Tom Hauswald

Automatic Code Synthesis of UML/SysML State Machines for airborne Applications

August 15, 2016

supervised by:
Dr. Sandro Schulze
Prof. Dr. Sybille Schupp
Dipl.-Ing. Detlef Götting

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Statutory Declaration

I, Tom Hauswald, solemnly declare that I have written this bachelor thesis independently, and that I have not made use of any aid other than those acknowledged in this bachelor thesis. Neither this bachelor thesis, nor any other similar work has been previously submitted to any examination board.
Abstract

This thesis covers the design and implementation of a code generator capable of transforming behaviours modelled with UML/SysML state machines to equivalent ANSI-C code that can be used in avionic systems. In order to decouple our code generator from a specific modelling tool, we choose the XML metadata interchange (XMI) format for input model specifications, which is supported across most popular tools. After motivating the use of model-based techniques in avionic systems engineering, we discuss several design aspects of the code generator. In the main part of the thesis we then elaborate a set of transformation rules and describe their implementation in the code generator. In the last part of the thesis, we validate the implemented code generator. We first perform a static code analysis on the generated code in order to show its compliance to relevant coding guidelines. Next, we validate the correctness of the implemented code mapping by performing a series of directed tests. Finally, we demonstrate the robustness of the code generator by providing a number of erroneous input models.
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Structure</td>
<td>2</td>
</tr>
<tr>
<td>1.3. Related Work</td>
<td>3</td>
</tr>
<tr>
<td>2. Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1. Model-based Systems Engineering</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1. Model-based Code Generation</td>
<td>5</td>
</tr>
<tr>
<td>2.2. Unified Modelling Language</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1. Systems Modelling Language</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2. UML/SysML State Machines</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3. XML Metadata Interchange</td>
<td>8</td>
</tr>
<tr>
<td>3. Design</td>
<td>11</td>
</tr>
<tr>
<td>3.1. Scope</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1. States</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2. Pseudostates</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3. Transitive Closure</td>
<td>17</td>
</tr>
<tr>
<td>3.1.4. Transitions</td>
<td>18</td>
</tr>
<tr>
<td>3.1.5. Events</td>
<td>18</td>
</tr>
<tr>
<td>3.1.6. Actions</td>
<td>19</td>
</tr>
<tr>
<td>3.2. State-based Code Patterns</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1. Implicit Representation (Table-based)</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2. Explicit Representation (Switch-based)</td>
<td>21</td>
</tr>
<tr>
<td>3.3. Code Mapping</td>
<td>21</td>
</tr>
<tr>
<td>3.3.1. Involved Entities</td>
<td>23</td>
</tr>
<tr>
<td>3.3.2. Generated Files</td>
<td>25</td>
</tr>
<tr>
<td>3.3.3. Generated Data Structures</td>
<td>26</td>
</tr>
<tr>
<td>3.3.4. State Machine Controller Implementation</td>
<td>34</td>
</tr>
<tr>
<td>4. Implementation</td>
<td>45</td>
</tr>
<tr>
<td>4.1. Frontend</td>
<td>45</td>
</tr>
<tr>
<td>4.2. Backend</td>
<td>46</td>
</tr>
<tr>
<td>5. Validation</td>
<td>49</td>
</tr>
<tr>
<td>5.1. Static Code Analysis</td>
<td>49</td>
</tr>
<tr>
<td>5.1.1. Results</td>
<td>49</td>
</tr>
<tr>
<td>5.2. Directed Testing</td>
<td>51</td>
</tr>
<tr>
<td>5.2.1. Stimulation Sequences</td>
<td>52</td>
</tr>
<tr>
<td>5.2.2. Test Cases</td>
<td>52</td>
</tr>
</tbody>
</table>
5.2.3. Results .......................................................... 53
5.3. Robustness Testing ............................................... 56
  5.3.1. Test Cases .................................................. 56
  5.3.2. Results .................................................... 57

6. Conclusion ......................................................... 59

7. Future Work ...................................................... 61

A. Figures .............................................................. 63
   A.1. Example State Machine .................................... 63
   A.2. Test Cases .................................................. 65
      A.2.1. Directed Testing ...................................... 65
      A.2.2. Robustness Testing .................................... 69

B. Listings ............................................................ 71
   B.1. Generated Example Code .................................. 71
   B.2. Test Case Stimulation Sequences ......................... 77

Bibliography ........................................................ 79
List of Figures

1.1. Envisioned MBSE process at Airbus. .............................. 2
2.1. Code generation using a handcoded code generator. .............. 6
2.2. Code generation using a template processor. ..................... 7
2.3. Scope of the SysML standard. .................................. 8
2.4. Abstract syntax of UML/SysML state machines. .................. 9
3.1. Examples of simple states: Atomic (left) and final. ............ 12
3.2. Example of a sequential state. .................................. 12
3.3. Example of a concurrent state. .................................. 13
3.4. Example of a junction pseudostate. ................................ 14
3.5. Example from Figure 3.4 expressed without junction. .......... 15
3.6. Example of fork and join pseudostates. ......................... 16
3.7. Relationship of the generated entities. .......................... 23
3.8. Generated Files ................................................. 25
3.9. UmlOperation class diagram. ..................................... 34
3.10. UmlOperation instance for ATM_appendUserInput function. ............... 35
3.11. State entry function pseudocode. ............................... 38
3.12. State exit function pseudocode. ................................ 39
3.13. Shared parent region pseudocode. ............................... 40
3.14. Transition function pseudocode. ................................ 40
3.15. Region event processor function pseudocode. .................... 44
4.1. Translate XMI document function pseudocode. .................... 46
4.2. Class diagram showing the subclasses of CodeGenItem. ........ 47
4.3. CodeGenState machine function pseudocode. ..................... 48
A.1. Example state machine model of an ATM. ......................... 63
A.2. Package hierarchy of ATM example. .............................. 64
A.3. Internal logic of Login sub-statemachine. ........................ 64
A.4. TC_SSimple. .................................................... 65
A.5. TC_SSequential. .................................................. 65
A.6. TC_SConcurrent. .................................................. 65
A.7. TC_SSeqConcNest. ............................................... 65
A.8. TC_SConcSeqNest. ................................................ 66
A.9. TC_SSubmachine. .................................................. 66
A.10. TC_PJunction. .................................................... 67
A.11. TC_PHistory. ...................................................... 67
A.12. TC_TInternal. ...................................................... 68
A.13. TC_TLocal. ........................................................ 68
A.14. TC_TExternal. ...................................................... 68
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.15</td>
<td>TC_TGuard</td>
<td>68</td>
</tr>
<tr>
<td>A.16</td>
<td>TC_TECompletion</td>
<td>68</td>
</tr>
<tr>
<td>A.17</td>
<td>TC_EChange</td>
<td>68</td>
</tr>
<tr>
<td>A.18</td>
<td>TC_ETime</td>
<td>68</td>
</tr>
<tr>
<td>A.19</td>
<td>TC_TECall</td>
<td>68</td>
</tr>
<tr>
<td>A.20</td>
<td>TC_ESignal</td>
<td>68</td>
</tr>
<tr>
<td>A.21</td>
<td>TC_RFork</td>
<td>69</td>
</tr>
<tr>
<td>A.22</td>
<td>TC_RJoin</td>
<td>69</td>
</tr>
<tr>
<td>A.23</td>
<td>TC_RDeepHistory</td>
<td>69</td>
</tr>
<tr>
<td>A.24</td>
<td>TC_RNoInitial</td>
<td>69</td>
</tr>
<tr>
<td>A.25</td>
<td>TC_RTwoInitials</td>
<td>69</td>
</tr>
<tr>
<td>A.26</td>
<td>TC_RTwoHistories</td>
<td>70</td>
</tr>
</tbody>
</table>
## List of Code Listings

3.1. Example of implicit state machine representation in C ........................................ 22
3.2. General structure of generated enumerations ...................................................... 27
3.3. Regions enumeration of ATM example ................................................................. 28
3.4. States enumeration of ATM example ................................................................. 28
3.5. Events enumeration of ATM example ................................................................. 29
3.6. Signals enumeration of ATM example ............................................................... 29
3.7. Callmodes enumeration of ATM example ......................................................... 29
3.8. Event queue data structure of ATM example ..................................................... 30
3.9. Event queue public interface .............................................................................. 31
3.10. Timer data structure of ATM example ............................................................... 31
3.11. Timer data structure public interface .................................................................. 32
3.12. Definition of the host data structure ................................................................... 32
3.13. Definition of the state machine internal data structure ........................................ 33
3.14. Definition of the state machine data structure ................................................... 33
3.15. Generated implementation of user-defined `ATM::appendUserInput` function ........ 36
3.16. Generated `guard` function for the transition from Initial to Sync ..................... 41
3.17. Generated `changeExpr` function for the transition from Initial to Sync ................. 41

5.1. Illustration of forward-declaration problem .......................................................... 51
5.2. Specification format of test stimuli ...................................................................... 52
5.3. Example of a stimulation sequence .................................................................... 52

B.1. Generated entry function of state Initial in region Lane A .................................... 71
B.2. Generated exit function of state Self Test inside region Main ............................. 72
B.3. Generated transition function for the transition from Self Test to Failure .............. 73
B.4. Generated `sendSignal` function ...................................................................... 73
B.5. Generated createChangeEvent function ............................................................. 74
B.6. Generated processEvent function for the main region of the Login submachine state. (Excerpt) ............................................................... 75
B.7. Generated state machine update function ............................................................ 76
B.8. Empty stimulation sequence ............................................................................. 77
B.9. Stimulation sequence for TC_EChange ............................................................ 77
B.10. Stimulation sequence for TC_ECall ................................................................. 77
B.11. Stimulation sequence for TC_ESignal ............................................................. 77
B.12. Stimulation sequence for TC_ETime ............................................................... 77
B.13. Stimulation sequence for TC_PJunction ........................................................... 77
1. Introduction

1.1. Motivation

While in the traditional environment of requirements-based systems engineering (RBSE) the properties and behaviour of a system have to be described by means of requirements specified in natural language, recent model-driven approaches to systems engineering accomplish this task using a standards compliant model representation of the system. Not only do engineers therefore no longer need to rely on ambiguous and misunderstandable natural language to express system requirements, but many other advantages arise as well, when employing model-based systems engineering (MBSE) techniques.

First of all, systems can be modelled in graphical languages, lowering the entry barrier of system specification tasks. Also, visual model representations can help teams of engineers to understand designed systems more easily and thoroughly. Furthermore, the possibility arises for engineers to reuse large parts of previous work, as completed components from the model library can be integrated into new projects without additional effort. All of the aforementioned properties of MBSE allow it to make systems engineering both less error-prone and much more time-efficient. For the reasons stated above, MBSE methods have nowadays become widely accepted in the industry. [Rum12, Wei06, Sch06, RSRB06, Est07, FGS07]

In addition to that, we are able to exploit the formal nature of graphically modelled system specification models in order to automate parts of the system implementation, simulation and verification phases. This yields yet another notable increase in efficiency, as the manual programming and testing workload is drastically reduced. Auto-generating code from system models is one way of automating system implementation and thus delivers the aforementioned benefits. Because of that, "automatically generated code can and is being used today in a variety of hard real time and embedded systems" [Dou99, p. 156].

Embedded systems in avionics often implement event-driven, state-based behaviour. Because UML state diagrams are used to describe exactly that type of behaviour in a standardized and easily comprehensible way, their automatic implementation and execution are of special interest for aviation engineers. In Figure 1.1, we show how automatic code generation integrates with the MBSE process envisioned by Airbus. The automatic code synthesis (ACS) step fills the gap between the theoretical model of the system and its physical implementation. Thereby, the required manual programming work is minimized. Also, an additional verification of the automatically generated code is only required until the ACS tools get qualified according to the DO-330 guidelines. After this qualification we can safely assume any generated code to be flight-ready, as is, resulting in a game-changing improvement regarding the efficiency of airborne software systems engineering.

Even though there already exist a handful of modelling tools that are able to automatically generate source code from state charts, the employed code generators are not designed with airborne software in mind. More specifically, they can neither ensure the
necessary software standards compliance nor enable the backtraceability of system requirements from the generated code to the original model. The described lack of critical features prevents the successful embedding of existing code generators for UML state machines in the engineering process of safety-critical and subject-to-authorization avionic systems.

Motivated by the aforementioned reasons, this bachelor thesis covers the design, implementation and subsequent validation of a code generator for UML/SysML state machines. Most importantly, a set of transformation rules is elaborated that maps input UML models to flight-ready ANSI-C code. This transformation algorithm will be designed such that it satisfies the aforementioned requirements and can therefore be embedded in the engineering process of airborne software systems at Airbus. Special attention will be paid to the compliance to relevant guidelines and coding standards for avionic software (DO-178C, MISRA-C). Also, we will enable the traceability of system requirements from inside the code back to the original model, which allows for an easier verification of the generated code prior to a DO-330 qualification of the code generator.

1.2. Structure

After introducing the reader to the background concepts encountered in this thesis, we start off by discussing the major design considerations for the code generator in Chapter 3. First of all, in order to provide a reasonable framework for this thesis, we define a sensible subset of UML/SysML modelling features that our code generator will support. As soon as we decided upon an allowed set of elements to be contained in input models, we work out a set of rules that concisely describes how each of these element is mapped to source code. In order to elaborate such a mapping, we first decide on a code pattern to use for the implementation of the modelled state-based behaviour.

After implementing the elaborated set of transformation rules as code generator in Chapter 4, we validate the correctness of our transformation in Chapter 5. We achieve this

Figure 1.1.: Envisioned MBSE process at Airbus.
1.3. Related Work

In this section, we describe other work related to the code generation based on UML/SysML state machine models. Due to the actuality of model-based approaches to system engineering, there exist many publications focussing on automatic code generation. However, because most proposed code generation solutions do not target safety-critical embedded systems, they predominantly produce outputs in object-oriented languages like C++ or Java. This holds true, for example, for the approach presented by Niaz and Tanaka [NT03, NT +04]. Benowitz et al. describe the code generator that is part of the state-based architecture and auto-coding for real-time systems (STAARS) process at the NASA Jet Propulsion Laboratory (JPL) [BCW06]. Despite of the presented solution being designed specifically for embedded architectures, it still generates object-oriented C++ code that relies on the Quantum [QL] framework.

