

RWTH Aachen University
Chair for Software Modeling and Verification

Bachelor Thesis

Generating Simulink Models from AADL system descriptions

Lukas Armbrorst
(321322)

28.09.2015

First Supervisor: Joost-Pieter Katoen

Second Supervisor: Thomas Noll

Advisor: Harold Bruintjes

I hereby declare that I have created the work at hand completely on my own and used no other sources or tools than the ones listed. Citations were marked accordingly.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.
Aachen, den 28. September 2015

Abstract

To verify the correctness of system software, developers use various types of models to represent the system. The SLIM language is one such modelling language. It is a dialect of the AADL (Architecture Analysis and Design Language), which is a standardised language to develop one single architecture-centric model of the entire system that can be used during the entire design process from the early requirement analysis until the deployment and maintenance. SLIM is used in the COMPASS tool-set, which offers various approaches to verify the safety and reliability of a system. The semantics of SLIM are formally defined, which allows for rigorous formal analyses. However, industrial applications more often use Simulink, which is an extension of MATLAB. Simulink can analyse and simulate block-oriented models, and offers automatic code generation. This thesis proposes a translation from SLIM to Simulink. With such a link, developers can use the COMPASS tool-set for system verification, and later they can use the generated Simulink model within their established engineering processes, for instance to generate code. The thesis describes the two languages involved, and investigates the commonalities and differences in their semantics. Based on this, the aspects of SLIM are mapped to Simulink features as accurately as possible. To demonstrate the viability of the translation, a case study is presented, which involves a physical system that can run software generated by Simulink. It is designed using the COMPASS tool-set, and gives an overview of the concrete application of the translation.

Contents

1	Introduction	3
2	Preliminaries	5
2.1	The AADL and its dialect SLIM	5
2.1.1	SLIM	6
2.2	Simulink	8
2.2.1	Stateflow	11
3	Transformation	12
3.1	General Aspects of SLIM	14
3.1.1	Data Types	15
3.1.2	Operators in expressions	16
3.2	Components and their Connections	18
3.2.1	Ports	18
3.2.2	Subcomponents	20
3.2.3	Port Connections	23
3.2.4	Flows	25
3.3	Modes and Mode Transitions	26
3.3.1	Stateflow Representation	27
3.3.2	Simulink and Stateflow semantics and SLIM semantics compared . .	31
3.3.3	Simulink Blocks Involved	38
3.4	Error Model	40
4	Case Study: a Lego Satellite	42
4.1	The Lego Robot	42
4.2	The SLIM and Simulink models	43
4.3	Remaining manual tasks	46
5	Conclusion	47
5.1	Future work	48
A	Auxiliary Systems	52
B	Stateflow Syntax Example	54
C	Robot Model	56
C.1	SLIM	56
C.2	Simulink	61

Chapter 1

Introduction

In modern days, computer systems become more and more powerful, especially in the form of embedded systems which are used in nearly every aspect of our every day life. With these expanding capabilities there comes an increase in complexity. Meanwhile, the use of such systems in critical applications like avionics impose strong requirements on the framework, for example concerning reliability and safety. To ensure such requirements are met, models of the complex system are used to simulate its behaviour and verify it. This has led to *Model-Based Engineering (MBE)*, where models are a crucial part throughout the entire development of the system, from early requirement analysis up to the maintenance of deployed software.

At different stages of development, different models might be used for varying purposes: In the requirement analysis, coarse models are constructed to test the viability and consistency of the requirements, while during the implementation fine-grained models are needed to accurately predict the execution time of a routine and check deadlines. But as requirements often change during the system's development, this may lead to inconsistencies between the models, and the results from early analyses become outdated. It takes considerable time, effort and resources to adapt those models and redo the analyses. AADL (*Architecture Analysis & Design Language*, [24]) is a standard for architecture-centric modelling, providing one unified model throughout the entire development. It already supports coarse early designs, and can be refined throughout the process, until reaching a detailed description of all the aspects of the system. This leads to one consistent up-to-date model during the entire course of development. Models for specific test can be generated from this central model, which simplifies the reiteration of early analyses after changes in the model. Additionally, one overall model allows to detect system-level problems at an early stage, which otherwise would only be found at integration tests after the system is operational. For example, two components might interact through an interface. In the early design phase, the concrete type of the data might not be specified yet. Later on, the components are modelled separately, so a type mismatch of the interface would not be detected until the finished components are integrated into one system. Fixing this requires large and expensive changes in the existing implementation. An up-to-date overall model would detect the inconsistency as soon as the types are fixed, avoiding the problem.

This thesis works on a dialect of AADL called SLIM (*System-Level Integrated Modelling*, [3]). It is based on a reduced set of AADL features, so that the semantics of the model can be specified in a very formal way. This allows for rigorous formal analyses of the model. The thesis describes a translation of SLIM models into Simulink models.

Simulink [25] is a toolbox within MATLAB [19] to simulate and analyse control systems in a numerical way. Simulink models consist of function blocks which represent the features of the system. There are various blocks to describe different operations, ranging from simple sums to black boxes representing an entire Simulink model and its function-

alities. Simulink is well integrated into MATLAB, so MATLAB can be used to examine the results of Simulink simulations, for instance to plot large amounts of data generated by the simulation. Besides, there are many tools to analyse Simulink models, for example the *Simulink Verification and Validation* tool [26], which among other features can check Simulink models against specified requirements. While Simulink models describe continuous data flows between the components, its extension *Stateflow* [27] can define finite state machines to model discrete behaviour of the system as well. These state machines are blocks themselves, and thereby easily integrated into the continuous system. To bridge the gap between the model and the actual implementation, Simulink offers automatic code generation. In general, the model might differ from the actual system and therefore, analysis results from the model might not be applicable to the real system. Code generation ensures the consistency between the model and its realisation.

First, this thesis will give an introduction into AADL in general and SLIM in particular (Section 2.1), as well as Simulink and Stateflow (Section 2.2). In the main part (Chapter 3), a translation is proposed, mapping the features of SLIM to Simulink, creating a model which is using Stateflow and which mimics the behaviour defined by SLIM as close as possible. While many features could be translated accurately, some concepts could not be described easily or at all. The description will focus especially on differences in the semantics of SLIM and Simulink due to different execution procedures. Additionally, the lack of non-determinism in Simulink plays an important role. In the end, a case study shows the viability of the translation (Chapter 4). The behaviour of a physical Lego model for a satellite is designed as a SLIM model, which is then translated into Simulink. Simulink models can be deployed onto Lego Mindstorms robots directly. That shows how a system can be designed and analysed in AADL, transformed into Simulink, simulated and analysed further and finally automatically transformed into code and deployed. The case study involves many features commonly used in SLIM, and brings attention to how the restrictions implied by the translation should be taken into account when preparing a model that should be translated later.

Chapter 2

Preliminaries

As this thesis centres on the translation from AADL models — and in particular SLIM models — to Simulink models, this chapter gives an introduction into the two systems involved. First, the AADL is described, along with the features and capabilities it offers and the characteristics of its dialect SLIM. Afterwards, Simulink is introduced, including its extension Stateflow. Stateflow offers the ability to model discrete behaviour within the otherwise continuous Simulink environment using state machines .

2.1 The AADL and its dialect SLIM

The AADL (*Architecture Analysis & Design Language*, [24]) is a standard for architecture-centric model-based engineering, published in 2004 by SAE International [22]. The two main concepts which constitute the model are on the one hand descriptions of the visible interfaces of the system’s components called *component type descriptions*. This is done by declaring ports, which can either be incoming, outgoing, or bidirectional. On the other hand, there are blueprints for the actual realisation of these components in *component implementation descriptions*, which include the hierarchical structure of subcomponents contained in a component, as well as definitions on how the parts of the component are connected and interact. There might be more than one implementation for the same component type, while the implementation must have exactly one type. So in early stages of the development, the general architecture of the system can be designed, defining the main components of the system and their interfaces. Over time, those are refined by adding implementations and necessary subcomponents. This approach ensures that there is a model for the entire system from the start, which enables developers to detect system-level problems early on, when they still can be fixed more easily compared to errors detected only after the implementation of the modules is finished.

To describe the properties of the components more accurately, the AADL divides them into *categories*. For example, a software component can be a **thread** or a **process**. Because the AADL was developed with a special regard towards embedded systems where the hardware is important, there are also hardware categories such as **processor** and **memory**. Incorporating the hardware into the model offers more rigorous analysis features. For example, to study scheduling and deadlines precisely, processor frequencies are necessary. **Devices** encapsulate interaction with the environment of the model, such as sensors and actuators. Hardware, software and devices together compose the **system**. Besides an increased clarity which results from this separations, the categories provide the ability to describe physical bindings between components. For example, a **process** might run on a certain **processor**, which accesses a **bus**. Therefore, model checking tools can for instance validate that two processors might only communicate if they access a common bus. Additionally, these categories impose further restrictions on the component, for example that **devices** cannot have further subcomponents.

These components can be grouped in *packages*, which allows the independent development of different modules of the system. Packages contain public definitions which are accessible from other packages, as well as private definitions which are hidden to the outside, so the user can control visibility, similar to object oriented languages. Another similarity is that the AADL offers inheritance relations between components via the **extends** operation, for example to add functionality. Thus, users might label components as abstract, and later refine them for instance either as a simple *device*, or as a detailed **system**. Components can have *properties* attached to them. The different component categories offer different predefined properties, such as frequencies for **processors** or deadlines for **processes**. Additionally, the user can specify his own properties to describe further aspects of his system. These properties are important to check certain characteristics of the system, for example whether it meets its deadlines using the current processor speed.

To describe the interaction of subcomponents, a component implementation can declare connections between the specified ports of these subcomponents. Besides these one hop connections, the user can define entire end-to-end data flows from a signal source through various connections and subsystems to a signal sink. Latencies can be attached to the connections, so that model checking tools might evaluate whether a flow fulfils requirements concerning its overall response time for safety or reliability reasons.

The component can be dynamically reconfigured using *modes*. These modes constitute a finite state machine as abstraction of the system's behaviour. Port connections, flows and subcomponents might be active only in certain modes, describing different configurations of the component. A typical example is a redundant system that can switch between two identical subcomponents if one of them fails. Properties can depend on the mode as well, such as processors which dynamically change the frequency. The change between modes is done with a transition, which is triggered by an event. That event might originate from the component itself, like an error occurring suddenly, or they are propagated through ports and port connections from another component.

Those component definitions only describe blueprints for the implementation of the system, like classes in object-oriented languages. Model checking tools using the AADL work on actual instances of the system. Therefore, the user has to declare a system to be the root component. From there on, instances for the subcomponents are created and evaluated. Hence, the user can specify any part of the system as a root to perform tests only on certain modules of the system.

