reused the pipeline of the C compiler. Once the C++ source code has been transformed into valid C, it reused the transformation into machine code.

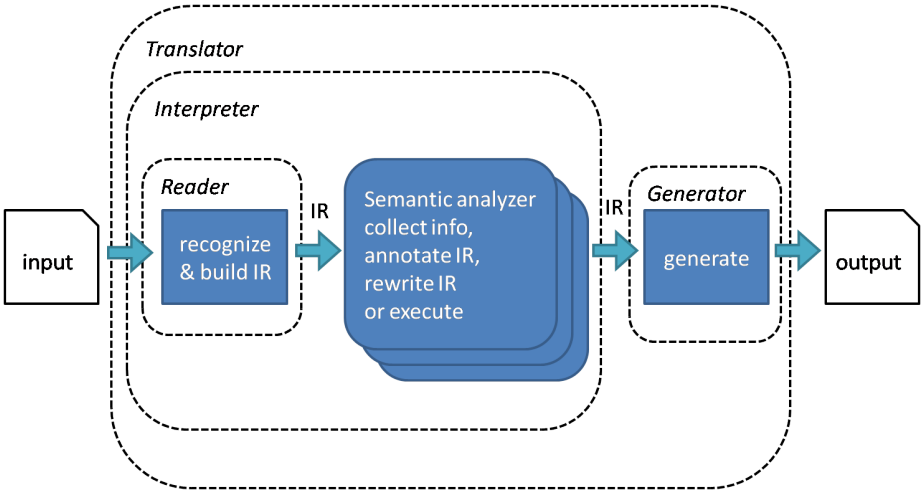

Figure 1.2: The multistage pipeline of a language application from [7]. IR is the abbreviation for intermediate representation, further called intermediate form.

[21] E.g., model transformations, graph transformations, compilation or generation.
[22] Or the Input-Process-Output+Storage model.

Categories of language applications

To distinguish between the executable elements of the pipeline, it is useful to categorize them according to [6, 7]:

Reader Recognizes the input data and builds an internal data structure. The reader component is also known as frontend.

Generator Walks the internal data structure and emits the output data. The generator is also known as backend.

Translator Is the combination of Reader and Generator. It reads input and emits output in the same or in different languages. If the result is in the same language, it is also called a rewriter.

Interpreter Is like a program that executes other programs. To do this it reads, decodes and executes instructions. E.g., the JVM is a byte code interpreter.

The common processes of the pipeline are categorized as follows, according to [7, 6]:

Parsing also known as Recognizing, is associated with the reader component. The process analyzes the input to check if it is a linguistic utterance of the source language. Thus, it checks if the input is a valid instance of a concrete syntax of the language, to pass it to the component that executes the syntactic mapping. Especially for textual notations applies, that for a language, the respective automaton can be used to check if a linguistic utterance belongs to the language.

The automaton can be derived from the grammar, thus the parser itself can be derived from the grammar[23]. Based on the underlying approach, it can be necessary to separate the word recognition process, called scanning, from the phrase recognition process called parsing [37].

Intermediate form construction If the parsing was successful, the process would enter the next stage to construct an intermediate form. Simple transformations waive an intermediate form. The consequence is a less complex algorithm with inferior scalability.

The intermediate form directly interrelates with the abstract syntax. Based on the underlying language specification, the intermediate form can be a tree or a graph. Because common language processing is based on trees, and a tree is always a graph, the word tree will further be synonymously used as the intermediate form.

Next to the recognition process of the parser, the parser creates a so called parse tree representing a recording of the parse process. To operate more efficiently with the tree, it will be transformed to an abstract syntax tree.

[23] So called parser generators or compiler compiler, like antlr (see http://www.antlr.org) or yacc (see http://dinosaur.compilertools.net/), generate a parser from a specific grammar.

The features of an abstract syntax tree are density, convenience and expressivity. It contains no unnecessary nodes, is easy to walk, and emphasizes operators, operands and relationships between them [19].

Semantic analysis After building the abstract syntax tree, the next stage is operating on it. This requires an algorithm to visit all elements. It can be a tree walker that is a set of recursive functions, a tree visitor that is an implementation of the visitor pattern [12], a tree grammar that is a specific grammar to generate a tree visitor, or a tree pattern matcher that is a kind of tree walker that triggers actions if it encounters specific patterns.

Those visitors can be adjusted to rewrite the tree to change its structure[24].

Interpreting can be split into two scopes. One is based on the created structures of previous stages. The other one is based on byte code instructions to gain higher efficiency. A byte code interpreter simulates a computer consisting of memory, CPU and registers.

Interpreting is an alternative to translating, but the processes can also complement each other. The consequence of interpreting is the runtime overhead with significant debugging support and fault tolerance.

Translating names the process of mapping input constructs to output constructs. A well known approach in linguistics is a literal translation. It replaces input symbols with output symbols without referencing the concept. The German phrase "Ich verstehe nur Bahnhof" could be mapped to "I only understand train station". But the correct translation, regarding the concept[25] of the phrase would be "It is all double dutch to me". Using an intermediate form evades this aspect.

Translation approaches divide into different scopes based on the granularity or even absence of the intermediate form. Syntax-directed translation emits immediate output from specific actions embedded into a grammar or parser. It processes the input in one pass. Therefore, it cannot perform sophisticated translations like the creation of forward references. The consequence is a less efficient translator that is easy to create and understand.

In contrast exists the rule based translation. It requires translation rules and a rule engine. Like pattern matching, it requires only rules for input constructs relevant in the translation process. Because the rule engine knows what to do based on the translation rules, it is very expressive and formal. Thus, it is able to perform the translation automatically. On the downside, the rule engine operates as black box, making it hard to comprehend what happens. Moreover, the translation process slows down significantly, the more rules are specified.

Next to that, the model-driven translation is the common translation approach. It focuses on the intermediate form. Depending on the complexity

[24] Furthermore, based on the visitors it is possible to identify and track symbols to create a symbol table and thus emulate an abstract syntax graph. Another option is the enforcement of static typing rules.

[25] As well as cultural aspects.

of the translator, it executes various task with multiple passes over the model before entering the generating stage. The simplest form creates an intermediate form to walk and emit output via print statements. The difference to the syntax-directed translation is walking the model instead of parsing a stream. The approach is highly scalable. Because once the model is created, it is possible to annotate, rewrite, or restructure it with interchangeable components. Based on the target language simple print statements could be insufficient. It requires to create output objects and structure them into an output model.

Collecting the output objects can be done output- or input-driven. The output-driven approach traverses the input model for each specific task, thus the output order dictates the translation process. The consequence is, walking the intermediate form frequently with an already correctly structured output model. The alternative is the input-driven approach that decouples the input from the output order at the cost of collecting output pieces. Consequently, the overhead of managing and collecting rises, the number of traversals falls, and the result is a less coupled structure for further processing.

Generating is the process of producing something. Depending on the domain, it can be power, a sentence, or a curve for example. In the context of this work, it is a linguistic utterance of a formally specified language. In the general context of generating software, it could be any software artifact. Usually, the generator is included in the translator, yet the generating approaches can differ significantly.

A so called code generator omits structured text from the output objects of the output model. Those objects can be strings, templates, or custom objects consisting of a string representation. Custom objects require a higher creation and annotation, or filling effort. Plain strings are easy to handle but inflexible. Templates are today's common approach to get the benefits of both worlds. Templates are like strings with specific holes to fill.

1.8 Characteristics of language transformation

The aforementioned section reflects the general multistage pipeline. If the prerequisites are satisfied, the translation stage will apply translational semantics as semantic mapping because semantics and code generators are alike. The default is the translation of a higher abstract language into a lower abstract language.

Another common transformation is the in-place transformation that does not change the language but rewrites the language instance. Thus, the transformation is semantic-driven. Providing multiple semantics would make the expected behavior of the transformation ambiguous. E.g., an informal language documentation of Java and the Java compiler are two semantics. Ideally, both are synchronous. But, what if a Java program does not work as expected? If the documentation is correct, the compiler may be buggy. If the compiler is correct, the documenta-

tion may be erroneous. Defining a leading semantic prevents such ambiguity and possible misunderstanding.

Changes in the source or target language require changes in the semantic mapping and thus changes in the code generator. If one or more involved languages are tailor-made, designing the code generator for adaptability will be crucial and should be considered right from the start.

In the context of model-driven software development, the model transformation is a key role. A code generator is a model to text transformation, as a special form of model transformations. A main application of model transformations is bridging the abstraction gap from more abstract instances or so called platform independent models to more concrete instances or so called platform specific models.

Creating a new model out of another one, as well as rewriting the model is a model to model transformation. The concrete representation of a model can be text too. Thus, the difference between a model transformation and a code generator, as well as the difference between an instance of a modeling language, like UML, and an instance of a programming language ,like C++, is just virtual [9]. Because of the different strategies, model transformations are categorized by their features according to [38]:

Specification Some transformations provide additional syntax elements for relations or conditions, like OCL for UML [28].

Transformation rule Represents the smallest unit of transformation and can be a tree rewriting rule, an implemented function, or a template for example. It consists of a count of domains, depending on how many source and target domains are involved. Usually, only one target and one source domain are concerned, but model weaving for example involves more than two domains.

Some rules require a separate syntax for target and source (e.g., a graph transformation). The opposite would be a rule implemented as Java function for example. Additionally, the rules could be parametrized, provide conditions, or even support reflections as well as aspects. As already mentioned in 1.7 on page 11, transformations can be based on intermediate forms, and especially relational approaches provide multidirection, meaning that the transformation could be executed vice versa.

Rule application control This feature contains the kind of location determination and the scheduling. Strategies to apply rules to a specific location can be deterministic, non-determenistic or interactive. The scheduling can be implicit or explicit, and consist of a selection strategy, potential phases and an iteration strategy like recursion or looping.

Rule organization The organizational aspect is relevant for reuse and adaptability. The transformation could provide a modularity and a reuse mechanism like inheritance or composition. Additionally, a domain could dictate the structure of the rules.

Source-Target relationship This feature determines if a new artifact needs to be created or if the existing one can be updated.

Incrementality Embraces the possibility of executing rules in an incremental way, depending on the changes in the target or the source. Some transformations provide the possibility to preserve user edited parts of the target (e.g., protected regions or empty subclasses).

Directionality Asserts uni- or multidirectionality which is directly dependent on the directionality of the transformation rule, as well as the scheduling logic. Especially synchronization requires multidirectionality.

Tracing Supports debugging and impact analysis. It could enable tracking of model synchronization. It could be plain logging to reconstruct the execution, or an interpreter functionality. It is based on so called traceability links that connect source and target elements. The creation of the links could be done manually or automatically with additional options to adjust them.

The major categories are, as already mentioned, split into model to text and model to model transformations. The simplest model to text approach is based on print statements. An alternative is based on the visitor pattern [12]. A far more superior approach is template based, supported by most of today's model-driven architecture tools. Those templates are target code enriched with meta code to fill in specific information of the source model.

Using a domain specific language (for details see 1.10 on page 18) supports the developer significantly in designing those templates because usually general purpose languages do not provide sophisticated string processing mechanisms. Templates resemble the generated code and are independent from the target language. Thus, templates can be created for any target language with the consequence of creating possible incorrect code fragments, and if not supported by a type system, no typing at all. Model to model transformations are divided in the following approaches according to [39]:

Direct manipulation Based on an internal representation and an API, the transformation rules target directly this representation. It is the least abstract approach because each rule, as well as the scheduling and tracing for example, has to be implemented. Using libraries and frameworks justifies this approach after all.

Structure driven This approach creates a structure based on the target model and subsequently annotates it with additional information like references. After this two phased sequence, the user defined rules are applied based on the framework determined scheduling. The approach aims for targets containing one to one or one to many references like database schemas.

Operational This approach is similar to direct manipulation, but it extends the metamodeling mechanism mentioned in 1.6.1 on page 6. E.g., QVT Operations add additional mappings to the transformation by OCL expressions [40].

Template based Similar to the model to text approach, a model template resembles the target consisting of the model and embedded metacode, like

stereotypes or OCL expressions. Consequently the transformation is easier to understand.

Relational Based on the mathematical concept of relations, this approach specifies the relations between source and target with constraints in a declarative manner. Some approaches use logic programming to implement the relational approach. The transformations are side effect free and thus support multidirectionality. Consequently, the performance is directly related to the constraints. The syntax requires mathematical comprehension.

Graph transformation based Graph transformations (for details see 1.8 on page 16) seem to be the obvious choice to realize model transformations because models and metamodels are like instance graphs and type graphs that are just formal representations of class models. The fundamentals of graph transformations are illustrated in [24, 22, 23].

The transformations are defined in source and target graph patterns, with additional conditions in the source graph to add logic. The scheduling can be modeled with state machines. Multidirectionality can be simulated using triple graph grammars. The disadvantages of graph transformations are the difficulty to use correlating the complexity to create and understand a graph transformation, the non-determinism, and possible inferior performance [30].

Hybrid Hybrid approaches combine aforementioned approaches on different levels of granularity.

Graph transformations

Graph transformations are based on models as representation of reality. The underlying data structure is the graph [3].

A graph is just a formal model for entities represented as vertices and their relationships represented as edges. Hence, it can be processed by computers. A general bottom up approach uses snapshots and scenarios to derive rules and concepts. Graphs, like models in UML, can represent those snapshots as well as concepts and rules.
A snapshot is represented by an instance graph, similar to an object diagram. A concept is the generalization of snapshots. It is represented as type graph, similar to a class diagram. If an instance graph can be typed over a type graph, it is a valid instance of the type graph.

Rules are extracted from transformation scenarios and represent generalized behavior. A rule consist of a left hand side, representing the precondition, and a right hand side representing the postcondition. Both sides are instance graphs typed over a type graph and whose structure is compatible. Next to generalization, rules can have a constructive meaning. But adding or removing elements requires additional logic to ensure the correctness of the graph. Removing a vertex could lead to a dangling edge, resulting in an invalid graph.

A solution could be to enforce the deletion of the edges of a vertex that is going to be removed. Because of the non-determinism, the result can be unexpected

and surprising. Adding additional concepts and thus additional complexity, evades the dangling edge problem. E.g., using constraints expressed in first order logic to express forbidden subgraphs. The application of multi-objects, known from object diagrams, could address collections of nodes with their responsible connections. Adding additional conditions to the rules adds more granularity and thus more control to the rule application (see [22, 24] for more details).

[41] illustrates and compares different graph transformations to transform UML models and opposes the results with QVT.

1.9 Language composition

A variety of transformations transform one source language instance in a less abstract target language instance. As mentioned in 1.8 on page 13, more than two languages could participate in a transformation. The consequence of using multiple software fragments is better version control, increased understandability, and reuseability regarding the single responsibility principle at the cost of referencing. Especially working with multiple developers in a software project, sharing one model as single point of truth, could result in a management disaster because of proprietary data formats, rendering the merge process impossible. Yet using multiple linguistic utterances to represent the source of information, requires referencing or linking.

Referencing can either be done by address respectively position, or by name respectively identifier. Using the address results in tight coupling but no name resolution overhead. Referencing by identifier should be the favored approach because of the benefit of loose coupling. Referencing a language component is like a use dependency known from UML, indicating that one model element requires another. It renders the referenced model element passive because it only provides its service. It renders the opposite site active because it needs the service to fulfill its duty.

Extending a language by language composition renders the active elements as additional language aspects from another domain, using the passive elements provided by the core language to partition the responsibilities, as depicted in 1.11 on page 19. The object oriented approach uses associations to represent references. Calling operations of associated objects requires access. A well designed class provides only access to public elements of itself to other classes. Thus, it hides its implementation. Using this information hiding principle in language composition, renders the referencing process easier for the language engineer. First of all, the innards of a language can change but the referencing elements should still work. In the second place, accessing only provided elements results in lesser coupling and a more scalable and flexible architecture.

The interface mechanism of object oriented design is an excellent implementation of information hiding because it separates the implementation from the signature or specification. The same applies for a language interface. The simplest language interface would be the abstract syntax model[26] of a language. Because

[26] A model representing the concepts of a language and their relationships to each other.

this does not hide anything it requires further refinement to control access to specific elements. An alternative would be a model and a transformation of the abstract syntax of the source language to a new model to simplify the provided information.

Known from design by contract, the offering side fulfills a contract and the consuming or required side requires a contract. The offering side can be considered as the passive language while the consuming side is an active language. E.g., OCL could be considered an active language, referencing parts of UML which is the passive language. Because OCL references only a small part of UML, each language providing this part could be used as passive language. It results in a language composition consisting of OCL and the provided language core.

The consequence of language composition is not only the loosened coupling but also the overhead of handling the references correctly. Thus, the IDE should check and resolve references in general and especially in case of changes. More details about language composition can be found in [9].

1.10 Domain Specific Languages

In linguistics each language is able to express all possible concerns in itself. While some languages require more effort to represent an aspect than others, they could be considered general purpose. But a physicist explaining the concepts of quantum mechanics to a doctor of literature, seems to be a physical impossibility even though they are speaking the same language.

In general it can be considered as a significant difference in the semantics, thus resulting in a problem of understanding. Two physicist talking about the same matter could understand each other, but because each semantics is subjective, it could result in a different understanding. Talking about an issue requires similar knowledge and terminology. Thus, the used language is a subset of the general purpose language. It consists only of elements necessary for this domain[27]. Such a domain specific language evolves from the people using it and the concepts it consists of. While linguistics do not consider these language subsets, computer scientist use this concept to provide a customized language for specific purposes.

Therefore, DSLs are highly specialized and problem oriented. Next to the characteristics of a language mentioned in 1.5 on page 3, they have additional categories[28] according to [9, 14, 11, 44]:

Audience or target domain The audience of a DSL is important because the hole purpose of the DSL is to support the user in expressing his will effectively and efficient. The audience are domain experts or domain users. It is crucial to reflect as much of the domain and the terminology as possible but with limited expressiveness. That means it should not be possible to express

[27] The developers of UML defined domain as follows: "Domain: An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area."[42]

[28] A more detailed list of desired properties can be found in [43].

something of another domain, but it should be expressive enough to make its intention clear to a user without programming skills for example. The notation needs to be tailored to fit the domain and should be developed in cooperation with the user.

Direction or source : DSLs can be classified as horizontal, also known as technical oriented, and vertical also known as business oriented. A horizontal DSL consists of technical characteristics shared by a group of applications. Frameworks and libraries are common implementations of horizontal DSLs. It is called horizontal because the contained concepts are all on the same abstraction layer. Because the domain knowledge is already known, it just needs to get formalized.

A vertical DSL is tailored to a business domain. It requires to mine the knowledge of the domain expert to create a domain model [45]. Based on this model the formal specification and tooling can be derived. It is called vertical because it references concepts on different abstraction layers. Yet it could be based on a horizontal DSL.

Classification : DSLs get classified by the way they are implemented in. So called internal DSLs are implemented in an existing language. Using such a host language renders them as embedded languages (see [43] for more details about implementation variants and their associated consequences).

Like the UML profiling mechanism, the language reuses elements of the host language. This makes it easy to create such a DSL. It delegates parsing and the like to the host language. Consequently, using an improper host language renders the DSL less effective because it is directly bound to the host language and its specification.

External DSLs on the other hand are developed ground-up. The implementation is rather complex, but it does not need to have the complexities of a full-blown language. It is arguable to add non-textual DSLs to the classification because it concerns only the notification discussed in 1.5 on page 3. Detailed information about the classification of DSLs can be found in [14, 11].

1.11 Language oriented programming

Language oriented programming tries to unite established software development concepts to gain their benefits [46]. Using a GPL to develop programs is a traditional software development approach. A GPL provides various language features and is Turing complete. Focusing different paradigms, a GPL can be imperative, object oriented, functional, declarative or even a combination of those paradigms. Most GPLs (e.g., Java) are hard to customize. Using frameworks and libraries supports the developer in writing more customized code, but the formal language specification will not be edited.

Combining Java with AspectJ[29] for example, adds aspect orientation to the

[29] http://www.eclipse.org/aspectj/

object oriented Java world at the cost of additional overhead. The runtime and compile time overhead can be compensated, but the additional AspectJ code adds more complexity to the source code. The developer needs to understand two languages in addition to their special interaction. While adding language features to a GPL is supported by inheritance, template mechanisms, composition, and modularization, it is not possible to remove functionality that will not be used by the developer in a specific project.

The main purpose of programming is to represent the developers mental model of the software as a linguistic utterance of the target programming language because: "A program is any precisely defined model of a solution to some problem in some domain, expressed using domain concepts..."[47].

Using a GPL delays this transformation process because the concepts differ significantly, since the GPL is too verbose and not tailored to the problem domain. Understanding and maintaining the resulting source code considers the same aspect vice versa. The developer needs to understand the source code and transform it back to a mental model. Using libraries and frameworks simplifie this process at the cost of learning and using an additional chunk of software.

Language oriented programming tackles the problem of the divergence between mental model and target language instance. A language consisting of the same concepts as the problem domain simplifies the transformation process. Tailor-made editors, to represent the linguistic utterance from different viewing angles, support developers to understand the ideas behind the source code. Thus, language oriented programming is a paradigm to provide the developer with one or more tailored languages and the respective tooling. In addition to that, language oriented programming provides a process of creating such a language and the tooling.

A common approach to create a language based on language oriented programming uses a domain model. A domain model can be deduced by domain driven design [45]. Thus, a language is designed object oriented and encapsulates one domain. Interweaving these languages in an aspect oriented fashion results in a composed language with definite concerns (see 1.9 on page 17). Usually one language represents the minimal core, providing interfaces to other languages. [45] defines the shared kernel pattern or, if a more pragmatic approach is necessary, the conformist pattern to design such an approach.

This modularization concept is similar to the profiling mechanism of the UML as well as the classical library or framework approach. Yet each language provides its own syntax, editor support, type system and transformation engine. Based on the design of the core language, the referencing language modules could be reused within other languages and imported as needed. Thus, keeping the actual language dense and problem specific, but flexible and expressive. More information can be found in [48, 13].

Language oriented programming can either be done projectional or parser based. The projectional approach renders the creation of editors simpler because of the underlying structure that is dictated by the editor.

1.11.1 Language modularization

As already coined in section 1.11 on page 19: "A modular language consists of a minimal language core, plus a library of language modules that can be imported and used in a program as needed based on the task at hand.[...] A language module is a little bit like a traditional framework or library, but it comes with its own syntax, IDE support, type system, and compiler or transformation engine."[48]

Present-day tools for language oriented programming, like MetaEdit+[30], Intentional Domain Workbench[31], Meta Programming System[32] and Xtext[33], aim for this modularization on different kind of levels and with different implementations, and their respective consequences.

First of all the editor can be projectional or parser based. A parser based approach requires parseable code and the projectional approach a specific format that is in general not parseable or human readable. Thus, both approaches could only communicate by a standardized exchange format. To create a representation in the target language the internal model needs a transformation engine and additional transformations. If the target language is textual like Java or C++, the projectional approach requires an additional model to text transformation because the models cannot be fed directly into the compiler or interpreter. More details can be found in [13].

Most language extension mechanisms use inheritance and thus adapt the Liskov substitution principle. A language extending another one is able to process linguistic utterances of the extended language. This could also mean to restrict the language concerning the base language. While Xtext provides single inheritance, MPS is less restrictive and thus renders the process of language extension easier and more efficient. Inheritance can also be applied to the generator or transformation engine to customize the transformation. Language composition can be done by adapters or facets [12].

Similar to the language interface approach from section 1.9 on page 17, the adaptor gets injected and thus a comfortable editing experience is provided to the user. Combining languages in this way or by cross references, will render itself useful if each concern was implemented as separate language. In contrast to reuse, language combination is less restrictive because designing a language for reuse restricts the language significantly. The language being reused is often a core language as mentioned in section 1.11 on page 19.

Embedding languages, like the concept of internal DSLs known from section 1.10 on page 18, provides a mechanism to combine completely independent languages (e.g., SQL embedded into a real estate DSL). The obvious consequence is adding more concepts than required and a possible interaction overhead with almost no effort. Furthermore, embedding languages with features like multiple inheritance within a target language like Java could lead to undefined behavior or surprising results.

[30] http://www.metacase.com/
[31] http://www.intentsoft.com/
[32] http://www.jetbrains.com/mps/
[33] http://www.eclipse.org/Xtext/

Language annotation is a less intrusive extension mechanism. Like using stereotypes in UML or annotations in Java, the annotated artifact gets enriched by metadata, that can be interpreted by transformations or tools[34]. Those annotations render useful for documentation, tracing and variability. See [50] for more details.

The modularization of languages by concerns or concepts require additional hooks. Known from aspect oriented programming, an advice defines the behavior. An advice relationship in language modularization could represent the behavior of executing additional rules before or after the extended part, or they could even override it. The join points and pointcuts are defined by the modularization implementation. More details can be found in [48, 51].

Yet the aspect language needs to be specific to the particular language, thus it requires a language specific aspect weaver [52].
Embedded software development benefits significantly from language modularization and language oriented programming because the integration and tooling problems required by modeling fade from the spotlight. Especially product line engineering (see the following section 1.11.2) combined with language annotation promotes the development of embedded systems.

1.11.2 Product line engineering and language modularization

Product line engineering or product family engineering is a concept to simplify the software development and deployment process of so called product families. The process to derive a software product line is divided into two phases. The first phase is called domain engineering to analyze and define the domain. It specifies the features of the product family and their relationships[35].

Product line engineering separates the problem space, respectively the problem domain, from the solution space respectively solution domain. "The former is concerned with end-user understandable concepts while the latter deals with the implementation of the product features using software technologies."[53]

Subsequently follows the product engineering to derive final products, similar to the application engineering of generative programming discussed in section 1.12 on page 25 as well as in [54]. Thus, product line engineering is about the variability of products of a product family. "In traditional SPLE [Software Product Line Engineering] approaches, variability is mainly handled using either mechanisms provided by the implementation language, such as patterns, frameworks, polymorphism, reflection, and pre-compilers or using configuration and build tools to set compile time variables and select variants of assets"[53].

The structural variability can be represented by a creative construction DSL often used in the solution domain. Non-structural variability is represented by a configuration DSL often used in the problem domain. The feature model defined

[34] A similar approach is described in [49], resulting in cascading model-driven software development.

[35] Domain driven design from [45], describes the creation of domain models to represent the problem domain containing the terminology and the relationships of the elements. It yields a similar representation to create a ubiquitous language.

in [54] provides a graphical notation to express configurable variability[36]. Figure 1.4 on page 27 shows an example of an abstract feature diagram.

Variability can be expressed either positive or negative. Negative variability removes parts from the creative construction model based on the configuration models. Positive variability starts with a minimal core and adds additional components using aspect weaving for example [53].

In addition to manage variability, product line engineering provides a mapping from problem to solution space. Using the modularization approach from section 1.11.1 on page 21 to create a language, enables such a mapping effectively. Language annotations can be used to represent features of the software component and thus enables feature modeling. A language workbench, like MPS, enables the language engineer to create a thorough software product line. It enables the developer to define a language, the editors and the transformation rules. It does also provide the possibility of language modularization and composition. Hence, it provides a simplified approach of product line engineering.