A topic that is very closely related to the automatic model-based code generation is the validation of the correctness of system models. We refer to this validation process as model-checking. In the work published by Knapp and Merz [KM02], the authors describe the design of the HUGO toolchain. HUGO provides a model checker that internally uses the SPIN model checker developed by Holzmann [Hol97].
2. Background

This chapter will introduce the reader to the most important background concepts used in this thesis. Section 2.1 explains the idea behind model-based systems engineering, including the concept of model-based code generation. Section 2.2 describes relevant standards for this thesis and introduces the term UML/SysML state machine.

2.1. Model-based Systems Engineering

Model-based approaches to systems engineering have first risen public awareness after an initiative was started by INCOSE in 2007 [FGS07, Est07]. The central idea behind MBSE methods is to replace the traditional requirements-based approaches by techniques that center around a standardized model of a system. The benefit of this approach is that the system requirements are implicitly defined by the model and thus do not need to be expressed in easily misunderstandable natural language.

Especially over the past decade, MBSE methodologies have matured and are nowadays used in many branches of cyberphysical systems engineering [Est07, Rum12]. Despite this positive development, the adoption of MBSE techniques in the aviation industry has not yet advanced as far as in comparable branches involving embedded systems engineering. This delay results from the intrinsically high safety and security requirements imposed on airborne systems.

2.1.1. Model-based Code Generation

Part of the MBSE process is the automatic implementation of a system based on its high-level model description. In terms of software systems, we refer to this automatic implementation as model-based code generation. Because the model of a system fully describes its execution semantics, MBSE tools are able to perform a model-to-text transformation to generate code that simulates the modelled behaviour. Given that the transformation is formally correct, the generated code does not need to be examined or modified. As a consequence, modelling languages take on the role of implementation languages in the context of MBSE. This paradigm shift is comparable to how assembly languages were supplanted by higher-level programming languages, after compilers had been introduced. [Sel03, FGS07]. According to Dalgarno, the four major advantages of automatic code generation can be summarized as follows [Dal06].

- Quality: The quality of the target code is fully controlled by the quality of the code generator. Thus, the optimization of a single program leads to quality improvements of the whole code base.
- Consistency: The code generator enforces code guidelines and e.g. naming conventions which leads to a highly consistent, homogenous code base.
- Productivity: Because code generation alleviates manual programming and validation work, the systems engineering process gets more productive.
Abstraction: Because code generators typically operate on abstract input models, they can be engineered to generate multiple different output fragments from the same input specification.

Model-based code generators usually comprise three major components. First, an importer is needed that can read the input model format. Next, the input model has to be parsed, creating an internal object model (IOM) representation. The code generator then processes this internal IOM with respect to a set of rules that defines the mapping from model to code. In a last step, the generated code is exported in a chosen output format. Despite of the same basic structure that all code generators share, we can still distinguish between two general categories. The first category of code generators is designed for one particular transformation, allowing their transformation rules to be implemented internally. Because code generators of this category are this specific, they offer good optimization potential. This, of course, comes at the cost of them having little to no flexibility. We refer to this first type of code generators as **handcoded**.

Code generators of the second category, on the other hand, do not implement any transformation logic themselves, but instead rely on externally specified templates. We call this kind of generators **template processors**. The advantage of template processors is their high flexibility, allowing them to be used in multiple projects. Figures 2.1 and 2.2 show the workflows of both types of code generators.

![Figure 2.1: Code generation using a handcoded code generator. Source: [Nac04](edited)](image)

2.2. Unified Modelling Language

According to its specification "the objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes" [Gro15]. It achieves this objective by defining standards for recurring concepts...
in software modelling, such as various types of actors and their behaviours. Especially interesting for this thesis is the specification of UML state diagrams or state charts. These diagrams can be used to formally express the state-based behaviour of a system.

2.2.1. Systems Modelling Language

The Systems Modelling Language (SysML) [Gro15b] is a modelling language based on the UML 2.0 specification. The SysML standard contains a subset of features described by the UML standard, referred to as UML4SysML, and is extended by some SysML-specific concepts, like system requirements. Also, a number of existing UML diagrams, e.g. activity and block diagrams are refined in SysML to better accommodate for systems engineering tasks. In Figure 2.1, we illustrate this correlation between UML and SysML. In this work, we are interested in the SysML language, as it allows us to embed externally engineered system requirements into a modelled UML state machine as SysML components. Our code generator can use this requirement information to allow for code-to-model backtraceability.

2.2.2. UML/SysML State Machines

According to the UML standard, UML state machines are used to "model discrete event-driven behaviours of a system using a finite state-machine formalism" [Gro15c][p. 345]. They can also be used to express valid interaction sequences, referred to as protocols. The terms behaviour state machine and protocol state machine are used to distinguish between the two types. As for this thesis, we are primarily interested in behaviour state machines, because we want to generate code that implements the behaviour of a modelled system. A behaviour state machine satisfies the abstract syntax shown in Figure 2.4. On the highest level, the state machine comprises one or more regions. Each of these regions contains a set of states, interconnected by transitions. We describe the structure of state machine models in more detail in Section 3.1.
2. Background

Because UML state machines are part of UML4SysML (see Section 2.2.1) and thus the SysML standard, they may also be referred to as SysML state machines. For this reason, we use the term *UML/SysML state machine* in this thesis.

2.2.3. XML Metadata Interchange

The XML Metadata Interchange (XMI) format is a standardized data exchange format whose specification is maintained by the Object Model Group (OMG) [Gro05]. The general purpose of XMI is the exchange of metadata in the form of instances of *meta-meta-models*. The only restriction on such meta-meta-models is that they must be expressable in *Meta-Object Facility* (MOF) [Gro15a]. Due to the fact that the meta-models of UML and SysML are specified in MOF, we can use the XMI format to represent the system models that contain our state machines. Choosing XMI as representation format for our state machine models has the advantage of decoupling them from the specific tool involved in their design. This effect stems from the import and export capabilities for XMI-encoded models provided by most major UML modelling tools, including the *PTC Integrity Modeler* software used for the modelling in this thesis. Another advantage of XMI is that it can be parsed and modified easily, using existing high-level APIs, because it is built on top of the popular XML standard. The possibility to losslessly store and distribute our models in XMI format, as well as the ease of its processing, make XMI a good choice of input format for our code generator.

---

Figure 2.3.: Scope of the SysML standard. *Source:* [Est07]
Figure 2.4.: Abstract syntax of UML/SysML state machines. Source: [Gro15c][p. 346] (modified)
3. Design

In this chapter, we will elaborate the design of a code generator that satisfies the requirements described in Chapter 1.1. After specifying, which diagram elements the code generator should support, in Section 3.1, we choose a general pattern for the generated code in Section 3.2. Based on these decisions, we then define a mapping from supported model elements to source code. This code mapping is described in detail in Section 3.3 and will be implemented in Chapter 4. Furthermore, the design decisions we make in this chapter contribute to Airbus internal modelling guidelines for future MBSE projects.

3.1. Scope

The first design consideration that we need to take into account, is the overall scope of the code generator. More specifically, we need to decide on the set of state diagram elements that we choose to allow to be included in input models. Therefore, we present the list of state diagram elements as per UML 2.5 specification [Gro15c] and discuss, whether we require the code generator to support them. We base this decision on multiple factors. First of all, we only allow features that can be mapped to the embedded target hardware. Also, there already exist experiences with the modelling of state-based behaviour in avionics, which we take into account, as well. Finally, the tradeoff between modelling comfort and implementation effort is a crucial aspect for our design consideration. We want to keep the transformation complexity as low as possible without notably increasing the modelling effort. For the same reason, we also elaborate additional restrictions that we enforce on future models. Such restrictions are sensible for different reasons. Most importantly, they help to increase the definiteness of system models, making them easier to understand for engineers. Other rules help to simplify the code generation process, while not notably increasing the modelling effort. In order to enforce regulatory conformity of future models, the introduced rules are integrated as part of the Airbus internal MBSE guidelines.

Sections 3.1.1 - 3.1.6 discuss the existing types of (pseudo-)states, transitions, events and actions with respect to their relevance for our code generator.

3.1.1. States

States are the fundamental building blocks of a state machine. Each state represents a different context of execution for the modelled system. One state has to be assumed inside each active region, at all times. In this section, we will examine the different types of states listed in the UML 2.5 specification [Gro15c].

- Simple States
- Composite States
- Submachine States
3.1.1. Simple States

Simple states are the most basic type of states. Simple states are completely self-contained, i.e. they do not contain any child states or inner transitions. Simple states can either be atomic or final states. While atomic states have no special meaning, final states, when active, indicate the completion of their parent region. On a more abstract level, the assumption of a final state inside a region represents the admissibility of a given input sequence. Figure 3.1 shows the transition from the atomic state Atomic to a final state.

![Figure 3.1: Examples of simple states: Atomic (left) and final.](image)

3.1.1.2. Composite States

Composite states build on top of simple states and may contain one or more orthogonal regions. Each such region contains any number of arbitrarily deeply nested states, as well as transitions connecting them. We call a composite state sequential state, given that it contains exactly one region and concurrent state, otherwise. Figures 3.2 and 3.3 show examples of sequential and concurrent states.

![Figure 3.2: Example of a sequential state.](image)

Because the nesting of states is necessary, in order to express any non-trivial behaviour in a visually comprehensible way, our code generator needs to support sequential and orthogonal states. Also, from now on we will consider a state machine equivalent to a
3.1. Scope

Figure 3.3.: Example of a concurrent state.

composite states with one orthogonal region. We call the region that encompasses the whole state-based behaviour the main region of the state machine.

3.1.1.3. Submachine States

Finally, submachine states allow a previously modelled UML state machine to reoccur as part of a larger state machine. Because submachine states thereby promote the reuse of existing models, they lead to a more efficient systems design process. This is a highly desired effect, hence why we choose to support submachine states, as well.

3.1.2. Pseudostates

In this section, we will have a look at the different kinds of pseudostates. The difference between states and pseudostates is that the latter can not be assumed by the state machine. Instead, pseudostates serve as shortcuts for the modeller to express complex behaviour in a much clearer way than with purely traditional states. Some pseudostates substitute large numbers of states and transitions in the state machine diagram. Others allow features to be modelled that would require the model engineer to adhere to additional guidelines, if he had to express them without pseudostates.

The following seven types of pseudostates must be taken into consideration.

- Initial States
- Junctions
- Choices
- Forks and Joins
3. Design

- Entry and Exit Points
- History states
- Termination States

3.1.2.1. Initial States

The most basic pseudostate is the initial state. An initial state is required in each orthogonal region of a composite state and is implicitly assumed with the start of the execution of the region. Also, a single outgoing transition from the initial state must be defined, that does neither have a trigger, nor a guard assigned to it. Figures 3.2 and 3.3 demonstrate the correct use of initial states. As we require initial states in order to correctly model composite states, our code generator needs to support them.

3.1.2.2. Junction Pseudostates

A junction pseudostate is used to form \( m \times n \) compound transitions from any combination of \( m \) incoming and \( n \) outgoing transitions (\( m, n \in \mathbb{N} \)). Each of the resulting compound transitions \( T_{ij} \), \( 0 \leq i < m \), \( 0 \leq j < n \) combines the trigger, guard and effect of the incoming transition \( T_i \) with the guard and effect of the outgoing transition \( T_j \). The outgoing transitions must not depend on any triggers. Only when both, the guard of the incoming transitions, and that of the outgoing transition, evaluate to true, a compound transition is taken. Because both guards have to be statically known to be true, beforehand, we refer to junctions as static conditional branches. In Figures 3.4 and 3.5, we show how the same behaviour is modelled with and without a junction. We choose to support junction pseudostates, as they can drastically reduce the required number of hand-modelled transitions.

![Figure 3.4: Example of a junction pseudostate.](image)

3.1.2.3. Choice Pseudostates

Very similar to junctions are choice pseudostates. In contrast to a junction, a choice implements a dynamic conditional branch. This means that incoming transitions to choices
3.1. Scope

![Diagram of state machine with states and transitions labeled with conditions.](image)

Figure 3.5.: Example from Figure 3.4 expressed without junction.

are taken, whenever possible, without checking in advance, whether any outgoing transition guard evaluates to true. If the model allows for situations to occur, in which an incoming transition to the choice is taken and no outgoing transition guard evaluates to true, we consider it ill-formed. The advantage of a dynamic conditional branch is that the outgoing transition guards can depend on the effect of the taken incoming transition. In other words, choice pseudostates provide additional contextual information. Existing modeling experiences at Airbus show that choice pseudostates are rarely required. Also, where applicable, junctions are considered the better practice, as they guarantee compound transitions to only be taken as a whole. For the reasons stated, we decide that our code generator will not support choice pseudostates. Instead, conditional compounds should be modeled using junction pseudostates.

3.1.2.4. Fork and Join Pseudostates

A fork pseudostate can be used in order to explicitly activate one or more orthogonal regions inside a composite state. The explicit activation of a region causes the state machine to skip the initial state of the region and directly assume the target state. This differs from the default activation of a region, which causes the state machine to assume the initial state of the region. When a state machine encounters a fork pseudostate, the explicit activation of the targeted orthogonal regions takes place. Default activation occurs for the remaining orthogonal regions. On the left side of Figure 3.6, we show a fork pseudostate, as well as transitions from it to inside the composite state Composite. Taking the outgoing transitions from the fork leads to the default activation of region B and explicit activations of regions A and C. More specifically, the state machine assumes states Foo and Final inside regions A and C, skipping the respective initial states. The execution of region B begins regularly at its initial state.