The AADL standard was designed to be easily extensible, so users can adapt it to their specific purposes. For example, there are abstract categories, so the user can create additional component categories. Additionally, several annexes were defined to extend the standard further. This includes a standard for a graphical definition of the otherwise text-based model description, but also additional features such as the possibility to model error behaviour. There also is a definition for an XML-based representation of the model, which permits easy interaction with other tools. The standard is defined formally using a context-free grammar, and can be purchased from SAE International [22]. A more hands-on introduction into the standard and its use offers *Model-Based Engineering with AADL* [7], which is co-authored by the technical lead and author of the AADL standard.

2.1.1 SLIM

This thesis will concentrate on a dialect of the AADL called SLIM (*System-Level Integrated Modelling*, [3]), which is based on a reduced feature set of AADL. This allows for a full formal definition of the semantics of the model, and thereby more rigorous model checking.

Listing 2.1 gives an example for such a SLIM model definition, using the most common features. Like in the AADL, a model in SLIM consists of components, whose interfaces are specified in *component types* in the form of ports, while the behaviour can be specified in *component implementations*. Unlike in AADL models, SLIM component types can

Listing 2.1: Exemplary definition of a SLIM model

```

system ExampleComponent

end ExampleComponent;

system implementation ExampleComponent.ExampleImplementation

end ExampleComponent.ExampleImplementation;

```

only specify ports, while flows and modes are entirely hidden within the implementation. Besides, the number of supported categories is reduced (see Section 3.1 for a complete list). One important difference to the AADL is the missing ability to attach *properties* to a component, which are used in model checking such as processor latency for timing analyses. The COMPASS toolset [5], which works on SLIM models, allows to define similar properties and validate them, but they are handled differently and those properties are kept separate from the actual SLIM model. Without property definitions, the categories lose a lot of their power. However, physical bindings such as a bus access can still be defined, and the restrictions regarding subcomponents still apply.

In SLIM, component types can only provide data ports of built-in data types and event ports. A combination of these two, such as the AADL **event data ports**, is not supported in this version of SLIM, and SLIM only supports unidirectional ports. Event connections are synchronous, so the triggering event is only allowed to happen if another component is ready to receive it. Event connections use multicast, so several components can receive the same event (fan-out). Analogously, a component can listen to multiple events through the same port (fan-in), triggering the internal event on any of the attached events. For data ports, only fan-out is supported, to avoid inconsistencies from concurring inputs. *Flows* have an entirely different meaning in SLIM: In the AADL, flows specify a sequence of connections through various subcomponents. In SLIM, flows only describe one step, similar to a data port connection. But in contrast to simple one-to-one port connections, flows allow to define a port's value through an expression involving logical and arithmetic operators. Thereby, SLIM provides a much more detailed description of what happens data-wise within a component implementation than AADL can. Only this concrete description makes the transformation into Simulink and actual program code possible.

Another restriction from standard AADL concerns components of the category **data**. While they can be specified like any other component in the AADL, including corresponding implementations and subcomponents, SLIM only supports predefined data types and therefore no data component implementations (see Section 3.1.1 for a list of supported types). Therefore, data subcomponents represent data variables of the given type.

SLIM offers the concept of *modes* similar to the one in AADL, to dynamically re-configure the component configuration by enabling and disabling subcomponents, port connections and flows. In addition to their functionality in AADL, *mode transitions* in SLIM specify not only source and target mode and a triggering event, but can also have *effects* on the data subcomponents and ports and a *guard* to enable the transition conditionally. Similar to the different definition of flows, this offers a better view on the concrete behaviour of the component concerning data. Another addition to AADL is the concept of *mode invariants*. They can constrain the amount of time spent in a particular mode using for instance **clocks**, and they can contain linear differential equations on data subcomponents of type **continuous** to let them change over time.

SLIM includes features from the AADL Error Model Annex [23]. Therefore, the user

can define *error model types* and their *error model implementations* like a special sort of component category. The error model type can specify *error states* which represent system configurations after certain errors occurred. The error model implementation is a, possibly probabilistic, machine which defines transitions between these error states, which are for example triggered by a sudden *error event*. The error models are defined separately from the nominal components, and can therefore be developed independently. To link them, the user has to specify *fault injections*, which are assignments that are active depending on the error state of the error model. They can override the value of a data element in the nominal component, to model the influence of failures such as broken wires. Through these fault injections, the user ties an error model to a certain nominal component. The nominal component can send a `reset` signal, which the corresponding error model receives, to model how the system can recover from errors. Error in one component can spread through **error propagations** to other error models, similar to event port connections in nominal SLIM components. Error propagations rely on the connections between the corresponding nominal components, so one error model can only propagate an error to another if the respective components are connected.

SLIM models provide an interface for a Fault Identification, Detection and Recovery (FDIR) mechanism. Through this interface, monitoring operations can perform diagnostics on the model, without influencing its behaviour. An example of this are **passive** event port connections. While events can only occur if another component can process them, passively connected components do not count as receiver. Thereby, monitors can listen without activating the event and thereby changing the behaviour. The FDIR mechanism was extended in the *Hardware And Software Dependability for Launchers* (HASDEL) project [8], which is an extension to the COMPASS toolset and which is the basis of this thesis. Another important change was the introduction of **nonblocking** outgoing event ports. Non-blocking ports can emit an event even if no component is ready to receive it. HASDEL introduced time units: Previously, the entire model used absolute numbers for timed constants, which implies one abstract time unit. With HASDEL, the model can use actual time units on its timed constants, for example seconds. This simplifies the handling of time, such as the comparison of two timed values. But they only pose as syntactic sugar, as the system internally converts all timed values into one base-unit (usually seconds). For more information on the HASDEL extensions, please refer to the official documentation [21].

2.2 Simulink

MATLAB (*Matrix Laboratory*, [19]) is a numerical tool developed by The MathWorks [12] commonly used for solving various numerical problems and for simulations. Simulink [25] is a graphical extension (*toolbox*) for MATLAB also developed by The MathWorks. It provides the possibility to model a system in a block-oriented way and analyse and simulate the behaviour of that model. An automated code generation allows the user to easily implement the system according to the model.

A Simulink model consists of functional blocks and the signal connections between them (see Figure 2.1). These blocks represent various functionalities, ranging from simple mathematical functions on the input values to complicated derivatives, but also control and data flow such as conditional statements and memory storage. A large block library offers blocks for commonly used functionalities like sums or numerical integration. For more advanced features, the user can define his own functions. Additionally, the user can group blocks together into a subsystem which appears as a black-box, hiding away the implementation details. These subsystems can be nested arbitrarily deep, which permits to model even considerably complex systems without losing clarity. Besides the better organisation, subsystems offer the additional feature that they can be enabled and disabled

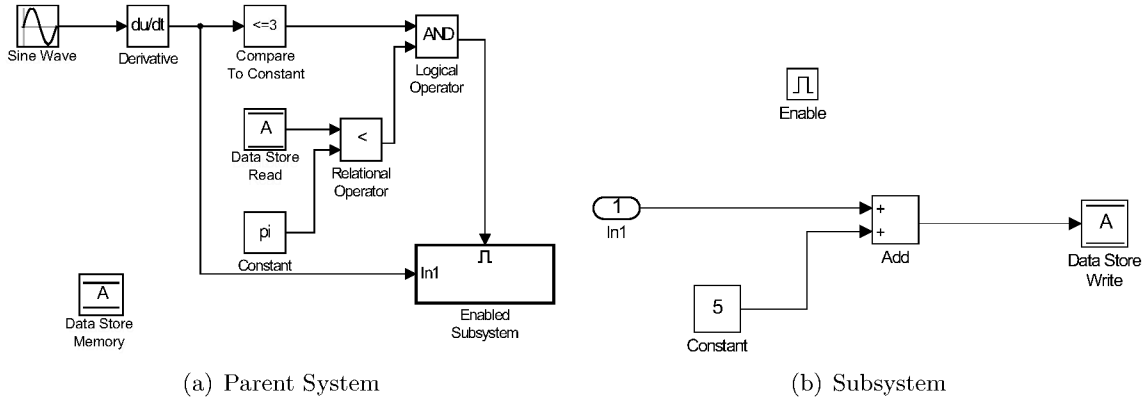


Figure 2.1: Exemplary Simulink system. It defines a data store named A with one reading and one writing block and two constants. Various relational, logical and arithmetic blocks use their output. A *Sine Wave* generator produces a periodic signal, and a *Derivative* operates on that. The result is used by a *Subsystem*, communicated through a port. The subsystem is enabled conditionally, only being executed when the condition on the data values of the parent system evaluates to *true*. Inside the subsystem, the value provided through the data port is incremented by a constant and stored in the data store A .

conditionally. Thus, the system can switch its behaviour depending on certain values. This block oriented view, as well as the graphical interface, renders the model more intuitive and clear than common text based approaches to programming.

For variables, there are *data stores*, which can hold one variable of a user-defined type. These data stores are defined using a `DataStoreMemory` block which names the data store. The surrounding system, as well as all its *Subsystems* can access the data store with `DataStoreRead` and `DataStoreWrite` blocks which are linked to the corresponding data store (see Figure 2.1). If a *Subsystem* defines a data store with the same name, this hides the data store from a higher level.

Despite the continuous nature of signal flows, Simulink operates numerically (like MATLAB), iterating through the simulation in discrete time steps and solving the problem (e.g. an integration) for each iteration. Simulink offers many different solvers, which use different procedures. The Simulink User’s Guide [17, ch. 21] gives an overview of solvers and helps picking the right one. The default solver is a Runge-Kutta (4,5) solver [6] called `ode45`, which is also commonly used in MATLAB. Other solvers use other methods like Backward Differentiation Formulas (BDF), and are suitable for different kinds of problems, for instance depending on the stiffness of the equation (how much it varies). In general, the guide suggests to first try the default `ode45`, which applies well to most problems. Simulink distinguishes between solvers with a fixed step-size, which always take a step of the same size, and those with a variable step-size, which automatically calculate additional values in between steps if the variation of the values in the model exceed a user-specified tolerance. This thesis does not investigate the different solvers, and uses the default `ode45` algorithm in the translation, which uses variable step-size.

Simulink is well integrated into MATLAB, being able to exchange data with the MATLAB workspace. Additionally, MATLAB can explicitly set block parameters. Together with the possibility to run the simulation of the Simulink model from MATLAB, the user can write entire MATLAB scripts to automatically simulate the system with varying parameters, such as using different solvers or changing model-specific values. Afterwards, all MATLAB functionalities can be applied to the results to further analyse the model. It is even possible to call MATLAB functions from within the model, allowing to incorporate nearly arbitrary MATLAB code into the model. Similar possibilities exist for C-code and Fortran for example to include existing code into the code generated from the model.