The transformations are piped in the kind explained in the section 1.7 on page 11 and require multiple transformation steps to create target code. Enriching the software components with features by language annotation is a positive variability. It requires a transformation in an aspect oriented manner because it weaves aspects to the result dependent on the particular models.

Bootstrapping language workbenches and language oriented programming enables the creation of meta-product lines[37] and software factories. Software factories combine component based development (see [49, 55] for more details), model-driven development and software product lines to create an integrated software development and deployment process.

Modern application development is supported by tools using abstraction and best practices. Software factories use the term domain specific assets to address this. Domain specific assets are used to complete domain specific tasks and thus mixing product line engineering and product development. Model-driven development uses UML to represent linguistic utterances. It is based on object oriented analysis and design to create the models. Using this approach, incorrectly assumes that the structure of the solution will match the structure of the problem as depicted in [56].

Thus, the primary use of UML is the visual representation of classes and their relationships to each other because UML is bound to object oriented concepts. Model-driven development processes the information of the models to automate the development tasks. It promises platform specific implementations similar to bytecode languages used by C# or Java. A lot of CASE tools, targeting model-driven development, propose a top-down process, thus forestalling rapid iteration. Roundtrip engineering tackles this problem at the cost of adding complexity to the model and thus rendering the modeling obsolete after one generation process. The main problem of model-driven development is the synchronization and resynchronisation of model and code base and thus the bridging problem between model

[36] In terms of the OMG the feature model would be the meta model and a concrete configuration would be the model [53].

[37] Product lines of a product line architecture. See [53] for more information.

and implementation.

Raising the abstraction layer by using an intermediate framework to address the problem domain targets the problem and results in domain specific models. An alternative to the framework could be a pattern language to implement the abstraction. Addressing the domain specific models with a DSL leads to various abstraction layers on top of each other and progressive transformations. Similar to feature models, a categorization and ordering is necessary to configure the approach in a deterministic and lucid way. Software factories define a matrix like figure 1.3 of the categorization of the models.

DSLs	Business	Information	Application	Technology
Conceptual	• Use cases and scenarios • Business Goals and Objectives	• Business Entities and Relationships	• Business Processes • Service factoring	• Service distribution • „Abilities“ strategy
Logical	• Workflow models • Role Definition	• Message Schemas and document specifications	• Service Interactions • Service Definitions • Object models	• Logical Server types • Service Mappings
Implementation	• Process Specification	• DB Schemas • Data Acess strategy	• Detailed design • Technology dependent design	• Physical Servers • Software Installed • Network layout

Figure 1.3: The layered grid for categorizing models from [57].

Filling the matrix results in a so called software schema, describing a set of specifications that must be developed to produce a software product. This product represents a software template that can be loaded into an IDE to create specific types of software also known as software factory for product families. The transformations of the models are divided into horizontal, vertical and oblique. Horizontal transformations are refactorings or delocalizations. Vertical transformations are classical refinements. Oblique transformations combine horizontal and vertical transformations. Delocalization is similar to language modularization by aspects. Aspects, like logging, are defined and the delocalizing transformation executes aspect weaving.

A software factory is based on abstraction, granularity and specificity. A problem of using abstraction is that: "Powerful abstractions that encapsulate large amount of low level code tend to address highly specialized domains."[57]

Abstraction always tended to be interpreted hierarchical like in the development of programming languages. Software factories use granularity to represent the amount of abstraction. Granularity is a measure of the size of software artifacts carrying abstraction. A flaw of this approach is the limited traceability. Designing a business component, like a service, and programming this component differs significantly due to the different abstractions. Using component based design with regard to composition, decomposition, partitioning and interaction remedies this deficiency. Component based design uses separate components to structure the resulting software. The characteristics of a component are functional isolation, de-

fined services and resources it provides, and defined services and resources required to provide the previous one.

> "A component, therefore, encapsulates its constituent features. Also as it is a unit of deployment, a component will never be expected to have access to the construction details of all components involved. For a component to be composable with other components by such a third party, it needs to be sufficiently self-contained. Also, it needs to come with clear specifications of what it requires and provides. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces."[58]

Völter defines in [49] three models, with their respective metamodels and relationships to each other, to specify components. A type model to describe components as data structures. A composition model to describe a logical system consisting of logical components referencing each other. A system model to define the hardware and process structure to deploy the logical components on.

Specificity is the scope of the abstraction. The more specific an abstraction gets, the more it contributes to the problem domain, but the less it is useful in other domains and vice versa.

While model-driven development and component based design are focused on building one software product, product lines define product families. Thus, it is necessary to distinguish between the product line developer and the product developer. The product line developer defines the product line scope and hence the domain models. The assets created by the product line developer are used by the product developer to create instances of the product family.

In the term of software factories the product line developer creates the software schemas and the software templates to create the software factory. The product developer is now able to use the software factory. In the context of software factories, the factory itself is the software template loaded into the IDE, thus making it possible to create software factories in a recursive manner. Using a language workbench to instantiate a language specific editor would enhance the process even further because of an IDE tailored to the specific problem domain with the consequence of creating an additional editor.

1.12 Generative programming

> "Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."[54]

Elements of generative programming are the problem space, the solution space and the configuration knowledge, to convert an instance of the problem space to

an appropriate instance of the solution space. This is the basic principle of Product Line Engineering as well as software factories. In general the configuration knowledge has a similar responsibility as a compiler. It contains rules for illegal combinations, default settings and dependencies, as well as optimization and construction knowledge. While object oriented design struggles to develop reusable Software fragments, generative programming tackles this problem in two steps.

First of all, the design and implementation of a generative domain model is focusing development for reuse. Secondly, it uses the generative model to produce a concrete system. Evans describes in [45] an informal method to create a domain model respective to the target software. Like most of the object oriented analysis and design methods, it is aiming for a single system. Domain engineering, on the other hand, addresses a multi system scope to deduce software for application engineering. The first step of domain engineering is the domain analysis. This phase is about collecting the domain knowledge, also known as domain scoping, and refine it to a coherent domain model known as domain modeling. The resulting domain model consists of the domain definition representing the scope, the domain lexicon, the concept models and the feature models.

A concept model describes the concepts of the domain and their relations to each other. The feature model defines requirements to specify the systems in a domain. It represents the configuration aspect of the concept models. The feature model prescribes the possible feature combinations, hence the variability. After the analysis phase follows the domain design to develop an architecture for the systems and a production plan. Finally, the domain implementation proceeds which contains the reusable components, domain-specific languages, generators and so forth.

To represent a concept model it is possible to use a class diagram of the UML to pursue the classical view. Representing a feature model requires a different syntax (i.e., the feature diagram depicted in [54]). The linchpin of this thesis is based on feature models. The Feature-Oriented Domain Analysis method defines three types of features: *Mandatory*, *Alternative* and *Optional* features. FODA prescribes two types of composition rules: *Requires* and *Mutually-exclusive with* rules. Furthermore, FODA distinguishes between various features (e.g., runtime, compile time features). A feature can be seen as a property of a concept. It describes the concept and displays the variation of the concept instances. Figure 1.4 on the following page shows a feature diagram of the car example from [54]. A car consists of a body, a transmission that is either automatic or manual, an engine that could be powered by electricity, by gasoline or by both, and it could pull a trailer.

Variable features need to be bound before use. Binding requires a site and a mode. While the site defines the link, the mode defines the behavior (e.g., static, changeable or dynamic binding). Feature diagrams and UML class diagrams can be used interchangeable to represent a feature model. To implement the variability, object oriented programming concepts, like inheritance and parametrization, can be used. Those approaches would lead to high complexity if used for features representing aspects, like security, because an aspect crosscuts with several modules.

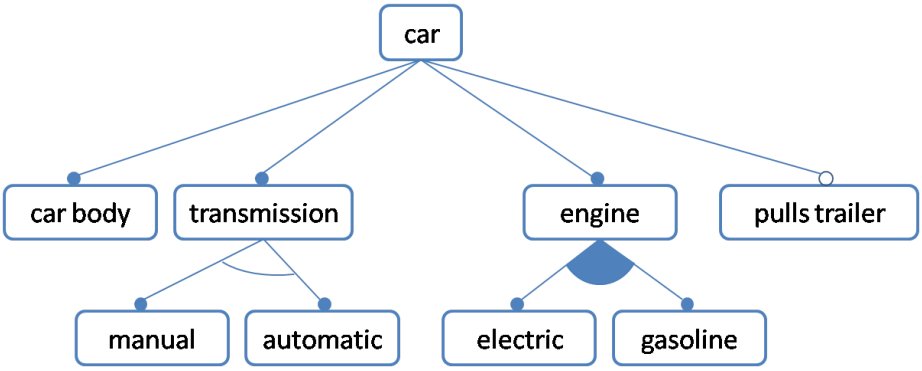

Figure 1.4: The car example feature diagram of [54].

Applying the separation of concerns principle from [59] for the aspects should result in understandability, adaptability, reusability and so forth. Alas, in the real world exist dependencies, as well as overlapping of issues, which cause either redundancy, due to localization, or lesser separation of concerns. To tackle the problem of redundancy, a software could create an internal representation and show the issues through different views. Yet editing elements in such a view needs to change the internal representation.

To derive features from concepts, two decomposition techniques are determined according to [54]:

Modular decomposition Decomposes a system into hierarchical units resulting in high cohesion and low coupling.

Aspectual decomposition "The main idea behind aspectual decomposition is to organize the description of a concept (e.g., a system, a domain, a component, a function, and so on) into a set of perspectives, where each perspective concerns itself with a different aspect and none of which is itself sufficient to describe the entire concept"[54].

This results in various models referring the same concepts. As a consequence, models reference each other which is known as crosscutting.

Both approaches complement each other and represent the natural modeling approach.

Techniques to implement variability

Inheritance is a static compile time variability mechanism. Single inheritance is ideal for non-simultaneous single variation points. It is also possible to represent non-singular variation points with the consequence of multiple implementations. Alternatively, multiple inheritance using mixins can be applied which results in more complicated relationships. More flexible than multiple inheritance is parametrized inheritance with the consequence of possible invalid compositions and differing semantics.

Static parametrization, like parametrized classes, implements statically bound simultaneous dimensions. Dynamic parametrization using dynamic method binding implements dynamic binding. Dependencies between features are represented as constraints and default dependency rules. Constraints specify valid feature combinations and default dependency rules which define default values for unspecified parameters. To express constraints in a UML model, the OMG specified the OCL. For concrete implementations it is useful to use a configuration system (i.e., a generator).

Generic programming represents an approach to realize static parametrization. The Standard Template Library represents an example of using generic programming to provide an abstract representation of efficient algorithms. Generic programming correlates with generic parameters, polymorphism and parametrized programming. "The idea of parametrized programming is to represent reusable software as a library of parametrized components, which can be combined in a vast number of ways."[54]

If a concept cannot be localized by generic programming or object oriented programming, it will be necessary to use aspect-oriented programming. It decomposes a problem into functional components and aspects which crosscut the functional components. Subsequently, it composes the components and aspects to gain an implementation. Aspect oriented decomposition is realized in different ways according to [54]:

Subject-oriented programming is based on different subjective views on an object to obtain a so called subject. "A subject is a collection of classes and/or class fragments (i.e., mixins) related by inheritance and other relationship (e.g., aggregation, association, and so on). Thus, a subject is simply a partial or a complete object model."[54]
The composition is specified by correspondence and combination rules.

Composition filters add a mechanism to separate functionality from message coordination to the object model by so called message filters. The object will be extended by an interface layer for input and output message filters. The filter intercepts the message to realize before and after actions. Furthermore, the filter realizes redirection, delegation and dynamic inheritance.

Adaptive programming separates behavior from the object structure. The behavior code will be written against a partial specification of classes. The partial specification is called a traversal strategy. It specifies the traversal of a given concrete class diagram. Based on that information, methods to pass information between classes can be generated.

Thus, it separates aspects from functional code to avoid code tangling. An alternative would be refactoring to patterns, yet it is not sufficient for real systems. The resulting fragments need to be composed by a so called aspect weaver. The aspect weaver executes a weaving process similar to compilation or generation. The aspect weaver itself can be implemented as generator and thus executes a static process, or as interpreter and thus executes the processing at runtime. The desired properties of the composition are minimal or even no coupling between

aspects and components, selectable binding times and binding modes between aspects and components, and non-invasive addition of aspects to the component.

So called join points are well defined points to interconnect aspects and components. Join points can be references by name to a language construct, references to the uses of a given construct, or references to patterns.
The binding time could be before or during runtime. The binding mode could be static or dynamic. E.g., inlining represents static binding and virtual methods represent dynamic binding.
Using a composition operator and defined hooks realizes non-invasive adaptions. E.g., inheritance is a composition operator with respect to the deriving class. The hooks could be represented by virtual methods.

Aspect orientation requires an abstract implementation to express aspects. The implementation can be a library or framework, a separate language like an external DSL, or a language extension similar to the mechanisms from section 1.11.1 on page 21. Additionally, aspect oriented programming requires an implementation of the weaving process to compose the aspects with linguistic utterances of the respective component or even other aspects. Because it is yet another language transformation, it can be realized with a transformation introduced in section 1.8 on page 13. An alternative implementation would be dynamic reflection that requires metaobjects of the target language.

Known from section 1.4 on page 2 and section 1.10 on page 18, languages consist of an abstraction and a specialization dimension to classify them. Aspect weaving adds an additional dimension of crosscutting. A higher level of crosscutting and abstraction requires a more complex weaving transformation, or extending the transformation pipeline similar to that of section 1.7 on page 10. More information can be found in [54].

1.13 Synopsis

This chapter introduced the principles of generating linguistic utterances of a target software language from one or more source languages. The concept is based on a multistage pipeline to partition the process into well defined stages. It starts with a reader component to parse the linguistic utterances of the source languages and creates an intermediate form. Based on the intermediate form and specified transformations, the intermediate form will be traversed, annotated and rewritten. Finally, an output model is forwarded to the generator to create software artifacts based on the target software language. The transformations can be categorized by their features and applied approaches. Consequently, the applied transformations are directly related to the requirements of the language application.

Because of the complexity of UML, it is reasonable to apply a model to model transformation to simplify the model. Using the resulting model as core model to enrich it with features from other models, embodies a flexible approach to generate a concrete software system from a plethora of options. Configuring the weaving component by choosing a generator empowers the product developer even

further. This approach is a simplified version of product line engineering as well as generative programming. It scatters the features to the active languages that reference the core model. Thus, a language workbench is required to create each language with its respective editor. Those extension languages represent positive variability. They should be reusable and non-invasive to support model-driven software development. Because the audience of those languages are developers, it is sufficient to provide a textual notation and thus a parser based editor. Hence, the languages can be specified by a context free grammar. Yet UML models are usually projected, thus the editor need to provide an export to a standardized format. Because UML is specified by a metamodel, the specification of an extension language should provide a transformation to a metamodel.

The approach requires at least two different language engineers. One language engineer for the core language. It requires no concrete syntax but a dense abstract syntax, designed for other language engineers. The semantics should be translational and adaptable. Another language engineer provides a default transformation of the core model to the target language. He needs to specify hooks for the language extensions and a configuration language to adapt the generation process. Each extension language can be build by a different language engineer. He has to use the specified join points but can design the language at will. He needs to provide at least one concrete syntax and one semantic description. The semantics should be pragmatical for the language user to simplify the configuration and extension process.

Each extension language should render one domain and comply the characteristics of a language (simple, minimal, distilled, composable). Vertical domains should be represented by a UML profile because of its pertinence of the communication between domain expert and developer. Technical domains should be represented by a non-invasive external DSL because the information is essential for the developer and the generator. Thus, concepts are modular decomposed by UML and each crosscutting aspect is represented by one or more extension languages. The variability is statically bound and parametrized. Because of the non-invasive approach, the features (and thus the product family) can be extended anytime by the language engineer. Hence, a language engineer is also a product line developer and the language user is a product developer.

Chapter 2

State machines

2.1 Motivation

As mentioned in the previous chapter, it is very useful to generate linguistic utterances of a less abstract language out of a more abstract and thus more expressive[1] language. Yet the abstract language needs to be formal to process it by a computer but usable and understandable for the audience. State transition systems are abstract machines based on directed graphs. They are used in theoretical computer sciences to characterize concepts of computer sciences. They are the origin of so called state machines and finite state automata and hence forefathers of UML state machines.

Despite the formal origin, its syntax is easy to understand and used in various domains like telecommunication, avionics and automotive. Especially so called embedded real time systems, or so called reactive systems[2], are represented by state machines to predict and comprehend the behavior of those systems depending on input from their environment. A concrete yet not standardized representation of a light switch as state machine is shown in figure 2.1.

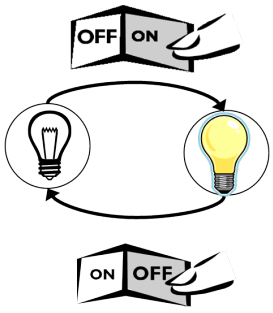

Figure 2.1: A light switch represented by a state machine.

[1] In the sense of expressing an issue dense, simple and thus more comprehendable.
[2] An object is reactive when its behavior is executed in response to received events [2].

2.2 Evolution of state machines

State transition systems consist of a tuple of a set of states, and a set of binary relations of the states called transitions. Additionally, a state transition system can contain a set of labels and thus a ternary relation of a set of labeled transitions. Labels can be interpreted as expected input, conditions, trigger or actions to perform. To represent such a state transitions system more human readable, it is possible to use a state transition table or a directed graph.

So called finite state machines or finite state automata[3] are similar to state transition systems, but they constrain the sets of states and transitions to be finite and define a start state and a set of final states. Finite state machines are models of computation. They are represented by so called state-transition diagrams. They can consume words of a regular grammar. Thus, they need a set of terminals or a so called alphabet. Instead of a set of transitions, they require a set of transition functions which combine states and terminals. Consuming a word and finishing the process in a final state results in the acceptance of the word. Those finite state machines are called acceptors because of the Boolean result.

Figure 2.2 shows the concrete representation of a state transition system, that accepts words with an odd count of the character '1', as directed graph. The same finite state machine is represented by the state transition table shown in figure 2.3 or by the formal representation shown in figure 2.4. A finite state machine producing output is called transducer. Instead of final states, it contains a set of output terminals and output functions which return terminals of the output alphabet. Two well known kinds of transducers are Moore machines and Mealy machines. The output functions of Moore machines depend only on the state. The output functions of Mealy machines depend on the state and the input alphabet. Thus, a Mealy machine usually contains lesser states than the respective Moore machine. See [18, 5] for more information.

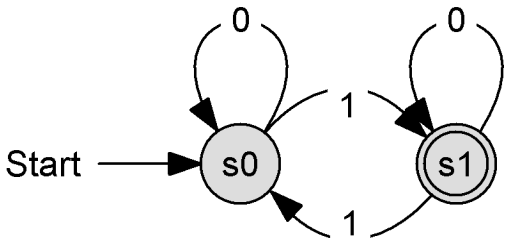

Figure 2.2: A deterministic finite state machine represented as directed graph.

It is possible to represent a finite state machine in a non-deterministic fashion To reduce its states. Lesser states result in more intelligible state machines for humans. Processing a non-deterministic state machine requires either a transformation into a deterministic state machine or the application of less effective

[3] From the Greek 'automatos' meaning self moving or self willed.

algorithms like backtracking. According to [18, 5], a non-deterministic finite state machine can be transformed in a deterministic state machine and vice versa.

δ	0	1
s0	s0	s1
s1	s1	s0

Figure 2.3: A deterministic finite state machine represented as state transition table.

$$M = (\{s0,s1\},\{0,1\},\delta,s0,\{s1\})$$

Figure 2.4: A deterministic finite state machine represented as formal definition.

Yet describing real systems with finite state machines results in a complex representation, because of the exponential growing of states and thus a hardly understandable state diagram. David Harel defined additional syntax elements and corresponding semantics to simplify the representation. He proofed that his statecharts are still formal and thus processable [60, 61]. He added the following concepts and elements to a finite state machine:

Hierarchy Because a traditional finite state machine is flat, it requires exponentially more states to represent containment relationships. Harel used encapsulation to express a hierarchy graphical. The semantics of such a superstate is an 'exclusive-or' for the substates. It provides an opportunity to cluster or refine states. Another advantage is the option to zoom-in or zoom-out a superstate.

Arrows represent directed transitions that can originate and terminate at any state level. Arrows can be labeled with events and parenthesized conditions.

Default states and arrows If nothing else is specified, a default state will represent the state entered. Default arrows are like start and final states of finite state machines.

History Represents the system's history as a circled H. The history state delegates the transition to the most recently visited state. A history state with an asterisk represents deep history to apply the history to the lowest level.

Orthogonality represents an 'and' decomposition and introduces parallelism, synchronization and independence to state machines. A state containing two or more orthogonal components can be in more than one state at the same time. The formal equivalent is the product of automata[4], but orthogonal components could have interdependencies. The syntactic separation of orthogonal components is represented by a dashed line.

[4] A product automaton is a disjoint product of two finite state automata.

Condition is a simplification of transitions represented by a circled C. Instead of two or more transitions triggered by the same event but labeled with different conditions, the event triggered transition points to the condition element. The conditioned transitions originate from this element.

Selection is another simplification element represented by a circled S. If the state that is going to be entered determined by a one-to-one fashion of the value of an event and thus the transitions are obvious, a selection can be used to hide these transitions.

Timeout and delay Harel added a special timeout event because real-time systems use timeouts on a regular basis. A timeout event is parametrized with a bound to derive the delay.

Unclustering Aims to simplify state machines by outsourcing composite states to state machines, thus representing the big picture of a state machine and the possibility to zoom-in to further details.

Action is a mechanism for the state machine to interact with its environment and influences other components. An action can be attached to transitions and have a zero execution time. Additionally, Harel defined a start and stop action for activities. Thus, Harel uses the syntax of Moore and Mealy automata to label transitions as well as states with actions.

Activity Are like durable actions. They can be controlled by the start and stop actions and require an active condition.

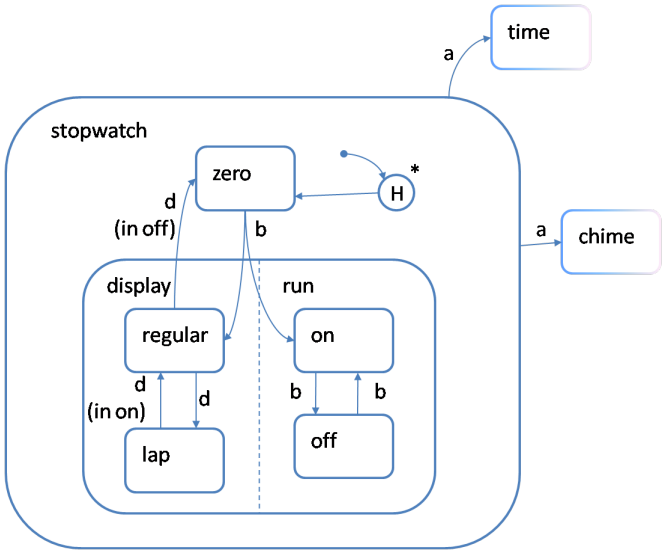

Figure 2.5: The statechart of the stop watch example from [61].

Harel indicates that using his statechart syntax can result in contradictions. It requires the designer to be cautious to avoid the introduction of inconsistencies into the system. Figure 2.5 shows the stopwatch example from [61] which is depicted with Harels statechart as concrete syntax.

2.3 UML state machines

The UML provides various diagrams and thus various concrete syntaxes to express different aspects of systems and software. E.g., the well-known class diagram to depict the structure of software. UML does also provide a syntax to represent dynamic aspects. Dynamics in UML are distinguished between behavior and interaction. Interaction implies instances working together in a synchronous or asynchronous way. It is modeled with sequence, communication, or timing diagrams.

Behavior belongs to an element or object. It is divided into simple, stateful and continuous behavior. Simple behavior is a function (e.g., a search or a sort operation) also known as functional behavior. It can be modeled with activity diagrams. Stateful behavior is assigned to objects possessing a finite set of states. The respective object must be in one state at a time. Because these objects react to events in a well-defined manner, they are also called reactive objects. Stateful behavior is modeled with state machine diagrams. Continuous behavior depicts objects with an infinite set of existence conditions (e.g., algorithmic objects). The behavior depends on the continuous past behavior. The OMG defined an extension of activity diagrams to model continuous behavior. See [2] for more details.

The `BehavioredClassifier` (that is a `Classifier`, that is an `Element`) hosts `Behavior`. E.g., an `Activity` is a parametrized behavior containing `Actions` as the most elemental `Behavior` of the UML. The UML specifies various actions to describe any kind of statement and thus representing the semantic equivalent to statements of a programming language. `Actions` have a so called run-to-completion semantics which means that they are not interruptable. Activities do not have run-to-completion semantics and thus can be interrupted between their actions.

State machine diagrams are the concrete syntax to model a `StateMachine`, which inherits from `Behavior`, to capture the behavior of a `BehavioredClassifier`. Sample diagrams are figure A.1, figure A.2, or figure A.3 contained in the appendix.

Figure 2.6 on the following page shows the metamodel of the `StateMachine` with all its referencing items. It is found on page 536 of [29] and belongs to the `BehaviorStateMachines` package. As already known from section 2.2 on page 32, a state machine consists of states and transitions.

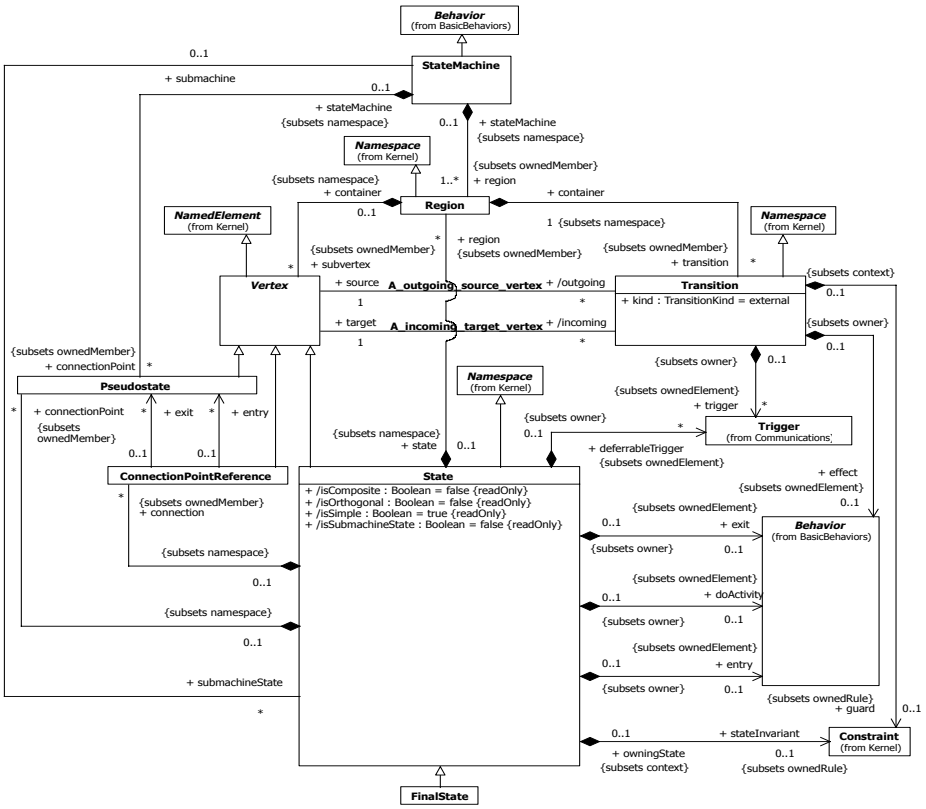

Figure 2.6: The abstract syntax of the UML state machine from [29].