Analogously, join pseudostates allow a subset of the orthogonal regions of a composite state to be exited explicitly, i.e. before their respective final states were reached. According to the UML specification, transitions to a join pseudostate may only be taken, as soon as the guard expressions of all incoming transitions to the join pseudostate evaluate to true. We show an example of a join pseudostate on the right side of Figure 3.6. In the case of the shown example, the join pseudostate is entered, as soon as states Foo and Bar
are assumed at the same time. The assumed state inside orthogonal region $C$ is ignored. As can be seen in the provided examples, fork and join pseudostates introduce a lot of additional complexity. Furthermore, fork pseudostates can reduce the clearness of intent, as they quickly obscure the graphical representation of the model. For the reasons stated, we choose to disallow forks and joins as parts of valid input models, for now. Support for them may be added during future improvements to the code generator.

![Figure 3.6: Example of fork and join pseudostates.](image)

### 3.1.2.5. Submachine Entry and Exit Points

As a result of the discussion in Section 3.1.1, we decided to support submachine states. The goal of submachine states is to allow model engineers to consider existing state machines as *black boxes*. In order to achieve this goal, we need to allow model engineers to specify a public interface for their state machines. For this reason, *entry* and *exit points* exist. A state machine may have several entry and exit points assigned that lead to different paths of execution inside it. Whenever a state machine is instantiated as submachine state inside a model, its entry points become valid target vertices and its exit points valid source vertices for transitions. The example state machine we show in Figure A.1 instantiates the `Login` state machine from Figure A.3 as a submachine state. As a consequence, the entry and exit points of the `Login` state machine become available transition vertices inside Figure A.1.

### 3.1.2.6. History States

Another kind of pseudostate that UML state machines may contain, are *history states*. A history state must be located inside a region of a composite state and only one history
3.1. Scope

A state is permitted per region. The UML standard distinguishes between two kinds of history states, shallow and deep history states. A shallow history state remembers the most recently active child state inside its parent region. A transition that targets a shallow history state will, during execution, target its latest remembered state. In addition, each shallow history state needs an associated default shallow history state. If no state was assumed yet in the parent region of the history state, and a transition is taken to the history state, its default state is assumed. Deep history states, on the other hand, do not only remember the last assumed state inside their parent region. Instead, they store the complete most recent state configuration inside the region, that is, the assumed states of all nested composite states. Because of this, deep history states can create arbitrarily high memory requirements, when used in deep state hierarchies. Also, existing modelling projects have shown that shallow history states suffice for typical avionic applications. Therefore, in terms of this thesis, we choose not to support deep history states, but only shallow ones. Likewise to fork and join pseudostates, deep history states may be added during future improvements to the code generator.

3.1.2.7. Termination Pseudostates

Finally, termination states are pseudostates that allow the execution of a state machine to be terminated, instantly, at any point of time. For safety reasons, we do not allow the abrupt termination of a state machine in such a way. Instead, the assumption of a well-defined error state is preferred. For the reasons stated, termination states are not supported.

3.1.3. Transitive Closure

In this section, we introduce the term transitive closure. According to Definition 3.1, the transitive closure of a state s consists of the s itself, combined with the transitive closures of all of its child regions. As a consequence, the transitive closure of any given non-composite state consists of only the state itself. Similarly, the transitive closure of an orthogonal region is defined by the union of the transitive closures of all of the child states of the region. We formally introduce the transitive closure of an orthogonal region r in Definition 3.2.

**Definition 3.1**

\[
\text{stateClosure}(s) := s \cup \bigcup_{r \in \text{childRegions}(s)} \text{regionClosure}(r).
\]

**Definition 3.2**

\[
\text{regionClosure}(r) := \bigcup_{s \in \text{childStates}(r)} \text{stateClosure}(s).
\]

The transitive closure of a state s can be graphically interpreted as the set of all states located within the boundaries of s inside the state chart. For example, the transitive closure of the state Composite shown in Figure 3.6 consists of the states \{ Composite, Foo, Final0, Bar, Final1, Baz, Final2 \}. 

\[\text{ss}\]
3.1.4. Transitions

The next key component of state machines are transitions. Transitions are directed links between a source and target state and can be written according to the regular expression 
\[\text{trigger}[,\text{trigger}]*\] /\text{guard}/ \text{effect}\], where:

- \text{trigger} is an existing event,
- \text{guard} is a boolean expression,
- \text{effect} is valid C code.

A transition may fire, as soon as one of the specified trigger events occurs and, at the same time, the associated guard statement evaluates to \text{true}. When multiple transitions are able to fire at the same time, the behaviour is undefined, according to the specification \cite{Gro15}. Therefore, the combinations of triggers and guards of outgoing transition should be mutually exclusive. Otherwise, the order in which the transitions occur in the XMI file decides, which transition is checked for readiness, first.

3.1.4.1. Transition Kinds

We differentiate between three kinds of transitions, based on the configuration of their source and target states. We use this section to explain these transition kinds.

First, a transition is called \textit{internal}, if both its source and target state are identical. Internal transitions do neither trigger the entry, nor the exit action of the state. (See \S 3.1.6 for an introduction to entry and exit actions.)

Next, there are \textit{local} transitions. Local transitions terminate on a target state that is contained within the transitive closure of their source state. (The term transitive closure is introduced in Section 3.1.3.) For this reason, local transitions can only originate from composite states. Because the target state of a local transition is located within its source state, no exit action needs to be invoked.

Finally, there are \textit{external} transitions. Other than local transitions, the source and target states of external transitions are not located inside a shared closure. Therefore, they cause the exit behaviour of the source state and the entry behaviour of the target state to be executed.

Our code generator is able to correctly generate the transition logic for all of the three aforementioned transitions kinds.

3.1.5. Events

The next concept that UML state machines rely on, are events. The occurrence of events is what causes the state machine to react. These reactions lead to the execution of the modelled behaviour. First of all, we have to differentiate between five types of events. There exist \textit{completion}, \textit{change}, \textit{signal}, \textit{call} and \textit{time} events. In the following paragraphs, we describe each type of event in detail.
3.1. Scope

3.1.5.1. Completion Events

A completion event is generated whenever the execution of the behaviour of a state is completed. For this reason, completion events for non-composite state are generated as soon as they are entered, because they terminate immediately. Completion events for composite states, on the other hand, can only be generated when a final state is reached within each of its child regions.

3.1.5.2. Change Events

Change events are used to notify the state machine about possible changes to any on-instance or on-class variables. A transition with a trigger of the form \texttt{when(expr)} listens to variable changes that influence the value of \texttt{expr}. Thus, change events remove the need to poll the values of all relevant variables, but allow the state machine to only reevaluate the expression, when necessary.

3.1.5.3. Signal Events

The state machine model may contain an arbitrary list of user-defined signal constants. Sending any of these signals to the state machine is one way to externally interact with it and is achieved using the \texttt{sendSignal} function that we describe in Section 3.3.4.8. Signal events are mapped to a specific signal and are triggered upon its reception.

3.1.5.4. Call Events

Call events can be associated to any of the user-defined functions. A call event is triggered, when the corresponding user-defined function is called.

3.1.5.5. Time Events

Transition triggers may also be of the form \texttt{after(duration)}. This means that the transition may only fire after the specified time has elapsed. A time event is created, accordingly, to inform the state machine about the elapsed duration.

3.1.6. Actions

Inside the modeller, the user has the ability to specify different \textit{actions} that are triggered under certain circumstances. Our code generator supports so called \textit{opaque actions}. These are C statements directly embedded into the model. In state machine models, there exist four kinds of actions. There are \textit{entry}, \textit{exit}, \textit{event} and \textit{do actions}. In the following sections, we explain the aforementioned actions.

3.1.6.1. Entry and Exit Actions

An entry and exit action can be associated to each state. These actions get triggered whenever this state is entered or exited, respectively.
3.1.6.2. Event Actions

Event actions can be associated to a state and get triggered whenever the corresponding event occurs. They are equivalent to unguarded internal transitions with the same associated trigger event.

3.1.6.3. Do Actions

Do actions can be used to model behaviour that should be performed in parallel to the default state execution. Because our primary deployment target is single-threaded embedded hardware, do actions are not of interest for this thesis.

3.2. State-based Code Patterns

Before we work out a code mapping for each of the allowed elements of input UML models, it makes sense to first decide on a general code pattern that we want to use for the implementation of state-based behaviour. This decision will influence the way in which we map certain model elements to code.

We can not choose the code pattern arbitrarily, as the generated code needs to be compliant with several relevant coding standards and authorization requirements. Thus, we first need to evaluate popular patterns with respect to these considerations. We can then decide on the method that is best applicable in an avionic environment.

3.2.1. Implicit Representation (Table-based)

An often used approach to represent finite state machines in code is the so called implicit or table-based representation. The implicit representation of the state machine uses a table of function pointers to store any state and transition logic. The approach can be extended to support entry, event and exit actions of states, as well as transition guards and effects. Dereferencing the state table at an index mapping to state \( s \) returns the state function of \( s \). The return value of the state function is the state that should be assumed next. In order to make this decision, the state function may evaluate different guard statements and check occurred events. The transition table, on the other hand, contains the transition effects of the transitions between any two states \( s \) and \( t \). Therefore, the transition table has to be dereferenced using a tuple of source and target states \((s, t)\).

An implicitly represented state machine is executed by repeatedly performing the following actions. First, the state table is dereferenced, returning the state function of the currently assumed state \( s \). Given that a call to the obtained state function returns a target state \( t \) for which there exists a transition \( s \Rightarrow t \), the transition table is dereferenced at the according indices. If a transition function was specified for the transition \( s \Rightarrow t \), it is called. Finally, the currently assumed state of the state machine instance is updated. These actions are repeated for as long as the state machine has not terminated. In Listing 3.1 we illustrate the implicit representation of a state machine.
3.3. Code Mapping

The implicit representation of state machines has the advantage of leading to very small code size, when compared to a switch-based representation. Also, a state table is very easy and fast to generate based on an input model. The downside of this approach, on the other hand, is that it breaks down quickly, as soon as we want to implement hierarchical states. This happens, because the coordinate space of the state table grows by one dimension for each orthogonal region in our model. Also, the extensive usage of function pointers poses a fundamental problem, as function pointers are considered insecure by various avionics-relevant security standards. For this reason, a table-based representation of state machines is not suitable for avionic environments.

3.2.2. Explicit Representation (Switch-based)

Another common approach to implement state-based behaviour in code is the explicit representation. The explicit representation uses a large construct of nested switch statements to encode the state and transition logic of a state machine. For each orthogonal region, a switch statement is required for the currently handled event. Each case of this switch statement consists of another switch statement that checks the currently assumed state inside the orthogonal region. The cases of this child switch statement then contain any transition guards that need to be evaluated. If the guards evaluate to true, the source state exit action, transition effect and, finally, target state exit action are invoked. We can execute an explicitly represented state machine by periodically invoking the event handler functions of each active orthogonal region.

The explicit representation of state machines has the advantage of allowing hierarchical states to be implemented, easily. Also, as opposed to the implicit representation, we do not require the usage of any function pointers. This greatly simplifies the verification of the generated code. The only downside of the switch-based approach is the bigger size of the generated code, compared to the table-based variant. Because of the aforementioned advantages, we choose a switch-based representation for our state machines. We describe the code mapping in more detail in Section 3.3.

3.3. Code Mapping

Taking into account the design goals introduced in Section 1.1, we now elaborate a mapping from each of the supported UML model elements to valid source code. We defined this set of supported model elements in Section 3.1. Based on our results from Section 3.2, an explicit, i.e. switch-based code representation is chosen for the state machines. In Section 3.3.1 we introduce the entities that our code generator generates in order to allow for the execution of the modelled behaviour. Next, in Section 3.3.2 we name the set of generated files and explain in which files the entities appear that we introduced in Section 3.3.1. Finally, in Section 3.3.4 we explain the generated state machine implementation, in detail.

In order to better visualize the implemented code mapping, we base our explanations on the example state machine model whose state chart we show in Figure A.1. The internal
typedef enum { ... , STATE_COUNT } state_t;
typedef struct { state_t state; ... } sm_t;
typedef void (* trans_func_t)(sm_t * sm);
typedef state_t (* state_func_t)(sm_t * sm);

// State function table.
state_func_t state_funcs[STATE_COUNT] =
{
&state_func_0,
&state_func_1,
...,
};

// Transition function table.
trans_func_t trans_funcs[STATE_COUNT][STATE_COUNT] =
{
/* TARGET --> */
/* SOURCE */ &trans_func_0_0, &trans_func_0_1, &trans_func_0_2, ...
/* | */ &trans_func_1_0, &trans_func_1_1, &trans_func_1_2, ...
/* v */ &trans_func_2_0, &trans_func_2_1, &trans_func_2_2, ...
...,
};

int main(int argc, char ** argv)
{
sm_t sm;
sm->state = initialState;

while(!terminated)
{
state_t next_state = state_funcs[sm->state](sm);
if(trans_funcs[sm->state][next_state] != NULL)
{
(* trans_funcs[sm->state][next_state])(sm);
}
sm->state = next_state;
}
return 0;
}

Listing 3.1: Example of implicit state machine representation in C.
structure of the contained submachine state Login is shown in Figure A.3. In Figure A.2, we additionally show the package hierarchy of the model which includes the set of defined data types, events and signals. It also shows the owning entity ATM with its attributes and functional interface.

3.3.1. Involved Entities

The generated code can be separated into multiple logical building blocks that we refer to as entities. In Figure 3.7 we show these entities and illustrate their mutual interactions via their provided interfaces. The figure also shows the generated application programmer interface (API) of the state machine. In the future, larger scale applications can integrate with auto-generated state machine implementations via this API.