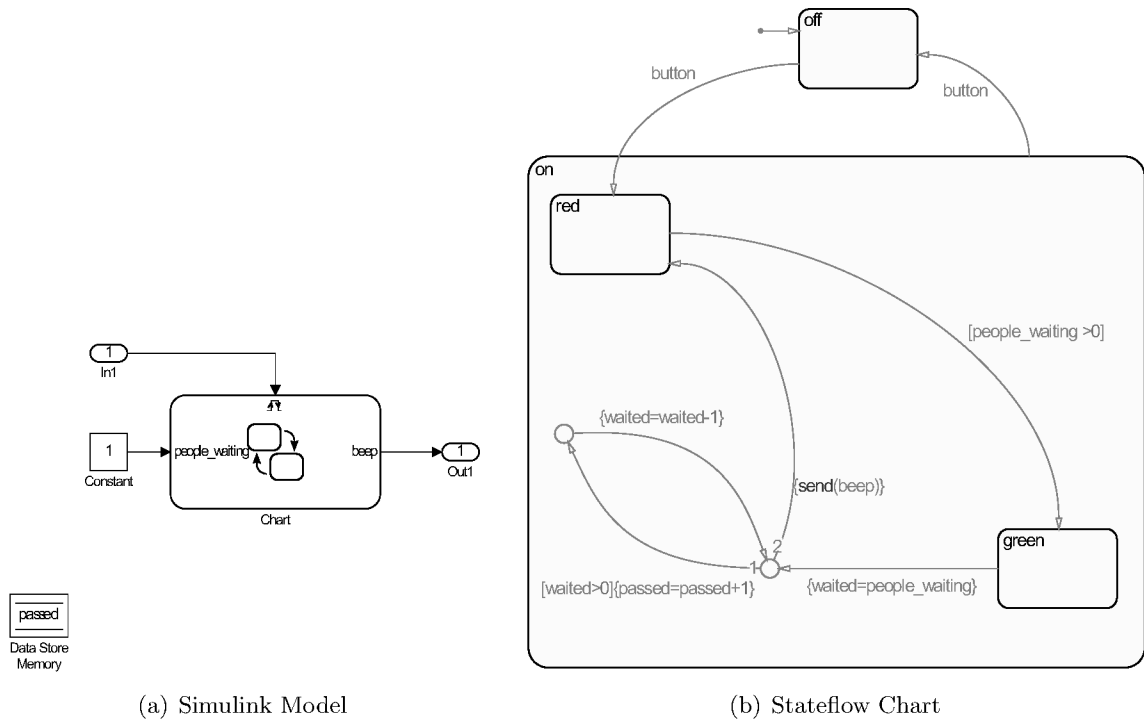


Figure 2.2: Exemplary Stateflow chart of traffic lights that count their users. Initially, the lights are off. After the external event *button*, the lights start to be red. Through a data input from the Simulink model, the chart knows how many people are waiting currently. The lights only turn green if people are waiting. When they switch back to red, they count the people who waited and increase the data store *passed* accordingly. This is done overly complicated with a loop. The numbers on the transitions leaving the first junction indicate the order in which they are checked. When the loop is finished and the lights switch to red, a warning tone is emitted via an event. If the external control sends the *button* event again, the lights turn off, regardless of whether they were green or red.

This is done via S-Function blocks (*system function*). Simulink treats them like black boxes which compute some output depending on the given input, while the actual behaviour of the function is specified in a separate file. These MEX files can either be written manually, or generated automatically for instance from a C++ source file. For more information on MEX files, please refer to [13, ch. 4]. S-Functions are described in detail in the S-Function documentation [14], which gives examples as well as guides on the usage.

To store Simulink models, there exists the default SLX file type that is a compressed package which contains the model encoded in the XML format, and the older MDL format, which is text-based. That format is still supported in recent MATLAB releases, but new features might only be incorporated into the new format (see [16] for details). Because the text-based MDL type is more easily manipulated than the binary SLX format, and because of its compatibility with older MATLAB / Simulink versions, the MDL file format is used here. Simulink offers automatic generation of executable code from the model. Often in model checking, a property has been validated for the model and then it depends on the similarity between the model and the actual system whether the property really holds. Automatic code generation from the model bridges that gap, so that model checking results can be applied to the real system more easily.

2.2.1 Stateflow

Stateflow is an extension to Simulink in the form of a block library for finite state machines to model discrete behaviour of the system. The Stateflow chart defines a finite number of states and transitions between them (see Figure 2.2(b)). States are either mutually exclusive, allowing only one mode at a time to be active, or they can be active in parallel to model concurrent parts of the system. Modes and transitions can be labelled in an *Action Language*, which allows for example to specify a condition that enables a transition depending on the current values of the model and actions that are executed whenever a transition is taken. For modes, the user can specify actions to be executed for instance whenever the mode is entered. Such actions of a mode or a transition might include triggering events, which can synchronise several Stateflow charts across the Simulink model. The chart can import data from the surrounding Simulink model and return data to Simulink, as well as manipulate the data stores of the model. The Action Language can even call MATLAB internal functions such as the sine and access the MATLAB workspace. Stateflow can also interact with the Simulink model by returning the mode which is currently active, enabling or disabling subsystems in Simulink. Therefore, Simulink can be used to develop hybrid systems, combining the quasi continuous behaviour within standard Simulink with the discrete one within Stateflow.

Within the Stateflow implementation, there is an object oriented view: Modes, transitions, events and variables are seen as child elements of the chart. Modes can be *supermodes*, which themselves contain child modes and transitions (see Figure 2.2(b)). If the supermode is active, at least one of its children must be active, and if the supermode becomes inactive, all its children are deactivated. This offers an easy way to group leaving transitions from several modes to the same target with the same transition label into a single transition (see Figure 2.2(b)). If the supermode gets activated with a transition, either a *history junction* lets it return to the last active child mode, or the supermode enters a default setting of its children. Transitions across the hierarchical levels are possible, for example connecting a child mode from one supermode to the child of another supermode.

Transitions always have to start and end in modes, but they can have junctions in between which can have several outgoing or incoming transitions. This allows not only to group transitions with overlapping conditions or actions, but it also allows to build loops as depicted in Figure 2.2(b). Whenever Stateflow recalculates, it either takes the entire transition path from the source mode to the destination, or none of it. So it is not possible to be in between modes at any point in time. If the chart has no modes at all, it is called a stateless flow chart, having only a default transition ending in a junction. In that case, the entire flow is executed whenever the chart is evaluated. If several transitions are leaving from a certain mode or from a junction, Stateflow has a fixed internal order and takes the first transition that is enabled by its condition (and triggered by the current event if there is one). Thus, the execution of a Stateflow chart is always deterministic.

For a practical introduction into using Simulink, *MATLAB – Simulink – Stateflow* [1] and *Matlab und Simulink* [2] (both in German) can be recommended. A very detailed description of all the features of Simulink and Stateflow offer the official user’s guides [17] and [18].

Chapter 3

Transformation

This chapter gives a detailed description of how each aspect of SLIM is translated into Simulink. After the initial paper introducing SLIM [3], several projects have been working on the COMPASS toolset and SLIM. The most recent one is HASDEL (see Section 2.1.1), so the syntax [21] and semantics [20] from that project are used as a definition of the SLIM language. HASDEL introduced several additional features, some of which are translated as well. Restrictions on Simulink are mainly drawn from the Simulink User's Guide [17] and the Simulink Reference [15], for Stateflow from its User's Guide [18].

The starting point for the transformation is an instance model of the given SLIM model. While component types correspond to interfaces in object oriented languages, and component implementations to classes, the instance model corresponds to actual objects that were created according to these specifications. It starts at a root component implementation, which is either determined by the toolset automatically (if there is exactly one component which is not a subcomponent of another), or specified by the user explicitly. From there on, the system is modelled in its hierarchical structure as described by the model. Therefore, component types and implementations which do not occur as dependent subcomponents the chosen root are not part of the instance model, while other component implementations are referenced more than once and thus instantiated multiple times (see Figure 3.1). All these instances combine the respective component type and implementation into one object adhering to the restrictions of both.

The translation is part of an entire tool-chain: It relies on the COMPASS toolset to parse the provided SLIM document and generate the instance model. The toolset also does some transformations on that tree structure to simplify it, such as converting all time units into seconds as common base unit. From the resulting tree, the translation presented in this chapter then generates an MDL file, which is the text-based format for Simulink models. This file can be loaded into Simulink to perform further analysis, for instance with the *Simulink Verification and Validation* [26] tool. Additionally, the Simulink model can be converted into executable code. Hence, code can be automatically generated from a SLIM model description. For more information on the code generation and possible restrictions, please refer to the user's guide [17].

Each section in this chapter gives a description of the SLIM aspect translated as well as its corresponding Simulink representation. The restrictions which apply to the specific features are described in the respective section. There are two general restrictions which apply: First, SLIM models the components running in parallel. When a transition influences several components, they all take the data values from before the step, and independently compute their next step. In opposition to that, Simulink evaluates its blocks successively in each step. Therefore, blocks use the current values from blocks which were already executed, and the values from the previous step otherwise. Simulink tries to order the blocks so that when evaluating a block, all the blocks it relies on are already computed. However, for instance read and write operations on a data store are

```

system Sys
end Sys;
system implementation Sys.Impl
  subcomponents
    m: memory Mem;
    p: process Proc;
  end Sys.Impl;

```

```

memory Mem
end Mem;
memory implementation Mem.Impl
end Mem.Impl;

```

```

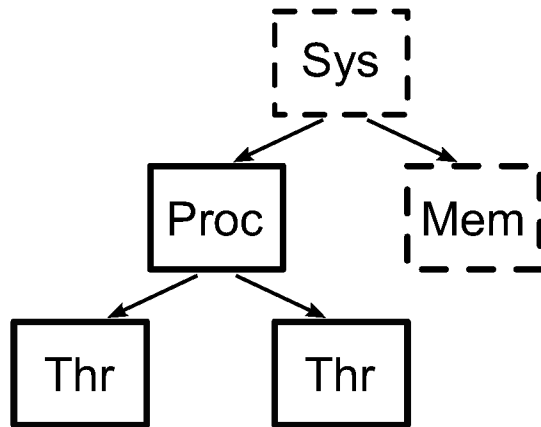
process Proc
end Proc;
process implementation Proc.Impl
  subcomponents
    t1: thread Thr;
    t2: thread Thr;
  end Proc.Impl;