A `State` represents a moment of the behavior of the `Classifier`. It models a situation when a specific invariant condition holds true. It can represent either a static or dynamic situation. It has compartments for its name, internal activities and internal transitions. A state is either active, that means as soon as it is entered, or inactive. According to [29], the UML defines four derived properties of a State:

isSimple Represents a state with no substates.

isComposite Represents a state with substates and thus a container of substates. If a composite state has more than one region, it is also orthogonal. A composite is active if one or more substates are active. The list of active states is called state configuration. The substates are displayed in the decomposition compartment that can be hidden to enable zoom-in a state machine.

isOrthogonal A state with more than one region is called orthogonal and thus each orthogonal state is a composite state. It is similar to Harels orthogonality concept and represents and-states. Thus, states contained in an orthogonal state, but in different regions, can be active at the same time. It enables parallelism, concurrency and independence of state configurations. Regions are separated by a dashed line in the decomposition compartment

of the state. Each region has its own initial and final state to execute tasks in parallel. A completion event (see page 40) is triggered after all regions of the state reached their final state.

isSubmachineState A submachine state is semantically equivalent to a composite state. A submachine state encapsulates another state machine and is made for reuse.

The `FinalState` is a specialized `State` to show that a `Region` is completed. Except for an incoming transition, it is constrained to have no other properties. If all regions of the state are completed, the state itself will be completed. If the region of a state machine is completed, the state machine itself will be terminated.

A transition represents the actual change in the configuration of a state machine. It can be labeled with a guard, a trigger and an effect.

A guard is a condition which indicates whether the transition is enabled for the incoming event. The guard condition is a constraint that is evaluated, "when an event occurrence is dispatched by the state machine"[29] and before the transition fires. It should not have any side effects and needs to evaluate to a Boolean result. The evaluation order of the guards is not defined[5].

A trigger is associated to an event. If it is enabled, it will cause the transition to fire.

The effect is a `Behavior`, usually an `Action` that is executed when the transition fires. The effect may explicitly generate a new event.

According to [29], the UML defines the following transition types:

Compound transition A path of one or more transitions which change a complete state machine configuration to another. It is a set of transitions and pseudostates which lead to a set of states.

High-level or group transition A transition from a composite state either out of the composite state or to another substate of the composite state. It causes the execution of all relevant exit activities, starting with the innermost[6]. If the transition leaves the compound state, its exit activity is executed as well, otherwise it is rendered as local transition[7].

Internal transition "An internal transition executes without exiting or re-entering the state in which it is defined. This is true even if the state machine is in a nested state within this state."[29]

Completion transition A transition without a trigger, but it can have a guard. It is implicitly triggered by a completion event. A completion event is generated

[5] Each transition can have only one guard, but there can be multiple transitions for the same event.

[6] Not explicitly contained in the specification is the entry behavior. It causes all relevant entry activities of the target state, starting with the outermost

[7] Only transitions inside a composite state can be local. They will not cause the exit activity of the composite state if triggered.

when a state finishes its activities or reaches its final state. To evade non-determinism, each completion transition of a state should have a mutually exclusive guard.

A transition from an external state to the border of a composite state is called default entry. It executes the entry activity and the default transition to a substate of the composite state. A transition from an external state to a substate of a composite state is called explicit entry. It executes the entry activity of the composite state as well as the targeted substate. An orthogonal composite state can be entered by default or explicitly. Entering a region explicitly triggers the default transitions of all other regions of the composite state. Leaving a composite state executes all exit activities, beginning with the innermost substate. Exiting an orthogonal composite state executes each exit activity of the active substates first and after that the composite state. The UML defined an additional concrete syntax called signal symbols to depict a transition oriented view. See [26] for more information.

As already mentioned, a transition can have an effect. This effect is normally an atomic behavior. A state can also be associated with functionality. It will be executed by the system if the state is active. Those activities are part of the internal activities of the state. They consist of a label and a trigger separated by a slash. According to [29] three labels are reserved by definition:

Entry is triggered as soon as the state is entered and thus executes before any other activity of the state. Entry activities should be atomic to avoid inconsistencies in the event processing.

Exit is triggered as soon as the state is left and thus executes as last activity before a transition out of the state occurs. Exit activities should be atomic to avoid inconsistencies in the event processing.

Do is an ongoing behavior "... performed as long as the modeled element is in the state or until the computation specified by the expression is completed."[29] If the do activity is completed, it will generate a completion event.

To simplify the state machines even further, [29] introduced so called pseudostates. Most of them are not crucial to a state machine, but they can illustrate various issues more explicitly.

Initial Pseudostate A solid filled circle to represent the source for a transition targeting the default entry. There can be only one initial pseudostate for each region.

History (shallow and deep) Similar to Harels history concept, a circled H with or without an asterisk is targeted by a transition. It resumes with the last state configuration of the composite state.

Entry point A circle usually at the border of a composite state as target for an entering transition to redirect it to a specific transition of a substate.

Exit point A crossed circle usually positioned at the border of a composite state to represent an exiting transition to redirect it to a specific outgoing transition. Exiting a region of a composite state by an exit point implies the exit of the composite state.

Fork and join A heavy bar to either split a transition for multiple regions or reunite multiple transitions from multiple regions of an orthogonal state.

Junction A solid filled circle that can act as a merger or splitter of transitions. The splitting is similar to Harels condition concept but the guards are evaluated statically, thus the transitions with a guard evaluating to false are disabled.

Choice pseudostate Is illustrated as a diamond and is similar to Harels condition concept. It realizes a dynamic conditional branch and splits the transition into multiple outgoing transitions with guard conditions. At least one guard condition should evaluate to true to avoid an ill-formed state machine.

Terminate node The cross depicts the immediate termination of the state machine and thus no exit activities are executed.

To execute a transition it needs to be enabled and at least one of its triggers must be satisfied by the associated event. "An event is the specification of some occurrence that may potentially trigger effects by an object."[29]

Hence, anything that happens is modeled as an `Event`. An event can either be external or internal. An external event is part of the interaction between objects. An internal event lives only inside one object. Events are further separated into asynchronous events and synchronous events. Synchronous events usually invoke an operation. The sender waits for the response of the receiver, thus the sender waits for the duration of the called operation. An asynchronous event is dispatched by the sender, and the sender continues its flow of control. Events can have parameters and attributes.

The event processing or event dispatching is defined by an event pool. A triggered event is added to the event pool and dispatched to the state machine. The dispatch method as well as the prioritization of the events is a semantic variation point of the UML. To keep the state machine in a well defined state, the UML defines that the event would be dispatched only if its predecessor was fully dispatched. If an event is dispatched and no transition is enabled, the event will be discarded and the next event will be dispatched. If more than one transition is enabled, only one will be fired based on the user defined prioritization. If multiple transitions of an orthogonal component are enabled, each transition will be fired, yet the order is undefined. States can define a list of events to postpone. Those deferred events will not be dispatched until they fire a transition, or the state machine enters a state where the events do not get deferred.

According to [29], the UML defines the following standard events which derive either from `Event` or `MessageEvent`:

SignalEvent An asynchronous event associated with a `Signal` that can contain additional information.

CallEvent A synchronous event associated with an `Operation` that will be called.

ChangeEvent An asynchronous event associated with a Boolean `Expression`. It represents a change in state or the satisfaction of a condition. A change event is generated explicitly, but the time of evaluation of the expression is undefined. A `ChangeEvent` is identified by 'when' followed by the expression.

TimeEvent An asynchronous event similar to Harels timeout concept. "A TimeEvent specifies a point in time. At the specified time, the event occurs."[29]

A `TimeEvent` can either be relative or absolute. It is associated with a `TimeExpression` which specifies the deadline. A relative `TimeEvent` is identified by 'after' followed by the `TimeExpression`. An absolute `TimeEvent` is identified by 'at' followed by the `TimeExpression`.

AnyReceiveEvent Is an asynchronous event triggered by the receipt of either a sent signal or a called operation that has no associated `SignalEvent` or `CallEvent` in the specific state. It is identified by the word 'all'.

The aforementioned completion event is yet another special case. It is dispatched immediately and thus prioritized before all other events in the event pool. The completion event has no parameters.

A `StateMachine` can be specialized by an extension mechanism similar to inheritance. It complies with the Liskov Substitution Principle. It is possible to add new states and transitions but not to delete states or transitions from the inherited state machine. Substates cannot change their superstate. Actions and activities can be added, removed and specialized. Transitions can be retargeted. Orthogonal components can be added to inherited states.

The `ProtocolStateMachine` is a specialized `StateMachine` used to specify the behavior of a protocol (e.g., SMTP, HTTP, and so on). It is not bound to an implementation, but it rather determines state changes and events of a protocol based communication. States represent a stable situation in the flow of the protocol, thus they have neither entry, exit, nor do activity and cannot contain history pseudostates.

Yet the `stateInvariant` of the state holds an additional condition for incoming and outgoing transitions. A `Transition` is further refined to a `ProtocolTransition`. A general `ProtocolTransition` is associated with an `Operation` of the owning `Classifier`, and it can have a pre- and a postcondition. Even though the `Operation` is derived from a call trigger, the UML defines a semantic variation point to specify other events on a `ProtocolTransition`.

Similar to the statecharts of Harel, contradictions can be introduced to a `StateMachine` by a reckless designer. UML calls those state machines ill-formed. Examples for ill-formed state machines consist of race conditions and overlapping guards. Some of those flaws could be found and thus indicated by the modeling tool (e.g., finding overlapping guards which is an NP-hard problem [2]). The work of Ines Nötzold [62] addresses the topic of how to detect and validate ill-formed state machines exemplarily.

Orthogonal components are an excellent representation for parallelism and concurrency in a state machine. However, they introduce a kind of multi-threading and

non-determinism into an event driven system. An alternative would be the application of the Active Object pattern. "The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control."[63] According to [63], it consists of:

- A proxy to access public methods.
- A method request triggered by a call of a proxy method. It contains the context information and a guard to determine when the request can be executed.
- A concrete method request for each method that requires synchronized access.
- An activation list where the proxy inserts the concrete method requests.
- A scheduler, running in a different thread than the proxy, executing the method requests.
- A servant defining the behavior of the active object. The method of the servant corresponds to the method of the proxy methods and thus is invoked when the associated method request object is processed by the scheduler.
- A future for the calling client to access the result of the proxy method call.

A model to model transformation from 1.8 on page 13 to refine the model, could transform each composite state with more than one region into a composite state with an active object for each region. A similar approach is depicted in [64] by using object composition instead of orthogonal regions. "Concurrency virtually always arises within objects by aggregation; that is, multiple states of the components can contribute to a single state of the composite object."[64]

The composite object needs to interact with its aggregates in a synchronous manner and dispatches the respective events. On the other hand, the aggregated parts need to communicate asynchronously with its container because they run in the same thread and would otherwise violate the run-to-completion semantics of the UML. This approach requires modeling interaction by a sophisticated designer and cannot be deduced by automated transformation. Another alternative to orthogonal components are the formal semantics of Harel to deduce a state machine similar to the algorithm of creating a product automaton [5].

Further approaches and efforts have been researched and performed to specify semantic variation points (see [65] for details), or to add formal semantics based on semantic profiles to extend the UML state machines (see [66] for details). Most of the model-driven approaches enrich the state machine with additional information and execute model refinement transformations, like flattening the state machine for further processing.

2.4 State machine implementations

A lot of software could benefit from a state machine approach because the context of the system is represented by the state instead of scattered variables. Yet most software systems intertwine the state machine with the concurrency model and an event-passing method. Extracting the state machine code decouples the system from the other software artifacts and increases its cohesion. A generic state machine should have at least an initialize operation, to provide a well-defined starting point, and a dispatch functionality, to dispatch events to the state machine. The dispatch functionality requires a uniform event representation which consists of the event signal and possible event parameters.

The nested switch implementation contains a dispatch operation. This operation consists of a switch statement based on the states of the state machine. Each state case statement consist of another switch to handle the specific events. The implementation is very simple, but it is not build to reuse and will result in a large monolithic operation if the state machine consists of many states and events.

The state table implementation renders the state machine as a table. A two dimensional table, consisting of the list of events and the list of states, contains a tuple of an action and the next state, that is the transition itself. Processing the events can be done generically because the specific details are captured in the table. The dispatch operation gets the actual state and the event to process. It determines the operations from the respective state table. Thus, it requires a vast number of operations, representing the actions, and relies heavily on pointer to functions. Yet the event processor can be reused and the event dispatching occurs in constant time. Especially embedded systems could swap the state table to a read only memory. The state table approach can either be implemented by inheritance or aggregation. Aggregation introduces another level of indirection, while inheritance requires the usual inheritance overhead. Alternatively, the state table can be one dimensional resulting in a more bloated table. But the table itself can be enriched with more information, like guard conditions, rendering the transitions more fine grained.

The state pattern from [12] is yet another implementation of a state machine. The approach is object oriented, and it uses polymorphism to dispatch events correctly. States are represented as classes containing their state specific behavior. The context class maintains a pointer to an abstract state class and the event handler function. This context class delegates the operation calls to the specific state class. Thus, the state pattern does not require a dispatch operation because of the event handler functions. Yet it is possible to introduce a weakly typed dispatch operation to provide a generic state handler operation. The dispatch operation of the concrete state classes contains a switch over the events, and it performs explicit downcasts to detect the signal. The approach is easily extendable, and it encapsulates the context class and the state classes. The abstract state class can be reused, but adding events results in changing its interface and thus requires to change all inheriting state classes.

The modal behavior chapter of [67] introduces objects for states, methods for states and collections for states. While objects for states refer to the state

pattern and methods for states refer to the state table, collections for states is an approach usually impractical for embedded systems. Collections are associated with specific states and contain the objects. The client can easily access all independent objects by the collection, and initiate a state change that removes the objects from one collection and adds them to another. Therefore, transitions are more or less connections of the different collections. Collections for states simplifies the access of multiple objects at the cost of a significant overhead.

In [64], Miro Samek combines the nested switch, the state table and the state pattern to provide a finite state machine as well as a hierarchical state machine implementation. It is a generic event processor named QEP[8]. It maps states directly to state handler operations. QEP defines a generic abstract state machine with a pointer to a state handler operation, which represents the current state, and its initialize and dispatch operation. The dispatch operation calls the state handler operation. The specific state handler operations of the state machine capture a state, and contain a switch over the respective events. Instead of the pointer to the state handler operation, it is possible to use a lookup table to detect the state which introduces an additional level of indirection. The hierarchical state machine implementation extends the approach by informing the event processor about a nested state. The state handler will inform the event processor about its super state if it cannot handle the event.

2.5 Event processing

A fundamental part of reactive systems and thus state machines are events to trigger state changes. Events could come from the systems environment, like user interactions or other systems, representing interaction. To distinguish between events from the environment and events from the system itself, they are called external and internal events.[9]

Messages should be transformed into events by an additional abstraction layer if the performance overhead is acceptable. [67] defines various patterns for messaging. The aforementioned abstraction layer should be implementation specific, but it depends on the requirements of the systems. Thus, it could be a broker, a publish-subscribe mechanism or a messaging middleware addressed in [67]. A lot of frameworks and CASE tools, like Rational Rhapsody, use a so called message bus that is one of the most scalable, but also one of the most complex solutions. The message bus is a messaging middleware that connects all participants virtually, and distributes messages in an asynchronous manner.

The processing of events and messages in an event-driven system is significantly different from sequential or multitasking systems. The processing is completely delegated to components outside of the system[10]. Usually, a part of the operating system or of the used framework implements this functionality and thus realizes an

[8] It is the event processor of Miro Sameks Quantum Platform.

[9] The author of this thesis prefers the word message for an external event and just event for an internal event.

[10] At least it is not directly in the hand of the developer of the system.

inversion of control[11]. The processing itself depends on the underlying execution model. The execution model is a set of policies to manage the central processing unit. According to [64], they are differentiated as follows:

Sequential system Is the simplest execution model. It consists of a main loop which calls functions and interrupt service routines handling interrupts. The model is straightforward to understand. The CPU utilization is poor because the loop needs to poll explicitly for inputs and thus the CPU is idle or waits most of the time.

Multitasking system Schedules which task or thread is executed in which order by the CPU. A so called kernel switches the CPU from one task to another by a process called context switching. Each task gets a memory stack to store and restore the CPU registers specific to the task. The context switches are activated by interrupts or by explicit calls to the kernel. The kernel provides mechanisms to block tasks, like semaphores, and thus is able to defer waiting tasks. That makes this execution model look more responsive than a sequential system even though it requires additional context switching overhead. Multitasking systems distinguish between preemptive and non-preemptive kernels to determine which task runs next.

A non-preemptive kernel gets control through explicit calls from the task. If an interrupt produces an event for a high-priority task, that is blocked waiting for an event, while a low priority task is running, the low running task will execute until it yields or it will make an explicit blocking call to the kernel. From that on, the kernel determines that the high-priority task is ready to run, and switches the context. This method is called cooperative multitasking which results in a non-deterministic way of execution. Yet the switching of a context will only perform in explicitly known calls to the kernel and thus makes it easier to share resources among tasks.

The preemptive kernel on the other hand, is called after the interrupt is executed and thus determines which task will be executed before the low-priority task yields or makes an explicit blocking call. The kernel switches the context to the high-priority task, and the interrupt returns to this task preempting the low-priority task. The high-priority task executes until it blocks via a call to the kernel. The kernel creates a fake interrupt to switch the context to the low-priority task. This approach guarantees deterministic task-level response, but it increases the complexity of sharing resources.

Event-driven system Consists of an event loop, an event dispatcher and an event queue. The running applications have event-handler functions. All occurring events are inserted in the event queue controlled by the event dispatcher. In the event loop, the event dispatcher pools the event queue, and it extracts events sequentially to call the associated event-handler functions. The event-handler function executes its code, and it returns to the event loop and thus realizes run-to-completion semantics. It is also called event-action paradigm because each event is mapped to the code that will be executed.

[11] Also known as Hollywood Principle.

Adding an abstraction layer between the execution model and the application, makes the software more portable. Therefore, [64] defines the active object computing model, depicted in figure 2.7. The core concept of those active objects applies multiple event-driven systems in a multitasking environment because each active object contains its own thread of control. Hence, an active object encapsulates an event loop, an event queue and a state machine. The event loop is simplified because the active objects extract the events directly from their event queue. The event loop calls the dispatch method of its active object to dispatch and process the event, like an event-handler function of the event-driven system. Because the dispatch method must complete before another event can be dispatched, run-to-completion semantics are provided implicitly. Active Objects do not distinguish between external and internal events because each event has to be queued into the specific event queue of the active object. The underlying state machine of the active object handles the dispatched event to complement the event-driven system, rendering the event dispatcher and the event-handler functions obsolete.

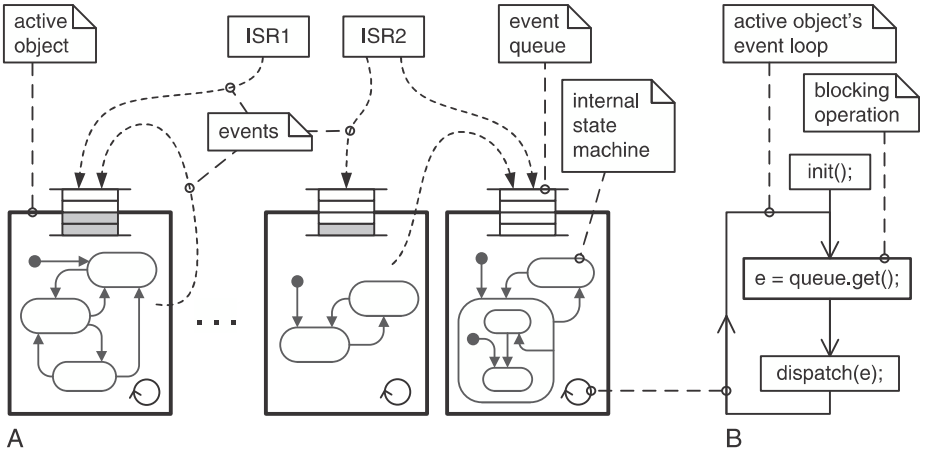

Figure 2.7: The active object computing model from [64]. (A) shows the Active object system and (B) the internal event loop.

The context of the active object is contained in itself, thus it integrates nicely into a system with context switching. Even the run-to-completion semantics would not be violated by a preemptive kernel because preempting a low-priority active object would only cause the active object to postpone its run-to-completion task. A simple non-preemptive kernel could use the event queues to determine which event will be dispatched next[12]. Each queue gets a priority, and the scheduler looks for the queue with the highest priority containing at least one event. Yet most of the device drivers and communication protocols require a more complex system because they use blocking calls. Because of the encapsulation of the active objects and thus the independence of the underlying execution model, they fit smoothly even into those environments[13].

[12] In [64] it is depicted as vanilla kernel.
[13] In fact, active objects could be based on protothreads (see [68] for details).

Delivering events and messages is distinguished between direct posting and one of the aforementioned messaging middleware systems. Direct event posting is a push-style communication because the sender 'pushes' the event to the recipient. Direct posting requires tight coupling because the sender needs to know the recipient and its specific events. It is rather inflexible because adding recipients requires to change the sender. On the contrary, direct event posting is straightforward, and does not require additional overhead.

However, the messaging middleware approach decouples sender and receiver. Common implementations use a publish-subscribe mechanism that realizes a pull-style communication. A recipient subscribes an event and therefore 'pulls' the information. The subscription can be done directly, similar to the Observer pattern, or via a mediator (see [12] for details). A mediator collects the events from registered publishers, and it distributes the events according to the subscriptions. Thus, the published events need to have the same semantics to all participants. It will be rendered as multicast event if more than one subscription exists for the same event. The approach is more flexible than the direct event posting because the participants can register and unregister themselves dynamically. The mediator is usually labeled as a software bus because it seems that the participants communicate by a bus. The mediator could represent either a classical hub and spoke architecture or could have intermediate mediators for scalability reasons, fault tolerance or high priority channels. The mediator itself could even be some kind of active object with its own thread and event queue.

As already mentioned, most of the events reside in a so called event queue. The event queue is a classical First-In-First-Out data structure to collect and distribute the events in the correct order. An alternative would be a priority queue that orders its elements based on their priority. To optimize the event management even further, it is possible to create a memory pool for events. This so called event pool is reserved for events only which renders all memory blocks of the pool equal. This results in fast access and allocation, no fragmentation and no overhead. Yet all events need to have the same structure and size. To support different event sizes it is necessary to create multiple event pools.

2.6 Related work

State machines correlate with event driven systems. Yet alternative approaches lead to similar results. [11] depicts that state machines do not require a graphical notation. Fowler uses state machines as example for a textual notation that is understandable as long as it is concerned with flat state machines and no syntactic sugar like orthogonal components.

[69] defines yet another textual DSL for state machines. The technique is low level oriented, and it maps state machines to event driven C code. It dissects the event handling in static and dynamic components. A special feature of this approach is the attribution of states to share information among actions.

ESTEREL[14] is a synchronous programming language, and has mathematical

[14] http://www.esterel.org/

semantics to describe reactive systems. It gets translated into finite state machines of low level languages like C.

According to [70], Argos is a synchronous language to create reactive systems similar to ESTEREL. Argos uses a graphical notation, like statecharts, and intuitive semantics.

SyncCharts[15] combine statecharts and Argos to a graphical formalism to model reactive systems. A syncChart can be translated to ESTEREL.

2.7 Synopsis

State machines, or state transition systems, have a long history, and are an established way of representing dynamic aspects of components. They are used by engineers because of the simple syntax consisting of states and transitions. The resulting statechart is used to communicate the behavior of a system to domain experts and developers. The underlying syntax (i.e., a graph or a table) can be processed by a computer because of its formal definition.

UML state diagrams define a concrete syntax for UML state machines to represent the dynamic behavior of a reactive object. They reuse Harels syntax which is based on Moore, Mealy and finite state machines. Yet UML specifies mostly informal semantics with semantic variation points. UML specifies a run-to-completion semantics, the entry and exit order, the generation of completion events and full event dispatching. [28] contains a state machine metamodel which defines different kinds of transitions, states and events. Semantic variation points are the evaluation of guards, the execution of multiple transitions, and the event dispatching and pooling. Either requirements dictate semantics and the realization or the developer has to determine them. The applied tool should not predefine any of these decisions. It should provide each possible kind of implementation and semantic description.

Implementing a state machine is based on the following techniques:

- The Nested switch which requires switch or nested if statements.
- The state table which requires function pointers.
- The state pattern which requires object orientation.
- The collections for states pattern which requires collections.
- A hybrid of those techniques.

Each state machine implementation should define a generic initialization and dispatch function to provide uniform access. The implementation directly depends on the event processing and thus on the execution model of the environment (i.e., sequential, multitasking, event-driven). The active object computing model from [64] can be applied as additional abstraction layer. The event dispatching

[15] http://www.i3s.unice.fr/~map/WEBSPORTS/SyncCharts/

can either be implemented by a push- or pull-style communication. The event queue is usually a FIFO or a priority queue. It could use one or more event pools for sophisticated event management. External events could be handled by a message dispatcher to transform them into internal events, and forward them to the respective event queue. Those features should be part of an extension language to configure the product. Because of the technical aspects, it should not be part of the respective UML model.

Semantics should be selectable within the configuration of the generation process. Requirements could dictate divergent semantics than the one specified by UML. The domain expert could request Harels formal semantics or other alternatives discussed in this chapter. Thus, the developer requires pragmatical semantics, provided by the respective language engineer.

The UML state machines syntax contains elements to simplify the diagram. The composition of states and most of the pseudostates can be transformed into elemental syntax elements by a model to model transformation. Regions could be transformed into active objects, or avoided by the application of aggregation. Thus, the core language could go without those syntactic sugar. On the other hand, if an adapter is available, it will be possible to support alternative source languages and event-driven systems.