![Figure 3.7: Relationship of the generated entities.](image)
3.3.1. Owning Entity

The *owning entity* is the part of the UML model that contains the state machine component, i.e. exposes the behaviour modelled by the state machine. It is used to define the interface that shall be exposed to other entities. This interface consists of an arbitrary number of *operations* that may either be publicly visible to other entities or be private to the entity itself. Additionally, the owning entity may have a number of *attributes* assigned to it. Likewise to operations, not all attributes have to be exposed to the public interface of the entity. Furthermore, we differentiate between *on-class* and *on-instance* attributes. On-class attributes share the same value for all instances of the owning entity type, while the values of on-instance attributes are local to each respective instance of the entity type.

3.3.1.2. Event Queue

The *event queue* is a data structure that we require to buffer a number of asynchronously occurring events, while preserving their temporal order. The event queue exposes, among others, *enqueue* and *dequeue* functions that allow its interaction with other entities.

3.3.1.3. Timer Management

The timer management entity allows the state machine controller to spawn and cancel timers that are required to model time events. In order to notify the state machine controller of elapsed time periods, the timer management entity interacts with the event queue by enqueuing respective time events. As the implementation of the actual timer logic is specific to the operating system of the target platform, the timer management provides a public *operating system abstraction layer (OSAL)* interface. This OSAL interface can be used to provide operating system specific timing capabilities to the timer management entity. We use the *WIN32* API as source of timing capabilities for the testing purposes in this thesis.

3.3.1.4. State Machine Controller

The state machine controller is the entity implementing the actual state and transition logic modelled by the synthesized state diagram. The controller interacts with the event queue via its *dequeue* function in order to process occurring events. The controller may also create events itself and *enqueue* them into the event queue. This mechanism is used to create change events, in order to notify the state machine instance of potential attribute changes. Furthermore, the controller interacts with the *timer management* entity, in order to handle time events. Because the state machine controller represents the core element of the generated implementation, later sections will focus primarily on this entity.
3.3. Code Mapping

3.3.2. Generated Files

In this section we explain the set of files that is generated by our code generator. As we show in Figure 3.8, the generated files are created inside a parent directory named \texttt{Ns_Name}, where both \texttt{Ns} and \texttt{Name} are placeholder strings. \texttt{Ns} is replaced by a user-specified namespace string and is required in order to prevent name clashes. Also, the use of a namespace is encouraged by Airbus-internal coding guidelines.

\texttt{Name} is the name that was given to the state machine inside the modelling tool. Both placeholders \texttt{Ns} and \texttt{Name} will reoccur in Section 3.3.4, where we describe the patterns of auto-generated code elements.

A common practice in software design for embedded systems is the separation of interface definitions into public and private header files. The public header files contain all declarations that shall be exposed to other compilation units, while the visibility of declarations inside the private header file is limited to the implementing compilation unit. For this reason, we suffix our header files by \_\texttt{private} and \_\texttt{public} accordingly, in order to display whether they contain public or private interface declarations.

\begin{verbatim}
Ns_Name/
  \_Ns_Name_shared.h
  \_Ns_Name_sm_private.h
  \_Ns_Name_sm_public.h
  \_Ns_Name_sm.c
  \_Ns_Name_host_private.h
  \_Ns_Name_host_public.h
  \_Ns_Name_host.c
  \_Ns_Name_timer_private.h
  \_Ns_Name_timer_public.h
  \_Ns_Name_timer.c
  \_Ns_Name_eventQueue_public.h
  \_Ns_Name_eventQueue.c
\end{verbatim}

Figure 3.8.: Generated Files

Next, we explain the contents of the generated files.

First, the \texttt{Ns_Name_shared.h} file contains the definitions of preprocessor constants that are shared across multiple other files. It is also used in order to include commonly required standard library header files.

The \texttt{Ns_Name_eventQueue_public.h} contains the definition of the event queue data structure, as well as its public interface declarations. Because the event queue data structure does not provide a private interface, no second header file is generated. The implementation of the event queue is located inside the \texttt{Ns_Name_eventQueue.c} file.

Next, the \texttt{Ns_Name_timer_private.h}, \texttt{Ns_Name_timer_public.h} and \texttt{Ns_Name_timer.c} files contain the private and public interface declarations of the timer
data structure, as well as their implementations.

Analogously, the declarations of the private and public interfaces of the host data structure are located inside the Ns_Name_host_private.h and Ns_Name_host_public.h files, and their implementations inside the Ns_Name_host.c file.

Finally, the Ns_Name_sm_private.h and Ns_Name_sm_public.h files contain the private and public interfaces of the state machine controller. For the reasons described in Section 3.3.1, the public interface declarations also include the user-defined functions provided by the owning entity. The Ns_Name_sm.c file contains the corresponding implementations.

### 3.3.3. Generated Data Structures

In this section, we describe the data structures that are generated by our code generator.

First, in Section 3.3.3.1, we explain the mapping of all applicable model elements to enumeration types. After that, in Sections 3.3.3.2 - 3.3.3.4, we describe the four data structures that we generate in order to represent the entities introduced in Section 3.3.1.

#### 3.3.3.1. Enumerations

In this section, we describe the code mapping of all model elements that we can represent by enumeration constants. We start off by illustrating the general structure of generated enumerations in Paragraph 3.3.3.1.1. Afterwards, in Paragraph 3.3.3.1.2, we list the model elements that we map to enumeration types and demonstrate the mapping using the example shown in Figure A.1.

##### 3.3.3.1.1. General Structure

In general, the auto-generated enumerations are of the form illustrated in Listing 3.2, where Ns, Name and Enum are placeholders for the namespace, the name of the state machine and the name of the enumeration, respectively, and $N$ denotes the number of enumeration constants. Note, that we choose preprocessor constants over C enums, because we want to be explicit about the storage size of the enumeration constants. While enums in C are bounded by the size of int, that is, a maximum of $2^{32}$ entries, the data type used to represent smaller size enumerations is fully compiler-dependent. We work around this problem by using the shown `#define` constructs in combination with a C `typedef` statement. We choose the data type that allows us to represent each enumeration with as little memory as possible. For each enumeration we also generate two meta constants, COUNT and UNDEFINED. The COUNT constant of each enumeration simply represents the number of enum entries, not considering the meta entries. The UNDEFINED constant, on the other hand, is used to represent an ill-defined state of a variable of the enumeration type. For example, the assumed state inside each orthogonal region is UNDEFINED, as long as the region was not yet entered. The value of the UNDEFINED constant is always $2^{bits} - 1$, where $bits \in \{8, 16, 32, 64\}$, $bits \geq \log_2 (N + 1)$, bits min.

Even though, we could assign human-readable names to the enumeration constants, in general we choose to name them after their value. We take this approach, because rule 5.1 of the MISRA-C:2012 guidelines requires identifiers to be distinguishable by their first 31
characters. Human-readable names often exceed the aforementioned 31-character limit, hence our decision.

Instead, we generate comments for all of the enumeration constants that enable engineers to backtrack them to the corresponding elements of the source model. In addition to that, we also generate references to any requirements that the model element satisfies. This allows for the desired code-level traceability of requirements.

### 3.3.3.1.2. Mapped Model Elements

In this paragraph, we explain which model elements we choose to map to enumeration constants. First of all, we decide to map the orthogonal regions of the state machine to an enumeration, as this allows us to use arrays for the storage of the currently assumed states and memorized state histories of each orthogonal region. The regions enumeration generated for our example state machine is shown in Listing 3.3. The states are mapped to enumeration constants, too, as this allows us to easily check the currently assumed states inside each region using `switch` statements. The state enumeration of the example state machine is shown in Listing 3.4. For the same reason, we choose to also represent the sets of consumable events, input signals and history states by enumerations. We show the event and signal enumerations of the ATM example in Listings 3.5 and 3.6. Because the ATM example does not contain any history states, no corresponding enumeration is generated. Finally, we introduce a `callmode` enumeration, consisting of `INTERNAL` and `EXTERNAL` callmodes. The significance of callmodes will be explained in detail in Section 3.3.4. We show the generated callmode enumeration of the example state machine in Listing 3.7.

### 3.3.3.2. Event Queue

The `Ns_Name_eventQueue_t` data structure represents the asynchronous event queue of a state machine instance. In Listing 3.8, we show the definition of the event queue data structure that is generated for the ATM example state machine. The public interface functions of the event queue are shown in Listing 3.9. Both, Listings 3.8 and 3.9 are excerpts from the `ACS_ATM_eventQueue_public.h` header file. The public interface of the event queue data structure exposes the four functions shown in Listing 3.9.

The `enqueue` and `dequeue` functions are used to add and remove events from the queue. Events may only be enqueued as long as a user-defined maximum capacity is not exceeded. Inside the generated code, this maximum capacity is represented by the `EVENTQUEUE_LIMIT`
### 3. Design

#### 3.1 regions

```c
#define ACS_ATM_REGION_0 0x00 /* Main. */
#define ACS_ATM_REGION_1 0x01 /* Serving Customer. */
#define ACS_ATM_REGION_2 0x02 /* Lane A. */
#define ACS_ATM_REGION_3 0x03 /* Lane B. */
#define ACS_ATM_REGION_4 0x04 /* Lane C. */
#define ACS_ATM_REGION_5 0x05 /* Submachine. */
#define ACS_ATM_REGION_COUNT 0x06 /* Number of enumeration constants. */
#define ACS_ATM_REGION_UNDEFINED 0xFF /* Ill-defined. */
typedef uint8_t ACS_ATM_region_t;
```

**Listing 3.3: Regions enumeration of ATM example.**

#### 3.2 states

```c
#define ACS_ATM_STATE_0 0x00 /* Initial (Main). */
#define ACS_ATM_STATE_1 0x01 /* Idle (Main). */
#define ACS_ATM_STATE_2 0x02 /* Maintenance (Main). */
#define ACS_ATM_STATE_3 0x03 /* Out of Service (Main). */
#define ACS_ATM_STATE_4 0x04 /* Serving Customer (Main). */
#define ACS_ATM_STATE_5 0x05 /* Final (Serving Customer). */
#define ACS_ATM_STATE_6 0x06 /* Initial (Serving Customer). */
#define ACS_ATM_STATE_7 0x07 /* Transaction (Serving Customer). */
#define ACS_ATM_STATE_8 0x08 /* Submachine (Serving Customer). */
#define ACS_ATM_STATE_9 0x09 /* Success (Submachine). */
#define ACS_ATM_STATE_10 0x0A /* Entry (Submachine). */
#define ACS_ATM_STATE_11 0x0B /* Cancelled (Submachine). */
#define ACS_ATM_STATE_12 0x0C /* Initial (Submachine). */
#define ACS_ATM_STATE_13 0x0D /* Typing (Submachine). */
#define ACS_ATM_STATE_14 0x0E /* Turning Off (Main). */
#define ACS_ATM_STATE_15 0x0F /* Failure (Main). */
#define ACS_ATM_STATE_16 0x10 /* Service Ended (Main). */
#define ACS_ATM_STATE_17 0x11 /* Self Test (Main). */
#define ACS_ATM_STATE_18 0x12 /* Initial (Lane A). */
#define ACS_ATM_STATE_19 0x13 /* Final (Lane A). */
#define ACS_ATM_STATE_20 0x14 /* Junction (Lane A). */
#define ACS_ATM_STATE_21 0x15 /* Initial (Lane B). */
#define ACS_ATM_STATE_22 0x16 /* Final (Lane B). */
#define ACS_ATM_STATE_23 0x17 /* Junction (Lane B). */
#define ACS_ATM_STATE_24 0x18 /* Initial (Lane C). */
#define ACS_ATM_STATE_25 0x19 /* Final (Lane C). */
#define ACS_ATM_STATE_26 0x1A /* Sync (Lane C). */
#define ACS_ATM_STATE_COUNT 0x1C /* Number of enumeration constants. */
#define ACS_ATM_STATE_UNDEFINED 0xFF /* Ill-defined. */
typedef uint8_t ACS_ATM_state_t;
```

**Listing 3.4: States enumeration of ATM example.**
3.3. Code Mapping

Listing 3.5: Events enumeration of ATM example.

```c
#define ACS_ATM_EVENT_CHANGE 0x00 /* Unprocessed variable change. */
#define ACS_ATM_EVENT_0 0x01 /* Completion event: Initial (Main). */
#define ACS_ATM_EVENT_27 0x1C /* Completion event: Sync (Lane C). */
#define ACS_ATM_EVENT_28 0x1D /* Signal event for signal Cancel. */
#define ACS_ATM_EVENT_33 0x22 /* Signal event for signal Shutdown. */
#define ACS_ATM_EVENT_34 0x23 /* Time event for duration 1000ms. */
#define ACS_ATM_EVENT_36 0x27 /* Time event for duration 4000ms. */
#define ACS_ATM_EVENT_UNDEFINED 0xFF /* Ill-defined. */
typedef uint8_t ACS_ATM_event_t;
```

Listing 3.6: Signals enumeration of ATM example.

```c
#define ACS_ATM_SIGNAL_CANCEL 0x00
#define ACS_ATM_SIGNAL_CARD 0x01
#define ACS_ATM_SIGNAL_CONFIRM 0x02
#define ACS_ATM_SIGNAL_KEYPRESS 0x03
#define ACS_ATM_SIGNAL_POWER 0x04
#define ACS_ATM_SIGNAL_SHUTDOWN 0x05
#define ACS_ATM_SIGNAL_COUNT 0x06 /* Number of enumeration constants. */
#define ACS_ATM_SIGNAL_UNDEFINED 0xFF /* Ill-defined. */
```

Listing 3.7: Callmodes enumeration of ATM example.

```c
#define ACS_ATM_CALLMODE_INTERNAL 0x00
#define ACS_ATM_CALLMODE_EXTERNAL 0x01
#define ACS_ATM_CALLMODE_COUNT 0x02 /* Number of enumeration constants. */
#define ACS_ATM_CALLMODE_UNDEFINED 0xFF /* Ill-defined. */
typedef uint8_t ACS_ATM_callmode_t;
```
3. Design

Listing 3.8: Event queue data structure of ATM example.

constant. This constant can be specified by the user of the code generator. Similarly, an event can only be dequeued, given that the queue is not empty.