```

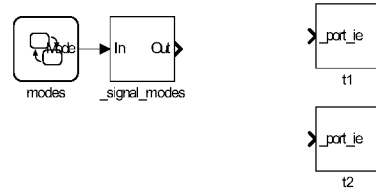
```

thread Thr
  features
    ie: in event port;
  end Thr;
thread implementation Thr.Impl
end Thr.Impl;

```



(a) Hierarchy



from SLIM
Component Implementation: Proc.Impl
package: None
category: process

(b) Simulink

Figure 3.1: SLIM instance models. On the left, an exemplary SLIM model. On the top right, a visualisation of its instance model when using the process as root: The system and the memory are not instantiated (dashed), since they are no subcomponents of the specified root. The thread is instantiated twice because the process has two such subcomponents. Both have an input port named *ie* as defined in the component implementation. On the bottom right the Simulink representation, including two Subsystems.

considered independent, and thus ordered arbitrarily. The translator tries to order the blocks in a fashion which mimics the behaviour of SLIM as close as possible (see especially Section 3.3.2 for details). To do so, some blocks are assigned a priority. The lower the priority value, the higher is the priority and the earlier Simulink tries to execute the block. However, if data dependencies require another execution order, the priority is overwritten. The translator also introduces dummy simulation steps at certain points to finish an execution entirely before doing anything else. This is done with a lock, blocking all other actions. For that purpose, the entire SLIM model is wrapped in a Simulink `Subsystem`, which also defines global auxiliary variables. That subsystem is called “*rootSystem*”. The actual SLIM model is placed within another `Subsystem` called “*slimModel*”, inside the “*rootSystem*”. Additionally, that wrapper system has an additional output port `_clock` which returns the time that Simulink has actually spent simulating the model (without the dummy steps). The leading underscore is introduced to avoid collisions with existing port names, as SLIM only allows identifiers starting with a letter. This concept will be used throughout this chapter whenever a helper function must be named and could potentially clash with an existing name. Unfortunately, there is one exception: Simulink data store names must begin with a letter as well. Therefore, the global lock which turns on and off the dummy steps is called `GLOBAL__EVENT__PHASE` (with double underscores). This name must therefore *not* be used in the SLIM model.

Second, SLIM allows for non-determinism. In the mode transition system (see Section 3.3), if there are several transitions enabled, the system can pick one of them at random (or none). Simulink does not support this concept. Whenever several transitions are possible in the Stateflow chart, it picks the first one. Additionally, due to the stepwise execution, there is a maximal amount of time which passes between two evaluations of a Stateflow chart. Since Simulink takes a transition whenever a transition is enabled, this limits the amount of time spent within a mode without taking any enabled transition. In contrast, SLIM can decide non-deterministically how much time passes between two mode transitions.

3.1 General Aspects of SLIM

Syntactically, the SLIM model *Specification* is defined as follows (from the documentation [21]):

$$\begin{aligned}
 \textit{Specification} & ::= \{\textit{PackageSpecification} \mid \textit{Declaration}\}^+ \\
 \textit{PackageSpecification} & ::= \textit{package identifier} \\
 & \quad \textit{public Declaration}^+ \\
 & \quad [\textit{private Declaration}^+] \\
 & \quad \textit{end identifier}; \\
 \textit{Declaration} & ::= \textit{ComponentClassifier} \mid \textit{ErrorClassifier} \\
 \textit{ComponentClassifier} & ::= \textit{ComponentType} \mid \textit{ComponentImplementation} \\
 \textit{ErrorClassifier} & ::= \textit{ErrorType} \mid \textit{ErrorImplementation} \\
 \textit{ComponentType} & ::= [\textit{fdir}] \textit{ComponentCategory identifier} \\
 & \quad [\textit{features Features}] \\
 & \quad \textit{end identifier}; \\
 \textit{ComponentCategory} & ::= \textit{SoftwareCategory} \mid \textit{HardwareCategory} \mid \textit{CompositeCategory} \\
 \textit{SoftwareCategory} & ::= \textit{process} \mid \textit{thread group} \mid \textit{thread} \mid \textit{data} \\
 \textit{HardwareCategory} & ::= \textit{processor} \mid \textit{memory} \mid \textit{device} \mid \textit{bus} \\
 \textit{CompositeCategory} & ::= \textit{system}
 \end{aligned}$$

$$\begin{aligned}
\text{Features} & ::= \text{PortSpec}^+ \\
\text{ComponentImplementation} & ::= [\text{fdir}] \text{ComponentCategory} \text{implementation Name} \\
& \quad [\text{subcomponents } \text{Subcomponents}] \\
& \quad [\text{connections } \text{Connections}] \\
& \quad [\text{flows } \text{Flows}] \\
& \quad [\text{modes } \text{Modes}] \\
& \quad [\text{transitions } \text{ModeTransitions}] \\
& \quad \text{end Name;} \\
\text{Name} & ::= \text{identifier.identifier}
\end{aligned}$$

On the top level, there are *Packages*. These are merely a method of organisation by providing namespaces, and have no further functionality. Each package can have a public part accessible from the outside, and a private part accessible only from within the package. These accessing rules are already checked by the tool-set when creating the instance model. Therefore, the translation does not need to consider them any more. But each *Subsystem* that is derived from a SLIM component contains an annotation, naming the package to provide traceability.

Component types and implementations can be tagged with `fdir`, to indicate that they are part of the mechanism for Fault Detection, Identification and Recovery (cf. Section 2.1.1). SLIM only offers the interface for monitor processes to observe the model's behaviour, the monitoring mechanism itself is beyond the scope of SLIM. Thus, most features related to FDIR, such as tagging a port as `observable`, have no meaning within the SLIM semantics themselves, and are therefore not translated into Simulink. For the same reason, the `fdir` tag itself is not translated, either.

Similarly, the *Categories* for components are of no concern for the translator. The restrictions they impose are already verified before the translation. The only distinction necessary here is between `data` subcomponents and the other, non-data components, because the former have no corresponding component declaration. The physical bindings like bus accesses are helpful to understand the underlying logic of the system and can be used to check certain properties (e.g. not too many processes should be stored in the same memory, so the binding helps monitoring memory demands). But they have no actual functionality in the model's semantics, and therefore are not translated into Simulink.

3.1.1 Data Types

The types from SLIM are formally defined in the documentation [20] as a set

$$\begin{aligned}
\text{Typ} & := \text{Ide} \cup \{[l..u] \mid l, u \in \mathbb{Z}, l \leq u\} \\
& \quad \cup \{\text{bool}, \text{clock}, \text{continuous}, \text{enum}(e_1, \dots, e_n), \text{int}, \text{real}\}
\end{aligned}$$

where *Ide* denotes the set of identifiers, $[l..u]$ an integer range, and where $n \geq 1$ and $e_i \in \text{Ide}$ for each $1 \leq i \leq n$ hold. Identifiers as a type are only allowed for components to indicate their component type and are thus no data type. Those were translated as follows:

- The data type `bool` from SLIM is translated as `boolean` into Simulink.
- The type `int` from SLIM becomes `int32` in Simulink. Simulink only offers bounded data types, so the arbitrarily large values that SLIM supports cannot be represented in Simulink. But depending on the settings of Simulink, an overflow can generate an error or warning during simulation. While the boundaries can become a problem if the model contains very large integers that risk to overflow, in most use cases a 32-bit signed integer suffices. In COMPASS, this word size can be changed arbitrarily.

Simulink only offers multiples of 8 as a word size, but they might be translated as integer ranges in the future. The option `Saturate on integer overflow` offered by arithmetic blocks like `Sum` is not used in the translation, so values do overflow like they would in actual physical systems.

- The types `clock`, `continuous` and `real` from SLIM are all represented by `double` in Simulink. As mentioned above, all time units are converted to seconds previous to the translation, allowing clocks to be treated as simple numbers. Similarly to the integers, the bounded range of `double` restricts the unbounded types from SLIM. In addition, `double` only offers values with a finite precision, while the SLIM types represent values with infinite precision. Again, these restrictions should not impede the models' functionality too gravely, because the step size of a `double` is very small. But if the model imposes bounds on such a continuous value and takes them more strictly than the `double` precision, then this may result in unwanted behaviour such as reporting a violated bound that a more precise measurement would have allowed.
- The *integer range* from SLIM is modelled by a simple `int32` integer which is restricted by an upper and a lower bound. The range might be small enough to fit into a smaller integer type, but assignments of an integer constant to a data element that uses a range would need a type conversion then, so the standard integer is used. Note that in mode transition effects, COMPASS transforms integer ranges into normal integers as well, so the Simulink translator uses those instead.
- MATLAB treats enumeration like classes, which require a separate MATLAB file that defines them. Therefore, the translator creates a new MATLAB `M` file for each `enum` type that occurs in the model. In that file, each enum value is identified with an integer, starting with 0. The enumeration is named *Enum_* followed by a list of all enumeration values, separated by underscores. This identifies the enumeration uniquely and offers easy traceability. Since Simulink / MATLAB demands that the name of the enumeration class must be unique among the names in the workspace (see [17, p. 54-4]), there should be no variable or data type with such a name in the user's workspace. Whenever an enumeration value is used in Simulink, it must be prefixed with the name of the enumeration, followed by a full stop. This is automatically done by the translator. Not every Simulink feature fully supports enumerated types. While this does not concern any part of the translation, it might impede the work which the user intends to do on the generated Simulink model, for instance logging in generated code. For more information, please refer to the user guide [17, p. 54-24].

3.1.2 Operators in expressions

In general, most operators from SLIM expressions can be mapped directly to operators in Simulink, which behave accordingly. Unfortunately, the translation differs depending on the usage of the expression: First, expressions in SLIM flows are translated into a tree of Simulink blocks (see Figure 3.2). Second, an expression in a SLIM mode transition has to be expressed in Stateflow Action Language, which resembles MATLAB. Finally, an expression in a SLIM default value has to be a MATLAB expression embedded in the respective Simulink block. Hence, different restriction apply to the different types of expressions.