Chapter 3

C++

3.1 Introduction

C++ is a widely used programming language. It is feared and worshiped because of its flexibility in combination with various language features. C++ evolved out of C to combine low level concepts, like pointers and preprocessor macros, with classes and objects. After that the language got enriched with features like constness, C++ templates, template libraries and different compiler implementations. The first standardization process occurred in 1998 by an ANSI/ISO committee. The latest standardization has been published as ISO/IEC 14882:2011 on August 12th, 2011. This process of language evolution leads to a vast number of supported features with disjoint targets.

> "Today's C++ is a multiparadigm programming language, one supporting a combination of procedural, object-oriented, functional, generic and metaprogramming features."[71]

Combining the features without understanding or knowing their characteristics will result in compiler or linker errors, if the programmer is lucky, or it will convey in undefined behavior and therefore result in a possible disaster. To prevent those outcomes, it is insufficient to assume that the programmer is fully aware of his doing. Tools like an integrated development environment and the compiler as well as the linker should be augmented by static code analyzation, optimization, code generation, validation of variants and coding guidelines. This part consists of language dissection and analyzation to deduce language aspects and assign those aspects to the proper field of application.

3.2 Basics

To derive language aspects from C++, it is essential to cover the basics. Because of the standardization of the language and differing semantics of concepts respective to other programming languages, this section contains terminology referenced in the next sections.

Translation unit "A translation unit is the source code giving rise to a single object file. It's basically a single source file, plus all of its `#include` files."[71] The translation unit is the input for a compiler to create object files.

Linking The linker uses the compiler created object files and binds them to an executable file, respectively a static or dynamic library. Because object files are created separately, it is possible to have multiple symbols of the same identifier. If identifiers are declared `extern`, it can result in undefined symbols. If the linker finds a doublet of an identifier, it will stop the binding process with an error. If the linker does not find a function, method or static element, it will also abort the binding process. Either the compiler consists of additional steps to clean object files or the linker consists of additional functionality to eradicate redundant elements (see [72] or [73] for details).

The linkage process can be differentiated between dynamic and static linking. Static linking binds all referenced elements into one executable file. Dynamic linking resolves the undefined symbols when the program is executed. The main advantage of dynamic linking is the reuse of often used libraries and thus only one software artifact per library which contains the information that will be referenced. Yet the main disadvantage is the renowned 'DLL-Hell'[1] which breaks executable files due to dependencies to a dynamic link library that got updated and replaced by an incompatible version. [74] explains the concept of dynamic code linking.

Preprocessor The preprocessor is a tool for text substitution. It runs before the compile process to substitute specific lines of code. Compiler can contain specific preprocessor directives, called `pragma` directives, to simplify programming or build tasks. If the compiler does not support the `pragma` directive, the compiler will ignore it. The concept of include guards for example is based on preprocessor directives to support the compile and development process. Microsofts Visual C++ Compiler supports the `#pragma once` directive which enables the same feature as include guards.

Declaration "A declaration tells compilers about the name and type of something, but it omits certain details." [71]

A function's declaration reveals its signature. The official C++ definition of 'signature' excludes the function's return type, but it is more convenient to consider it as part of the signature.

Definition "A definition provides compilers with the details a declaration omits." [71]

In C++ it is possible to divide the class definition into a header and an implementation file. Generating a modeled UML class should result in a header file of the class, which contains all member in the respective section of the access modifier, and an implementation file which references the header and contains the member function bodies.

[1] DLL-Hell is the Microsoft Windows specific dependency hell.

One Definition Rule Using an object or a function requires to define it first[2]. Yet there should only exist one definition to avoid ambiguity. A translation unit for example will not compile if it includes more than one definition for a class. One option is the application of forward declarations. Another option uses include guards and yet another solution could use compiler specific `pragma` directives.

Initialization "Initialization is the process of giving an object its first value." [71]

C++ provides different possibilities to initialize an object. First of all is the default constructor. It can be called without any arguments if not redefined by the programmer. The copy constructor will be called if given a reference to a different object of the same kind. The copy assignment operator will be called if the programmer intends to copy the value from one object to another of the same kind. The most important kind of initialization is using the member initialization list. It is more efficient because it does not call the default constructors of the member.[3]

Undefined behavior "For a variety of reasons, the behavior of some constructs in C++ is literally not defined: you can't reliably predict what will happen at runtime." [71]

E.g., dereferencing a null pointer or referring to an invalid array index. The behavior of the program will be unpredictable. An unprobable yet possible outcome could be the erasure of the hard drive for example.

Memory allocation With C++ the programmer has the possibility to put data structures on the stack or on the heap which is also called dynamic memory allocation. Stack objects will be deleted automatically when the scope is exited. Those variables are called local variables. With dynamic memory allocation the programmer extends the lifespan of an object beyond the scope. The disadvantage is that the programmer has to take care of the destruction of the heap object or it results in a memory leak.

Function objects Objects that act like functions because of overloading `operator()`. Function objects are also called functors in C++.

Smart pointer Is a pointer like object whose destructor calls the destructor of the object it is pointing to. The STL provides `auto_ptr`, `shared_ptr`, and `weak_ptr` for this issue, respective to the ownership. The object ownership is about the responsibility of the actual object (see item 13 of [71]).

RAII Is the abbreviation for Resource acquisition is initialization. It is common to use an object like a smart pointer to manage a resource. Normally the developer acquires a resource and initializes the managing object in the same statement (see item 13 of [75]).

Constness Despite other programming languages, the keyword `const` is remarkably versatile and thus valuable in C++. It is not only possible to create

[2] Referencing on the contrary requires only a declaration.

[3] The C++11 standard introduced move semantics and so called move constructors to avoid unnecessary memory allocations for object initialization.

constant primitives and objects, but also pointer and references, to specify semantic constraints. Thus, `const` allows the developer to communicate to other developers as well as to the compiler (see item 15 of [75]).

Member functions can be enriched with `const` to ensure immutable access. Yet the compiler will only check for bitwise constness, checking only for changes of the associated variables[4]. Developers on the other hand prefer logical constness that means internal changes are tenable as long as the client cannot detect it. Using the keyword `mutable` for data members of the class provides this functionality and satisfies the compiler.

It is also possible to cast away or add constness at runtime with cast operators. Requiring this option usually implies bad design and should be used judiciously. More information about constness can be found in item 3 of [71].

Friendship Functions and classes can be declared as friends of other classes, making them access protected, private and public member of that class. Friend functions or classes usually indicate a design flaw and should be avoided for the sake of encapsulation. Yet, especially type conversion by non-member functions require the friend declaration (see item 24 and 46 of [71]).

Inlining C++ provides the syntactic feature of inlining function calls. It is similar to a preprocessor macro because it substitutes function calls with the actual lines of code. Yet inlining is just a request to the compiler. The compiler will inline short functions like getter methods implicitly. The developer can add the keyword `inline` to the function signature to make an explicit inline request. As item 30 of [71] coins, inlining should be used judiciously and it depends highly on the build environment.

3.3 Advanced Concepts

To get the most out of C++ it is necessary to understand the consisting language features and how to combine them properly. Most of those concepts are eminent for effective C++ programming and require insight of high level and low level programming.

3.3.1 Dynamic polymorphism

Object oriented programming languages contain a language concept called polymorphism. Based on the inheritance it is possible to derive the type of the object at runtime. Because C++ additionally contains compile time or static polymorphism (see section 3.3.2 on page 54), this concept is called runtime or dynamic polymorphism. Dynamic polymorphism in C++ is based on virtual tables (vtbls) and virtual table pointer (vptrs) combined with inheritance. The virtual pointer of an object points to the virtual table of the class. Declaring one function virtual,

[4] Incorrect use of `const` in combination with a reference to internal data yields changeable data members (see item 28 of [71] for details).

induces the compiler to create a vtbl and a vptr for each object pointing to the vtbl. Using dynamic polymorphism for small objects like events or points wastes resources and thus should be used judiciously (see item 24 of [76]).

> "Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design."[75]

Using inheritance renders inlining for most compilers useless because creating the translation unit requires knowledge of the actual implementation. To deduce the object type at runtime, the compiler adds a `type_info` object to the respective vtbl and thus provides runtime type information (RTTI).

In contrast to object oriented languages like Java, C++ provides multiple inheritance with all its consequences. Multiple inheritance puts the developer in the position to use so called mix-in classes (see [77]) to easily extend the functionality of classes. Yet it leads to the renowned diamond problem. Deriving from two classes that both derive from the same class creates a diamond shaped inheritance graph and usually leads to unexpected behavior. Common inheritance would copy the member of the base class for each derived class. Using virtual inheritance avoids additional copies. Yet virtual inheritance defines odd initialization and assignment rules. If it is not really necessary, it should be avoided or at least should not contain any data (see item 40 of [71]).

Similar to the interfaces of Java and .NET are virtual base classes with pure virtual functions. Making a function pure virtual prohibits instantiation of classes not redeclaring the function. It is possible to provide a definition for a pure virtual function, but the call needs to be qualified.

An (impure) virtual function should not be public because it "inherently has two different and competing responsibilities, aimed at two different audiences."[75]

It is part of the public interface accessible for the rest of the world, and it is a customization point due to its virtual nature. The non-virtual interface (NVI) idiom targets exactly this problem and defines a non-virtual public function which delegates to a virtual non-public function. Similar to the template method defined in [12], it provides an internal hook and well defined before and after parts. Because C++ lets the developer redefine private virtual functions it is a common approach, except for derived virtual functions, to call their base class counterparts and virtual destructors. Yet another alternative to implement is, using the strategy pattern either classical (see [12]) or via function pointers or function objects (see item 35 of [71] for details).

If a class is designed as a polymorphic base class, it should define a virtual destructor (see item 8 of [71]). The C++11 standard provides a no derivation-prevention mechanism similar to Java's `final`, but most of the current compiler do not support the functionality. It is possible to inherit from a class with a non-virtual destructor. That could lead to object slicing and undefined behavior which usually results in a resource leak. Because the C++ compiler will always add a non-virtual public destructor to a class if it was not declared it explicitly[5], item 50 of

[5] The C++11 standard defines the delete keyword to prevent the compiler from creating default constructors or destructors.

[75] suggests to define public virtual destructors for base classes with polymorhpic deletion, or protected non-virtual destructors for base classes without polymorphic deletion like policy classes from section 3.3.2 at page 57. Furthermore, item 54 of [75] considers cloning to avoid slicing. Disabling the copy constructor and the copy assignment constructor prevents object slicing. An NVI based clone member function provides the functionality of deep copies.

Dynamic polymorphism requires type casts to provide down- as well as up-casting of types[6]. Type casts in C++ differ significantly from type casts of other object oriented languages like Java.

For low level operations, and usually unportable results, C++ provides `reinterpret_cast` (e.g., casting a pointer to an int). The `reinterpret_cast` operator does not perform any type checking and should be avoided (see item 92 of [75]).

The `static_cast` operator provides type conversion with compile time type checking. It is possible to perform explicit conversions and force implicit conversions. Yet a cast like `static_cast<Window>(*this).onResize()` behaves different from the equivalent Java cast. This cast calls `onResize` of a completely different object and should be replaced by a qualified function call like `Window::onResize()`. Because `static_cast` does not perform runtime type checking, it should be used judiciously on pointers of polymorphic objects because the result could be erroneous (see item 93 of [75]).

The `dynamic_cast` operator performs runtime type checking and thus should be preferred for such downcasts. It will yield a null pointer if the pointers are incompatible. It will yield a bad cast exception if it is performed on incompatible references.

Most `dynamic_cast` implementations are less efficient than `static_cast` implementations because of the additional runtime type check. The `dynamic_cast` requires RTTI. A possible implementation could use `strcmp` on the `type_id` to perform the type check and thus results in a significant performance loss (see item 27 of [71]). Yet item 93 of [75] provides a combination of `dynamic_cast` and `static_cast` because: "Using static_cast instead of dynamic_cast is like eliminating the stairs night-light, risking a broken leg to save 90 cents a year."[75]

3.3.2 Static polymorphism

Static polymorphism or compile time polymorphism is similar to preprocessor directives except that the compiler decides whether to expand the source code or not[7]. Static polymorphism is realized by the template mechanism. The template mechanism is itself Turing-complete and based on generic programming.

The main purpose of generic programming is to write reusable code relying on the syntax of the type. In contrast to object oriented programming, that is

[6] [72] defines an additional cast called side-cast that can only be applied in a multiple inheritance hierarchy.

[7] Preprocessor directives, or macros, define plain text substitutions without any syntax analysis. The substitution process will be executed before the template mechanism. Thus, it is possible to use macros within templates.

based on explicit interfaces centered on function signatures, templates are based on implicit interfaces. An implicit interface is based on a valid expression. Sutter and Alexandrescu name those valid expressions customization points (see item 65 of [75]). A customization point can be a member function, a typedef, a non-memberfunction or a (trait) specialization.

During compile time, the compiler tries to substitute the types by fitting the syntax. If the type does not have the appropriate syntax, the compile process will stop. Yet if there is no appropriate type but the template gets never called, the compiler will not try to fit the type and thus the source code is valid and will get compiled.

Generic programming

Czarnecki and Eisenecker outline generic programming as follows:

> "Generic programming is a subdiscipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. [...] Generic programming focuses on representing families of domain concepts."[54]

Thus, Generic programming is a different paradigm to represent software fragments compared to object oriented programming. Both approaches try to tackle the same problems, thus it is usually possible to represent a software fragment either object oriented or generic.

Generic programming in C++ is implemented as unbounded static polymorphism. I.e., the interfaces of the involved types are not predetermined and thus the approach is non-invasive, and "the binding of the interface is done at compile time"[73].

The consequences of applying generic programming compared to object oriented programming are (see [73]):

- Interface commonalities do not require base classes and thus collections are easily implemented.
- The generated code requires more memory, but it will be executed faster because of no additional indirections.
- The code base needs to be published to integrate it in other projects.
- Types implementing only parts of the implicit interface can be used if the called templates do not require the entire interface.

The standard template library (STL) of C++ is an application of generic programming. It consists of containers, algorithms, and iterators. The containers are completely detached from the algorithms. The iterators are used to combine both of them and thus the STL provides a multitude of combinations (see [78] and [79] for details).

Template mechanism

C++ templates have been introduced in the late 1980s. The first compiler that could really parse template definitions was developed in the mid-1990s by Taligent. It was bought by Hewlett-Packard and thus became the aC++ compiler[8]. C++ templates became part of the C++98 standard. A lot of missing features have been added to the C++11 standard.

By the time of writing this thesis, most of the common C++ compiler do not support the C++11 standard completely and thus relying on the new features results in possible portability issues. Yet the concepts represented by the C++ templates have not been changed significantly since its introduction.

C++ templates are divided into function templates and class templates. Function templates are functions with additional template parameters. A template parameter is either a built-in type (e.g., `int`, `char`, and so on) or a class. Using a function template requires the developer to fill in the concrete types, called template arguments, either explicit or implicit by deduction. Function templates can be overloaded to provide specific implementations for specific types, similar to type based if-else branches.

Class templates are classes with additional template parameters. A container class to manage elements of certain types is usually implemented as a class template. Template based member functions will only be instantiated if they are used within the application. Similar to function overloading, a class template can be specialized to provide optimized implementation for specific types. A specialization requires the specialization of all member functions of the class template. In addition to function templates, class templates can be partially specialized. I.e., it is possible to specialize only some of the template parameters[9].

Class templates can contain default template arguments to simplify the access. E.g., the container classes of the STL define an allocator template to refine the memory management of the container. In most of the cases the default is just fine, but if the developer needs to allocate specific objects on a special heap, he must overwrite the template argument with his own allocator.

Member functions, as well as nested classes, can have additional template parameters in respect of their surrounding class. Only virtual member functions and destructors can not have template parameters.

Template parameters do not have to be types. A template parameter can be a so called non-type template parameter. It is more like a value than a type (e.g., a fixed size for a container). A template parameter can also be a template parameter that is called template template parameter. E.g., injecting a generic container into a template.

Because templates differ significantly from ordinary code, the handling is less intuitive. Normally, declaration is separated from definition, yet templates require a specific inclusion and instantiation order to link them correctly. common

[8] http://www.hp.com/go/acc

[9] Function templates have to be totally specialized. The workaround is to provide an additional overloaded function template.

approaches are, either instantiatiang the templates explicitly or the so called inclusion model[10] (see [73]). The inclusion model either includes the definition in the declaration or puts the whole template in a header file. The developer includes the template file, and the linker will link each object file of the translation units. If the template includes additional header files, the size of the template will grow significantly and thus it increases the build time. Another linker problem that could occur, are multiple function definitions. Each translation unit will require the definition if it uses the template and thus violating the one-definition rule.

To tackle the build time problem, it is advisable to use precompiled headers. A precompiled header is a compiler specific option. It creates object files, consisting of non-changing software fragments, before the compilation process to reuse instead of recompile those fragments.[80]

The instantiation of C++ templates is based on the two-phase lookup because names cannot be resolved when templates are parsed. The first phase uses ordinary lookup rules and argument dependent lookup to look up non-dependent and unqualified names. The second phase occurs at the instantiation of the template, to look up the qualified and unqualified names, to complete the look up.

A detailed explanation of the template mechanism is contained in [73].

Traits and policies

C++ Templates are the bedrock to encapsulate characteristic features, and properties of software fragments, and bind them at compile time. Traits, trait templates or trait classes "represent natural additional properties of a template parameter." [73]

> "A traits template is a template class, possibly explicitly specialized, that provides a uniform symbolic interface over a coherent set of design choices that vary from one type to another."[81]

Basic traits are either fixed traits, value traits, or parametrized traits like in the accumulation example of [73]. More advanced traits are promotion traits to deduce the return value of a template based operator overloading[11]. "An important use of traits is as 'interface glue' — universal non-intrusive adapters. If various classes implement a given concept in slightly different ways, traits can fit those implementations to a common interface."[81]

A Part of the STL consists of traits like the `std::iterator_traits`. The C++11 standard, as well as the boost libraries[12], introduce even more traits to create more sophisticated C++ templates (see [77]). In item 47 of [71], Meyers defines the properties of traits as type information at compile time. It admits if-else tests at compile time by using function overloading. An application of traits is,

[10] A former approach called separation model was removed from the C++ standard because of the complexity for compiler manufacturers to implement it.

[11] The C++11 standard introduced the decltype and the auto keyword to simplify these circumstances.

[12] `http://boost.org`

using a master function or function template that calls trait specific (overloaded) worker functions or function templates.

Policies or policy classes are similar to traits but they are more behavior specific.

> "Policies represent configurable behavior for generic functions and types (often with some commonly used defaults)."[73]

They are passed as template parameters and need to be collected in classes or class templates. Thus, policies are orthogonal to each other and cannot be treated as standalone software fragments. The accumulation example could be parametrized by a policy to determine the accumulation algorithm (e.g., summation, multiplicity) (see [73]). A thorough introduction to policy based design can be found in [82]. Alexandrescu relates policies with a compile time based strategy pattern [12].

> "... multiple inheritance and templates foster complementary trade-offs. Multiple inheritance has scarce mechanics; templates have rich mechanics. Multiple inheritance loses type information, which abounds in templates. Specialization of templates does not scale, but multiple inheritance scales quite nicely. You can provide only one default for a template member function, but you can write an unbounded number of base classes. This analysis suggests that a combination of templates and multiple inheritance could engender a very flexible device, appropriate for creating libraries of design elements."[82]

Thus, policy based design is based on templates and multiple inheritance to get a synergy of both worlds. The result is a highly flexible design. It is based on combining individual policies within a host class in a typesafe manner.

The symbiosis of policies and traits are called policy traits that are unique properties of a template parameter and thus usually implemented as type or member function. E.g., the performance based policy trait to deduce copy, swap and move of a type from [73]. Alexandrescu defined in [82] typelists to combine policies efficiently with inheritance and avoid ambiguities. The C++11 standard introduced variadic templates to handle this issue more sophisticated.

The C++ standard specifies that an empty base class does not need to allocate memory as long as it does not cause memory allocation to an address of an object of the same type. Because `typedefs` do not require memory, and a lot of template parameters are substituted with empty classes, the compiler can optimize the respective class if inheritance is used instead of a data member for the template parameter (see item 39 of [71] for details about the empty base optimization).

Combining inheritance with templates to factor out specific implementations of member functions is also known as the "Curiously Recurring Template Pattern"(CRTP). A templatized base class specifies member functions in regard to the template parameter. Another class inherits from the base class, and it forwards itself as template argument and thus has to provide the functionality required by the base class.[83]

Template metaprogramming

Metaprogramming considers programs as data and thus generates or modifies source code of the target language. Metaprogramming is usually based on a meta object protocol (MOP) to manipulate the code in a consistent way (see [84] for details). The meta object protocol will either be evaluated at compile time, like in Lisp, or at runtime like in Groovy. Because C++ templates were not envisaged for metaprogramming, template metaprogramming does not directly consist of a meta object protocol[13].

Because templates are evaluated at compile time, template metaprogramming is also evaluated at compile time. One reason to use template metaprogramming in C++ is, to move computations from runtime to compile time, improving the runtime behavior at the cost of longer compile time[14].

Because metadata is immutable and metafunctions can not have side effects, template metaprogramming is called a pure functional language. Metadata is either types or non-types[15] used as template parameters. Especially traits provide type based information for the template as template parameter.

The parameters and nested types of the traits template are used as function parameters and return values of metafunctions. Template specialization provides a mechanism to alter the type of the traits template. Traits templates capture multiple nested types and thus metafunctions can have multiple return values. [85] defines a metafunction as either "a class template, all of whose parameters are types[,] or a class with a publicly accessible nested result type called type."[85]

C++ Frameworks and libraries, like boost[16] or Blitz++[17], provide domain specific languages based on template metaprogramming. Blitz++ is a library to perform mathematical calculations efficiently. It is based on so called expression templates[18] to delay the evaluation of the expression. Expression templates represent a compile time parse tree, thus the compiler can optimize the expression based on the so called return value optimization (RVO). RVO is a compiler specific optimization to avoid the creation of temporary objects. Using CRTP for the template arguments of the expression templates and operator overloading results in an embedded domain specific language within C++.

The metaprogramming library (MPL) of boost provides an API for metaprogramming that is also used within other boost libraries like Meta State Machine. [85] defines a state machine framework based on MPL. A templatized state machine class captures a state transition table, and concrete state machines derive from the state machine class using CRTP. The event processing is also templatized to invoke a function based event dispatching. Boost.Statechart, as well as Meta State Machine, is either based or at least inspired by this implementation (see section 3.4 on page 62 for further details).

[13] OpenC++ provides a meta object protocol for C++ to modify the target language at compile time.

[14] The additional code bloat is another disadvantage that is considered in item 44 of [71].

[15] Because of the restriction of the C++ standard, non-types are integer or bool constants.

[16] http://www.boost.org/

[17] http://sourceforge.net/projects/blitz/

[18] Move semantics and rvalue references introduced by the C++11 standard render expression templates obsolete.

3.3.3 Class design

Item 32 of [75] is named: "Be clear what kind of class you're writing."
Within the item Sutter and Alexandrescu distinguish classes in C++ as follows:

Value class Is intended to be used as a concrete class with public destructor and constructor but without virtual functions. An example would be `std::vector` that gets usually put on the stack or is a member of another class.

Base class Is part of a class hierarchy and provides an interface through virtual functions. The destructor should be public virtual and the copy constructor non-public (see item 50 and 53 of [75]). Concrete derived classes of base classes are usually dynamically created on the heap.

Trait class Carries information about types and consists of typedefs and static functions. Traits will not get instantiated, are unmodifiable and stateless. See section 3.3.2 on page 57.

Policy class Is a pluggable behavior which consists of a state. Policies are a powerful design principle to split classes into multiple fragments and thus are not used as standalone objects but as member or base. "...policy-based class design fosters assembling a class with complex behavior out of many little classes (called policies), each of which takes care of only one behavioral or structural aspect."[82]
See section 3.3.2 on page 57.

Exception class Should virtually derive from `std::exception`, and it has a public destructor and a no-fail constructor. Exceptions in C++ are thrown by value and catched by reference (see item 73 of [75] and item 13 of [76]).

Ancillary class "typically support specific idioms (e.g., RAII see Item 13)."[75]

If possible, non-member non-friend functions should be prefered (see item 44 of [75]). Yet sometimes objects and thus classes, like smart pointer, are required. "They should be easy to use correctly and hard to use incorrectly (e.g., see Item 53)."[75]

Item 19 of [71] recommends to "treat class design as type design". Because C++ provides operator and function overloading, adapting memory allocation and deallocation, as well as object initialization and deinitalization, a class can be designed to be seamlessly integrated into C++. The creation and destruction is not only specified by its constructor and destructor but also by operator new, operator delete, the new-handler, placement new and placement delete[19] (see chapter 8 of [71] for details).

C++ will create various constructors and operators silently if not explicitly declared otherwise (see [77] and item 5 and 6 of [71]). That could lead to unexpected behavior. E.g., C++ creates a non-virtual public destructor for a class that does not contain a destructor. If the class operates as base class, polymorphic deletion will lead to undefined behavior.

[19] As well as the respective new[] and delete[] operators for arrays.

The difference between object initialization and assignment has to be designed with care for the user of the class. Passing objects by value calls the copy constructor which results in additional memory allocation. As coined in section 3.3.2 on page 59, some compilers are able to avoid the construction of temporary objects by the application of RVO. If the copy constructors consists of side effects, this optimization can cause unexpected behavior[20].

> "Any class that overloads the function call operator (i.e., operator())
> is a functor class."[79]

Instances of functor classes can be applied by the user like functions, but they can have an additional state. Especially the STL algorithms make extensive use of instances of function classes called function objects. A function class returning a bool is called a predicate class. Predicate functions and predicate classes are used as parameters for conditional algorithms. More details about functors and functor classes can be found in chapter 6 of [79].

Enabling operators by overloading them is as fundamental as disabling them because the operators are the implicit semantics of the class. Enabling a copy constructor for a singleton, or disabling the streaming operator for strings, is unintuitive for the user. Yet operators like `&&` or `||` should not be overloaded because of their special semantics (see item 7 of [76] for details). Some operator functions, like the streaming operator, are non-member functions which require friend access to the class.

Next to the specific operators of a class are type conversions. Type conversion can either be defined explicit or implicit. Explicit type conversion prevents implicit type conversions and thus minimizes the chance of unintended type conversions. Yet an implicit type conversion is more convenient for the user (see item 15 of [71]).

Additional information for the user of the class are so called guarantees. All STL algorithms and container have to have a specific runtime behavior. It enables the user to choose the right algorithm or container, based on his requirements. C++ enables the class designer to specify additional information of possible exceptions that could occur when a function will be executed. Adding the keyword `throw()` to a function declaration, lets the compiler know that this function should not throw any exception. It is called the nothrow guarantee and promises to never throw an exception. if an exception occurs in the function, the 'unexpected function' will be called which will usually terminate the program (See item 29 of [71] for details about exception safety and guarantees. See [86] item 8 to item 17 and chapter 3 of [76] for details about exceptions, exception handling and exception safety).