The `front` and `back` functions serve similar purposes. They, too, return the events that currently reside in the foremost and backmost positions of the queue, respectively, but do not remove them from the queue.

### 3.3.3.3. Timer

The `Ns_Name_timer_t` data structure represents a platform-independent timer resource. We use these timers in order to keep track of time events that need to be triggered in the future. In Listing 3.10, we show the timer struct that is generated for the ATM example state machine. The shown definition is located inside the `ACS_ATM_timer_public.h` header file. The timer data structure decapsulates the state machine controller from any platform-specific timing API. Given that Windows is selected as target platform for the generated code, like in the ATM example, the timer data structure manages a `WIN_timer_t` instance. Depending on the target platform, the code generator may replace this member variable with other platform-specific timer resources or delegate the management task to another abstraction layer.

The public interface of the timer data structure is shown in Listing 3.11. The `init` function is used to initialize a timer instance. This initialization process involves the reservation of an operating system timer resource. Additionally, the desired duration of the timer in milliseconds is stored, as well as the time event constant associated to the timer instance. The `start` function is used to start a previously initialized timer instance. The function specified by the `callback` parameter will be invoked after the started timer has timed out. The `cancel` function, on the other hand, is used to cancel a previously started and still running timer instance. Cancelling a timer causes the corresponding time event to not be triggered and the platform-specific resources managed by the timer instance to be released.
3.3. Code Mapping

Listing 3.9: Event queue public interface.

```c
/* Adds a new event to the tail of the queue. */
extern bool_t
ACS_ATM_eventQueue_enqueue(ACS_ATM_eventQueue_t * const queue,
    ACS_ATM_event_t event);

/* Removes the foremost event from the queue. */
extern bool_t
ACS_ATM_eventQueue_dequeue(ACS_ATM_eventQueue_t * const queue,
    ACS_ATM_event_t * const event);

/* Returns the foremost event in the queue. 
   Does not remove the event from the queue. */
extern bool_t
ACS_ATM_eventQueue_front(ACS_ATM_eventQueue_t * const queue,
    ACS_ATM_event_t * const event);

/* Returns the backmost event in the queue. 
   Does not remove the event from the queue. */
extern bool_t
ACS_ATM_eventQueue_back(ACS_ATM_eventQueue_t * const queue,
    ACS_ATM_event_t * const event);
```

Listing 3.10: Timer data structure of ATM example.

```c
/* Timer callback function. */
typedef void (* ACS_ATM_timer_cb_t)(ACS_ATM_timer_t *, ACS_ATM_host_t *);

/*** 
* This structure holds information about an OS-specific timer. 
* After a timer instance timed out, its associated event will 
* be enqueued in the state machine instance’s event queue. 
***/
typedef struct __ ACS_ATM_timer
{
    ACS_ATM_host_t    * host;
    ACS_ATM_sm_t      * sm;
    ACS_ATM_event_t   event;
    uint32_t           duration;
    bool_t             initialized;
    ACS_ATM_timer_cb_t callback;
    WIN_timer_t       * win32Timer; /* OS-specific. */
} ACS_ATM_timer_t;
```
3. Design

```c
/* Initializes a new timer instance. */
extern void
ACS_ATM_timer_init(ACS_ATM_timer_t * const timer,
                    ACS_ATM_host_t * const host,
                    ACS_ATM_sm_t * const sm,
                    ACS_ATM_event_t event,
                    uint32_t duration);

/* Starts a previously initialized timer. */
extern void
ACS_ATM_timer_start(ACS_ATM_timer_t * const timer,
                     ACS_ATM_timer_cb_t callback);

/* Cancels a previously started timer. */
extern void
ACS_ATM_timer_cancel(ACS_ATM_timer_t * const timer);
```

Listing 3.11: Timer data structure public interface.

```c
typedef struct __ACS_ATM_host
{
    ACS_ATM_timer_t timers[ACS_ATM_HOST_MAX_TIMERS];
} ACS_ATM_timer_t;
```

Listing 3.12: Definition of the host data structure.

3.3.3.4. State Machine

We use two data structures to represent instances of a modelled state machine. We generate the \texttt{Ns\_Name\_smInt\_t} data structure that holds any internal information of a state machine instance that is not of interest for user code. This includes the event queue of the state machine, as well as any information relevant for the state machine controller, like the currently assumed states inside each orthogonal region. We also generate the \texttt{Ns\_Name\_sm\_t} data structure which encapsulates the internal information of the state machine instance inside its \texttt{internal} member variable. In addition to that, it also contains any user-defined on-instance attributes of the state machine. By separating the internal information from the user data, we greatly reduce the risk of user code accidentally overwriting internal data. As a consequence, user-defined functions are much less prone to error. Another small benefit of this separation is that the number of reserved names is reduced to just 'internal'. This allows for a more free choice of on-instance attribute names. In addition to that, a third data structure called \texttt{Ns\_Name\_host\_t} is generated. The host is used as an additional layer of abstraction between the state machine and the platform-specific timer management. It is especially important when targeting embedded systems that run real-time operating systems.

In Listings 3.12 - 3.14 we show the definitions of the \texttt{host}, \texttt{smInt} and \texttt{sm} data structures generated for the ATM example.
3.3. Code Mapping

```c
/**
 * Internal state machine data.
 * Stores the currently assumed state in
 * each orthogonal regions and keeps track
 * of all history states.
 */
typedef struct __ACS_ATM_smInt
{
    ACS_ATM_host_t        * pHost;
    ACS_ATM_eventQueue_t  eventQueue;
    ACS_ATM_callmode_t    callmode;
    ACS_ATM_state_t       currentState[ACS_ATM_REGION_COUNT];
    ACS_ATM_state_t       stateHistories[ACS_ATM_HISTORY_COUNT];
} ACS_ATM_smInt_t;
```

Listing 3.13: Definition of the state machine internal data structure.

```c
/**
 * State machine instance.
 * Contains all internal state machine data
 * as well as the user-defined on-instance
 * attributes.
 */
typedef struct __ACS_ATM_sm
{
    ACS_ATM_smInt_t  internal;
    uint8_t        * pUserInput;
    uint8_t        userInputLen;
    bool_t         * pDevicesWorking;
    bool_t         requiresService;
    uint8_t        selfTestResult;
    uint8_t        error;
} ACS_ATM_sm_t;
```

Listing 3.14: Definition of the state machine data structure.
3.3.4. State Machine Controller Implementation

In this section, we describe the generated implementation of the state machine controller. The implementation of the state machine controller is separated into multiple types of functions, which we explain in Sections 3.3.4.1 - 3.3.4.11. In general, all of the state machine controller functions operate on instances of the \texttt{Ns\_Name\_sm\_t} data structure that we described in Section 3.3.3.4. A pointer to this instance is passed to the controller functions via their \texttt{sm} parameter.

3.3.4.1. User-Defined Functions

As we introduced in Section 3.3.1, the owning entity of a state machine may define private and public operations in the form of \texttt{UmlOperation} instances. In the context of code generation, we refer to these operations of the owning entity as user-defined functions. In Figure 3.9, we show the composition of the \texttt{UmlOperation} class. As can be seen, the signature and body of a user-defined function are part of the \texttt{UmlOperation} instance data. But, even though the implementation of a user-defined function can be extracted from the model data, we need to apply some modifications to it in order to achieve its correct behaviour in the execution context of the state machine.

![Figure 3.9: UmlOperation class diagram.](image)

For example, model engineers would not be able to access on-instance attributes of the state machine from inside user-defined functions. In order to solve this problem, we automatically prepend an argument called \texttt{sm} to the argument list of each user-defined function. The \texttt{sm} parameter is of type \texttt{Ns\_Name\_sm\_t \* const} and points to the state machine instance inside whose context the user-defined function shall be executed. Accesses to any on-instance attributes from inside user-defined function bodies are realized...
by dereferencing the \texttt{sm} pointer.
Furthermore, the state machine may need to be informed about calls to user-defined functions. Given that there exist transitions that are triggered by a call event associated to a given user-defined function, we need to generate a corresponding call event with each function call. For this reason, we append code that enqueues the corresponding call event into the event queue of the state machine to the bodies of each applicable user-defined function.
Finally, because user-defined functions may arbitrarily change the values of state machine attributes, we need to recheck any change event expressions. For this reason, we append a call to the \texttt{update} function of the state machine to the generated implementation. Because we only want to recheck change event expressions after calls to user-defined function from an external source, that is not from inside the \texttt{update} function itself, we additionally check the \texttt{callmode} of the state machine instance. We refer to calls to the \texttt{update} function from within user-defined functions as \textit{implicit updates}.
As can be seen in the package hierarchy we show in Figure A.2, the owning entity of the ATM example state machine provides a functional interface consisting of the six user-defined functions \texttt{ATM\_shutdown}, \texttt{ATM\_startup}, \texttt{ATM\_readCard}, \texttt{ATM\_ejectCard}, \texttt{ATM\_clearUserInput} and \texttt{ATM\_appendUserInput}. For demonstration purposes, we show the \texttt{UmlOperation} instance that maps to the user-defined \texttt{ATM\_appendUserInput} function in Figure 3.10. Our code generator maps this example function to the implementation shown in Listing 3.15.

![Instance: UmlOperation]

\begin{verbatim}
Name : "ATM\_appendUserInput"
Visibility : Public
Parameters : [ ]
{ 
Name = "character"
Type = "uint8_t"
}]
ReturnType : "void",
Body : "if(sm->userInputLen < 127u)
{ 
  sm->pUserInput[sm->userInputLen++] - character;
  sm->pUserInput[sm->userInputLen] = '\0';
}"
\end{verbatim}

Figure 3.10.: \texttt{UmlOperation} instance for \texttt{ATM\_appendUserInput} function.
extern void
ATM_appendUserInput(ACS_ATM_sm_t * const sm, uint8_t character)
{
    /* User defined function body: */
    {
        if (sm->userInputLen < 127u)
        {
            sm->pUserInput[sm->userInputLen++] = character;
            sm->pUserInput[sm->userInputLen] = '\0';
        }
    }

    /* Notify state machine instance of possible attribute changes. */
    ACS_ATM_sm_createChangeEvent(sm);

    if (sm->internal.callmode == ACS_ATM_CALLMODE_EXTERNAL)
    {
        /* Trigger implicit update. */
        ACS_ATM_sm_update(sm);
    }
    else
    {
        NOOP;
    }
}

Listing 3.15: Generated implementation of user-defined ATM_appendUserInput function.
3.3. Code Mapping

3.3.4.2. State Entry Functions

We generate entry functions for each state satisfying at least one of the following requirements. First, if an entry action was specified for the state inside the modeller, we require an entry function for it, in order to perform the modelled behaviour. An entry function is also required for all composite states, because we need to assume the initial states of all contained orthogonal regions, as soon as the composite state is entered. Finally, all history pseudostates require entry actions, because they might memorize a state that requires its entry function to be called. History state entry functions differ from usual state entry functions and are thus explained separately, in Section 3.3.4.3.

The generated entry functions implement the pseudocode shown in Figure 3.11. Inside the entry function of a given state, we first call the entry function of its parent state. This process is repeated, until the state entry function of a state inside the shared parent region of transition source and target state was called. This state is specified by the \texttt{topmostState} parameter of the state entry function.

After all parent entry functions have finished, we need to perform the entry action of the state, if there was one specified inside the modeller.

Given that the entered state is a composite state, we next need to initialize all orthogonal child regions. This initialization includes assuming the initial states of each region and calling their respective state entry functions.

Next, we need to start all required timers for any of the time events that may occur while the just entered state is assumed. Finally, we need to handle the generation of a completion event for the entered state. Given that we just entered a non-composite state, we simply create a completion event for this state. In addition to that, if a final state was just entered, we need to check whether the state machine has assumed final states inside all of the other parallel regions of the parent state. If this is the case, we can additionally create a completion event for the parent state of the entered state.

In Listing B.1, we show the generated entry function for the \texttt{Self Test} state of the ATM example state machine.

3.3.4.3. History Entry Functions

For the reasons stated in Section 3.3.4.2, we require a special kind of state entry function for history states. Because a history state serves as a placeholder for the latest assumed state inside its parent region, the entry function of a history state has to invoke the entry function of the latest memorized state. Of course this requires an entry function for the memorized state to exist. If this is not the case, no special action is performed at history state entry.

3.3.4.4. State Exit Functions

Analogously to entry functions, exit functions are generated for all states satisfying either of the following requirements. Either the state has an exit action on its own, or is the orthogonal parent of any state with an exit action. Or, there exists at least one transition originating from the state that is triggered by a time event. The exit function of each
Pseudocode: \textit{State entry function}.
Input Data: \textit{sm, state, topmostState}.

\begin{verbatim}
if state \neq topmostState then
    stateEntryFunction(sm, parentState(state), topmostState);
end if
if stateHasEntryAction(state) = TRUE then
    performEntryAction(state, sm);
end if
if isCompositeState(state) = TRUE then
    for all \( r \in \text{childRegions(state)} \) do
        initializeRegion(r);
    end for
end if
if stateContainsTimeEventListeners(state) = TRUE then
    startRequiredTimers(state, sm);
end if
if isCompositeState(state) = FALSE then
    createStateCompletionEvent(state, sm);
end if
if isFinalState(state) = TRUE then
    siblingRegionsTerminated \leftarrow \text{TRUE};
    for all \( r \in \text{childRegions(parentState(state)), r \neq parentRegion(state)} \) do
        if finalStateAssumedInRegion(r) = FALSE then
            siblingRegionsTerminated \leftarrow \text{FALSE};
        end if
    end for
    if siblingRegionsTerminated = \text{TRUE} then
        createCompletionEvent(parentState(state), sm);
    end if
end if
\end{verbatim}

Figure 3.11.: State entry function pseudocode.

state implements the pseudocode shown in Figure 3.12. First, any running timers are stopped, given that they are no longer required. Next, all currently assumed child states inside its orthogonal regions are exited. This involves the invocation of their respective state exit functions. Finally, the exit action of the state itself is performed. In Listing B.2, we show the generated exit function for the \textit{Self Test} state of the ATM example state machine.
3.3. Code Mapping

Pseudocode: STATE EXIT FUNCTION.
Input Data: sm, state.

if stateContainsTimeEventListeners(sm) then
    stopRunningTimers(state, sm);
end if
if isCompositeState(state) then
    for all r ∈ childRegions(state) do
        exitAssumedState(r, sm);
    end for
end if
if stateHasExitAction(state) then
    performExitAction(state, sm);
end if

Figure 3.12.: State exit function pseudocode.