Arithmetic Operators In flow expressions, all arithmetic operators (addition, subtraction, multiplication, division, unary minus and modulus) can be used, as there exists a Simulink block for each of them. The above-mentioned restrictions for the types should be kept in mind. With integer ranges, the boundaries are only implicit: When adding

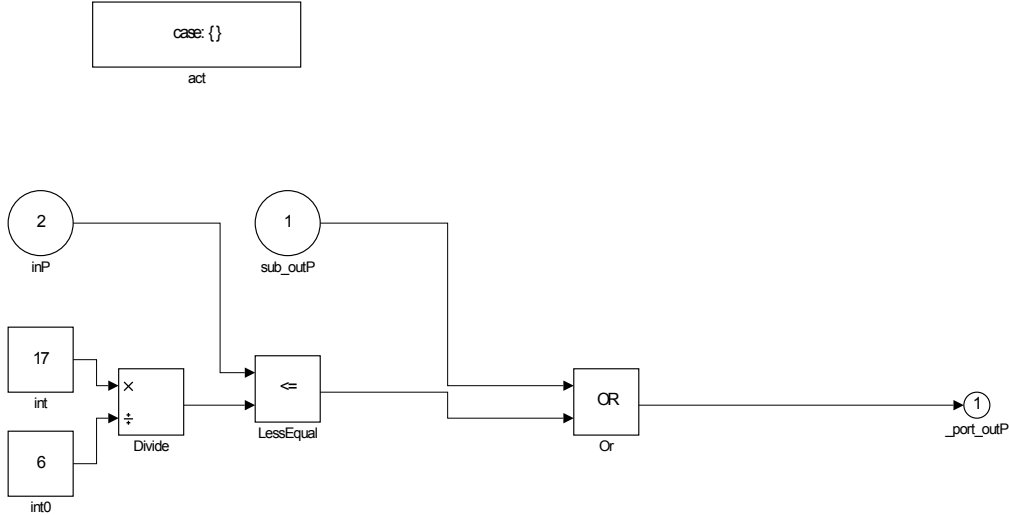


Figure 3.2: Simulink representation of the flow $outP := sub.outP$ or $inP \leq 17/6$, using a block for each operator in the expression.

two values of type *integer range*, the result in SLIM is again of type *integer range*. In Simulink, this is an addition of two integers, which each have bounds. So the result is an integer, which can only be within the bounds derived from the inputs, but is not explicitly restricted, which formally differs from the SLIM definition.

To allow assignments from one integer range to another, SLIM defines an assignment $d := expr$ to a data element d of the range type $[l..u]$ as $d := (expr - l) \bmod (u - l + 1) + l$ in mode transition effects. This is done prior to the translation into Simulink by the COMPASS toolset. For the default values, integer division has to be used explicitly: MATLAB interprets numbers as *doubles* by default, and entirely evaluates an expression, before casting it into the target type. Thus, an expression like $(5/4)*4$ would be evaluated as a *double* calculation, and cast into an integer afterwards. Since that leads to the result of 5 instead of the 4 that SLIM would give, the expression is transformed into $(\text{divide}(\text{int32}(5)/\text{int32}(4)))*4$ to actually be an integer division. This is done whenever both sides of a division in SLIM have an *int* or range type. Otherwise, the same restrictions apply in mode transitions and default values as for flows.

Relational and Boolean Operators All relational operators (equality, inequality, greater, less, greater or equal, less or equal) are fully supported in flows, mode transitions and default values. The boolean operators **and**, **or**, **not** and **xor** are expressed by the according function or Simulink block. For **xnor** there is a Simulink block which is used for flows, while there is no corresponding function in MATLAB, so default values and mode transitions use a negated xor. **If** and **imp** cannot be expressed directly, and are transformed into $a == b$ and $(\text{not } a)$ or b , respectively.

Case Operator The **case** operator from SLIM is only partially supported. It describes an *if ... else if ... else ... end* expression, for which there is no direct counterpart in Simulink. MATLAB supports such expressions. Simulink itself has an if block, but it can only be used to enable and disable subsystems, so each conditional expression would have to be placed in a subsystem and connected to the rest via ports, which would require considerable effort. For mode transitions, the Action Language does not offer a comparable operation, because control flow logic such as MATLAB's *if* is explicitly disallowed in transition actions. So only with a lot of effort it would be possible to mimic the behaviour of the **case** operator (e.g. by defining an external function which is called

in that transition, or by splitting the transition into one transition for each conditional statement and adapt the guard and effect accordingly). Therefore, the `case` operator in general is disallowed.

However, one special case is allowed: If the resulting value of the case expression is of type `bool`, the expression `case b1:e1; ...; bn:en otherwise e0 end` can be transformed into a boolean expression *(b₁ implies e₁) and ((not (b₁) and b₂) implies e₂) and ... and ((not (b₁) and ... and not(b_n)) implies e₀)*. This expression is then translated into the according Simulink or MATLAB form to be used for instance in a mode transition guard.

3.2 Components and their Connections

Formally, components are defined in the semantics [20] as a set $Cmp \subseteq Ide^+$, which is partitioned into the set of data components $DCmp := \{c \in Cmp \mid category(c) = \mathbf{data}\}$ and the set of control components $CCmp := Cmp \setminus DCmp$. Each $c \in Cmp$ has a (component or data) type $typ(c) \in Typ$ (see Section 3.1.1) and an implementation $imp(c) \in Ide.Ide$. Control components are translated into `Subsystems` in Simulink as seen in Figure 3.1(b) (for data components, see Section 3.2.2). Thus, the root implementation is also wrapped in a `Subsystem` “*slimModel*”, which itself is placed within the top-most `Subsystem` “*rootSystem*” besides global definitions such as the simulation time clock (see beginning of this chapter). So at the top level of the Simulink model, there is exactly one `Subsystem` block which represents the entire SLIM instance model. The advantage of this approach is that one can easily build further systems around the SLIM model to represent the environment with which it interacts (see Chapter 4 for an example).

3.2.1 Ports

SLIM component instances have the ports of the corresponding component type. In the SLIM component type, incoming and outgoing event and data ports are syntactically specified as follows (from the documentation [21]):

```

PortSpec ::= identifier: PortType;
PortType ::= in event port | out event port[nonblocking] |
             in data port DataType[Default] |
             out data port DataType[Default][PortTag]
Default ::= default value
PortTag  ::= alarm | observable

```

These are represented in Simulink by `Inports` and `Outports` (together with a *data store* in case of data ports). If a `Subsystem` contains such a port block, Simulink automatically adds a corresponding port to the `Subsystem` block. Simulink `Inports` and `Outports` can have the following attributes:

Name The name of the port block within the `Subsystem`. It is also used on the level above on the `Subsystem` block.

Number The port number is displayed inside the port block symbol. It defines the order in which the ports are displayed on the `Subsystem` block on the level above.

Data Type (optional) The data type of the port can be specified explicitly. Alternatively it can be set to *Inherited*, meaning Simulink automatically derives it from the type of blocks connected to it.

Minimum (optional) A lower bound for the values that are sent through the port can be specified. Range checking must be activated in the Simulink settings for this parameter to take effect. If no value is given, the boundaries of the data type apply.

Maximum (optional) An upper bound analogous to the lower bound. It is possible to only assign one of them and leave the other one unspecified.

Initial Output (optional) If the Output is contained within a conditional subsystem such as a `SwitchCaseActionSubsystem`, this value is sent out as long as the system was not activated yet.

Output when disabled (optional) This option only exists for Outputs of conditional subsystems as well. It can either be *reset*, so the initial output is sent whenever the system is disabled, or *held* which continues to send the last output value as long as the system is disabled.

Each incoming event port $p \in IEPr(c)$ of a component $c \in CCmp$ (cf. [20]) is translated into an Inport with the name of the SLIM port, prefixed with “_port_”. The number is assigned in an arbitrary order. The optional parameters concerning the data type and the output are not set. The Stateflow chart produces a boolean signal which flips whenever an event occurs, so the data type is derived by Simulink to be `boolean`. But that is an implementation detail of Stateflow, and therefore left out of the port definition. The component’s outgoing event ports $OEPr(c)$ are similarly translated into Outputs.

In SLIM, transitions with an outgoing event port as trigger can only be executed if that port is connected so that there is a transition receiving that event. Only ports tagged with `nonblocking` can emit an event that no one receives. In Simulink, there is no easy way to determine if anybody is receiving the event, so all outgoing event ports are automatically considered non-blocking. Formally, this results in the set of non-blocking outgoing event ports being $OEPrNB(c) = OEPr(c)$. The corresponding tag thus becomes superfluous and is not translated.

Each incoming data port $p \in IDPr(c)$ is translated into an Inport whose name is the name of p in SLIM, prefixed with “_Inport_”. Like with event ports, the number assigned arbitrarily but unique. The data type is set depending on the SLIM data type $typ(c, p)$ according to Section 3.1.1, with the minimum and maximum only set for integer ranges. Analogously, the outgoing data ports $ODPr(c)$ are translated into Outputs with the according prefix “_Outport_” in their name.

The default value of a SLIM port is hard to mimic, because there is no direct equivalent in Simulink. Outputs have the *Initial Output* value, but only if they are contained in a conditional Subsystem which can be disabled. Otherwise, no such option exists. For Inports, there is no way to define default values at all. Therefore, for data ports a Subsystem is added which provides the default value. It has one Inport and one Output and is placed between the actual Inport of the component and all its receivers, or between all the senders and the actual Output of the subsystem. This Subsystem is called like the SLIM port p , adding the prefix “_port_”. This is the reason for the difference in naming data ports and event ports: The blocks inside the system always connect to the “_port_<SLIM port name>” block and need not distinguish between event ports and data ports anymore.

Inside that Subsystem for data ports, there is a `DataStoreMemory` block which stores the current value of the port (see Figure A.1 for the actual implementation). If the component receives a new value through its Inport, that value is sent into this Subsystem and there it is stored in the data store. The blocks of the component then receive that new value from the Subsystem. The write block is given a lower priority than the read block, causing it to be executed earlier (if possible) to avoid read-before-write inconsistencies. Thus, it seems to the other blocks as if they were connected immediately to the Inport. If there has not been any input to the component yet, the Subsystem provides the component with the

initial value of the data store. This is set to be the default value of the SLIM port (if no such value is specified in the SLIM model, it is set to θ ; this differs from the undefined value in SLIM, which cannot be mimicked). That way, the Simulink port mimics the SLIM port's behaviour when it is first activated, as much as possible. If the component is disabled and gets reactivated, there are three possible scenarios: If in the parent system, there is an active port connection or flow driving it, that value is adopted. If all these connections are currently deactivated, a separate flow is added to provide the default value of the port. If no such connections are defined, the data store never received any value and therefore still emits the default value of the port. This behaviour corresponds to the one of SLIM defined in the semantics [20].