To minimize compilation dependencies and thus minimize the recompilation effort[21] in C++, a class should only depend on declarations of other classes. De-

[20] It could even occur that the program will work as expected if it is compiled with debug information. Yet compiling the same program in release mode, could trigger the compiler to apply RVO.

[21] If a class depends on a class and that class depends on another class, it is called cascading compilation dependency. It leads to long build times, even though only minor changes in the implementation have been done.

pending on the declaration makes it possible to work with pointers of those classes and thus makes it possible to use forward declarations.

Implementing a class with respect to minimal compilation dependencies is either done with an interface class or a handle class. An interface class is a class declaration which consists of pure virtual function declarations and a factory function for clients.

A handle class is based on the pointer to implementation (pImpl) idiom. The pImpl idiom hides the implementation within an implementation class. The handle class consists of necessary forward declarations, function declarations and a pointer to the implementation class. The function definitions of the handler class delegate the call to the implementation class through the pointer. Thus, a level of indirection is added, but the declaration is separated from the implementation. [75] suggests to pImpl judiciously because of the additional complexity. The handle class is similar to the bridge pattern (see [12]), and it is also called compilation firewall or Cheshire Cat. More details can be found in [86] item 26 to item 30 and in [71] item 31.

3.4 Implementations of state machines in C++

Known from section 2.4 on page 42 are the basic implementations like nested switch, state table, state pattern and hybrids like Sameks QEP. Yet, because of their large field of application, state machines have been implemented by various developers. Those implementations are usually contained in libraries or frameworks.

Qt State Machine Framework

The Qt Project[22] contains an object oriented implementation of a state machine framework. It is based on State Chart XML (SCXML)[23]. The implementation integrates itself flawlessly into the Qt framework because it is tightly coupled with Qt's Meta-Object System. The Qt Meta-Object System is a MOP defined by Qt, adding additional language features to C++ by an additional compile step which is executed by the Meta-Object compiler. The use cases of Qt state machines are state-based (fluid) User Interfaces, asynchronous communication, the controller of the Model-View-Controller pattern and other high-level applications. It is not meant to be used for performance critical or low level applications because of the runtime overhead introduced by Qt. Qt state machines are easy to use, and they provide most of the functionality UML state machines define (see section 2.3 on page 35). The Qt state machine requires a running event loop to execute asynchronously, but it contains an internal event loop as well as an event queue (see [87] for details).

Listing 3.1 defines a state machine, wich communicates with the user interface, corresponding to the UML diagram A.1:

[22] http://www.qt-project.org
[23] http://www.w3.org/TR/scxml/

```
//defining application, button, etc. ...

QStateMachine machine;

QState *compound = new QState();
QState *state1= new QState(compound);
QState *state2= new QState(compound);
QFinalState *quitState = new QFinalState();

state1->addTransition(&button, SIGNAL(clicked()), state2);
state2->addTransition(&button, SIGNAL(clicked()), state1);

state1->assignProperty(&button, "text", "State 1");
state2->assignProperty(&button, "text", "State 2");

QObject::connect(state2, SIGNAL(entered()), &button, SLOT(showMaximized()));
QObject::connect(state2, SIGNAL(exited()), &button, SLOT(showMinimized()));

compound->setInitialState(state1);
compound->addTransition(&quitButton,SIGNAL(clicked()),quitState);

machine.addState(compound);
machine.addState(quitState);
machine.setInitialState(compound);
QObject::connect(&machine, SIGNAL(finished()), QApplication::instance(), SLOT(quit()));

machine.start();
```

Listing 3.1: An adapted example of [87].

Boost.Statechart

As mentioned in section 3.3.2 on page 59, boost consists of two implementations to create state machines. The first one is Boost.Statechart, formerly known as boost.fsm. It is a C++ library for finite state machines. It is possible to map most of the features of UML state machines, and its corresponding semantics, to state machines of Boost.Statecharts and vice versa[24].

Listing A.1 defines a simple Boost.Statechart state machine which corresponds to the UML diagram A.2. The code snippet 3.2 depicts the event definition. Snippet 3.3 shows the definition of a state as well as a state machine.

```
struct EvStartStop : sc::event< EvStartStop > {};
struct EvReset : sc::event< EvReset > {};
struct EvGetElapsedTime : sc::event< EvGetElapsedTime >{
  public:
    EvGetElapsedTime( double & time ) : time_( time ) {}
    void Assign( double time ) const {time_ = time;}
  private:
    double & time_;
};
```

Listing 3.2: The event definition of the refined `StopWatch`.

```
struct Active;
// StopWatch statemachine with active as initial State
struct StopWatch : sc::state_machine< StopWatch, Active > {};
struct Stopped;
// composite active state of stopwatch with Stopped is initial state
struct Active : sc::simple_state< Active, StopWatch, Stopped >{
  public:
    typedef sc::transition< EvReset, Active > reactions;
    Active() : elapsedTime_( 0.0 ) {}
    double & ElapsedTime() {return elapsedTime_;}
    double ElapsedTime() const {return elapsedTime_;}
  private:
    double elapsedTime_;
};
```

Listing 3.3: The definition of the state machine and the `Active` state.

The implementation makes use of template metaprogramming, CRTP, traits and policies. Due to this design, it is possible to create state machines on a highly

[24] see http://www.boost.org/doc/libs/1_54_0/libs/statechart/doc/uml_mapping.html

flexible basis which are build at compile time. It is even possible to distribute state machines on multiple translation units[25]. The default state machine applies an optimized RTTI. By using an RTTI-policy, it is possible to use the native C++ RTTI, or if necessary, alternative functions to get state type information provided by the library. Because boost.Statechart is statically configured, the compiler is able to optimize and validate the state machines at compile time. The double dispatch does not require a dynamic table and thus an event dispatch will only call a virtual function with an additional linear search for the transition[26].

The default state machine is not thread safe. Boost.Statechart provides also asynchronous state machines, using a scheduler policy that consists of a variable worker policy for thread safety. The library consists of an additional concept called state local storage which makes it possible to create state based context fragments. Usually the state machine itself contains the context which results in a monolithic class and thus in a hot spot for changes. Because most of the context fragments are only relevant in a specific state, and state local storage provides a mechanism to distribute the context elements to the states, the developer is able to unclutter the state machine by the application of state local storage. Snippet 3.4 shows the destructor of the `Running` state which accesses the `ElapsedTime` property of the compound `Active` state by state local storage.

```
~Running(){context< Active >().ElapsedTime() = ElapsedTime();}
```

Listing 3.4: The destructor of the `Running` state.

Despite UML's Run-To-Completion semantics, Boost.Statechart is able to handle exceptions at any time and thus supports unstable states and unstable state machines. Yet, "[the libraries] exception event processing rules ensure that a state machine is never unstable when it returns to the client"[88]. A specific propagation mechanism will delegate the `exception_thrown` event to the state that caused the exception if the state machine is stable, or to the outermost unstable state of the unstable state machine. Similar to the STL container, each state machine has an allocator trait that can be customized for hard real time requirements. [88]

Meta State Machine

The Meta State Machine (MSM) of Christophe Henry is the other state machine library provided by Boost. State machines defined by Meta State Machine are based on template metaprogramming similar to Boost.Statechart. Compared to Boost.Statechart, the Meta State Machine library is far superior at runtime, but it requires more effort at compile time.

The state machines are divided into frontend and backend. The frontend is a DSL to represent the state transition table of the state machine. The backend represents the policy based engine. Because of the policy based design, it is highly

[25] Depending on the compiler.
[26] Boost.Statechart defines so called reactions which imply transitions.

flexible. The default backend is optimized for runtime speed, thus increasing the compile time. Especially larger state machines will not compile by default because of the maximum template recursion depth, as well as MPL and Boost.Fusion limitations.

At the moment Meta State Machine provides three frontends to define state machines. The basic frontend is inherited from [85]. The transition table of the player example from [85] is contained in listing A.2 that is the source of the following code snippets (see figure A.3 for the respective UML diagram). The snippet 3.5 shows the declaration of the `Empty` state:

```
    a_row < Empty , open_close  , Open     , &p::open_drawer >,
      row < Empty , cd_detected , Stopped  , &p::store_cd_info ,&p::good_disk_format >,
      row < Empty , cd_detected , Playing  , &p::store_cd_info ,&p::auto_start >,
```

Listing 3.5: The state table entry of the `Empty` state for the basic frontend.

```
void store_cd_info(cd_detected const&) { cout << "player::store_cd_info\n"; }
```

Listing 3.6: The definition of `store_cd_info` transition for the basic frontend.

```
bool good_disk_format(cd_detected const& evt){
 if (evt.disc_type != DISK_CD){cout << "wrong disk" << endl;return false;}
 return true;
}
```

Listing 3.7: The definition of `good_disk_format` guard for the basic frontend.

Ommitting elements from a row requires the developer to use a specific row, like `a_row` in the snippet 3.5. Transitions are `void` member functions like in snippet 3.6. Guards are `bool` member functions like in snippet 3.7. The definition of events like in snippet 3.8, the definition of states like in snippet 3.9, and the definition of the backend like in snippet 3.10 can be used in the basic frontend as well as the functor frontend. Meta State Machine allows the developer to mix the frontends within a state machine.

```
struct open_close {};
enum DiskTypeEnum{DISK_CD=0,DISK_DVD=1};
struct cd_detected{
  cd_detected(string name, DiskTypeEnum diskType)
    : name(name),disc_type(diskType){}
  string name;
  DiskTypeEnum disc_type;
};
```

Listing 3.8: The MSM event definition of `open_close` and `cd_detected` for the basic and the functor frontend.

```
// front-end: define the FSM structure
struct player_ : public msm::front::state_machine_def<player_>{
 struct Empty : public msm::front::state<>{
  template <class Event,class FSM> void on_entry(Event const&,FSM&){cout << "entering: Empty" <<
       endl;}
  template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Empty" <<
       endl;}
 };
```

Listing 3.9: The MSM definition of the state machine and the `Emtpy` state for the basic and the functor frontend.

```
   typedef msm::back::state_machine<player_> player;
```

Listing 3.10: The registration of the backend.

The functor frontend is more user friendly than the basic frontend. At the moment, the functor frontend is the preferred frontend. The same player example as before is implemented in listing A.3 with the functor frontend. The following code snippets are from listing A.3. The snippet 3.11 shows the state table entry of the `Empty` state. As depicted in the previous code snippet, instead of member functions, the frontend uses functors. The Snippets 3.12 and 3.13 show the transition and the guard as functor. The functor syntax can also be applied to states, like in snippet 3.14.

```
  Row < Empty , open_close  , Open    , open_drawer   , none >,
  Row < Empty , cd_detected , Stopped , store_cd_info , good_disk_format >,
  Row < Empty , cd_detected , Playing , store_cd_info , auto_start >,
```

Listing 3.11: The state table entry of the `Empty` state for the functor frontend.

```
struct store_cd_info{template <class EVT,class FSM,class SourceState,class TargetState> void
    operator()(EVT const&,FSM& fsm ,SourceState& ,TargetState& ){cout << "player::store_cd_info" <<
    endl;}};
```

Listing 3.12: The definition of `store_cd_info` transition for the functor frontend.

```
struct good_disk_format{
 template <class EVT,class FSM,class SourceState,class TargetState>
 bool operator()(EVT const& evt ,FSM&,SourceState& ,TargetState& ){
  if (evt.disc_type != DISK_CD){cout << "wrong disk" << endl;return false;}
  return true;
 }
};
```

Listing 3.13: The definition of `good_disk_format` guard for the functor frontend.

```
struct Empty_Entry{template <class Evt,class Fsm,class State> void operator()(Evt const& ,Fsm&
    ,State& ){cout << "entering: Empty" << endl;}};
struct Empty_Exit{template <class Evt,class Fsm,class State> void operator()(Evt const& ,Fsm&
    ,State& ){cout << "leaving: Empty" << endl;}};
struct Empty : public msm::front::euml::func_state<Empty_Entry,Empty_Exit>{};
```

Listing 3.14: The functor definition of the `Emtpy` state.

The experimental eUML frontend is the newest Meta State Machine frontend. Its syntax is more UML like and contains less syntactic noise. Yet another implementation of the player example implemented with the eUML frontend is depicted in listing A.4. The following snippets are extracted from this listing. For each definition exists a macro that is usually expanded to the functor frontend.

The former event definition can be rewritten like in snippet 3.15. Guards and transitions are defined as actions, depicted in snippet 3.16 and 3.17. The state definition is simplified to the state macro, depicted in snippet 3.18. The exit and entry behavior can be defined as action as well. Using the macro syntax, requires the developer to explicitly declare the state machine as depicted in listing 3.19. The state transition table looks more user friendly, like in snippet 3.20. It could also be written with the alternative notation, beginning with the target state.

```
BOOST_MSM_EUML_EVENT(open_close)
enum DiskTypeEnum{DISK_CD=0,DISK_DVD=1};
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(string,cd_name)
BOOST_MSM_EUML_DECLARE_ATTRIBUTE(DiskTypeEnum,cd_type)
BOOST_MSM_EUML_ATTRIBUTES((attributes_ << cd_name << cd_type ), cd_detected_attributes)
BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES(cd_detected,cd_detected_attributes)
```

Listing 3.15: The event definition of `open_close` and `cd_detected` using the eUML macro syntax.

```
BOOST_MSM_EUML_ACTION(good_disk_format){
  template <class FSM, class EVT, class SourceState, class TargetState>
  bool operator()(EVT const& evt,FSM&,SourceState& ,TargetState& ){if
       (evt.get_attribute(cd_type)!=DISK_CD){cout << "wrong disk, sorry" << endl;return false;}
    return true;
  }
};
```

Listing 3.16: The definition of `good_disk_format` guard using the eUML macro syntax.

```
BOOST_MSM_EUML_ACTION(store_cd_info){template <class EVT, class FSM, class SourceState, class
     TargetState> void operator()(EVT const&,FSM& fsm ,SourceState& ,TargetState& ){cout <<
     "player::store_cd_info" << endl;}};
```

Listing 3.17: The definition of `store_cd_info` transition using the eUML macro syntax.

```
BOOST_MSM_EUML_STATE(( Empty_Entry,Empty_Exit ),Empty)
```

Listing 3.18: The definition of the `Empty` state with the respective eUML macro.

```
    Empty + open_close                    / open_drawer   == Open ,
    Empty + cd_detected [good_disk_format] / store_cd_info == Stopped ,
    Empty + cd_detected [auto_start]       / store_cd_info == Playing ,
```

Listing 3.19: The state transition table entry of the `Emtpy` state using the eUML frontend.

```
BOOST_MSM_EUML_DECLARE_STATE_MACHINE(( transition_table, //STT
                  init_ << Empty, // Init State
                  no_action, // Entry
                  no_action, // Exit
                  attributes_ << no_attributes_, // Attributes
                  configure_ << no_configure_, // configuration
                  Log_No_Transition // no_transition handler
                  ), player_) //fsm name
```

Listing 3.20: The definition of the player state machine using the eUML macro syntax.

Because of the separation between frontend and backend, it is possible to define user specific DSLs to represent state machines in C++. It is also possible to provide different semantics by the backend. Because Meta State Machine calculates most of the state machine aspects at compile time, the compiler is able to optimize and validate[27] the state machine. The compile time optimization results in a highly efficient state machine. The compile time validation is able to check that orthogonal regions are really orthogonal and that each state is reachable.

The double dispatch is executed at constant-time. The run-to-completion semantics depend on the configuration of the backend (like enabled exception handling, or message queuing) and the structure of the state machine. Because each state machine is represented as state transition table, which uses the row template, it is possible to substitute states with sub state machines and thus enable the developer to create a highly scalable state machine.

Meta State Machine supports the complete UML state machine syntax and full UML state machine semantics. Each concept can be directly mapped to Meta State Machine source code. The default behavior corresponds to the defined UML behavior. Additionally, Meta State Machine provides interrupt states for

[27] The validation is done by the compile time graph library mpl_graph.

sophisticated error handling, and Kleene events to simplify the transitions of a state machine if any event should trigger the transition.

Yet another useful feature are flags to simplify state conditions. Flags enable the user to define specific conditions which apply for specific states. Instead of querying the state machine for its states, the user is able to query the flags (see [89] for details).

3.5 Synopsis

C++ is called a multiparadigm programming language because it consists of various language features and concepts. Generating C++ supports the developer to combine the features correctly. It can also prevent vicious combinations. Mapping UML classes to C++ classes seems to be easy, but C++ consists of more than one representation for a class (e.g., base, value, policy, trait, exception, anxcillary class). While dynamic polymorphism can be modeled in UML pretty straighforward, static polymorphism and meta programming requires either a profile or an external model extension. Other options like constness, friendship, inlining, guarantees and memory alllocation should not be part of the model because they are details of a lower abstraction layer. The same applies for compiler specific options (e.g., applying RTTI, Empty Base Optimization, RVO, pre-compiled header) and corresponding implementation patterns (e.g., NVI, pImpl, CRTP). Modeling a class that is based on policy based design, or references traits, requires a sophisticated parametrization approach. Thus, it is only resasonable to provide at least one language extension for C++.

While the Qt state machine framework can be generated straigthforward, the boost state machine frameworks need to be enriched with more information. Boost.statecharts is based on template metaprogramming and uses policy based design and CRTP to provide a flexible and configurable implementation. Meta State machine differentiate between frontend and backend. The generator should let the developer choose the frontend DSL and the backend configuration. The semantics should also be selectable because of possible restrictions determined by the specifications, requirements or the system environment. An additional language extension for the configuration of implementation specific state machines will enable the developer to do this in a more sophisticated manner than UML provides at the moment.

Chapter 4

Implementation

4.1 Mapping problem

As depicted in the previous chapters, C++ provides multiple state machine implementations and thus multiple target representations. UML state machines provide just one source representation[1] with semantic variation points. A solution would be to specify the requirements and pick the most sophisticated target representation. This is done in [90]. That solution does not scale well and the language users could have divergent requirements for their products. The alternative would be to enrich the model with additional information. Outlined in the first chapter, enriching a model with ancillary information is usually done with an additional language or by adapting the source language. The extension language needs to be tailored to the domain, and it requires to reference UML model elements (see section 1.4 on page 2 and section 1.5 on page 3).

The classical MDA approach proposes the profiling mechanism (see [26]). This solution is feasible, yet the model will contain technical details not relevant for its purpose. Adding details to a model lowers the abstraction level and thus the expressiveness of the model. The alternative of adapting the metamodel is more intrusive and less portable. The preferable approach should render domain specific views of the model which represent only relevant information. It should weave all the information together to create a concrete representation of the target language. Language oriented programming as well as language modularization, introduced in section 1.11 on page 19, represent such an approach.

4.2 Solution approach

Based on the principles of language oriented programming and language composition, one core language provides a well defined interface for other languages, so they can reference the core language elements. Using a language workbench,

[1] It is possible to model the same state machine with different concepts, but UML provides only one concrete syntax.

like MPS or Intentional Domain Workbench, demands to implement the concrete syntax of UML within the language workbench from the language engineer. But most of the domain experts, and system architects, use a UML editor to model the system. A UML editor is usually a projectional editor. If the language workbench is also a projectional editor, it will require an import of the UML model. Because of incompatibility issues, this import is not provided out of the box, and it is usually not wanted by the manufacturer of the language workbench.

Yet the Eclipse Modeling Framework[2] (EMF) is a full featured modeling framework, which provides an XMI interface, to read and write models [91]. UML2 of the Model Development Tools[3] provides an EMF based UML implementation. If the UML editor has an EMF UML serialization, it will be possible to work with the exported model in an EMF environment. Within the Eclipse ecosystem, those models can be referenced and adapted by Eclipse based editors. Xtext, for example, is a framework to develop languages. It uses one or more context free grammars, similar to the extended backus naur form, to create EMF components. Because each context free grammar can be transformed into a metamodel, Xtext provides a transformation to transform the grammar into a metamodel (see section 1.6.1 on page 5). Therefore, those languages integrate themselves smoothly into the Eclipse ecosystem. Within those languages, it is possible to reference other languages by their metamodels. Thus, it is possible to create a non-invasive language extension for a UML model based on the abstract syntax model. Figure 4.1 depicts the solution approach explained in the following sections.

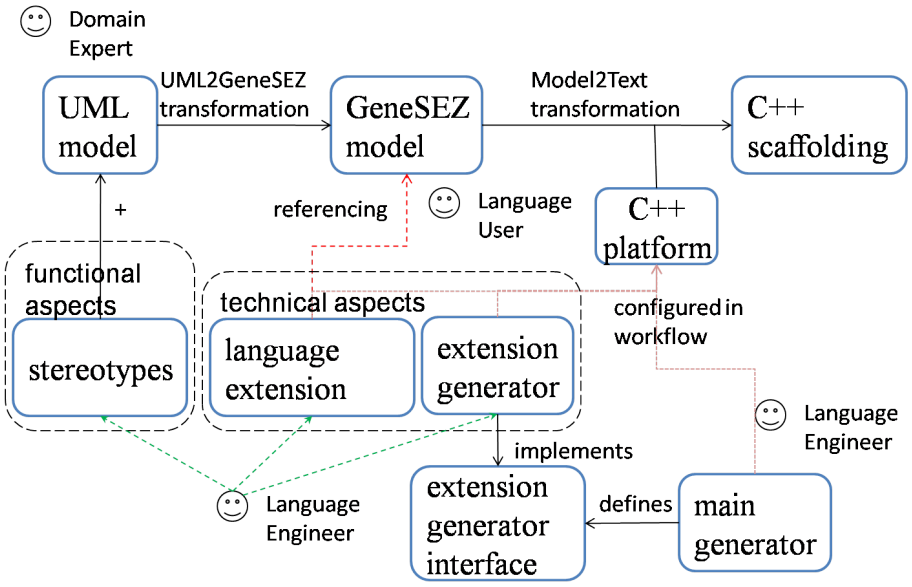

Figure 4.1: A draft of the scalable, yet flexible, solution approach to generate C++ out of UML.

[2] http://www.eclipse.org/modeling/emf/
[3] http://www.eclipse.org/modeling/mdt/

4.2.1 Model transformation

The UML metamodel based on its superstructure [29] and infrastructure [28] is very complex. This abstract syntax does not suit the characteristics of a language, which have been discussed in section 1.5 on page 3, because the intention of UML is general purpose modeling. Hence, UML is not tailored to a specific domain. It violates simplicity, minimalism, introduces accidental complexity and thus impedes the language engineer to create proper language transformations.

The GeneSEZ[4] framework provides a dense metamodel which consists of transformation relevant language elements. It simplifies and minimizes the language interface and thus simplifies the generation process, as well as the language composition and extension known from section 1.9 on page 17. A supplied model to model transformation of the GeneSEZ framework transforms a serialized UML model into an instance of the `gcore` metamodel. Using the abstract syntax of the GeneSEZ metamodel as core language, and referencing it within Xtext grammars, results in a scalable extension mechanism.

It is also possible to use a different source language as long as a transformation to a GeneSEZ compliant model exists[5]. Figure 4.2 shows the state machine part of the `gcore` metamodel. Compared to figure 2.6 on page 36, which shows the state machine part of the UML metamodel, the `gcore` metamodel is clearer and contains more information (e.g., events and transitions), models states slightly different and spares regions.

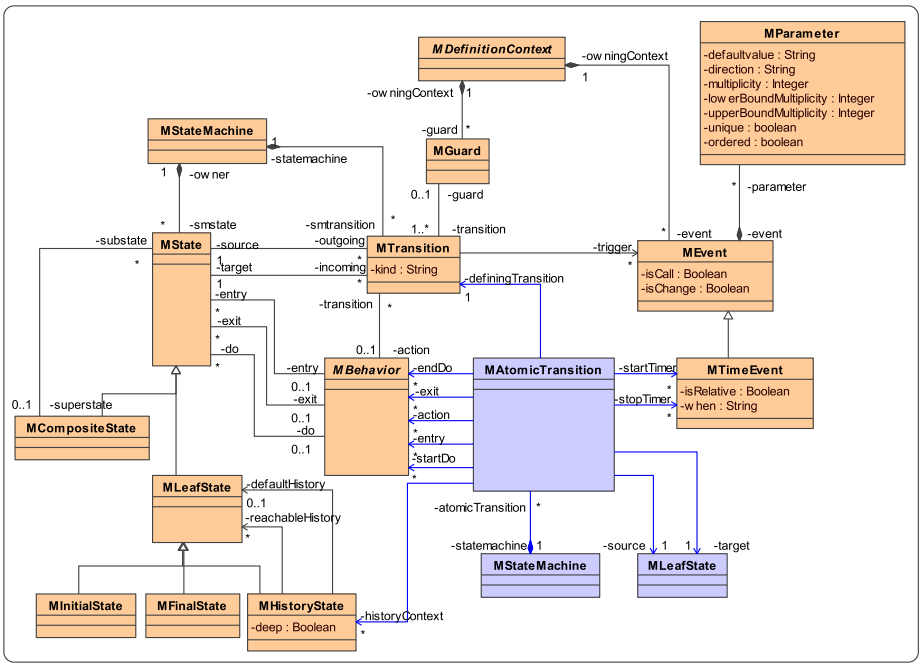

Figure 4.2: The state machine part of the gcore metamodel.

[4] http://genesez.org/

[5] The GeneSEZ term for such a transformation is called adapter.

4.2.2 Language extension

Each language extension depends directly on the applied technology. An Xtext based extension requires a grammar file to derive a metamodel. This process is adaptable by exchanging, and extending, the workflow components of the workflow to create the software artifacts. An alternative would be EMFText[6] that derives its artifacts from an existing metamodel. Yet another technology is Epsilon[7]. It introduces an additional abstraction layer which consists of the Epsilon Model Connectivity and the Epsilon Object Language.

For the sake of simplicity the language extensions are based on Xtext. The abstract syntax can be represented by a grammar file. Other models can be referenced. The editor can be customized. There is almost no configuration overhead.

Because the audience will be developers, a text based editor within the Eclipse IDE is sufficient to express the will of the user efficient and effective. This approach requires EMF and thus excludes more sophisticated language workbenches (e.g., Meta Programming System, Intentional Domain Workbench) because of the incompatibility of the abstract syntax. If the language workbench is based on EMF, or provides an EMF interface, it should be possible to integrate it and its extensions within the multistage pipeline.

Each extension consists of an Xtext project and its respective user interface project for the editor. Because a workflow is provided to configure the extension, each aspect of the editor and the language generation is adaptable. `Gcore` needs to be registered within the workflow. It needs to be imported into the grammar to reference the `gcore` language elements. Because the GeneSEZ metamodel is already instantiated, it is not possible to derive from the metamodel[8]. Hence, additional rules need to encapsulate the respective core elements. Because this is similar to the language annotation approach, discussed in section 1.11.1 on page 21, this approach will only provide positive variability (see section 1.12 on page 27, [54] and [56] for details).

The language extension itself can be specialized for target language specific aspects. E.g., the state machine extension can be specialized for C++ to provide additional mappings for boost.Statechart, Meta State Machine or the Qt State Machine Framework. This enables a more granular configuration for state machines that could be mapped to implementation specific details and policy configurations.