3.3.4.5. Transition Functions

For each of the modelled state transitions we generate a respective transition function. These transition functions implement the pseudocode shown in Figure 3.14. When a transition function is called, we first exit the source state configuration. As we introduced in Section 3.1.4.1, the set of exited states depends on the transition kind. More specifically, given a transition from state \( s \) to \( t \), we need to exit the state inside the \textit{shared parent region} of \( s \) and \( t \), whose transitive closure contains \( s \). The pseudocode in Figure 3.13 shows, how the shared parent region of any two given states \( s \) and \( t \) is obtained. After all required states were exited, we perform the transition effect, given that one was specified inside the modeller. Next, the state machine struct instance is updated, such that the currently assumed state inside the transition region becomes the target state of the transition. Similarly, we update the history state of the transition region, if there exists one.

Finally, the target state configuration is entered, causing the appropriate entry behaviour. Again, the transition kind determines, which states need to be entered. Given a transition from state \( s \) to \( t \), the entry function of state \( t \) is invoked. In order for the parent states of \( t \) to be entered correctly, we pass the state inside the shared parent region of \( s \) and \( t \), whose transitive closure contains \( t \), as \textit{topmostState} parameter.

In Listing B.3, we show the transition function that is generated for the external transition from \textit{Self Test} to \textit{Failure} of the ATM example state machine. In this example, the shared parent region of \textit{Self Test} and \textit{Failure} is the main region of the state machine. The state inside the main region, whose transitive closure contains \textit{Self Test}, is \textit{Self Test} itself. Therefore, we exit the \textit{Self Test} state at the beginning of the transition function. The state inside the shared parent region, whose transitive closure contains the
target state *Failure* also is the target state itself. Therefore, at the end of the shown transition function, we enter the *Failure* state.

---

**Pseudocode:** FIND SHARED PARENT REGION.
*Input Data*: *s, t.*

```plaintext
spr ← parentRegion(s);
while \{s, t\} \∉ regionClosure(spr) do
    spr ← parentRegion(parentState(spr));
end while
return spr;
```

Figure 3.13.: Shared parent region pseudocode.

---

**Pseudocode:** TRANSITION FUNCTION.
*Input Data*: *sm, transition.*

```plaintext
exitSourceStateConfiguration(sm);
if hasTransitionEffect(transition) = TRUE then
    performTransitionEffect(transition, sm);
end if
currentState(sm) ← targetState(transition);
if regionContainsHistoryState(parentRegion(targetState(transition))) then
    updateHistoryState(sm, parentRegion(targetState(transition)));
end if
enterTargetStateConfiguration(sm);
```

Figure 3.14.: Transition function pseudocode.

---

### 3.3.4.6. Guard Functions

For each of the transition guards that occur within the state machine model, we generate a respective *guard* function. The generated functions have a return type of `bool_t` which can accept any of the two values `TRUE` and `FALSE`. The body of a guard function simply consists of a `return` statement. The return value of this statement is the evaluation result of the guard expression that was specified inside the modelling tool as inline C code. In Listing 3.16, we show the *guard* function that is generated for the transition from *Self Test* to *Failure* of the ATM example state machine.
3.3. Code Mapping

3.3.4.7. Change Expression Functions

For every transition that is triggered by a change event, we generate a changeExpr function. The return value of these function is either TRUE or FALSE depending on the evaluation result of the associated change expression. Just like the guard functions we introduced in Section 3.3.4.6, changeExpr functions consist of a single return statement, as well. In Listing 3.17, we show the changeExpr function that is generated for the change event that triggers the transition from Initial to Sync of the ATM example state machine.

3.3.4.8. Send Signal Functions

Given that an input model contains transitions that are triggered by signal events, we need to provide an interface to the user that allows him to send signals to the state machine instance. For this reason, we introduce the sendSignal function of the state machine controller. In addition to the sm parameter, the sendSignal function receives a signal parameter which represents the signal that should be sent. The value of the signal parameter may be any of the enumeration constants of type Ns_Name_signal_t. The body of the sendSignal function consists of a single switch statement with respect to the value of signal switch cases are generated for all signal constants that are mapped to signal events. Note that a signal constant is only mapped to a signal event, given that there exists a transition that is triggered by the reception of the signal. Inside each of the generated switch cases, we simply enqueue the appropriate signal event into the event queue of the state machine instance.

Because we modified the event queue, a call to the update function is required at the end of the function.

In Listing B.4, we show the generated sendSignal function of the ATM example state machine.
3.3.4.9. Event Processor Functions

In order to recognize when transitions have to fire, we need to regularly process incoming events. For this reason, we introduce the `processEvent` functions. We generate such a function for each of the orthogonal regions of the state machine. In addition to the pointer to the state machine instance `sm`, the `processEvent` functions receive a second parameter, `event`, which is the enumeration constant that represents the event to be processed. The return value of the `processEvent` function is of type `bool_t` and we return a value of `TRUE`, given that a transition fired from within the function. In this case we say the event was consumed. Accordingly, `FALSE` is returned from the function, if the event could not be consumed.

The `processEvent` functions of all orthogonal regions of the model implement the pseudo-code shown in Figure 3.15. Inside the `processEvent` function of a given region, we first pass on the event to all applicable child regions. Only if none of the child regions could consume a given event, i.e. all their `processEvent` functions returned `FALSE`, we process the event inside the considered region itself.

In this case, we first generate a `switch` statement with respect to the value of `event`. We generate `switch cases` for all events that can possibly be consumed by the considered region. If no events can be consumed by the region, the `switch` statement is omitted, completely. Inside each of the `cases` of the outer `switch` statement, we generate an inner `switch` statement. This inner `switch` statement takes into account the value of the currently assumed state inside the region. We generate `cases` for the inner `switch` statement for each state inside which the state machine can consume an instance of the considered event.

Inside each of the generated inner `switch cases`, we determine if any transition should fire. Given that the transition is triggered by a change event, we first need to check whether the associated change expression evaluates to `TRUE`. In addition to that, if the transition is guarded, we need to check whether the associated guard expression evaluates to `TRUE`. Given that the aforementioned conditions are met, we can fire the transition by calling its corresponding transition function. Because only one transition may fire during each call to `processEvent`, we exit the function as soon as a transition was triggered.

In Listing B.6, we show the generated `processEvent` function for the region encompassed by the Login submachine state. Even though, due to its length we only show an excerpt of the function, the overall structure of the `processEvent` functions should still be recognizable.

3.3.4.10. CreateChangeEvent Function

As per UML specification [Gro15c], transitions may fire based on the values of one or more on-instance and on-class attributes of a state machine. In order to prevent having to poll all change event expressions periodically, we introduce the `createChangeEvent` function. The `createChangeEvent` function enqueues a `Ns_Name_EVENT CHANGE` constant into the event queue, given that the tail element of the queue is not already a change event. We do not need multiple consequent change events inside the event queue, because
whenever a transition was triggered by a change event, a new change event instance is created.
The `createChangeEvent` may also be called by the user, in order to notify the state machine of changes to any attributes of the state machine that happened outside of the state machine controller logic.
In Listing B.5 we show the `createChangeEvent` function that was generated for the ATM example.

### 3.3.4.11. Update Function

In order to execute a state machine model, we need to regularly process all events inside the asynchronous event queue. For this task we introduce the `update` function. The main part of the `update` function consists of a `while`-loop that runs for as long as the event queue contains event instances. During each iteration of the `while`-loop, we call the `processEvent` function of the main region, which automatically distributes the event instance to all nested regions. A return value of `TRUE` of the `processEvent` function indicates that a transition fired due to the processed event. In this case, we need to generate a change event instance, because the transition effect may have changed the values of one or more attributes of the state machine. It is worth noting that we switch the state machine instance into `CALLMODE_INTERNAL` before we start with the processing of the event queue.
As we explained earlier, whenever the state machine is in `CALLMODE_INTERNAL`, calls to user-defined functions do not trigger an implicit update of the state machine. In this way, we prevent the `update` function from being called from within itself. Because the `update` function always handles all elements of the event queue, we can still be sure that newly generated events will be processed, even though only one call to `update` is made. After we finished the processing of the event queue, we switch back to `CALLMODE_EXTERNAL`, in order for user functions to trigger implicit updates again.
In general, the update function is called implicitly by the state machine controller, whenever there are unprocessed events in the queue. Though, if the user manually creates a change event using the `createChangeEvent` method, he needs to call the `update` function himself in order to initiate the processing of the new event.
Pseudocode: Region event processor function.
Input Data: sm, region, event.

\[
\text{childFeedback} \leftarrow \text{FALSE}; \\
\text{transitionTriggered} \leftarrow \text{FALSE}; \\
\]

\[
\text{for all } s \in \text{childStates}(\text{region}) \text{ do} \\
\text{for all } r \in \text{childRegions}(s) \text{ do} \\
\quad \text{if } \text{regionIsActive}(r) = \text{TRUE} \text{ then} \\
\quad \quad \text{childFeedback} \leftarrow \text{childFeedback} \ || \ \text{processEventFunction}(r, \text{sm}, \text{event}); \\
\quad \text{end if} \\
\text{end for} \\
\text{end for} \\
\text{if } !\text{childFeedback} \text{ then} \\
\text{for all } e \in \text{applicableEvents}(\text{region}) \text{ do} \\
\quad \text{if } \text{event} = e \text{ then} \\
\quad \text{for all } s \in \text{childStates}(\text{region}) \text{ do} \\
\quad \quad \text{for all } t \in \text{transitions}(\text{sm}) \text{ satisfying sourceState}(t) = s \text{ do} \\
\quad \quad \quad \text{if } \text{transitionTriggered} = \text{FALSE} \text{ then} \\
\quad \quad \quad \quad \text{transitionReady} \leftarrow \text{TRUE}; \\
\quad \quad \quad \quad \text{if } \text{transitionIsGuarded}(t) \text{ then} \\
\quad \quad \quad \quad \quad \text{if } \text{evaluateGuard}(t, \text{sm}) = \text{FALSE} \text{ then} \\
\quad \quad \quad \quad \quad \quad \text{transitionReady} \leftarrow \text{FALSE}; \\
\quad \quad \quad \text{end if} \\
\quad \quad \text{end if} \\
\quad \quad \text{if } \text{triggeredByChangeEvent}(t) = \text{TRUE} \text{ then} \\
\quad \quad \quad \text{if } \text{evaluateChangeExpression}(t, \text{sm}) = \text{FALSE} \text{ then} \\
\quad \quad \quad \quad \text{transitionReady} \leftarrow \text{FALSE}; \\
\quad \quad \text{end if} \\
\quad \quad \text{end if} \\
\quad \quad \text{if } \text{transitionReady} = \text{TRUE} \text{ then} \\
\quad \quad \quad \text{transitionFunction}(t, \text{sm}); \\
\quad \quad \quad \text{transitionTriggered} \leftarrow \text{TRUE}; \\
\quad \quad \text{end if} \\
\quad \text{end if} \\
\text{end for} \\
\text{end for} \\
\text{end for} \\
\text{return } \text{transitionTriggered}; \\
\]

Figure 3.15.: Region event processor function pseudocode.
4. Implementation

In this chapter, we describe the implementation of a code generator that applies the transformation rules we elaborated in Chapter 3 to a given XMI-encoded input model. First, we separate the code generator into frontend and backend. The frontend of the code generator is responsible for the processing of the input XMI file. Its output is a UmlModel object that represents the parsed model data. The implementation of the frontend is described in Section 4.1.

The backend of the code generator is responsible for the actual code generation based on a previously constructed UmlModel instance. The code that is generated by the backend satisfies the mapping that we elaborated in Section 3.3. The implementation of the backend is described in Section 4.2.

For both frontend and backend we choose C# as the implementation language. This allows us to use the robust and well-documented utilities provided by the Microsoft .NET libraries. Most notably, we benefit from the provided solutions for the parsing of XML files. We use the Microsoft Visual Studio integrated development environment (IDE), as it offers a very productive workflow, as well as powerful debugging features.

4.1. Frontend

In this section, we describe the implementation of the frontend of the code generator. The implementation of the frontend is based on the libXMI library. The libXMI library encapsulates commonly used UML model elements and allows for the parsing of XMI files. The included XmiParser class provides a Parse method that accepts a path to an input XMI file path and returns a parsed XmiDocument instance. The returned XmiDocument consists of a hierarchy of XmiNode instances. A previously parsed XmiDocument instance can be translated to a UmlModel instance using the TranslateDocument method of the XmiTranslator class. This translation process of the document can be controlled by the user by specifying different callback actions.

First, a number of preprocessing actions can be specified. These preprocessing actions operate on the XmiDocument instance and can be used to apply structural permutations to the document or to create and remove nodes from it, after it has been parsed. This mechanism is useful to e.g. narrow down the scope of subsequent processing steps.

Next, translation actions can be specified for each of the existing types of XmiNodes. Translation actions are used to translate all XmiNodes of a given type occurring inside the document to UmlObjects. All of the created UmlObjects are aggregated inside a UmlModel instance.