For **Outports**, something similar happens: As long as the component is inactive, the *Initial Output* option of the Simulink **Outport** provides the surrounding system with the default value of the SLIM port. But SLIM does not allow references to deactivated sub-components in port connections, flows or mode transitions, so this has no effect. When the component is activated, there are four possibilities: First, if there are active port connections or flows, they determine the value of the port. Second, if there are no active port connections or flows and the component has an **activation** mode, the entire **Subsystem** is reset, including the data stores of its outgoing data ports which return their default values. Third, if there is no **activation** mode and all port connections and flows are deactivated, then the dedicated Simulink block is active and provides the default value (see Section 3.2.3). Finally, if there is no **activation** mode and no port connection or flow defined, the **Outport** continues to return the last value it provided. This is either the last value assigned to it in a mode transition, or otherwise the default value. Again, all this models the behaviour of SLIM as defined in the semantics [20].

Simulink does contain an **IC** block type for initial conditions for signals. But this block forces the signal's value to be the defined initial condition at the beginning of the simulation, while a SLIM default value can be overwritten from the first moment on and never actually be used. So this block can provide wrong values at the beginning of the simulation and is thus not applicable.

All the names for **Inports**, **Outports** and the **Subsystem** for the default value are guaranteed not to clash with any names from SLIM. Due to the leading underscore, they cannot occur as names in the SLIM model. Since SLIM requires all ports within a component to have distinct names, the names are unique within the **Subsystem**. The name of the SLIM port on itself might not be unique, because subcomponents can have the same name as ports (except data subcomponents and data ports, see the documentation [21]). But the prefixes prevent any conflicts.

The last aspect to be considered are the port tags **alarm** and **observable** from SLIM, which are part of the **FDIR** syntax. Since this entire mechanism is omitted in the transformation, these tags have no translation and are simply ignored.

3.2.2 Subcomponents

SLIM components take their hierarchical component structure from the corresponding component implementation description in the SLIM model. It is defined as follows (from the documentation [21]):

$$\begin{aligned} \textit{Subcomponents} & ::= \textit{Subcomponent}^+ \\ \textit{Subcomponent} & ::= \textit{DataSubcomponent} \mid \textit{OtherSubcomponent} \end{aligned}$$

Data subcomponents are comparable to a variable of a built-in type, while non-data subcomponents are instances of other component implementations defined within the same SLIM model. The former are translated into Simulink data stores, and in case of continuous components there are additional **Subsystems**. The non-data subcomponents are converted into **Subsystems** for the according component.

Data Subcomponents

Syntactically, data subcomponents are defined in the documentation [21] as

```
DataSubcomponent ::= identifier: data DataType [Default] [InModes] [observable];  
| identifier: data clock [TimeUnit] [InModes] [observable];
```

Each of these data subcomponents $d \in DCmp$ is translated into a Simulink *data store*. For each of them, a `DataStoreMemory` is created. It has the following options (see [17, ch. 56]):

Name The name of the block.

Data Store Name the name of the data store, independent of the name of the `DataStoreMemory` block which encapsulates it.

Initial Value The initial value of the data store which all its `DataStoreRead` blocks provide as long as no other value is written to the data store. A reset of the surrounding system resets the data store to this value.

Data Type (optional) The data type can be specified explicitly, or Simulink derives it from the blocks connected to it and the default value. To use the data store in Stateflow charts, it has to be defined explicitly.

Minimum (optional) A lower bound can be specified for the values that are stored in the data store. Range checking must be activated in the Simulink settings for this option to take effect. If no minimum is specified, the boundaries of the data type apply.

Maximum (optional) An upper bound analogous to the minimum. None of the two values requires the presence of the other one.

Signal Type (optional) The signal type can either be *real* or *complex*, or it is determined automatically.

In the translation, the name and data store name are the name of the SLIM data subcomponent. As Stateflow charts can access the data store directly by referring to its name, without the need of ports, this naming makes the access in transitions the most transparent. Since ports are renamed (see Section 3.2.1) and all subcomponents have to have unique names, there cannot be any name clashes. Although data stores from higher levels are visible in a `Subsystem` as well, Simulink resolves possible collisions automatically by using the lowest definition (which are hiding higher definitions). Because SLIM components can only see their own data subcomponents without any hierarchical access, this behaviour does not interfere with the SLIM definition. But the hierarchical visibility is used in `Subsystems` explicitly created for Simulink, for instance in updaters (see below). It could also be used to simplify the access to data subcomponents: If a non-data subcomponent wants to access a data subcomponent of one of its ancestors, the value must be handed down through ports in SLIM. This could be omitted in Simulink, placing a `DataStoreRead` directly in the reading `Subsystem`. But for traceability, the translator keeps the behaviour of SLIM, using ports. It also increases the clarity on who is accessing the data store, since there cannot be any access hidden arbitrarily deep in the hierarchy.

The data type is set according to Section 3.1.1, using the minimum and maximum option for integer ranges (and leaving them unspecified otherwise). The signal type is set to *real*, because it must be specified explicitly for Stateflow charts. The initial value is a MATLAB expression derived from the default value of the SLIM data subcomponent as described for the respective operators in Section 3.1.2.

There are three ways a SLIM data subcomponent can change its value: First, a mode transition can assign a value to it in its effect. That case is described in Section 3.3. Second, the data component can change its value over time because it is either a clock, or it is a continuous component and a trajectory equation causes it to change linearly. That is described in the next paragraph (for the definition of trajectory equations see Section 3.3). Finally, the subcomponent is reset to its default value when it is deactivated. To achieve that in Simulink, the translator creates a `Subsystem` if there are modes in which the subcomponent is inactive, which prints the default value into the respective data store. This `Subsystem` is only activated in those modes in which the data subcomponent is deactivated. Therefore, the `DataStoreMemory` block itself can remain active in all modes. This is crucial, because otherwise it would have to be placed in a `Subsystem` to be activated and deactivated, and thus it would not be accessible from the current system.

A special kind of data subcomponents are continuous ones and in particular clocks, which increase their value steadily during the execution. While Simulink has clock blocks, these can only report the total time since the simulation started, and cannot be reset like clocks in SLIM. Therefore, in addition to the data store, the translator adds a separate `Subsystem` to update their value (see Figure A.2). As they still are simple data stores, the handling of continuous components such as clocks does not differ too greatly from that of other data subcomponents, for instance when referenced in mode transitions or flows. The updater *Subsystem* for the clock $d \in Clk(c)$ is named `_update.d`. For continuous data subcomponents $d \in Cnt(c, m)$ for which the invariant of mode $m \in Mod(c)$ specifies a trajectory equation, it is called `_traj-<d>_in-<m>`. Those names cannot clash, because all subcomponents must be named uniquely and only one updater is needed for a clock and only one for each pair of continuous data component and mode. These subsystems have the highest priority in the model, and should therefore be executed as early in each step as possible. Thus, all other components can access the clock value corresponding to that time step, instead of old ones. Nevertheless, Simulink sometimes puts them after other blocks such as the Stateflow chart, recognising a dependency even if there is none. So this might be the reason if clocks and other continuous data components give unexpected values.

The updater consists mainly of an `Integrator` block. This kind of Simulink block takes an external signal and integrates it over time, updating its output in each simulation step of Simulink accordingly. Because Simulink simulates the system in discrete time steps, the subcomponent's value does not increase perfectly continuously, but to the system it seems so. The `Integrator` block offers the following parameters (taken from the references [15, p. 1-929ff.]):

- The *External reset* can either be *None*, meaning there is no external reset, or the `Integrator` has a port for external resets. Then this parameter describes whether the reset happens on rising edges of the signal on that port, on falling edges, or both.
- The *Initial condition source* can be either *internal* or *external*. The latter causes another port on the `Integrator` block to appear, from which the initial condition is taken. This initial value only matters at the beginning of the integration and on each reset, changes during the integration are ignored.
- The *Initial condition* can be set if the *initial condition source* is set on *internal*. Then this MATLAB expression is taken as initial condition for the integration.

Additionally, it has one input port for the signal which is to be integrated. In our case, this signal is coming from a `Constant` block producing a 1 for clocks, or a tree of blocks that produce the value specified in the trajectory equation. The *external reset* is set to *either*. Whenever a mode transition effect assigns a value to a continuous component, for example a clock reset, the Stateflow chart generates an event (see Section 3.3). The type

of that event is either edge as well, and it is connected to the updater and its Integrator. Thereby, the Integrator reevaluates its initial condition and restarts the integration. The *initial condition* of the Integrator is a 0 for clocks, which is the default value for them in SLIM and the only value to which they can reset. Because this value is a constant, it is set within the Integrator block, so the *initial condition source* is set to *internal*. For trajectory equations, the initial condition source is external, because the initial condition is read from the data store, which holds the value that was just assigned to the continuous data component. The integration result is then written into the respective data store. Therefore, it contains the current time in seconds as a **double** in case of clocks. If the clock is only active in certain modes, the updater subsystem is activated only in those modes as well, while the DataStoreMemory always remains active. The trajectory equation updater is only active in the mode whose invariant defines it. Additionally, an EnabledSubsystem checks whether the simulation is currently locked, so time can only proceed during actual simulation steps (see Section 3.3.2).

The time unit attached to a clock has no semantic effect, and is therefore omitted. The tag **observable** is part of the FDIR mechanism, and thus of no consequence here. The **in modes** option is described in Section 3.3.

Non-Data Subcomponents

A non-data component can have other control components as subcomponents. The resulting component hierarchy must be a tree, so no circular dependencies are allowed. Syntactically, these subcomponents are defined in the documentation [21] as

$$\begin{aligned}
 \textit{OtherSubcomponent} & ::= \textit{identifier} : \textit{ComponentCategory} \textit{ UniqueNameOrIdentifier} \\
 & \quad [\textit{Bindings}] [\textit{InModes}]; \\
 \textit{UniqueNameOrIdentifier} & ::= [\textit{identifier} ::] \textit{NameOrIdentifier} \\
 \textit{NameOrIdentifier} & ::= \textit{Name} \mid \textit{identifier} \\
 \textit{Bindings} & ::= \textit{Binding}^+ \\
 \textit{Binding} & ::= \{\textit{stored in} \mid \textit{running on} \mid \textit{accesses}\} \textit{identifier}
 \end{aligned}$$

Like any control component, each control subcomponent $s \in CSub(c)$ of component c (see the semantics [20]) is translated into a Simulink **Subsystem** which has the same name as s . These names do not clash, because SLIM subcomponents must be named uniquely and ports are renamed (see Section 3.2.1). But the information which component implementation they instantiate would be lost. To maintain traceability, the name of the component implementation is added as an **Annotation** inside the **Subsystem**, together with the package and the category. As already mentioned in Section 3.1, this is the only effect the category has on the Simulink model. The SLIM bindings hold no meaning on the computational level of Simulink and only describe logical connections between the components. Hence, they are not translated. If the subcomponent is active only in certain blocks, an **ActionPort** is added, which activates and deactivates the **Subsystem**. It is controlled by a **SwitchCase** block, depending on the state of the Stateflow chart (see Section 3.3).