Then again, an abstract language extension can be reused for other platforms to create a product line or even a software factory (see section 1.11.2 on page 22). If intended, even the core language could be exchanged, because each extension references only specific elements[9] of the core language.

The result is an editor within Eclipse to create one or more extension files. The editor itself is text based, and the extension requires access to the serial-

[6] http://www.emftext.org/index.php/EMFText

[7] http://www.eclipse.org/epsilon/

[8] In this case, EMF defines inheritance as bidirectional. Xtext enables inheritance from a grammar because the metamodel will yet be generated.

[9] In this case, those elements are depicted in figure 4.2.

ized model. Compared to a UML profile, such an extension should be used for technical domain aspects. Especially functional aspects should be represented by stereotypes within the model because this information is relevant for the domain expert. Stereotypes can alter and control the `UML2GeneSEZ` transformation step. Because the language extensions reference an already transformed model, they cannot alter this transformation. Hence, only following transformations can be adapted. The obvious disadvantage is the required presence of a `gcore` instance model.

The resulting DSLs are external, horizontal and target the application developer. While creating the UML model follows the modular decomposition approach, each language extension should follow the aspectual decomposition[10]. I.e., a class is modeled modular. If the class uses policy based design, each policy, like threading or logging, will be configured by one or more extensions and therefore will be based on aspectual decomposition. Another aspect of those language extensions are the semantic variation points of the UML (e.g., the event processing of UML state machines). Those variation points are independent from the model but crucial for the resulting software. The language user should be able to choose the appropriate semantics in a pragmatic manner. The semantics should be easy to use and understand and thus ideally be pragmatic semantics (see page 9).

4.2.3 Generation

Subsequent transformations could either be model to model or model to text transformations. Because Xtext provides the transformation language Xtend2, it is only reasonable to use it for the following generation process. Because of its template syntax inherited from Xpand[11], its Java like syntax[12] and the direct Guice[13] support, the opportunity arises to use this language for the model to text transformation[14]. Xtend2 is able to manipulate the model directly in a uni-directional manner. Thus, it requires the language engineer to write each transformation rule by himself. Tracing needs to be implemented by the language engineer, too. Because Xtend2 is interpreted, it simplifies the debugging of the transformation process at runtime.

For this prototypical implementation a model to text transformation based on Xtend2 is sufficient. If required, it could be replaced, or complemented, seamlessly by an alternative transformation approach (e.g., graph transformation, operational transformation, relational transformation, and so on) and its respective EMF based implementation (e.g., Epsilon Transformation Language, ATL Transformation Language, QVT Declarative, and so on)[15]. The main advantage of alternative transformation approaches are, more sophisticated model to model transformations to weave the information into the model, and thus create an in-

[10] Yet specializing a language extension follows the modular decomposition approach.
[11] http://www.eclipse.org/modeling/m2t/?project=xpand
[12] Xtend2 compiles completely to Java and can be used interchangeably.
[13] https://code.google.com/p/google-guice/
[14] It could also be used for model to model transformations.
[15] see the model transformations of http://www.eclipse.org/modeling/

termediate model to simplify the model to text transformations. Or more sophisticated model to text transformations provided by the respective implementation.

The model to text transformation, based on Xtend2, is separated into the main generator (see figure 4.3) and the language extension generators (see figure 4.4 and figure 4.5). The main generator project contains a `CppGeneratorComponent` to configure the generator setup as well as the language extensions. To create a specific `Injector` by Guice, the respective `RuntimeModule` is injected into the main generator setup. The `RuntimeModule` binds the specific `Generators` and the protected region resolver.

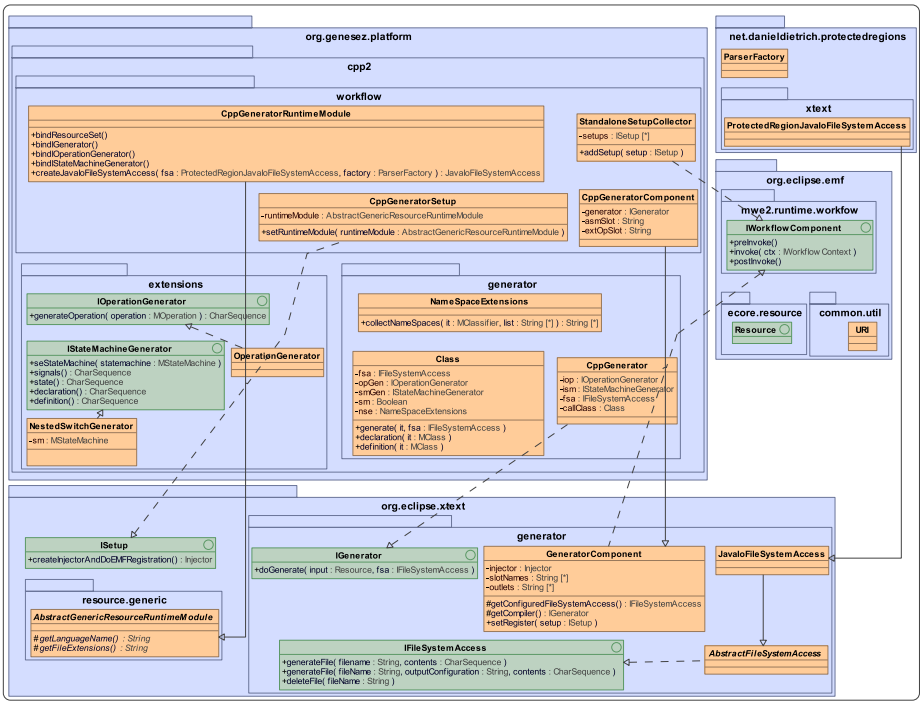

Figure 4.3: A structural overview of the C++ platform project.

A protected region resolver makes so called protected regions available to provide an alternative to subclassing. A protected region is a tagged code segment for user specific implementation details. The Xtext Protected Region Plug-in from Daniel Dietrich[16] extends the Xtext framework by protected regions. It overwrites the default behavior of the `JavaIoFileSystemAccess`, and it provides a `ParserFactory` to register language specific parsers to create protected regions based on the target language.

The main generator overrides the `doGenerate` method of the `IGenerator` interface and contains a reference to each extension generator. The main generator project provides an interface for each extension generator as well as a default implementation. It injects each extension generator and protected region resolver

[16] https://github.com/danieldietrich/xtext-protected-regions

into the main template class by setter injection. Subsequently, the generator calls the templates `generate` method.

The template class creates the resulting character stream and delegates parts of the creation to the extension generators. Because the main generator provides the extension generator interface, the language engineer of the extension can provide different generators, thus different semantics for the language user, if required.

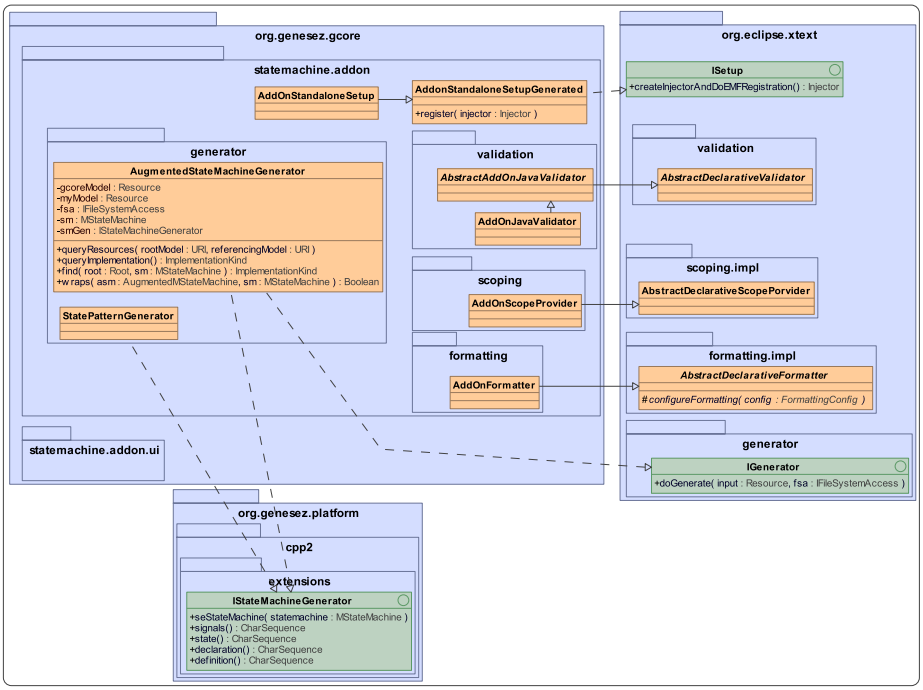

Figure 4.4: A structural overview of the state machine extension platform. This extension is not C++ specific, but can be specialized for C++ specific implementations.

Each extension generator implements the `IGenerator` as well as its extension generator interface. Because of a flaw in the workflow engine, the `doGenerate` method of the `IGenerator` interface is misused to inject the model resource[17] at the invocation of the workflow component.

Because EMF uses URIs to identify objects, it is necessary to query the resources by the model element to get the correct `gcore` model and the correct extension model. Otherwise, EMF is not able to resolve the model element correctly, and it is not able to find the corresponding model element in the `gcore` model. With the model element and the correct model instances, it is possible to check for eventual extensions and alter the generation process respective to the extension generator. Figure 4.3 shows the structure of the C++ platform project. Figure 4.4 and 4.5 show the structure of the extension projects. The class diagrams omit specific details for brevity.

[17] The same applies for the protected region resolver.

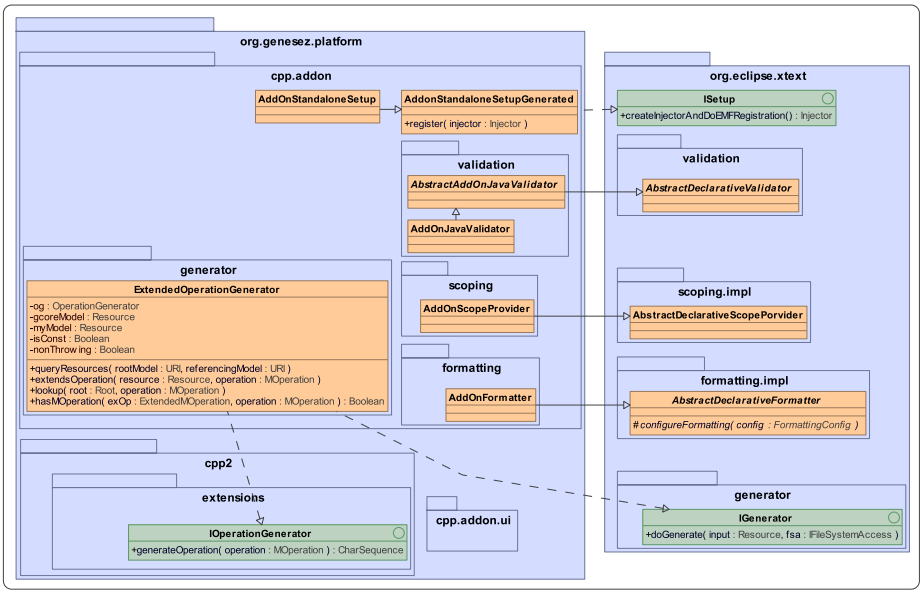

Figure 4.5: A structural overview of the C++ extension platform. At the moment only `MOperations` are enriched with additional information.

4.2.4 Configuration

It is common to use a workflow or configuration language to configure the transformation process. It could be done with any general purpose language, like Java, but this approach is usually not very intuitive for the language user. The used workflow language is MWE2 which is also part of the Xtext framework. An MWE2 workflow is controlled by components which are plain Java classes. These components implement at least the `IWorfklowComponent` with its `preInvoke`, `invoke` and `postInvoke` methods, to provide a sequential flow of execution.

If the user wants to use different generators for the language extension, he must create a project specific runtime module to overwrite the default bindings. If he uses the language extension, he must register each `StandaloneSetup` so that the workflow runtime can read the extension instances. Additionally, he has to register the `GcoreSupport`, or the references to the `gcore` model will not be resolved by Xtext. Because an `IGenerator` requires an EMF `Resource`, the respective core model has to be serialized. With all this prerequisites the `CppGeneratotComponent` can be configured with the project specific `RuntimeModule`, as well as the respective models or resources.

Thus, the user of the generator is able to adapt the generation process respective to his requirements. If he requires the service of a language extension, he must create an extension file and register it at the workflow. Additionally, he configures the `RuntimeModule` with the specific generator. This approach enables non-intrusive positive feature variability in an aspect oriented fashion. Each language extension captures a specific aspect. The configured generation process adds the information to the result. The passive core model is referenced actively

by the respective extension models. If the syntax or semantics of the language extension is insufficient, the language engineer can adapt either the specific language or provide an alternative generator. This enables generative programming in a product line engineering fashion.

The main drawback is the tight coupling between the extension generator interfaces and the calls within the main generation process. An advantage is the flexible mapping from problem to solution domain. The language user is able to create and configure a product line similar to a feature model. Thus, he can generate a specific target language instance from a product family.

4.3 Reference implementation

The reference implementation uses an adapted version of the `TimeBomb` example from [64]. Figure A.4 and A.5 show the respective class and state machine model. The example is GeneSEZ-like divided into the actual project and the generator project. The generator project contains the model, the workflow and the extension files. Executing the `SerializeCoreModel` workflow creates a `gcore` instance of the model and exports the respective XMI representation.

For now, a runtime Eclipse is necessary to deploy the language extension with its editor to an Eclipse instance. This instance is able to interpret files with the registered file extensions correctly as language instances. To write a language extension file, the project needs a reference to the serialized `gcore` instance (see figure 4.6).

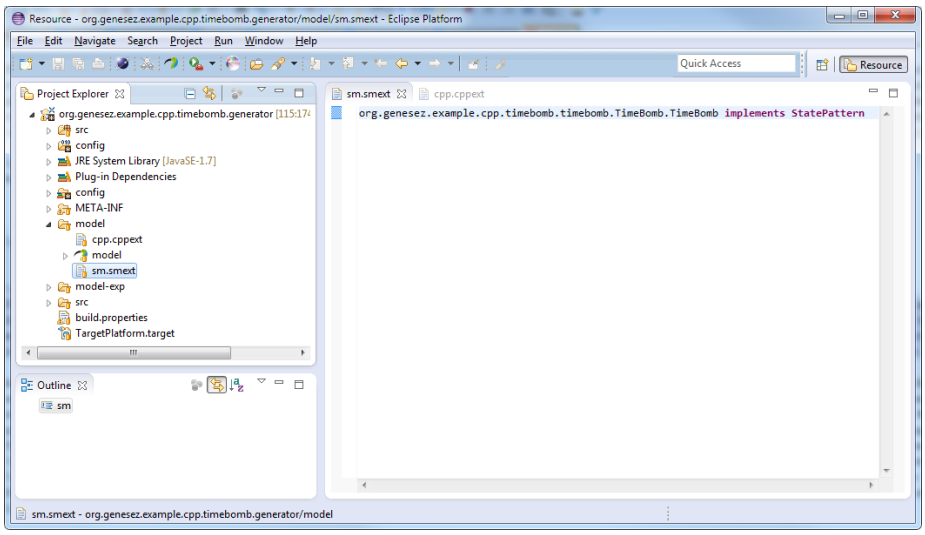

Figure 4.6: A runtime Eclipse with the deployed language extensions. Each extension file references the serialized model of the project.

As depicted in section 4.2.2 on page 72, each language extension consists of two projects[18]. The `org.genesez.gcore.statemachine.addon` project is the actual language extension for the state machines. The respective `ui` plugin configures the editor and will be used by default[19]. Within the `org.genesez.gcore.statemachine` package of the Xtext project is a workflow, a grammar, a `RuntimeModule` and a `StandaloneSetup`. Listing 4.2 on page 79 shows the grammar example of the language extension for state machines.

The first line specifies the grammar name and the inheritance of terminals like 'ID'. The second line tells Xtext to create a metamodel. The third line imports `gcore`[20]. The fifth line contains the starting rule. The `StateMachine` rule encapsulates an `MStateMachine` of `gcore` within an `AugmentedMStateMachine`. The `AugmentedStateMachine` rule adds the additional information to the `AugmentedMStateMachine`. The grammar enables the language user exemplarily to configure the implementation kind of the generated state machine (see section 2.4 on page 42).

Figure 4.7 shows the respective metamodel that will be generated from the grammar by Xtext.

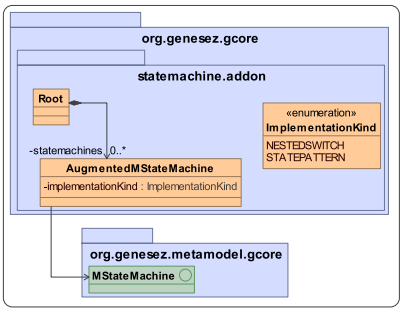

Figure 4.7: The respective metamodel of the state machine grammar from listing 4.2 on the following page.

The workflow contains the grammar URI, the file extension, the generator and various project specific details. To generate working EMF resources, the respective `gcore` resources have to be registered, like in snippet 4.1 from the `GenerateAddOn.mwe2` workflow of the project.

```
registerGenModelFile =
    "platform:/resource/org.genesez.metamodel.core/model/gcore.genmodel"
registerGeneratedEPackage = "org.genesez.metamodel.gcore.GcorePackage"
```

Listing 4.1: The `gcore` import for the workflow to generate the state machine extension Plug-in.

To gain text based auto completion and syntax highlighting of the `gcore` instance within the language extension editor, the `org.genesez.gcore.resource`

[18] Xtext projects consist usually of an additional sdk and test project. These are omitted for the sake of brevity.

[19] Because of a flaw in Xtend2.3 the projects cannot contain the word extension and thus are named addon instead.

[20] This is possible due to a Plug-in dependency to `org.genesez.metamodel.core`

Plug-in[21] needs to be deployed. It registers the XMI file extension at Xtext, hence Xtext is able to access and parse serialized gcore models.

```
grammar org.genesez.gcore.statemachine.AddOn with
     org.eclipse.xtext.common.Terminals
generate addOn "http://www.genesez.org/gcore/statemachine/AddOn"
import "http://genesez.org/metamodel/core" as gcore

Root:
  statemachines+=AugmentStateMachine*;
StateMachine returns AugmentedMStateMachine:
  statemachine =[gcore::MStateMachine|QualifiedName];
QualifiedName:
  ID ("." ID)*;
AugmentStateMachine returns AugmentedMStateMachine:
  StateMachine 'implements' implementationKind = ImplementationKind;
enum ImplementationKind:
  NESTEDSWITCH = 'nestedSwitch'|
  STATEPATTERN = 'StatePattern';
```

Listing 4.2: The Xtext grammar file of the org.genesez.gcore.statemachine.addon project

The org.genesez.gcore.statemachine.generator package contains an AugmentedStateMachineGenerator and a StatePatternGenerator (see figure 4.4 on page 75). Both implement the IStateMachineGenerator interface of the org.genesez.platform.cpp2 project. The AugmentedStateMachineGenerator is also an IGenerator and the entry point for the generation process of the language extension. As outlined in section 4.2.3 on page 73, the doGenerate method is misused to inject the extension model and the IFileSystemAccess which handles the protected regions (see listing 4.3).

```
  override doGenerate(Resource input, IFileSystemAccess fsa) {
      setResource(input)
      this.fsa=fsa
  }
```

Listing 4.3: The doGenerate operation of the AugmentedStateMachineGenerator.

The moment the setStateMachine method is called, a ResourceSet of EMF is queried to get the proper gcore and extension instances[22] (see listing 4.4).

```
  def void queryResources(URI rootModel,URI referencingModel)
  {//a freshly instantiated ResourceSet can access registered resources
    var ResourceSet rs = new ResourceSetImpl()
    gcoreModel = rs.getResource(rootModel,true)
    myModel = rs.getResource(referencingModel,true)
  }
```

Listing 4.4: The queryResources operation of the AugmentedStateMachineGenerator.

[21] Together with its respective ui Plug-in.

[22] Despite the already injected extension model, EMF requires the reference of the model instance, that is present in the ResourceSet, together with the gcore instance to resolve calls to model elements.

With the correct model instances, it is possible to query the extension model. If it contains a reference to the injected `MStateMachine`, the respective data will be evaluated. Currently, the `xmiGuid` of the `MStateMachine` is used to identify the correct state machine (see listing 4.5).

Each time a state machine is set, the extension generator queries the extension model which is similar to an output-driven traversing approach (see the item Translating on page 12). The implementation is inefficient, but to phrase [92]: "... performance is not a high priority because the code generator itself is not in production. It is the generated code that must match the performance metrics of the project. Memory space efficiency is also of little concern because the generator is run offline during development."

Depending on the queried information, the extension generator creates a specific generator, to delegate the code generation calls, and injects the `MStateMachine` into this generator.

```
def dispatch ImplementationKind find(Root root, MStateMachine sm){
  for(AugmentedMStateMachine asm : root.statemachines){
    if(asm.wraps(sm)){return asm.implementationKind}
  }
  return ImplementationKind::NESTEDSWITCH
}
def boolean wraps(AugmentedMStateMachine asm,MStateMachine sm){
  if(asm.statemachine.xmiGuid==sm.xmiGuid){return true}
  else{return false}
}
```

Listing 4.5: The evaluation of a possible extension of an `MStateMachine` within the `AugmentedStateMachineGenerator`.

E.g., The `StatePatternGenerator` contains templates to create a state machine based on the state pattern (see [12] and section 2.4 on page 42). Because it implements the `IStateMachineGenerator`, each method call can be forwarded. The default generator is the `NestedSwitchGenerator`. It will be the fallback generator if the state machine is not enriched by the language extension (see listing 4.5 and 4.6).

```
override setStateMachine(MStateMachine statemachine) {
  queryResources(statemachine.eResource.URI,myModel.URI)
  sm = statemachine
  impl = queryImplementation
  switch(impl){
    case ImplementationKind::NESTEDSWITCH: smGen = new
         NestedSwitchGenerator
    case ImplementationKind::STATEPATTERN: smGen = new
         StatePatternGenerator
    default: smGen = new NestedSwitchGenerator}
  smGen.setStateMachine(sm)}
```

Listing 4.6: The choice of the template generator, respective to the augmentation of the `MStateMachine`, within within the `AugmentedStateMachineGenerator`.

An alternative approach is realized with the `org.genesez.platform.cpp.addon` project that extends `MOperation`[23] with C++ specific information. It is

[23] An `MOperation` is the gcore representation of a UML `Operation`

similar to the org.genesez.gcore.statemachine.addon project. The difference is the omission of the additional level of indirection for specific implementation kinds. Because of the required information within an ExtendedMOperation, the ExtendedOperationGenerator handles the generateOperation call directly. It is illustrated in the class diagram 4.5 on page 76. An example of the grammar and the respective metamodel can be found in the appendix (see listing A.5 and figure A.6).

The org.genesez.platform.cpp2 project represents the main generator for C++. It consist of a workflow, an extension and a generator package. The workflow package contains workflow components, as well as the default CppGeneratorRuntimeModule and the CppGeneratorSetup to register a RuntimeModule. The StandAloneSetupCollectorComponent collects each StandaloneSetup of the language extension, to establish an EMF registration, necessary for the project specific workflow (see section 4.2.4 on page 76). The CppGeneratorComponent configures and invokes the generation process. It requires a RuntimeModule to configure the generator bindings for Guice, and it needs at least the core model (see listing 4.7).

The additional extension resources are optional and prototypically set by explicit slots. Each generator is queried by the Guice injector. If the slots for the extension resources are set, the extension resource, as well as the IFileSystemAccess, will be injected by the doGenerate method into the specific generator (see listing 4.8 on the following page).

Finally, the doGenerate method of the CppGenerator is called to start the generation process.

```
  public void preInvoke() {
    super.preInvoke();
    if (slotNames.isEmpty()) {throw new IllegalStateException("no 'slot'
        has been configured.");}
    if ((extOpSlot == null || (extOpSlot.isEmpty())) &&
        (injector.getInstance(IOperationGenerator.class) instanceof
        IGenerator)) {
      throw new IllegalStateException("Extension Generator " +
          injector.getInstance(IOperationGenerator.class).toString()
         + " incorrectly configured with empty resource");}
    if ((asmSlot == null || (asmSlot.isEmpty())) &&
        (injector.getInstance(IStateMachineGenerator.class) instanceof
        IGenerator)) {
      throw new IllegalStateException("Extension Generator " +
          injector.getInstance(IStateMachineGenerator.class).toString()
         + "  incorrectly configured with empty resource");}
  };
```

Listing 4.7: The preInvoke method of the CppGeneratorComponent to check for an available Injector as well as an available coremodel slot.

```
public void invoke(IWorkflowContext ctx) {
  IGenerator cppGenerator = injector.getInstance(IGenerator.class);
  IFileSystemAccess fsa = getConfiguredFileSystemAccess();
  IOperationGenerator operationGenerator =
       injector.getInstance(IOperationGenerator.class);
  if (extOpSlot != null && !(extOpSlot.isEmpty())) {
    Object cppExtensionModel = ctx.get(extOpSlot);
    ((IGenerator) operationGenerator).doGenerate(((EObject)
         cppExtensionModel).eResource(), fsa);
  }
  ((CppGenerator)
       cppGenerator).setIOperationGenerator(operationGenerator);
  IStateMachineGenerator smGenerator =
       injector.getInstance(IStateMachineGenerator.class);
  if (asmSlot != null && !(asmSlot.isEmpty())) {
    Object smExtensionModel = ctx.get(asmSlot);
    ((IGenerator) smGenerator).doGenerate(((EObject)
         smExtensionModel).eResource(), fsa);
  }
  ((CppGenerator) cppGenerator).setIStateMachineGenerator(smGenerator);
```

Listing 4.8: The configuration of each generator within the invoke method of the `CppGeneratorComponent`.

The `CppGenerator` is part of the generator package. It is yet another level of indirection to configure and call the `Class` template class. As already discussed in section 3.2 on page 49, the resulting source code is divided into the header specific declaration part and the definition part. If the respective class contains an `ownedBehavior` of type `MStateMachine`, the specific interface methods of the `IStateMachineGenerator` will be called to weave in the state machine aspect (see listing 4.9).

```
def hasStateMachine(MClass it){
  if(ownedBehavior.size==1){
    var behavior = ownedBehavior.get(0)
    switch(behavior){
      MStateMachine:{
        smGen.setStateMachine(behavior)
        return true}
    }
  }
  return false}
```

Listing 4.9: The `hasStateMachine` method of the `Class` template class to evaluate if the respective `MClass` has a `MStateMachine`.

Each interface and default implementation of an extension generator is contained in the extensions package. Because Xtend2.3 is not able to support interfaces, the extension interfaces are plain Java interfaces. E.g., The `IStateMachineGenerator` is a simple interface with one method to set an `MStateMachine`, a declaration and a definition method, as well as a method for signals and states returning a `CharSequence`.