Finally, postprocessing actions can be specified for each subclass of the UmlObject class. The specified postprocessing actions are invoked on each of the UmlObject instances contained by the UmlModel constructed during the translation phase. Postprocessing actions are primarily useful in cases where cross-references have to be established between different UmlObject instances, as this requires the whole UmlModel to be constructed, beforehand.
The described translation process taking place inside the TranslateDocument method of the XmiTranslator class is summarized by the pseudocode we show in Figure 4.1. This approach to the implementation of the frontend allows for iterative improvements to it in the future. For example, the set of supported model elements can easily be extended by adding new preprocessor, translation and postprocessing actions to the XmiTranslator instance.

Pseudocode: Translate XMI document function.
Input Data: document, preprocessorActions, translationActions, postprocessorActions.

```plaintext
for all preprocess ∈ preprocessorActions do
    preprocess(document);
end for
model ← createEmptyModel();
for all node ∈ documentNodes(document) do
    for all translate ∈ translationActions do
        object ← translate(node);
        addObjectToModel(object, model);
    end for
end for
for all object ∈ model do
    for all postprocess ∈ postprocessorActions do
        postprocess(object);
    end for
end for
return model;
```

Figure 4.1.: Translate XMI document function pseudocode.

4.2. Backend

In this section, we describe the backend of our code generator, which is responsible for the generation of code based on a previously constructed UmlStateMachine instance. The input UmlStateMachine instance is obtained from the UmlModel instance populated by the XmiParser and XmiTranslator. The translation process of an XMI document to a UmlModel instance is described in Section 4.1. The CodeGenStateMachine method of theCodeGenEngine class serves as the entry point for the code generation. The engine also provides code generation methods for common code constructs. Among others, these include comments, conditional branches and loops. For more complex code constructs, we introduce the CodeGenItem base class. In Figure 4.2, we show the four subclasses of the CodeGenItem class. The code generation items
are separated into files (CodeGenFile), data structures (CodeGenStruct), enumerations (CodeGenEnum) and functions (CodeGenFunction). Instances of the CodeGenFile class are specialized to CodeGenHeaderFile or CodeGenSourceFile, respectively, depending on their contents. All of the files, data structures, enumerations and functions that we discussed in Section 3.3, are mapped to the corresponding subclasses of the CodeGenItem class.

![Class diagram showing the subclasses of CodeGenItem.](image)

The CodeGenStateMachine function of the CodeGenEngine class first creates CodeGenFile instances for each of the files to generate. We described this set of generated files in Section 3.3.2. Next, the CodeGen method of each of the created CodeGenFile instances is called. This leads to the instantiation of the CodeGenEnum, CodeGenStruct and CodeGenFunction classes, depending on the respective file contents. Finally, the code generation logic of the CodeGenStruct, CodeGenEnum and CodeGenFunction instances is executed. We summarize the code generation process taking place inside the CodeGenStateMachine method of the CodeGenEngine class in the pseudocode we show in Figure 4.3.
Pseudocode: GENERATE CODE FOR STATE MACHINE.
Input Data: $sm$

for all file ∈ filesToGenerate($sm$) do
    for all enum ∈ enumsToGenerate(file, $sm$) do
        addCodeToFile(file, generateEnumCode(enum, $sm$));
    end for
    for all struct ∈ structsToGenerate(file, $sm$) do
        addCodeToFile(file, generateStructCode(struct, $sm$));
    end for
    for all decl ∈ funcDeclsToGenerate(file, $sm$) do
        addCodeToFile(generateFuncDeclCode(decl, $sm$));
    end for
    for all impl ∈ funcImplsToGenerate(file, $sm$) do
        addCodeToFile(generateFuncImplCode(impl, $sm$));
    end for
    saveFile(file);
end for

Figure 4.3: CodeGenStateMachine function pseudocode.
5. Validation

In this chapter, we validate the generated state machine implementations with respect to the avionics-relevant MISRA-C:2012 coding guidelines. We achieve this validation using a static code analysis tool. We present the results of the static code analysis in Section 5.1.

Furthermore, we want to ensure that the state machine implementations created by our code generator exhibit the exact modelled behaviour. Thus, in order to demonstrate the correctness of our transformation, we perform a series of directed tests. The testing process, as well as the obtained results, are described in Section 5.2.

Finally, we perform a series of robustness tests of our code generator, in order to ensure its correct behaviour in the case of erroneous input models. We describe the used test cases, as well as the obtained results of the robustness tests in Section 5.3.

5.1. Static Code Analysis

In this section, we describe the static code analysis procedure that we use to validate the compliance of our generated code with relevant standards. We check the code against the MISRA-C:2012 guidelines for the use of the C language in critical systems. The focus of the MISRA-C guidelines is to increase the reliability and maintainability of embedded software, both of which are highly important factors in avionic environments.

We perform the analysis using the commercial off-the-shelf tool Understand by SciTools. Understand takes a collection of syntactically correct C files and checks them against a selection of standards and guidelines. Among other checks, Understand analyzes the possible execution paths of the program, identifying e.g. dead code or function-level recursion. We choose the code generated for the ATM example state machine shown in Figure A.1 as input for the analyzer, because it contains a large number of model elements. This ensures the generation of a representative amount of code. We describe the results of the static code analysis with respect to the MISRA-C:2012 guidelines in Section 5.1.1.

5.1.1. Results

In this section, we describe the results of the static code analysis with respect to the MISRA-C:2012 guidelines. In the summary of the results in Table 5.2, it can be seen that the generated code contains a total of 17 deviations from the MISRA-C:2012 guidelines. Specifically, there exist violations of rules 2.4, 2.5, 5.6 and 5.7.
5. Validation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>A project should not contain unused tag declarations.</td>
<td>__tag_ACS_ATM_eventQueue &lt;br&gt;__tag_ACS_ATM_smInt &lt;br&gt;ACS_ATM_SIGNAL_COUNT &lt;br&gt;ACS_ATM_SIGNAL_UNDEFINED &lt;br&gt;ACS_ATM_EVENT_9 &lt;br&gt;ACS_ATM_EVENT_COUNT</td>
</tr>
<tr>
<td>2.5</td>
<td>A project should not contain unused macro declarations.</td>
<td>ACS_ATM_EVENT_UNDEFINED &lt;br&gt;ACS_ATM_REGION_UNDEFINED &lt;br&gt;ACS_ATM_STATE_COUNT &lt;br&gt;ACS_ATM_CALLMODE_COUNT &lt;br&gt;ACS_ATM_CALLMODE_UNDEFINED</td>
</tr>
<tr>
<td>5.6</td>
<td>A typedef name shall be a unique identifier.</td>
<td>ACS_ATM_sm_t &lt;br&gt;ACS_ATM_timer_t &lt;br&gt;__tag_ACS_ATM_host</td>
</tr>
<tr>
<td>5.7</td>
<td>A tag name shall be a unique identifier.</td>
<td>__tag_ACS_ATM_sm &lt;br&gt;__tag_ACS_ATM_timer</td>
</tr>
</tbody>
</table>

Table 5.2.: Results of static code analysis.

5.1.1.0.1. Rule 2.4 of the MISRA-C:2012 guidelines advises programs to not contain unused tag declarations. This rule is violated by the definition of the ACS_ATM_eventQueue_t data structure. The struct definitions created by our code generator declare two tags, each. We require both of them, in order to be able to forward-declare the struct in other files. As a consequence, unused tags are introduced for all data structures that are not forward-declared elsewhere. We allow the deviation from rule 2.4 in the aforementioned case, because it can be considered harmless and allows us to keep the uniform appearance of all data structure definitions.

5.1.1.0.2. Rule 2.5 of the MISRA-C:2012 guidelines advises programs to not contain unused macro declarations. This rule is violated by a number of auto-generated enumeration constants inside the ACS_ATM_shared.h header file. As we described in Section 3.3.3.1, we generate the meta-constants COUNT and DEFINED for each of the produced enumerations. While these constants may not be used by the state machine implementation, they may be used by user code interacting with the state machine. Also, event and signal constants are generated for all events and signals occurring in the model. Given that the state machine implementation does not use the corresponding events or signals, the produced enumeration constants are unused. The deviations from rule 2.5 caused by
5.2. Directed Testing

the aforementioned reasons can neither be avoided, nor do they cause any harm.

5.1.1.0.3. Rules 5.6 and 5.7 of the MISRA-C:2012 guidelines require the names of typedef aliases and struct tags to be unique identifiers. Because we need to forward-declare some of the generated data structures in a number of files, we cannot circumvent the redeclaration of the used typedef aliases and struct tags. In Listing 5.1, we illustrate how this problem arises in the case of the forward declaration of the ACS_ATM_sm_t data structure. Because we require a number of forward declarations in our program, we cannot circumvent the resulting deviations from rules 5.6 and 5.7. Because the redeclared typedef aliases and struct tags refer to the same data type as their initial declarations, we can consider the described deviations to be harmless.

Listing 5.1: Illustration of forward-declaration problem.

```
/* File: ACS_ATM_sm_public.h */
typedef struct __tag_ACS_ATM_sm
{
  ...
} ACS_ATM_sm_t;

/* File: ACS_ATM_timer_public.h */
typedef struct __tag_ACS_ATM_sm ACS_ATM_sm_t;
/* - Violation of rules 5.6 and 5.7 - */
typedef struct __tag_ACS_ATM_timer
{
  ACS_ATM_sm_t * sm;
  ...
} ACS_ATM_timer_t;
```

5.2. Directed Testing

In this section, we describe the testing process that we use to validate the source code that is generated by our code generator.
In order to simulate a generated state machine implementation, we need to stimulate it with external events. In Section 5.2.1, we therefore introduce a format in which we specify stimulation sequences.
Next, we elaborate a set of test cases in Section 5.2.2. Each test case consists of a state machine model and a stimulation sequence. We design the test cases such that they cover all chart elements that we chose to support in Section 3.1.
Finally, we describe the obtained results of the test cases in Section 5.2.3.
5. Validation

5.2. Stimulation Sequences

In this section, we describe the format in which stimulation sequences are defined.
A stimulation sequence may consist of an arbitrary number of stimuli, where each such
stimulus has two attributes. First, a stimulus is associated to a specific point in time at
which it should be triggered. Second, it defines a set of C statements that are executed
as soon as the stimulus is triggered. We show the notation of how a test stimulus is
specified in Listing 5.2.
A stimulation sequence is specified by concatenating one or more stimulus specifications.
We show an example of a stimulation sequence in Listing 5.3. This example sequence
stimulates the state machine by modifying an on-instance attribute (ll. 1-4), calling a
user-defined function after one second (ll. 6-8) and finally, sending a signal after two
seconds of elapsed time (ll. 10-12).

5.2.2. Test Cases

In this section, we describe the set of elaborated test cases that we perform in order to
validate the correctness of our code generator.
We separate the test cases into different categories based on the model elements that
they focus on. These four categories are states, pseudostates, transitions and events. We
describe the test cases of each respective category in Sections 5.2.2.1 - 5.2.2.4.
We show the state machine diagrams used for each of the test cases in Figures A.4 ff.
and the according stimulation sequences in Listings B.8 ff.
Note, that the empty stimulation sequence shown in Listing B.8 is used for all of the test cases to which no other stimulation sequence is assigned explicitly.

5.2.2.1. States

With state-related test cases, we focus on the functionality of the different state types introduced in Section 3.1.1. In TC_SSIMPLE, we test the correct behaviour of atomic and final states. Next, in TC_SSEQUENTIAL and TC_SCONCURRENT we test the correct order of execution of sequential and concurrent states. The test cases TC_SSEQCONCNEST and TC_SCONCSEQNEST test these model elements more thoroughly, as they contain sequential and concurrent states nested within each other. Finally, we test the correct functionality of submachine states in test case TC_SSUBMACHINE.

5.2.2.2. Pseudostates

Similarly as with states, test-cases for pseudostates cover the functionality of the supported pseudostate types introduced in Section 3.1.2. We use test case TC_PJUNCTION in order to test the correct behaviour of junction pseudostates. After that, using test case TC_PHISTORY, we validate the functionality of history pseudostates.

5.2.2.3. Transitions

Next, we describe the test cases focussing on the correct implementation of transitions. We use test cases TC_TINTERNAL, TC_TLOCAL and TC_TEXTERNAL in order to test the behaviour of the different transition kinds explained in Section 3.1.4.1. Additionally, we use test case TC_TGUARD to validate that transition guards are taken into account, correctly.

5.2.2.4. Events

Finally, the following event-related test cases address the correct implementation of the various event types introduced in Section 3.1.5. First, we use test case TC_ECOMPLETION in order to test the generation and consumption of completion events of simple and composite states. Next, test case TC_ECHANGE tests the correct implementation of change events. Test cases TC_ECALL and TC_ESIGNAL stimulate the state machine with a call to a user-defined function and an input signal, respectively. Thereby we validate the implementation of call and signal events. Finally, we use test case TC_ETIME to test the implementation of time events.