3.2.3 Port Connections

The SLIM component implementation can specify connections between the ports of the component and the ports of its subcomponents. They can only connect ports of the same type, that is event ports only with event ports and data ports only according to their data

type. They are defined in the documentation [21] as

$$\begin{aligned} \text{Connections} & ::= \text{PortConnection}^+ \\ \text{PortConnection} & ::= \text{EventPortConnection} \mid \text{DataPortConnection} \\ \text{EventPortConnection} & ::= \text{event port NameOrIdentifier} \rightarrow \text{NameOrIdentifier} \\ & \quad [\text{InModes}] [\text{passive}]; \\ \text{DataPortConnection} & ::= \text{data port NameOrIdentifier} \rightarrow \text{NameOrIdentifier} [\text{InModes}]; \end{aligned}$$

In the semantics [20], the set of event port connections $ECon(c, m)$ of the component c that are active in mode m is formally defined as

$$\begin{aligned} ECon(c, m) & \subseteq IEPr(c) \times IES(c, m) \times \{false\} \\ & \quad \cup OES(c, m) \times OEPrt(c) \times \{false\} \\ & \quad \cup OES(c, m) \times IES(c, m) \times \mathbb{B} \end{aligned}$$

where IES denotes the set of incoming event ports of subcomponents which are active in m and OES their outgoing event ports. The boolean flag indicates whether the connection is passive. Analogously, the set of data port connections $DCon(c, m)$ is defined as

$$\begin{aligned} DCon(c, m) & \subseteq IDPr(c) \times IDS(c, m) \\ & \quad \cup ODS(c, m) \times ODPrt(c) \\ & \quad \cup ODS(c, m) \times IDS(c, m) \end{aligned}$$

with IDS and ODS being the incoming and outgoing data ports of subcomponents which are active in mode m .

The `passive` attribute of event connections is used for monitor functions for FDIR (see Section 2.1.1). Since the translator assumes all ports to be non-blocking (see Section 3.2.1), this tag becomes superfluous and is not translated.

Therefore, we can consider all connections to be pairs $(sc_1.p_1, sc_2.p_2)$, with $sc_1, sc_2 \in \{\varepsilon\} \cup Act(c, m)$ being either the current component or one of its subcomponents that are active in the respective mode. From the definition of the sets $ECon$ and $DCon$, we can deduce three kinds of connections:

- *In to in*: Port connections which go from an incoming port of the current component to an incoming port of a subcomponent.
- *Out to out*: Port connections which go from an outgoing port of a subcomponent to an outgoing port of the current component.
- *Out to in*: Port connections which connect an outgoing port of one subcomponent with an incoming port of another subcomponent (connections to itself are disallowed).

In the first two cases, the port connection in Simulink links the `Subsystem` representing the respective subcomponent with the `Subsystem` of the corresponding data port (see Section 3.2.1 on ports), or directly with the `Inport` or `Outport` in case of an event port connection. In the third case, the connection links the `Subsystems` involved. *In to out* connections are not allowed to avoid circular dependencies. However, in case of data connections they can be expressed as flows if they do not create a dependency loop.

If the connection is active in all modes and not *out to in*, it is realised by a simple line between the Simulink blocks. If the `in modes` option is present, then the system is reconfigurable at runtime. The translator realises this by using a `Subsystem` for the connection and adding a `SwitchCaseBlock`, which activates it depending on the Stateflow's state (see Section 3.3). That `Subsystem` contains only one `Inport` and one `Outport` which are

connected directly. The **Subsystem** is named *_Con<unique number>*. This name cannot occur as a name in SLIM, so it is unique within the **Subsystem**. The **Outputport** of the **Subsystem** is reset to the respective port default value upon deactivation. To ensure data consistency, SLIM only allows one data connection or flow to drive each port in each mode (no data fan-in). If several connections or flows drive the port in different modes, their values are merged with a **Merge** block. This block recognises the one signal that is connected to an active **Subsystem**, and overwrites the inactive signals. If there are port connections or flows driving a port, but there are modes when none of them are active, a separate subsystem is added which returns the port's default value to mimic the reset performed in SLIM. The **Merge** block is also used for event ports, which support fan-in: It recognises that only the edges of the signal matter, and can therefore merge the signals (there is at most one event in each time step, see Section 3.3.2). To support fan-out, the line originating at the corresponding port is simply branched.

Subsystems which are activated by a **SwitchCase** block, are treated as an atomic unit. While **Subsystems** in general are only virtual groupings of blocks, and the blocks are considered to be on the same level of the execution hierarchy as the surrounding system, atomic subsystems are seen as one block. This can lead to circular dependencies: If two such subsystems use data from each other, they might use it independently without causing a data dependency cycle. But on the higher level, they are both seen only as black boxes, and the independence of the input and output data within the system is not detected. To break such cycles, a data store is placed in each *out to in* connection to buffer the signal. *In to in* and *out to out* do not need buffers, as they already contain data stores for the default values in case of data ports, so the cycle is already broken. For event ports, there cannot be *in to out* connections, so on the highest level there have to be only *out to in* connections, which are buffered already. Alternatively, the Stateflow chart of that component is involved. As all outgoing signals of charts are buffered (see Section 3.3.3), the chain is broken there. Inside the buffering **Subsystem**, there is only a **DataStoreMemory** block together with one corresponding read and write block each (see Appendix A). The write is prioritised over the read, so the data is fed right through if possible. However, if a dependency cycle is detected, the read is executed before the write and one of the systems is using the data from the previous time step. But because data ports can only depend on timed values through mode transitions, this does not differ from a successive execution in the same time step.

3.2.4 Flows

In contrast to the AADL, flows in SLIM are comparable to data port connections. But while port connections can only directly forward values from one port to another, flows allow more elaborate expressions. Syntactically, these are defined in the documentation [21] as

$$\begin{aligned}
 \textit{Flows} & ::= \textit{Flow}^+ \\
 \textit{Flow} & ::= \textit{NameOrIdentifier} := \textit{FlowExpression} [\textit{InModes}]; \\
 \textit{FlowExpression} & ::= \textit{NameOrIdentifier} \mid \textit{value} \mid \textit{Expression} \textit{operator} \textit{Expression}
 \end{aligned}$$

Their target still has to be a single data port (outgoing of the current component or incoming of a subcomponent), but they allow arbitrary source expressions over readable ports (incoming of current component or outgoing of a subcomponent), as long as the resulting type matches the data type of the target. These SLIM expressions are transformed into one Simulink block for each operator involved (according to Section 3.1.2) as well as for each constant. The blocks are named with their operator or with their data type (in case of a constant). If several blocks with the same operator or type are needed, a unique number is added to their name for distinction, making all names unique. These blocks

are then interconnected with simple lines. An example can be seen in Figure 3.2. Like port connections, flows can be activated depending on the mode. In that case, a `SwitchCaseBlock` is added, and all the Simulink blocks representing the expression are defined within a `Subsystem` which is activated accordingly. Analogously to port connections, the `Subsystem` is named `_Flow<unique number>`. Again, the fanning rules for data ports from Section 3.2.3 apply.

Because flows allow nearly arbitrary expressions as a source, it is not possible to detect *out-to-in* connections. Therefore, all flows that target a subcomponent are buffered as a precaution. Hence, *in-to-in* connections which are described as a flow are buffered twice: Once for the default value of the port, and once by the flow. Due to the general write-before-read policy in the translation, the middle section connects a low-priority read with a high-priority write, which causes Simulink to partly abandon the priority. This can lead to delays in the data stream, even if there is no dependency cycle and no delay would be expected. However, as described in the previous section, such delays should not cause problems in most models.

3.3 Modes and Mode Transitions

SLIM component implementations can define *Modes* and *Transitions* between them. As described above, subcomponents, port connections and flows can be defined to be active only in certain modes, enabling developers to dynamically reconfigure the system. Changes between those modes are defined in the transitions. Syntactically, they are defined as follows (from the documentation [21]):

```

Modes ::= Mode+
Mode ::= identifier: [ModeType] mode [urgent in Delay]
        [while Invariant];
ModeType ::= initial | activation | error
Invariant ::= Expression | TrajectoryEquation
        | Invariant and Invariant
Expression ::= identifier | value | Expression operator Expression
TrajectoryEquation ::= identifier' = number [per TimeUnit]
ModeTransitions ::= ModeTransition+
ModeTransition ::= SourceMode -[ [ModeTrigger] [when ModeGuard]
        [within Delay to Delay] [then ModeEffect] ]
        -> identifier;
SourceMode ::= identifier | *
ModeTrigger ::= NameOrIdentifierDisjunction
        | reset | @activation
NameOrIdentifierDisjunction ::= NameOrIdentifier {or NameOrIdentifier}*
ModeGuard ::= Expression
ModeEffect ::= Assignment{; Assignment}*
Assignment ::= identifier := Expression
InModes ::= in modes (IdentifierList)
Delay ::= number [TimeUnit]

```

Mode invariants can provide linear equations for continuous data subcomponents by specifying their constant derivative. Additionally, they can restrict the time spent in a mode

using clocks and continuous data subcomponents. As described below, Simulink / Stateflow has no concept of non-determinism and therefore takes transitions as soon as they are enabled. Thus, mode invariants are always adhered to during the modes activation, but they can prohibit entering it. The handling of trajectory equations is described in Section 3.2.2, while restrictions are incorporated into transition guards (see Section 3.3.2). The * as a source mode indicates that the transition can be taken in any mode. The mode trigger `reset` is used to interact with the error model (see Section 3.4). The trigger `@activation` is an addition for HASDEL. These transitions are triggered when a subcomponent is reactivated. This feature is not part of this translation, which focuses on the aspects of the original SLIM language.

Simulink itself has no way to model such discrete behaviour. But the extension *Stateflow* provides a `Chart` block which does exactly that. It is a graphical representation of a finite state machine, consisting of states and transitions between them. Thus, a `Chart` block is inserted into each `Subsystem` that represents a SLIM component. It is named “*modes*”. Since this is a keyword in SLIM, it is disallowed for names, so there are no collisions with other blocks. Even if the component implementation does not define any modes, the COMPASS toolset generates a default mode previous to this translation. Therefore, there also is a Stateflow chart inserted into the `Subsystem` representing such a component.