The TimeBomb example is configured with a state machine extension and an operation extension (see listing 4.10 and listing 4.11).

```
org.genesez.example.cpp.timebomb.timebomb.TimeBomb.TimeBomb implements
    StatePattern
```

Listing 4.10: The state machine extension file to configure the TimeBomb example with a StatePattern implementation.

```
org.genesez.example.cpp.timebomb.timebomb.TimeBomb.getPower const throw()
org.genesez.example.cpp.timebomb.timebomb.TimeBomb.isArmed const
```

Listing 4.11: The operation extension file of the TimeBomb example with C++ specific details.

The project specific RunTimeModule binds the specific generators (see listing 4.12). Because it derives from the CppGeneratorRuntimeModule, it also consists of the adapted IFileSystemAccess to generate protected regions.

```
public class TimeBombRuntimeModule extends CppGeneratorRuntimeModule {
  public TimeBombRuntimeModule() {}

  @Override
  public Class<? extends IOperationGenerator> bindIOperationGenerator()
  {
    return org.genesez.platform.cpp.generator
      .ExtendedOperationGenerator.class;
  }
  @Override
  public Class<? extends IStateMachineGenerator>
      bindIStateMachineGenerator()
  {
    return org.genesez.gcore.statemachine.generator
      .AugmentedStateMachineGenerator.class;
  }
}
```

Listing 4.12: The RuntimeModule of the TimeBomb example to bind the correct extension generators.

The Generate workflow of the TimeBomb example configures and executes the generation process (see listing A.6 in the appendix). The first component executes the UML2GeneSEZ transformation to instantiate a gcore compliant model from the UML model. The second component collects, and executes, all necessary StandaloneSetups of the language extensions to register the language extensions at EMF. The GcoreSupport component makes gcore available for Xtext, hence EMF is able to resolve referenced model elements. The Serializer component creates a resource from the core model, created with the first component. The following two Reader components map the extension files to resources for EMF. The last component is the CppGeneratorComponent to configure and execute the model to text transformation. To create the correct injector for the extension generators, the project specific RuntimeModule is injected into the CppGeneratorSetup. Each extension resource, the core model and the output path for the IFileSystemAccess are set.

The generation will result in a header and source file. Snippet 4.13 from the header shows the enriched methods of the UML model, containing the keywords `const` or `throw` respective to the information of the extension file.

```
  public:
    int getPower() const throw();
    TimeBomb(int power, int defuseCode);
    bool isArmed() const;
```

Listing 4.13: The method declaration of the `TimeBomb` example with the additional extension information.

Because the extension file for the state machine defines the `StatePattern` implementation kind for the state machine of the `TimeBomb`, it is declared like in snippet A.7. The virtual base class `TimeBombState` defines each available transition empty. Each state referring to a transition overwrites it. The `state` pointer captures the state of the `TimeBomb`. Each event gets dispatched to the respective event handler function. The dispatch method provides a standardized interface to forward events to the state machine. The `onArm` event handler function of the `timing` state, depicted in snippet 4.14, contains the respective guard and a state change.[24] If the guard evaluates to true, the state change will be executed. The snippet 4.15 from the `StatePatternGenerator` evaluates the kind of transition modeled by the developer, according to [29], and generates the respective lines of code.

```
void TimeBomb::timingState::onArm(TimeBomb* ctx)
{
  if(evalDisarm())
  {
    //PROTECTED REGION ID (disarm_arm_evalDisarm) ENABLED START
            // add your code in between this section
            //PROTECTED REGION END
    ctx->state = &(ctx->setting);
  }
}
bool TimeBomb::timingState::evalDisarm()
{
//PROTECTED REGION ID (_wsg9ElswEeKkL_Cf_2btpg) ENABLED START
// add your code in between this section
//PROTECTED REGION END
}
```

Listing 4.14: The `onArm` event handler function of the `timinig` state.

```
def CharSequence evalKind(MTransition it){

if(!target.outgoing.empty){
switch(kind)
{
  case "ext":return '''ctx->state = &(ctx->«target.name»);'''
  case "int":return "//internal transition"
  case "loc":return ""
}

}
return "//no outgoing transition on target"
}
```

Listing 4.15: The evaluation of the kind of an `MTransition`.

[24] There seems to be a flaw in the protected regions suppport because sometimes the indentations will not be generated as expected.

4.4 Synopsis

The introduced solution provides a mapping from a UML model to the respective C++ representation. Yet UML is an abstract modeling language, and C++ is a multiparadigm programming language. Based on the requirements of the product, one of many C++ implementations needs to be selected. The solution is based on EMF to import UML models from projectional UML editors. Because the `gcore` metamodel is an effective syntax for a language transformation, the imported model will be transformed into a `gcore` model. This intermediate form is enriched with information by so called language extensions.

Each language extension with its respective generators is designed to be orthogonal to each other, similar to policies or aspects. Thus, a language extension defines one or more features and each project specific configuration is an instance of those features. The language user interacts with the specific features. The language engineer defines the features and makes them available with its language extension and generators. The language engineer of the main generator provides the interfaces and hooks for the language extensions.

The language extensions are highly adaptable and can be specialized if required. The specialization provides a customizable granularity. The more abstract an extension is, the more reusable it is. The extensions complement the modeling approach of UML. The generation process is configured by a configuration DSL. Thus, the transformation is adaptable. The extension file and the semantics of the language extensions can be interchanged at will. Thus, the solution approach represents a simplified version of generative programming.

Chapter 5

Conclusions

Using state machines to represent the dynamic behavior of reactive systems is the current method in various domains. Because of the expressive, yet simple, syntax it is used by engineers and computer scientists to communicate. As depicted in the second chapter the concrete syntax of state machines has a deep history that influenced the syntax of UML state machines. The syntax of UML state machines is precisely defined, yet the OMG introduced semantic variation points that have to be defined for the implementation of the system. The resulting products have diverging requirements (e.g., embedded systems, software for visualization, accounting systems, and so on). Those requirements are necessary to choose the correct implementation for the variation points. C++ provides various language features for the developer to implement a state machine. Not only the regular implementations can vary, but several frameworks and libraries provide an implementation for specific use cases.

This thesis introduces an approach to solve this mapping problem by product line engineering and generative programming. The domain expert creates an UML model that will be transformed into a corresponding `gcore` model to minimize the accidental complexity. The `gcore` metamodel specifies a sophisticated syntax to simplify the following transformations for the language engineer. This model acts as core model and will be referenced by additional languages. Each of these languages add positive variability to the model and capture one or more technical features. The language engineer does not only provide the syntax, but he is also able to provide different semantics for these domain specific languages for the transformation process. If the language user requires additional features he configures the generation process by adding additional models and registers the respective generators at the workflow.

The reference implementation is based on the Xtext framework, but it could be exchanged or even combined by alternative EMF based frameworks to enrich the transformation process. At the moment only two simple extension languages are implemented to demonstrate the approach. Most of the features of C++, depicted in the third chapter, are not yet implemented. Thus, future work will be the implementation of more transformation rules as well as the examination of combining transformations of different frameworks.

Appendix A

Appendix

A.1 UML diagrams

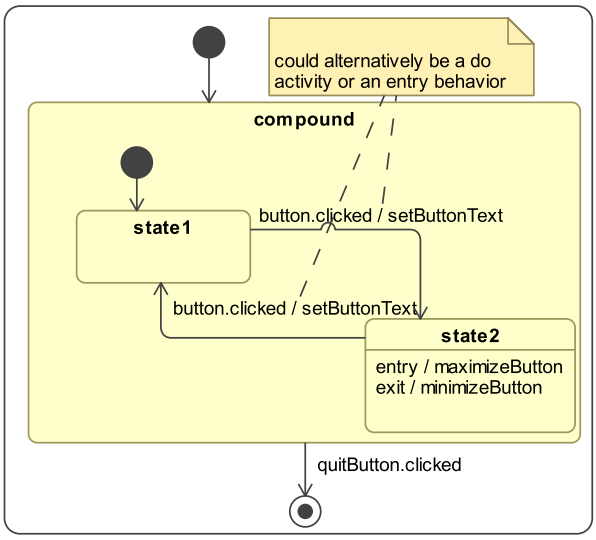

Figure A.1: The UML diagram of the Qt state machine as basis for listing 3.1

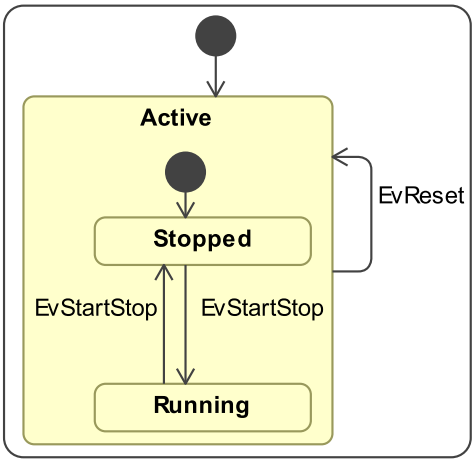

Figure A.2: The UML diagram of the Stop watch state machine as basis for listing A.1

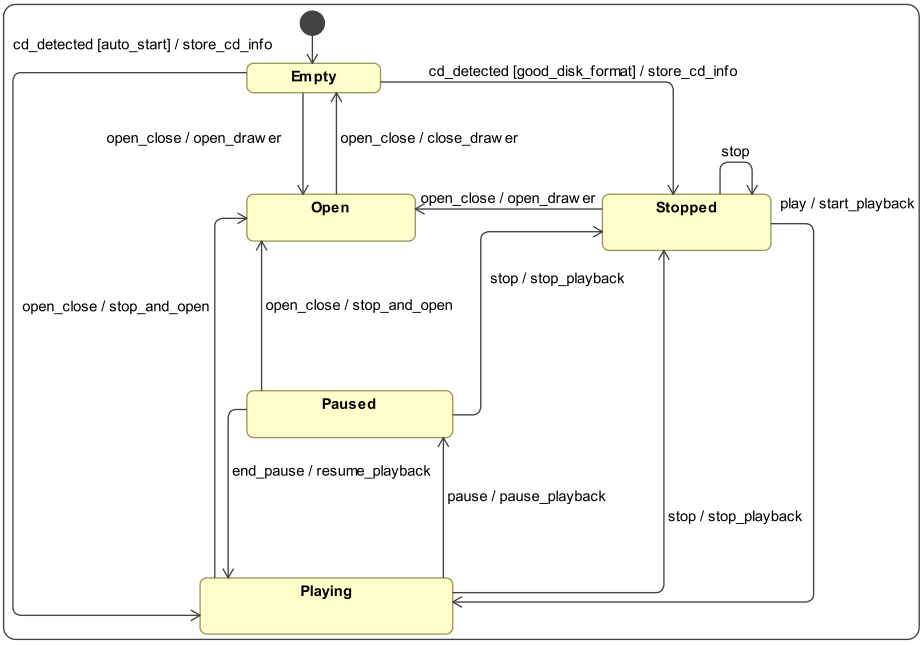

Figure A.3: The UML diagram of the CD player state machine as basis for the listings A.2, A.3 and A.4

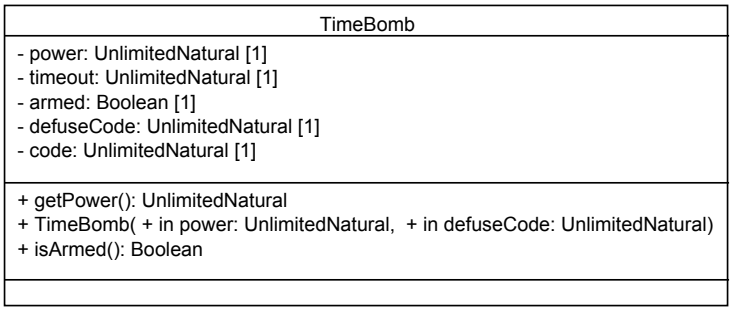

Figure A.4: The adapted Timebomb class from [64]

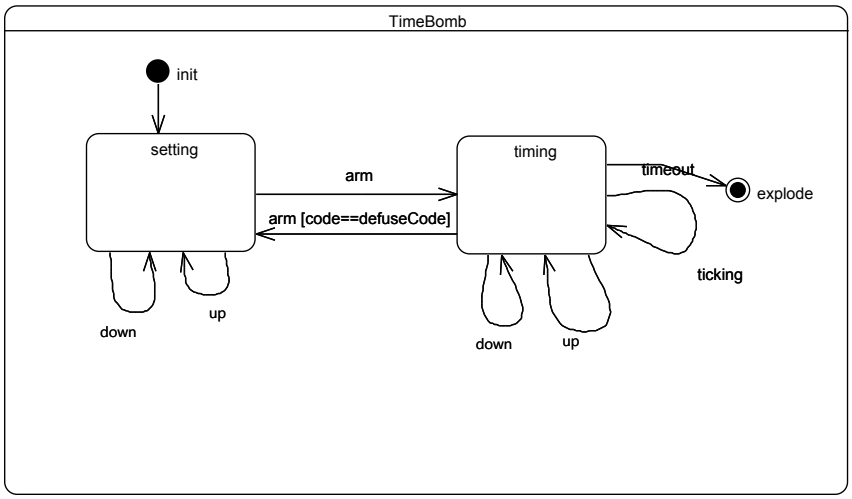

Figure A.5: The adapted Timebomb state machine from [64]

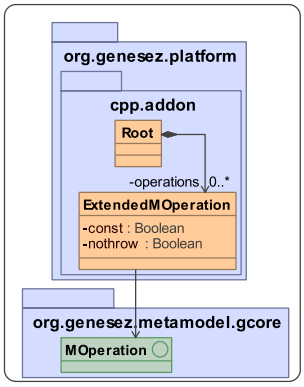

Figure A.6: The respective metamodel of the C++ grammar from listing A.5.

A.2 Source code listings

```
namespace sc = boost::statechart;
namespace mpl = boost::mpl;

// Event declaration and definition
struct EvStartStop : sc::event< EvStartStop > {};
struct EvReset : sc::event< EvReset > {};
struct EvGetElapsedTime : sc::event< EvGetElapsedTime >{
  public:
    EvGetElapsedTime( double & time ) : time_( time ) {}
    void Assign( double time ) const{time_ = time;}
  private:
    double & time_;
};

struct Active;
//StopWatch statemachine with active as initial State
struct StopWatch : sc::state_machine< StopWatch, Active > {};

struct Stopped;
// composit active state of stopwatch with Stopped is initial state
struct Active : sc::simple_state< Active, StopWatch, Stopped >{
  public:
 //reset transition
    typedef sc::transition< EvReset, Active > reactions;
    Active() : elapsedTime_( 0.0 ) {}
    double & ElapsedTime(){return elapsedTime_;}
    double ElapsedTime() const{return elapsedTime_;}
  private:
    double elapsedTime_;
};
//running state within active state
struct Running : sc::simple_state< Running, Active >{
  public:
    typedef mpl::list<
      sc::custom_reaction< EvGetElapsedTime >,
      sc::transition< EvStartStop, Stopped >
    > reactions;

    Running() : startTime_( std::time( 0 ) ) {}

    ~Running(){
   //access the context variable of the compound state
      context< Active >().ElapsedTime() = ElapsedTime();
    }
  //handling the SignalEvent
    sc::result react( const EvGetElapsedTime & evt ){
      evt.Assign( ElapsedTime() );
      return discard_event();
    }
  private:
    double ElapsedTime() const{
      return context< Active >().ElapsedTime() +
        std::difftime( std::time( 0 ), startTime_ );
    }
    std::time_t startTime_;
};

struct Stopped : sc::simple_state< Stopped, Active >{
  typedef mpl::list<
    sc::custom_reaction< EvGetElapsedTime >,
    sc::transition< EvStartStop, Running >
  > reactions;

  sc::result react( const EvGetElapsedTime & evt ){
    evt.Assign( context< Active >().ElapsedTime() );
    return discard_event();
  }
};
```

Listing A.1: The refined stop watch example of [88].

```
// Copyright 2010 Christophe Henry
// henry UNDERSCORE christophe AT hotmail DOT com
// This is an extended version of the state machine available in the boost::mpl library
// Distributed under the same license as the original.
// Copyright for the original version:
// Copyright 2005 David Abrahams and Aleksey Gurtovoy. Distributed
// under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

// see https://github.com/wuhao5/boost/blob/master/libs/msm/doc/
// HTML/examples/SimpleTutorial.cpp for the full example
// includes and using directives omitted for brevity

namespace{
 // events
 struct play {};
 struct end_pause {};
 struct stop {};
 struct pause {};
 struct open_close {};
```

```
enum DiskTypeEnum{DISK_CD=0,DISK_DVD=1};
struct cd_detected{
 cd_detected(string name, DiskTypeEnum diskType)
  : name(name),disc_type(diskType){}
 string name;
 DiskTypeEnum disc_type;
};
// front-end: define the FSM structure
struct player_ : public msm::front::state_machine_def<player_>{
 struct Empty : public msm::front::state<>{
  template <class Event,class FSM> void on_entry(Event const&,FSM&){cout << "entering: Empty" <<
       endl;}
  template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Empty" <<
       endl;}
 };
 struct Open : public msm::front::state<>{
  template <class Event,class FSM> void on_entry(Event const& ,FSM&) {cout << "entering: Open" <<
       endl;}
  template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Open" << endl;}
 };
 struct Stopped : public msm::front::state<>{
  template <class Event,class FSM> void on_entry(Event const& ,FSM&) {cout << "entering: Stopped" <<
       endl;}
  template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Stopped" <<
       endl;}
 };
 struct Playing : public msm::front::state<>{
  template <class Event,class FSM> void on_entry(Event const&,FSM& ) {cout << "entering: Playing" <<
       endl;}
  template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Playing" <<
       endl;}
 };
 struct Paused : public msm::front::state<>{};
 // the initial state of the player SM. Must be defined
 typedef Empty initial_state;
 // transition actions
 void start_playback(play const&) { cout << "player::start_playback\n"; }
 void open_drawer(open_close const&) { cout << "player::open_drawer\n"; }
 void close_drawer(open_close const&) { cout << "player::close_drawer\n"; }
 void store_cd_info(cd_detected const&) { cout << "player::store_cd_info\n"; }
 void stop_playback(stop const&) { cout << "player::stop_playback\n"; }
 void pause_playback(pause const&) { cout << "player::pause_playback\n"; }
 void resume_playback(end_pause const&) { cout << "player::resume_playback\n"; }
 void stop_and_open(open_close const&) { cout << "player::stop_and_open\n"; }
 void stopped_again(stop const&) {cout << "player::stopped_again\n";}
 // guard conditions
 bool good_disk_format(cd_detected const& evt){
  if (evt.disc_type != DISK_CD){cout << "wrong disk" << endl;return false;}
  return true;
 }
       bool auto_start(cd_detected const&){return false;}
       typedef player_ p;
       // Transition table for player
 struct transition_table : mpl::vector<
  // Start Event Next Action Guard
  // +--------+-------------+---------+---------------------+----------------------+
   a_row < Stopped , play , Playing , &p::start_playback >,
   a_row < Stopped , open_close , Open , &p::open_drawer >,
    a_row < Stopped , stop , Stopped >,
  // +--------+-------------+---------+---------------------+----------------------+
   a_row < Open , open_close , Empty , &p::close_drawer >,
  // +--------+-------------+---------+---------------------+----------------------+
   a_row < Empty , open_close  , Open    , &p::open_drawer >,
     row < Empty , cd_detected , Stopped , &p::store_cd_info ,&p::good_disk_format >,
     row < Empty , cd_detected , Playing , &p::store_cd_info ,&p::auto_start >,
  // +--------+-------------+---------+---------------------+----------------------+
   a_row < Playing , stop , Stopped , &p::stop_playback >,
   a_row < Playing , pause , Paused , &p::pause_playback >,
   a_row < Playing , open_close , Open , &p::stop_and_open >,
  // +--------+-------------+---------+---------------------+----------------------+
   a_row < Paused , end_pause , Playing , &p::resume_playback >,
   a_row < Paused , stop , Stopped , &p::stop_playback >,
   a_row < Paused , open_close , Open , &p::stop_and_open >
  // +--------+-------------+---------+---------------------+----------------------+
       > {};
   };
   // Pick a back-end
   typedef msm::back::state_machine<player_> player;

   static char const* const state_names[] = { "Stopped", "Open", "Empty", "Playing", "Paused" };
   void pstate(player const& p){
 cout << " -> " << state_names[p.current_state()[0]] << endl;
   }
   void test(){
 player p;
       // needed to start the highest-level SM. This will call on_entry and mark the start of the SM
       p.start();
       // go to Open, call on_exit on Empty, then action, then on_entry on Open
       p.process_event(open_close()); pstate(p);
       p.process_event(open_close()); pstate(p);
       // will be rejected, wrong disk type
       p.process_event(cd_detected("louie, louie",DISK_DVD)); pstate(p);
       p.process_event(cd_detected("louie, louie",DISK_CD)); pstate(p);
 p.process_event(play());
       // at this point, Play is active
       p.process_event(pause()); pstate(p);
       // go back to Playing
       p.process_event(end_pause()); pstate(p);
       p.process_event(pause()); pstate(p);
```

```
        p.process_event(stop()); pstate(p);
        // event leading to the same state
        // no action method called as it is not present in the transition table
        p.process_event(stop()); pstate(p);
        cout << "stop fsm" << endl;
        p.stop();
    }
}
int main(){test();return 0;}
```

Listing A.2: The state machine of an adapted player example of [85] from [89]. It is defined with the basic frontend of Meta State Machine.

```
// Copyright 2010 Christophe Henry
// henry UNDERSCORE christophe AT hotmail DOT com
// This is an extended version of the state machine available in the boost::mpl library
// Distributed under the same license as the original.
// Copyright for the original version:
// Copyright 2005 David Abrahams and Aleksey Gurtovoy. Distributed
// under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

// see https://github.com/wuhao5/boost/blob/master/libs/msm/doc/HTML/
// examples/SimpleWithFunctors.cpp for the full example
// includes and using directives omitted for brevity

namespace{
 // events
 struct play {};
 struct end_pause {};
 struct stop {};
 struct pause {};
 struct open_close {};
 enum DiskTypeEnum{DISK_CD=0,DISK_DVD=1};
 struct cd_detected{
  cd_detected(string name, DiskTypeEnum diskType)
   : name(name),disc_type(diskType){}
  string name;
  DiskTypeEnum disc_type;
 };
 // front-end: define the FSM structure
 struct player_ : public msm::front::state_machine_def<player_>{
  struct Empty_Entry{template <class Evt,class Fsm,class State> void operator()(Evt const& ,Fsm&
       ,State& ){cout << "entering: Empty" << endl;}};
  struct Empty_Exit{template <class Evt,class Fsm,class State> void operator()(Evt const& ,Fsm&
       ,State& ){cout << "leaving: Empty" << endl;}};
  struct Empty : public msm::front::euml::func_state<Empty_Entry,Empty_Exit>{};
  // struct Empty : public msm::front::state<>{
   // template <class Event,class FSM> void on_entry(Event const&,FSM& ) {cout << "entering: Empty" <<
        endl;}
   // template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Empty" <<
        endl;}
  // };
  struct Open : public msm::front::state<>{
   template <class Event,class FSM> void on_entry(Event const& ,FSM&) {cout << "entering: Open" <<
        endl;}
   template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Open" << endl;}
  };
  struct Stopped : public msm::front::state<>{
   template <class Event,class FSM> void on_entry(Event const& ,FSM&) {cout << "entering: Stopped" <<
        endl;}
   template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Stopped" <<
        endl;}
  };
  struct Playing : public msm::front::state<>{
   template <class Event,class FSM> void on_entry(Event const&,FSM& ) {cout << "entering: Playing" <<
        endl;}
   template <class Event,class FSM> void on_exit(Event const&,FSM& ) {cout << "leaving: Playing" <<
        endl;}
  };
  struct Paused : public msm::front::state<>{};
  // the initial state of the player SM. Must be defined
  typedef Empty initial_state;
  // transition actions
  struct start_playback{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::start_playback" <<
       endl;}};
  struct open_drawer{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::open_drawer" <<
       endl;}};
  struct close_drawer{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::close_drawer" <<
       endl;}};
  struct store_cd_info{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const&,FSM& fsm ,SourceState& ,TargetState& ){cout << "player::store_cd_info" <<
       endl;}};
  struct stop_playback{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::stop_playback" <<
       endl;}};
  struct pause_playback{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::pause_playback" <<
       endl;}};
  struct resume_playback{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::resume_playback" <<
       endl;}};
```

```
  struct stop_and_open{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::stop_and_open" <<
       endl;}};
  struct stopped_again{template <class EVT,class FSM,class SourceState,class TargetState> void
       operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::stopped_again" <<
       endl;}};
  // guard conditions
  struct good_disk_format{
   template <class EVT,class FSM,class SourceState,class TargetState>
   bool operator()(EVT const& evt ,FSM&,SourceState& ,TargetState& ){
    if (evt.disc_type != DISK_CD){cout << "wrong disk" << endl;return false;}
    return true;
   }
  };
  struct auto_start{
   template <class EVT,class FSM,class SourceState,class TargetState>
   bool operator()(EVT const& evt ,FSM&,SourceState& ,TargetState& ){return false;}
  };
  // Transition table for player
  struct transition_table : mpl::vector<
   // Start Event Next Action Guard
   // +--------+-----------+--------+-------------------------+----------------------+
   Row < Stopped , play , Playing , start_playback , none >,
   Row < Stopped , open_close , Open , open_drawer , none >,
   Row < Stopped , stop , Stopped , none , none >,
   // +--------+-----------+--------+-------------------------+----------------------+
   Row < Open , open_close , Empty , close_drawer , none >,
   // +--------+-----------+--------+-------------------------+----------------------+
   Row < Empty , open_close  , Open    , open_drawer   , none >,
   Row < Empty , cd_detected , Stopped , store_cd_info , good_disk_format >,
   Row < Empty , cd_detected , Playing , store_cd_info , auto_start >,
   // +--------+-----------+--------+-------------------------+----------------------+
   Row < Playing , stop , Stopped , stop_playback , none >,
   Row < Playing , pause , Paused , pause_playback , none >,
   Row < Playing , open_close , Open , stop_and_open , none >,
   // +--------+-----------+--------+-------------------------+----------------------+
   Row < Paused , end_pause , Playing , resume_playback , none >,
   Row < Paused , stop , Stopped , stop_playback , none >,
   Row < Paused , open_close , Open , stop_and_open , none >
   // +--------+-----------+--------+-------------------------+----------------------+
  > {};
 };
    // Pick a back-end
    typedef msm::back::state_machine<player_> player;

    static char const* const state_names[] = { "Stopped", "Open", "Empty", "Playing", "Paused" };
    void pstate(player const& p){cout << " -> " << state_names[p.current_state()[0]] << endl;}

    void test(){
  // see the test function of the basic frontend
    }
}

int main(){test();return 0;}
```