5.2.3. Results

As we show in Table 5.4, our code generator passes all of the provided test cases we described in Section 5.2.2. As a consequence, we assume that the generator correctly implements the code mapping described in Section 3.3.
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Output</th>
<th>Obtained Output</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC_SSimple</td>
<td>123</td>
<td>123</td>
<td>✓</td>
</tr>
<tr>
<td>TC_SSequential</td>
<td>1234</td>
<td>1234</td>
<td>✓</td>
</tr>
<tr>
<td>TC_SConcurrent</td>
<td>12223334456</td>
<td>12223334456</td>
<td>✓</td>
</tr>
<tr>
<td>TC_SSeqConcNest</td>
<td>1234555678</td>
<td>1234555678</td>
<td>✓</td>
</tr>
<tr>
<td>TC_SConcSeqNest</td>
<td>1223344556789</td>
<td>1223344556789</td>
<td>✓</td>
</tr>
<tr>
<td>TC_SSubmachine</td>
<td>Start, Left, Right, Turn, Bottom, Top</td>
<td>Start, Left, Right, Turn, Bottom, Top</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Center, Bottom, Top, Turn, Left, Right, End</td>
<td>Center, Bottom, Top, Turn, Left, Right, End</td>
<td>✓</td>
</tr>
<tr>
<td>TC_PJunction</td>
<td>Start, ACG, Start, Parent Entry, Inner, Entry Action, Exit Action, External</td>
<td>Start, ACG, Start, Parent Entry, Inner, Entry Action, Exit Action, External</td>
<td>✓</td>
</tr>
<tr>
<td>TC_PHistory</td>
<td>Start, Parent Entry, Inner, Internal</td>
<td>Start, Parent Entry, Inner, Internal</td>
<td>✓</td>
</tr>
<tr>
<td>TC_TInternal</td>
<td>Start, Parent Entry, Inner, Internal</td>
<td>Start, Parent Entry, Inner, Internal</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Table 5.4.: Results of directed tests.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Output</th>
<th>Obtained Output</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC_TLocal</td>
<td>Start</td>
<td>Start</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Parent Entry</td>
<td>Parent Entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner</td>
<td>Inner</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Child Entry</td>
<td>Child Entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Child Exit</td>
<td>Child Exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>Final</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Local</td>
<td>Local</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parent Entry</td>
<td>Parent Entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sibling Entry</td>
<td>Sibling Entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sibling Exit</td>
<td>Sibling Exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parent Exit</td>
<td>Parent Exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>End</td>
<td>End</td>
<td></td>
</tr>
<tr>
<td>TC_TExternal</td>
<td>Start</td>
<td>Start</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Parent Entry</td>
<td>Parent Entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Child Entry</td>
<td>Child Entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Child Exit</td>
<td>Child Exit</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Parent Exit</td>
<td>Parent Exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>External</td>
<td>External</td>
<td></td>
</tr>
<tr>
<td></td>
<td>End</td>
<td>End</td>
<td></td>
</tr>
<tr>
<td>TC_TGuard</td>
<td>1234</td>
<td>1234</td>
<td>✓</td>
</tr>
<tr>
<td>TC_ECompletion</td>
<td>12</td>
<td>12</td>
<td>✓</td>
</tr>
<tr>
<td>TC_EChange</td>
<td>12</td>
<td>12</td>
<td>✓</td>
</tr>
<tr>
<td>TC_ECAll</td>
<td>$5 + 3 = 8$</td>
<td>$5 + 3 = 8$</td>
<td>✓</td>
</tr>
<tr>
<td>TC_ESignal</td>
<td>12</td>
<td>12</td>
<td>✓</td>
</tr>
<tr>
<td>TCETIME</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.9s passed</td>
<td>4.9s passed</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.1s passed</td>
<td>5.1s passed</td>
<td>✓</td>
</tr>
</tbody>
</table>
5.3. Robustness Testing

In this section, we describe the results of the robustness tests that we performed on our code generator. We provided a total of 6 erroneous input models to our code generator. Each of these models ignores one of the major imposed model restrictions that we introduced in Section 3.1. We describe the test cases that we used for the robustness testing of our code generator in Section 5.3.1 and the obtained test results in Section 5.3.2.

5.3.1. Test Cases

In this section, we describe the test cases that we used for the robustness testing of our code generator. First, we describe the test cases focusing on unsupported model elements in Section 5.3.1.1. Next, in Section 5.3.1.2, we describe the test cases focusing on invalid numbers of initial states inside orthogonal regions. And finally, in Section 5.3.1.3, we describe the test cases focusing on invalid numbers of shallow history states inside orthogonal regions. The set of performed tests cannot guarantee that our code generator will identify any error contained by input models. Instead, we choose test cases that cover the errors that are most probably made by model engineers.

5.3.1.1. Unsupported Model Elements

The model used for the TC_RDeepHistory, TC_RFork and TC_RJoin test cases contain a deep history-, fork- and join pseudostate, respectively. As we mentioned in Sections 3.1.2.6 and 3.1.2.4, our code generator does not currently support any of these three model elements. Therefore, the expected output of the aforementioned test cases is an error message.

5.3.1.2. Initial Pseudostates

The model used for the TC_RNoInitial test case contains an orthogonal region that does not contain an initial state. The model for the TC_RTWOINITIALS test case, on the other hand, contains an orthogonal region that contains two initial states. As we mentioned in Section 3.1.2.1, we require each orthogonal region of the model to contain exactly one initial state. For this reason, the expected output of the aforementioned test cases is an error message.

5.3.1.3. History Pseudostates

The model used for the TC_RTWOHISTORIES test case contains an orthogonal region that contains two shallow history pseudostates. According to the UML standard, no more than one shallow history pseudostate may be contained by any given orthogonal region [Gro15c][pp. 310-311]. Therefore, the expected output of the aforementioned test case is an error message.
5.3.2. Results

As we show in Table 5.6, our code generator passes all provided test cases, except TC_RDeepHistory. We fail this test, because the XMI documents exported by the used version of the PTC Integrity Modeler software do not contain any information about whether a given history pseudostate is deep or shallow. We will contact the publisher of the modelling tool in order to solve this issue in the near future.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Output</th>
<th>Obtained Output</th>
<th>Passed</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC_RFork</td>
<td>Error Message</td>
<td>Error Message</td>
<td>✓</td>
</tr>
<tr>
<td>TC_RJoin</td>
<td>Error Message</td>
<td>Error Message</td>
<td>✓</td>
</tr>
<tr>
<td>TC_RDeepHistory</td>
<td>Error Message</td>
<td>Generated Code</td>
<td>✗</td>
</tr>
<tr>
<td>TC_RNoInitial</td>
<td>Error Message</td>
<td>Error Message</td>
<td>✓</td>
</tr>
<tr>
<td>TC_RTwoInitials</td>
<td>Error Message</td>
<td>Error Message</td>
<td>✓</td>
</tr>
<tr>
<td>TC_RTwoHistories</td>
<td>Error Message</td>
<td>Error Message</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.6.: Results of robustness tests.
6. Conclusion

In this thesis, we propose an approach to design and implement a code generator capable of transforming XMI-encoded UML/SysML state machine models to executable code. In Section 3.1, we limited the scope of the code generator to a reasonable set of supported model elements. We also expressed a number of additional restrictions on input models that increase their clarity and promote a homogenous appearance of the model database. The elaborated rules are included in Airbus-internal modelling guidelines and serve as reference for future model-based projects.

After having decided on the scope of the code generator, we elaborated a systematic mapping between these model elements and C code patterns in Section 3.3. We described the implementation of this mapping in Chapter 4. We separated the code generator into a frontend and backend, in order to allow for easy exchange of input and output formats. Also, the chosen modular approach to the parser implementation allows it to be easily extended by additional supported model elements, in the future.

The testing results we obtained in Chapter 5 proved that the transformation rules we elaborated in Chapter 3 lead to correct implementations of arbitrarily complex state machine models, given that they satisfy our imposed restrictions. As we demonstrated in Section 5.3, our code generator can also prevent model engineers from generating code for input models that contain commonly made errors.

Not only does our code generator greatly reduce the manual programming workload, but it also simplifies the verification process of generated implementations by adhering to relevant coding guidelines and taking into account Airbus-internal standards. This effect is increased even further by the code-level traceability of system requirements that our code generator achieves.

On top of these advantages, the readability of the generated code benefits from its uniform appearance. As a consequence, the code generated by our tool is far easier to understand for model engineers than handcoded solutions, allowing for more efficient maintenance.
7. Future Work

In this chapter, we summarize the potential future work on the code generator. In the future, we may extend the capabilities of our code generator by adding more supported chart elements. For the reasons described in Section 3.1, we currently do not support e.g. fork, join or deep history states pseudostates. Due to the chosen implementation pattern that we described in Chapter 4, future support for additional model elements can easily be added iteratively. Furthermore, as we motivated in Section 1.1, in the near future we will try to achieve a DO-330 certification of our code generator. This certification will allow us to fully automate the verification process of the generated state machine implementations, eliminating the need for manual verification tasks. A successful DO-330 certification will lead to another significant improvement of the efficiency of the systems engineering process at Airbus.
A. Figures

A.1. Example State Machine

Figure A.1.: Example state machine model of an ATM.
Figure A.2: Package hierarchy of ATM example.

Figure A.3: Internal logic of Login sub-statemachine.
A.2. Test Cases

A.2.1. Directed Testing

A.2.1.1. States

Figure A.4.: TC_SSimple.

Figure A.5.: TC_SSequential.

Figure A.6.: TC_SConcurrent.

Figure A.7.: TC_SSeqConcNest.
Figure A.8.: TC_SConcSeqNest.

Figure A.9.: TC_SSubmachine.
A.2. Test Cases

A.2.1.2. Pseudostates

Figure A.10.: TC_PJunction.

Figure A.11.: TC_PHistory.
A.2.1.3. Transitions

Figure A.12.: TC_TInternal.

Figure A.13.: TC_TLocal.

Figure A.14.: TC_TExternal.

Figure A.15.: TC_TGuard.

A.2.1.4. Events

Figure A.16.: TC_TECompletion.

Figure A.17.: TC_EChange.

Figure A.18.: TC_ETime

Figure A.19.: TC_TECall.

Figure A.20.: TC_ESignal.
A.2.2. Robustness Testing

A.2.2.1. Unsupported Model Elements

Figure A.21.: TC_RFork.

Figure A.22.: TC_RJoin.

Figure A.23.: TC_RDeepHistory

A.2.2.2. Initial Pseudostates

Figure A.24.: TC_RNoInitial.

Figure A.25.: TC_RTtwoInitials.
A.2.2.3. History Pseudostates

Figure A.26.: TC_RTTwoHistories
B. Listings

B.1. Generated Example Code

```c
/**
 * Auto-generated state entry function.
 *
 * Entered state: Initial (Lane A).
 *
 * @param sm
 * Pointer to currently running state machine instance.
 *
 * @param topmostState
 * Parent state inside shared parent region of transition
 *
 * source and target states.
 *
 **/ static void ACS_ATM_sm_entry_19(ACS_ATM_sm_t * const sm,
 ACS_ATM_state_t topmostState)
 {
 if (topmostState != ACS_ATM_sm_STATE_19)
 { ACS_ATM_sm_entry_18(sm, topmostState);
 }
 // Start timer(s).
 ACS_ATM_sm_createTimer(sm, ACS_ATM_EVENT_34, 1000);
 // Create completion event.
 ACS_ATM_eventQueue_enqueue(&sm->internal.eventQueue, ACS_ATM_EVENT_19);
 }
```

Listing B.1: Generated entry function of state Initial in region Lane A.
Listing B.2: Generated exit function of state Self Test inside region Main.
B.1. Generated Example Code

```java
static void ACS_ATM_sm_transition_33(ACS_ATM_sm_t * const sm) {
    /* Exit state 'Self Test'. */
    ACS_ATM_sm_exit_19(sm);
    /* Update current state. */
    sm->internal.currentState[ACS_ATM_REGION_0] = ACS_SM_STATE_17;
    /* Update history state. */
    sm->internal.stateHistories[ACS_ATM_HISTORY_0] = ACS_SM_STATE_17;
    /* Enter state 'Failure'. */
    ACS_ATM_entry_17(sm, ACS_ATM_STATE_17);
}
```

Listing B.3: Generated transition function for the transition from Self Test to Failure.

```java
extern void ACS_ATM_sm_sendSignal(ACS_ATM_sm_t * const sm,
                                   ACS_ATM_signal_t signal) {
    switch(signal) {
        case ACS_ATM_SIGNAL_CANCEL:
        {
            ACS_ATM_eventQueue_enqueue(
                &sm->internal.eventQueue,
                ACS_ATM_EVENT_29
            );
            break;
        }
        ...
        case ACS_ATM_SIGNAL_SHUTDOWN:
        {
            ACS_ATM_eventQueue_enqueue(
                &sm->internal.eventQueue,
                ACS_ATM_EVENT_34
            );
            break;
        }
        default:
        {
            NOOP;
            break;
        }
    }
    ACS_ATM_sm_update(sm);
}
```

Listing B.4: Generated sendSignal function.
extern void ACS_ATM_sm_createChangeEvent(ACS_ATM_sm_t * const sm)
{
    ACS_ATM_event_t event;
    bool_t required;

    /* Event queue is not empty. */
    if(ACS_ATM_eventQueue_tail(&sm->internal.eventQueue, &event))
    {
        /* Tail element of event queue is not a change event. */
        if(event != ACS_ATM_EVENT_CHANGE)
        {
            required = TRUE;
        }
        else
        {
            required = FALSE;
        }
    }
    else
    {
        /* Event queue is empty. */
        required = TRUE;
    }
    if(required)
    {
        ACS_ATM_eventQueue_enqueue(
            &sm->internal.eventQueue,
            ACS_ATM_EVENT_CHANGE
        );
    }
}
Listing B.6: Generated processEvent function for the main region of the Login submachine state. (Excerpt)
Listing B.7: Generated state machine update function.

```c
extern void ACS_ATM_sm_update(ACS_ATM_sm_t * const sm)
{
    ACS_ATM_event_t event;

    /* Temporarily switch to internal call mode. */
    sm->internal.callmode = ACS_ATM_CALLMODE_INTERNAL;

    while(ACS_ATM_eventQueue_dequeue(&sm->internal.eventQueue, &event))
    {
        if(ACS_ATM_sm_process_0(sm, event))
        {
            /* Notify state machine instance of possible attribute changes. */
            ACS_ATM_sm_createChangeEvent(sm);
        }
        else
        {
            NOP;
        }
    }

    /* Switch back to external call mode. */
    sm->internal.callmode = ACS_ATM_CALLMODE_EXTERNAL;
}
```
B.2. Test Case Stimulation Sequences

Listing B.8: Empty stimulation sequence.

Listing B.9: Stimulation sequence for TC_EChange.

Listing B.10: Stimulation sequence for TC_ECall.

Listing B.11: Stimulation sequence for TC_ESignal.

Listing B.12: Stimulation sequence for TC_ETime.

Listing B.13: Stimulation sequence for TC_PJunction.
Bibliography


[QL] LLC. Quantum Leaps. Quantum platform in c++.