Listing A.3: The state machine of an adapted player example of [85] from [89]. It is defined with the functor frontend of Meta State Machine.

```
// Copyright 2010 Christophe Henry
// henry UNDERSCORE christophe AT hotmail DOT com
// This is an extended version of the state machine available in the boost::mpl library
// Distributed under the same license as the original.
// Copyright for the original version:
// Copyright 2005 David Abrahams and Aleksey Gurtovoy. Distributed
// under the Boost Software License, Version 1.0. (See accompanying
// file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

// see https://github.com/wuhao5/boost/blob/master/libs/msm/doc/HTML/
// examples/SimpleTutorialEuml2.cpp for the full example
// includes and using directives omitted for brevity

namespace
{
    // events
 BOOST_MSM_EUML_EVENT(play)
 BOOST_MSM_EUML_EVENT(end_pause)
 BOOST_MSM_EUML_EVENT(stop)
 BOOST_MSM_EUML_EVENT(pause)
 BOOST_MSM_EUML_EVENT(open_close)
 enum DiskTypeEnum{DISK_CD=0,DISK_DVD=1};
 BOOST_MSM_EUML_DECLARE_ATTRIBUTE(string,cd_name)
 BOOST_MSM_EUML_DECLARE_ATTRIBUTE(DiskTypeEnum,cd_type)
 BOOST_MSM_EUML_ATTRIBUTES((attributes_ << cd_name << cd_type ), cd_detected_attributes)
 BOOST_MSM_EUML_EVENT_WITH_ATTRIBUTES(cd_detected,cd_detected_attributes)
 // front-end: define the FSM structure
 BOOST_MSM_EUML_ACTION(Empty_Entry){template <class Evt,class Fsm,class State> void operator()(Evt
       const& ,Fsm& ,State& ){cout << "entering: Empty" << endl;}};
 BOOST_MSM_EUML_ACTION(Empty_Exit){template <class Evt,class Fsm,class State> void operator()(Evt
       const& ,Fsm& ,State& ){cout << "leaving: Empty" << endl;}};
 BOOST_MSM_EUML_ACTION(Open_Entry){template <class Evt,class Fsm,class State> void operator()(Evt
       const& ,Fsm& ,State& ){cout << "entering: Open" << endl;}};
 BOOST_MSM_EUML_ACTION(Open_Exit){template <class Evt,class Fsm,class State> void operator()(Evt
       const& ,Fsm& ,State& ){cout << "leaving: Open" << endl;}};
 BOOST_MSM_EUML_ACTION(Stopped_Entry){template <class Evt,class Fsm,class State> void operator()(Evt
       const& ,Fsm& ,State& ){cout << "entering: Stopped" << endl;}};
```

```
BOOST_MSM_EUML_ACTION(Stopped_Exit){template <class Evt,class Fsm,class State> void operator()(Evt
      const& ,Fsm& ,State& ){cout << "leaving: Stopped" << endl;}};
BOOST_MSM_EUML_ACTION(Playing_Entry){template <class Evt,class Fsm,class State> void operator()(Evt
      const& ,Fsm& ,State& ){cout << "entering: Playing" << endl;}};
BOOST_MSM_EUML_ACTION(Playing_Exit){template <class Evt,class Fsm,class State> void operator()(Evt
      const& ,Fsm& ,State& ){cout << "leaving: Playing" << endl;}};

BOOST_MSM_EUML_STATE((),Paused)
BOOST_MSM_EUML_STATE((  Empty_Entry,Empty_Exit ),Empty)
BOOST_MSM_EUML_STATE((  Open_Entry,Open_Exit ),Open)
BOOST_MSM_EUML_STATE((  Stopped_Entry,Stopped_Exit ),Stopped)
BOOST_MSM_EUML_STATE((  Playing_Entry,Playing_Exit ),Playing)
// transition actions
BOOST_MSM_EUML_ACTION(start_playback){template <class FSM,class EVT,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM&,SourceState& ,TargetState& ){cout <<
      "player::start_playback" << endl;}};
BOOST_MSM_EUML_ACTION(open_drawer){template <class EVT,class FSM,class SourceState,class TargetState>
      void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout << "player::open_drawer" <<
      endl;}};
BOOST_MSM_EUML_ACTION(close_drawer){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout <<
      "player::close_drawer" << endl;}};
BOOST_MSM_EUML_ACTION(store_cd_info){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const&,FSM& fsm ,SourceState& ,TargetState& ){cout <<
      "player::store_cd_info" << endl;}};
BOOST_MSM_EUML_ACTION(stop_playback){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout <<
      "player::stop_playback" << endl;}};
BOOST_MSM_EUML_ACTION(pause_playback){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout <<
      "player::pause_playback" << endl;}};
BOOST_MSM_EUML_ACTION(resume_playback){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout <<
      "player::resume_playback" << endl;}};
BOOST_MSM_EUML_ACTION(stop_and_open){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout <<
      "player::stop_and_open" << endl;}};
BOOST_MSM_EUML_ACTION(stopped_again){template <class EVT,class FSM,class SourceState,class
      TargetState> void operator()(EVT const& ,FSM& ,SourceState& ,TargetState& ){cout <<
      "player::stopped_again" << endl;}};
// guard conditions
BOOST_MSM_EUML_ACTION(good_disk_format){
 template <class FSM,class EVT,class SourceState,class TargetState>
 bool operator()(EVT const& evt,FSM&,SourceState& ,TargetState& ){if
      (evt.get_attribute(cd_type)!=DISK_CD){cout << "wrong disk, sorry" << endl;return false;}
  return true;
 }
};
BOOST_MSM_EUML_ACTION(auto_start){
 template <class EVT,class FSM,class SourceState,class TargetState>
 bool operator()(EVT const& evt ,FSM&,SourceState& ,TargetState& ){return false;}
};
// Transition table for player
BOOST_MSM_EUML_TRANSITION_TABLE((
    Stopped + play / start_playback == Playing ,
    Stopped + open_close / open_drawer == Open ,
    Stopped + stop == Stopped ,
    // +------------------------------------------------------------------------------+
    Open + open_close / close_drawer == Empty ,
    // +------------------------------------------------------------------------------+
    Empty + open_close                      / open_drawer   == Open ,
    Empty + cd_detected [good_disk_format] / store_cd_info == Stopped ,
    Empty + cd_detected [auto_start]       / store_cd_info == Playing ,
   // +------------------------------------------------------------------------------+
    Playing + stop / stop_playback == Stopped ,
    Playing + pause / pause_playback == Paused ,
    Playing + open_close / stop_and_open == Open ,
    // +------------------------------------------------------------------------------+
    Paused + end_pause / resume_playback == Playing ,
    Paused + stop / stop_playback == Stopped ,
    Paused + open_close / stop_and_open == Open
    // +------------------------------------------------------------------------------+
    ),transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(( transition_table, //STT
          init_ << Empty, // Init State
          no_action, // Entry
          no_action, // Exit
          attributes_ << no_attributes_, // Attributes
          configure_ << no_configure_, // configuration
          Log_No_Transition // no_transition handler
          ), player_) //fsm name
// Pick a back-end
    typedef msm::back::state_machine<player_> player;

    static char const* const state_names[] = { "Stopped", "Open", "Empty", "Playing", "Paused" };
    void pstate(player const& p){cout << " -> " << state_names[p.current_state()[0]] << endl;}
    void test(){
 // see the test function of the basic frontend
    }
}

int main(){test();return 0;}
```

Listing A.4: The state machine of an adapted player example of [85] from [89]. It is defined with the eUML frontend of Meta State Machine.

```
grammar org.genesez.platform.cpp.AddOn with
     org.eclipse.xtext.common.Terminals
generate addOn "http://www.genesez.org/platform/cpp/AddOn"
import "http://genesez.org/metamodel/core" as gcore

Root:
  operations+=ExtendOperation*;

Operation returns ExtendedMOperation:
  operation =[gcore::MOperation|QualifiedName];
QualifiedName:
    ID ("." ID)*;
ExtendOperation returns ExtendedMOperation:
  Operation const?='const'? nothrow?='throw()'?;
```

Listing A.5: The Xtext grammar file of the `org.genesez.gcore.cpp.addon` project

```
module org.genesez.example.cpp.timebomb.serialize
import org.eclipse.emf.mwe.utils.*

var modelpath = "model/"
var model = "model.uml"
var checkScript = "org::genesez::adapter::uml2::uml2constraints"
Workflow {

 //instantiate a core model
 component = org.genesez.adapter.uml2.Uml2GeneSEZ {
  model = "${modelpath}${model}"
  umlCheckScript = "${checkScript}"
 }

 //- execute required setups for adding the models to EMF
 component =
      org.genesez.platform.cpp2.workflow.StandAloneSetupCollectorComponent
      {
  setup = org.genesez.platform.cpp.AddOnStandaloneSetup {}
  setup = org.genesez.gcore.statemachine.AddOnStandaloneSetup {}
 }

 //required for the generator to navigate within the gcore model
 //registers xmi files for Xtext to parse
 component = org.genesez.gcore.resource.GcoreSupport{}

 //creates a resource for the coremodel
 component = org.genesez.platform.common.workflow.Serializer {
  file = "model-exp/model.xmi"
 }

 //add a cppResource to EMF
 component = org.eclipse.emf.mwe.utils.Reader{
  uri = "model/cpp.cppext"
  modelSlot ="cppResource"
 }

 //add a smResource to EMF
 component = org.eclipse.emf.mwe.utils.Reader{
  uri = "model/sm.smext"
  modelSlot ="smResource"
 }

 //maps the coremodel to a coreresource
 //component = org.genesez.platform.cpp2.workflow.ModelToResourceMapper
      {}

 //Workflow component to configure the main generator
 component = org.genesez.platform.cpp2.workflow.CppGeneratorComponent {

  //register main generator setup to create injector
```

```
  register = org.genesez.platform.cpp2.workflow.CppGeneratorSetup {
   //project specific runtimeModule to register alternative Generators
        and register @ EMF
   runtimeModule =
        org.genesez.example.cpp.timebomb.generator.TimeBombRuntimeModule
        {}
  }
  //register models/resources
  extOpSlot = 'cppResource'
  asmSlot = 'smResource'
  slot = 'coremodel'
  outlet = {
   path = "../org.genesez.example.cpp.timebomb/src-gen"
  }
 }
}
```

Listing A.6: The `Generate` workflow of the `TimeBomb` example.

```
    class TimeBombState
    {
      public:
      virtual void onArm(TimeBomb*){}
      virtual void on_ANON_(TimeBomb*){}
      virtual void onUp(TimeBomb*){}
      virtual void onDown(TimeBomb*){}
      virtual void onTick(TimeBomb*){}
      virtual void onTimeout(TimeBomb*){}
    };
    class settingState : public TimeBombState
    {
      public:
      virtual void onArm(TimeBomb* ctx);
      virtual void onUp(TimeBomb* ctx);
      virtual void onDown(TimeBomb* ctx);
    };
    class timingState : public TimeBombState
    {
      public:
      virtual void onArm(TimeBomb* ctx);
      bool evalDisarm();
      virtual void onUp(TimeBomb* ctx);
      virtual void onDown(TimeBomb* ctx);
      virtual void onTick(TimeBomb* ctx);
      virtual void onTimeout(TimeBomb* ctx);
      bool evalTimeoutEvent();
    };
    class initState : public TimeBombState
    {
      public:
      virtual void on_ANON_(TimeBomb* ctx);
    };
    private:
      TimeBombState* state;
      settingState setting;
      timingState timing;
      initState init;
      void onArm(){state->onArm(this);}
      void on_ANON_(){state->on_ANON_(this);}
      void onUp(){state->onUp(this);}
      void onDown(){state->onDown(this);}
      void onTick(){state->onTick(this);}
      void onTimeout(){state->onTimeout(this);}
    public:
      void dispatch(TimeBombSignal event)
      {
        switch(event)
        {
          case arm : onArm(); break;
          case _ANON_ : on_ANON_(); break;
          case up : onUp(); break;
          case down : onDown(); break;
          case tick : onTick(); break;
```

```
            case timeout : onTimeout(); break;
          }
        }
    };
```

Listing A.7: The declaration of the state machine in the `TimeBomb` example.

A.3 Contents of the data storage device

The enclosed data storage device contains the practical part of the master thesis. It is structured as follows:

eclipse The directory contains the Eclipse IDE with all necessary Plug-ins for the workspace.

workspace The directory contains the created and referenced projects to either instantiate a runtime Eclipse or generate the scaffolding of the example project.

masterthesis.ebook.pdf The ebook version of this thesis.

masterthesis.print.pdf The print version of this thesis.

A.3.1 Eclipse projects

The Eclipse workspace contains the following projects:

org.eclipse.emf.ecore.adapter.uml Provides an UML to Ecore transformation to create a `gcore` metamodel.

org.genesez.adapter.uml2 Provides the UML to gcore transformation to create `gcore` complaint models from UML models.

org.genesez.example.cpp.timebomb The example project which contains the generated C++ source code.

org.genesez.example.cpp.timebomb.generator The example project which contains the models, the project specific `RuntimeModule` and the workflow to configure the generation process.

org.genesez.gcore.resource A Plug-in project to register the XMI file extension at Xtext. Deploying this Plug-in enables Xtext to parse a serialized `gcore` model.

org.genesez.gcore.resource.ui The respective user interface project of the `org.genesez.gcore.resource` Xtext project.

org.genesez.gcore.statemachine.addon The Xtext project contains a language extension for state machines.

org.genesez.gcore.statemachine.addon.ui The respective user interface project of the `org.genesez.gcore.statemachine.addon` Plug-in project.

org.genesez.metamodel.core Provides the `gcore` metamodel.

org.genesez.platform.common Provides common components and functions (e.g., the `Serializer` component to serialize a `gcore` compliant model).

org.genesez.platform.cpp.addon The Xtext project contains a language extension for C++ specifics.

org.genesez.platform.cpp.addon.ui The respective user interface project of the `org.genesez.platform.cpp.addon` Xtext project.

org.genesez.platform.cpp2 The main generator project for C++.

A.3.2 Workflows

GenerateAddon.mwe2 of the org.genesez.gcore.statemachine.addon The workflow to generate the metamodel from the grammar file and to create necessary software artifacts for the deployable Plug-in project.

GenerateAddon.mwe2 of the org.genesez.platform.cpp.addon The workflow to generate the metamodel from the grammar file and to create necessary software artifacts for the deployable Plug-in project.

Generate.mwe2 of the org.genesez.example.cpp.timebomb.generator The project specific workflow to generate C++ from the specified models with the respective configuration.

SerializeCoreModel.mwe2 of the org.genesez.example.cpp.timebomb.generator performs a UML to `gcore` transformation on the respective model and serializes the `gcore` compliant model to the `model-exp` directory.

List of Figures

List of listings

Bibliography

[1] Crossway Bibles. The Holy Bible, English Standard Version (with Cross-References), 2 2011.

[2] Bruce Powel Douglass. *Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)*. Addison-Wesley Professional, 3 edition, 2 2004. ISBN 9780321160768.

[3] Alfred V. Aho, J. D. Ullman, J. E. Hopcroft. *Data Structures and Algorithms (Addison-Wesley Series in Computer Science and Information Pr)*. Addison Wesley Pub Co Inc, 1 1982. ISBN 9780201000238.

[4] Bjarne Stroustrup. *C++ Programming Language*. Addison Wesley Pub Co Inc, 0004 edition, 3 2013. ISBN 9780321563842.

[5] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Longman, Amsterdam, 3rd edition. international edition. edition, 3 2007. ISBN 9780321514486.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman, Amsterdam, 2nd ed. edition, 9 2006. ISBN 9780321491695.

[7] Terence Parr. *Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages*. Pragmatic Programmers, 1 edition, 1 2010. ISBN 9781934356456.

[8] Steven Pinker. *The Language Instinct: How the Mind Creates Language (P.S.)*. Harper Perennial Modern Classics, 1 reprint edition, 9 2007. ISBN 9780061336461.

[9] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Longman, Amsterdam, 1 edition, 12 2008. ISBN 9780321553454.

[10] Martin Fowler. Refactoring: Improving the Design of Existing Code. In *XP/Agile Universe*, 256. 2002.

[11] Martin Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, 10 2010. ISBN 9780321712943.

[12] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 11 1994. ISBN 9780201633610.

[13] Markus Völter. From Programming to Modeling - and Back Again. *IEEE Software*, 28(6):20–25, 2011.

[14] Debasish Ghosh. *DSLs in Action*. Manning Publications, pap/psc edition, 12 2010. ISBN 9781935182450.

[15] Frederick Phillips Brooks. *The mythical man-month: Essays on software engineering*. Addison-Wesley, Boston, Mass., anniversary ed. [with 4 new chapters], 33. printing. edition, 2008. ISBN 0201835959.

[16] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[17] Noam Chomsky. *Aspects of the Theory of Syntax*. The MIT Press, 3 1969. ISBN 9780262530071.

[18] Gottfried Vossen, Kurt-Ulrich Witt. *Grundkurs Theoretische Informatik: Eine anwendungsbezogene Einführung für Studierende in allen Informatik-Studiengängen*. Vieweg+Teubner Verlag, 4, verb. und erw. aufl. 2006 edition, 3 2006. ISBN 9783834801531.

[19] Terence Parr. *The Definitive Antlr Reference: Building Domain-Specific Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, 1 edition, 5 2007. ISBN 9780978739256.

[20] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[21] Kim Marriott, Bernd Meyer. *Visual Language Theory*. Springer, 1998 edition, 7 1998. ISBN 9780387983677.

[22] Reiko Heckel. Graph Transformation in a Nutshell. *Electr. Notes Theor. Comput. Sci.*, 148(1):187–198, 2006.

[23] Grzegorz Rozenberg, editor. *Foundations: Foundations Vol 1 (Handbook of Graph Grammars and Computing by Graph Transformation)*. World Scientific Pub Co (, 2 1997. ISBN 9789810228842.

[24] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer. *Fundamentals Of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, softcover reprint of hardcover 1st ed. 2006 edition, 12 2009. ISBN 9783642068317.

[25] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, 2., aktualisierte und erweiterte auflage. edition, 5 2007. ISBN 9783898644488.

[26] Dan Pilone, Neil Pitman. *UML 2.0 in a Nutshell: A Desktop Quick Reference (In a Nutshell (O'Reilly))*. O'Reilly Media, 1 edition, 6 2005. ISBN 9780596007959.

[27] Stephen J. Mellor, Kendall Scott, Dirk Weise. *MDA Distilled*. Addison-Wesley Longman, Amsterdam, 3 2004. ISBN 9780201788914.

[28] Object Management Group. *The UML Infrastructure Specification 2.4*, November 2010. http://www.omg.org/spec/UML/2.4/.

[29] Object Management Group. *The UML Superstructure Specification 2.4*, November 2010. http://www.omg.org/spec/UML/2.4/.

[30] Berthold Hoffmann, Mark Minas. Defining Models - Meta Models versus Graph Grammars. *ECEASST*, 29:13, 2010.

[31] Tobias Haubold. *Konzepte zur Integration einer Komponente zur Traceability von Anforderungen in ein MDA Framework*. Master's thesis, Westsächsische Hochschule Zwickau, May 2010.

[32] Nico Herbig. *Konzeption und Integration einer TYPO3-Plattform in das MDA-Framework GeneSEZ zur Entwicklung von Extensions in Extbase und Fluid*. Master's thesis, Westsächsische Hochschule Zwickau, March 2012.

[33] Andre Pflüger. *Entwicklung einer EJB3-Cartridge für die modellgetriebene Softwareentwicklung und Entwurf eines komponentenbasierten Verwaltungs- und Kommunikationsplattform-Prototyps als Referenzimplementierung*. Master's thesis, Westsächsische Hochschule Zwickau, November 2008.

[34] Marcus Alanen, Ivan Porres. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, Turku Centre for Computer Science Åbo Akademi University, 2003.

[35] David Harel, Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[36] C. K Ogden. *The meaning of meaning;: A study of the influence of language upon thought and of the science of symbolism, (International library of psychology, philosophy, and scientific method)*. Harcourt, Brace & Co, later edition edition, 1923.

[37] Lennart C. L. Kats, Eelco Visser, Guido Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In *OOPSLA*, 918–932. 2010.

[38] S. Helsen K. Czarnecki. Feature-based survey of model transformation approaches, 2006.

[39] Simon Helsen Krzysztof Czarnecki. Classification of Model Transformation Approaches, 2003.

[40] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification*, January 2011. http://www.omg.org/spec/QVT/1.1/.

[41] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan De Lara, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, et al. Model Transformations by Graph Transformations: A Comparative Study, 2005.

[42] Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1 edition, 9 1998. ISBN 9780201571684.

[43] Christian Hofer, Klaus Ostermann, Tillmann Rendel, Adriaan Moors. Polymorphic embedding of dsls. In *GPCE*, 137–148. 2008.

[44] Markus Völter. MD* Best Practices. *Journal of Object Technology*, 8(6):79–102, 2009.

[45] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software (DOMAIN-DRIVEN DESIGN: TACKLING COMPLEXITY IN THE HEART OF SOFTWARE) BY Evans, Eric(Author) on Aug-30-2003 Hardcover*. Addison-Wesley Professional, 8 2003.

[46] M. P. Ward. Language Oriented Programming. *Software-Concepts and Tools*, 15(4):147–161, 1995.

[47] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm, 2004.

[48] Konstantin Solomatov Markus Voelter. Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In *SLE*. 2010.

[49] Markus Völter. Modellgetriebene, Komponentbasierte Softwareentwicklung, Teil 1. *JavaMagazin*, 10, 2005.

[50] Markus Voelter. Embedded Software Development with Projectional Language Workbenches. In Dorina C. Petriu, Nicolas Rouquette, Øystein Haugen, editors, *MoDELS (2)*, volume 6395 of *Lecture Notes in Computer Science*, 32–46. Springer, 2010. ISBN 978-3-642-16128-5.

[51] Markus Völter. Language and IDE Modularization, Extension and Composition with MPS. In *GTTSE*. 2011.

[52] Markus Völter. Architecture as Language, Part 2: Expressing Variability. *voelter.de*, 2008.

[53] Iris Groher, Markus Völter. Aspect-Oriented Model-Driven Software Product Line Engineering. *T. Aspect-Oriented Software Development VI*, 6:111–152, 2009.

[54] Krzysztof Czarnecki, Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN 978-0-201-30977-5.

[55] Markus Völter. Modellgetriebene, Komponentbasierte Softwareentwicklung, Teil 2. *JavaMagazin*, 11, 2005.

[56] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley Professional, 1 edition, 10 1998. ISBN 9780201824674.

[57] Jack Greenfield, Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In Ron Crocker, Guy L. Steele Jr., editors, *OOPSLA Companion*, 16–27. ACM, 2003. ISBN 1-58113-751-6.

[58] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley Professional, 2 edition, 11 2002. ISBN 9780201745726.

[59] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[60] David Harel, Amir Pnueli, Jeanette P. Schmidt, Rivi Sherman. On the Formal Semantics of Statecharts (Extended Abstract). In *LICS*, 54–64. 1987.

[61] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[62] Ines Nötzold. *Validierung von UML Statemachine Modellen: für die Verwendung mit den GeneSEZ State machine Metamodell*. Bachelor thesis, Westsächsische Hochschule Zwickau, 2009.

[63] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, volume 2 edition, 9 2000. ISBN 9780471606956.

[64] Miro Samek. *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. CRC Press, 2 edition, 10 2008. ISBN 9780750687065.

[65] Nguyen Trung Thanh. A Novel Implementation for UML StateMachine and Some Issues to Improve State Machine Semantics.

[66] Kevin Lano, David Clark. Direct Semantics of Extended State Machines. *Journal of Object Technology*, 6(9):35–51, 2007.

[67] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, volume 4 edition, 5 2007. ISBN 9780470059029.

[68] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys*, 29–42. 2006.

[69] Oliver Kasten, Kay Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN*, 45–52. 2005.

[70] Florence Maraninchi, Yann Rémond. Argos: an automaton-based synchronous language. *Comput. Lang.*, 27(1/3):61–92, 2001.

[71] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (Addison-Wesley Professional Computing)*. Addison-Wesley, München, 3rd ed. edition, 5 2005. ISBN 9780321334879.

[72] Ralf Schneeweiß. *Moderne C++ Programmierung: Klassen, Templates, Design Patterns (Xpert.press) (German Edition)*. Springer, 2. überarb. aufl. 2012 edition, 4 2012. ISBN 9783642214288.

[73] David Vandevoorde, Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, 11 2002. ISBN 9780201734843.

[74] Gordon Letwin. *Inside OS/2*. Microsoft Press, first edition edition, 2 1988. ISBN 9781556151170.

[75] Herb Sutter, Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 1 edition, 11 2004. ISBN 9780321113580.

[76] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman, Amsterdam, 12 1995. ISBN 9780201633719.

[77] Marc Gregoire, Nicholas A. Solter, Scott J. Kleper. *Professional C++ (Wrox Professional Guides)*. Wrox, 2 edition, 10 2011. ISBN 9780470932445.

[78] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference (2nd Edition)*. Addison-Wesley Professional, 2 edition, 4 2012. ISBN 9780321623218.

[79] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, 1 edition, 6 2001. ISBN 9780201749625.

[80] Tara Krishnaswamy. Automatic Precompiled Headers: Speeding up C++ Application Build Times. In *WIESS*, 57–66. USENIX, 2000. ISBN 1-880446-15-4.

[81] Andrei Alexandrescu. Traits: The else-if-then of Types. *C++ Report*, 12(4)(12), April 2000.

[82] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 1 edition, 2 2001. ISBN 9780201704310.

[83] Stanley B. Lippman, editor. *C++ Gems: Programming Pearls from The C++ Report (SIGS Reference Library)*. SIGS, 320 edition, 12 1997. ISBN 9780135705810.

[84] Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 7 1991. ISBN 9780262610742.

[85] David Abrahams, Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 12 2004. ISBN 9780321227256.

[86] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley Professional, 11 1999. ISBN 9780201615623.

[87] Qt State Machine Framework. http://qt-project.org/doc/qt-5.0/qtcore/statemachine-api.html.

[88] Andreas Huber Dönni. Boost.Statechart, 2007. http://www.boost.org/doc/libs/1_54_0/libs/statechart/doc/.

[89] Christophe Henry. Meta State Machine (MSM), 2010. http://www.boost.org/doc/libs/1_54_0/libs/statechart/doc/.

[90] Daniel Michel. *Code Generator for UML State Machines*. Semester thesis, HSR University of Applied Science Rapperswil MRU Software and Systems, 2011.

[91] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2nd revised edition, 12 2008. ISBN 9780321331885.

[92] Jack Herrington D. *Code Generation in Action*. Manning Publications, revised edition, 7 2003. ISBN 9781930110977.

Selbständigkeitserklärung gem. § 21 Absatz 5 MPO

Hiermit versichere ich, Peter Huster, dass ich die vorliegende Masterarbeit mit dem Titel

Generating C++ out of UML state machines

selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Zwickau, 11.11.2013

Peter Huster