3.3.1 Stateflow Representation

Currently, there is no official documentation of the Stateflow syntax in an MDL file. Thus, most of this section is based on manual inspection of generated MDL files. The Stateflow chart is basically an `S-Function` block, which represents a function defined outside the Simulink model (cf. Section 2.2). In this case, the behaviour of the function is defined in the Simulink model file after the Simulink model itself. This definition consists of one large list containing all entities related to any Stateflow chart in the model, such as states, transitions and the charts themselves (see Appendix B for an example). Each entry in the list has a unique ID, which is used to distinguish between elements of different charts: Many elements have a `linkNode` attribute that is a triple whose first entry is the ID of the parent element, which is usually the chart in which the element resides; the second entry is the ID of a previously defined element of the same type from the same parent, or zero if no such element exists, and the last entry is an analogous pointer to the next element of the same type from the same parent. Thus, for instance all transitions within one chart form a double linked list. The parent then only needs a pointer to the first entry of that list to know all elements. In general, the name for Stateflow objects are taken from the SLIM model. Stateflow objects must not have Stateflow keywords like `enter` or `hasChanged` for their name. For a list of keywords to avoid, please refer to the user guide [18, p. 2-4ff.].

machine and target The first element in the definition is a `machine`, of which there is only one for all charts in the model. Its name is the name of the model file. It has a pointer `firstTarget` to the last entry of the definition, which is of type `target`. That element provides the handle needed by the `S-Function` blocks that use this definition. The `target` is named “*sfun*”, and the `S-Function` uses “*sf_sfun*” as `functionName` to reference it. So together, `machine` and `target` provide the interface for the Stateflow definition.

chart and instance Each Stateflow chart block is represented by two elements in the definition: a `chart` element and an `instance`. They both have the full path of the corresponding chart block in the Simulink model as their `name`. The `instance` contains besides its `name` and `id` only a reference to the `machine` and the `chart` element. The `chart` stores all parameters of the Stateflow chart:

- A reference to the first entry of the double linked list of transitions in that chart (`firstTransition`), which in our case is always the default transition indicating the initial state.
- A reference to the first entry of the list of events (`firstEvent`), if any events are defined in the chart.
- A reference to the first entry of the list of data elements (`firstData`), if any data elements are used in the chart.
- The `updateMethod`, which is set here to *Inherited*, so the chart is updated whenever an input event occurs (or with the rate of the fastest input data if no input events are defined). In the following section, it is detailed how this option is used.
- `executeAtInitialization` is set to one to execute the Stateflow chart when the simulation starts. Otherwise, the state would be undefined until the first step of the simulation.
- The `actionLanguage` is set to *1*, which corresponds to the C version (*2* would be MATLAB). This has slight influence on the syntax of transition labels, although the general language is very similar in both cases. Since MATLAB does not allow the leading underscores which the translator uses to avoid name clashes, the action language here is chosen to be C.
- The parameter `disableImplicitCasting` is set to one. This has the disadvantage that values have to be cast explicitly in mode transitions. But without this, Simulink casts input values implicitly when providing them to the Stateflow chart, which then causes a type error inside Stateflow.
- In case of charts which represent the mode behaviour of a SLIM component, the current state is provided as a data output. Therefore, the `activeStateOutput` attribute is added. This output uses the custom name *“Mode”*. Additionally, the `enumTypeName` for the state output is given explicitly. That enumeration is named after the component implementation, followed by *“_ModeType”*. Because a full stop separates the enumeration type name from the enumeration value, the full stop in the component implementation’s name is replaced with an underscore. Stateflow does not allow two different enumerations with the same name in the model, but if the component implementation is the same, so are the defined modes in the enumeration. Problems can occur only if two packages define component implementations of the same name with different modes. Therefore, the enumeration name is prefixed with the package name if the component implementation is defined within a package.
- the `outputData` is then associated with the corresponding output data element using its ID.

state The states of a Stateflow chart are represented by `state` elements in the definition. Besides their ID, they have the following attributes:

- The `labelString` contains the label of the state, which consists of its name and possible actions, which are for instance to be taken when the state is entered. The translation uses no such actions, so the label is simply the name of the state. This is taken directly from the name of the SLIM mode.
- The chart holds the ID of the `chart` element that represents the Stateflow chart which contains this state. This is the only link between a chart and its states, a reference from the chart to the states is not necessary.

- The **position** of the state is an array of four values. The first two are indicating the position of the upper left corner, the latter two give the size of the state. The translator places the states in a diagonal row from the upper left corner of the chart towards the lower right. The ordering of the states is arbitrary (the order in the memory).

transition The states are interconnected via **transition** elements. In addition to their ID, they can specify

- a **labelString** that contains the label of the transition (see below), using the action language of the respective chart. Here, C action language is used (see above). The initial mode or activation mode of the component is indicated by a default transition with an empty label.
- a **labelPosition**, which indicates the placement of the label in the chart by specifying the upper left corner and the size of the label. The translator places the label at the midpoint of the transition. Since two transitions which connect the same states are placed above each other, the labels collide as well (see placement of transitions below).
- a reference to the ID of the corresponding **chart**.
- the source of the transition **src**, which consists of the ID of the source state and an **intersection** array to characterize the starting point. The ID is omitted for the default transition which has no source state. The array has eight values:
 - The first indicates on which side of the state the transition originates: The value can go from *1* for top to *4* for left. It is *0* for the source of the default transition.
 - The second and third value provide the same information in a different manner: The side is indicated in an x-y-description, being for example *1 0* for the right side or *0 -1* for the top. It is *1 0* for the start of the default transition.
 - The fourth value describes the ratio at which the transition divides the respective side: The translator uses *0.5*, which means it originates in the centre of the side. A value of *0.75* would mean three quarters along the side in a clockwise direction (so on the right side of the state it would be in the lower half, while on the left side it would be in the upper half). To avoid collisions with other transitions, the default transition is placed at *0.75* on the left side of the initial state, slightly above the other transitions.
 - The fifth and sixth value specify the absolute coordinates in the chart where the transition starts. They give *x* and *y* values respectively, originating in the upper left corner of the chart.
 - The last two values are internal Stateflow values which are set to zero.
- the destination of the transition **dst** analogous to the **src**: It provides the ID of the target state and an **intersection**. Transitions created by the translator from a higher state in the diagonal arrangement to a lower state go from the centre of the right side of the source to the centre of the top rim of the target. Transitions from a lower state to a higher state go from the left of the source to the bottom of the target. Self-loops go from the right to the top of the state. As mentioned above, the default transition is on the left of the initial state.
- a **linkNode** consisting of a reference to the chart and to two other transitions of the chart, as described above.

- the x and y coordinates of the **midpoint** of the transition. They are chosen so that the default transition is a straight line, while other transitions have their midpoint approximately at the same x position as the target and around the same y position as the source. Therefore, they always bend to the right.

transition label The label for Stateflow transitions must be defined in the action language specified for that chart. For both action languages, it has the following form (cf. guide [18, p. 2-16ff.]):

$$\begin{aligned} \textit{Label} &::= [\textit{Events}] [[\textit{condition}]] [\{ \textit{conditionAction} \}] [/\{ \textit{transitionAction} \}] \\ \textit{Events} &::= \textit{event} \{ | \textit{event} \}^* \end{aligned}$$

The *event* must refer to a local event or an input event of the chart, although the translator does not use local events (see below). The disjunction of events *Events* defines all the events that can cause the transition to happen. If none is specified, any event can trigger the transition (including the “tick”, see Section 3.3.3). The *condition* must be a boolean expression. The transition can only be taken if the condition evaluates to *true*. If no condition is given, it is implied to be *true*. The *conditionAction* is executed as soon as the condition is confirmed. In contrast, the *transitionAction* is only executed when the transition is actually taken. If the transition consists of several segments that connect via **Junctions** and if only the condition of the penultimate segment is not fulfilled, then the transition cannot be taken. Therefore, no transition action of any segment is executed, while Stateflow does execute the condition actions of all segments but the last two. The transition action is enclosed within curly braces.

data If the transition label refers to a data port of a component or to a data subcomponent, there has to be a corresponding **data** element in the Stateflow definition. For ports, they correspond to the ports of the Simulink chart block, while data stores of the surrounding system can be used directly. In addition to an ID, **data** elements are defined with the following attributes:

- The **name** of the port or data store. It corresponds to the name used in the SLIM transition (except full stops in ports of subcomponents being replaced with underscores).
- The **linkNode** references the chart and other data elements as described above.
- The **scope** is either *INPUT_DATA*, *OUTPUT_DATA* or *DATA_STORE_MEMORY_DATA*, depending on whether the data element is a data store, a port which is read from or a port which is written to.
- The **reference** to the machine is always *1*, which is the only machine used in the translation.
- If the data type of this element is an integer range, the **range** is specified with its minimum and maximum.
- The **dataType** itself is given, according to Section 3.1.1. If this is the state output element, the **dataType** is set to the enumeration as defined in the chart.
- If this data element is the output of the current state of the chart, **outputState** is added with a reference to the chart whose state is reported.

event Transitions can be triggered by incoming events and they can emit events themselves. The event element in the definition contains as attributes

- a **name** which is taken from the SLIM event port involved. If it is a port of a subcomponent, the full stop in the name is replaced with an underscore.
- a **linkNode** referring to the chart and to other events as explained above.
- the **scope** of the event, which is either *INPUT_EVENT* or *OUTPUT_EVENT*.
- the **type** of the trigger, which is always *EITHER_EDGE_EVENT* for the translation. This is the only type of event a chart can produce apart from *FUNCTION_CALL_EVENT*. But function calls are harder to handle, as they cannot be passed through normal ports. Edge events are represented by a boolean signal that changes upon an event, and can therefore be used more easily. That is why the translator always uses *EITHER_EDGE_EVENTS*.

For each **continuous** and **clock** data subcomponent in SLIM which is set in a mode transition, there is also generated such an output event with the name *_reset-<subcomponent name>*. That event is emitted whenever the subcomponent is assigned, to reset the Integrator block in the respective updater Subsystem (see Section 3.2.2). If the Stateflow chart has input events, it is only executed when such an event occurs, so a “tick” event is sent in every simulation step (see Section 3.3.3). For that, an input event is created like described above.

3.3.2 Simulink and Stateflow semantics and SLIM semantics compared

For SLIM, the semantics are formally defined in [20] in the form of an *Event-data automaton (EDA)* for each component, and a *network of event-data automata (NEDA)* to describe their interaction. Unfortunately, there is no such formal definition for the behaviour of Stateflow. Informally, it is described in the guide [18].

Event-data automaton

Each SLIM component $c \in CCmp$ formally is a tuple $\mathfrak{A} = (M, m_0, X, v_0, \chi, \varphi, E, \longrightarrow)$ with

- the set of modes $M := Mod(c)$,
- a starting mode $m_0 \in M$ which is the **initial** or **activation** mode $stm(c)$ of the component,
- the set of variables X consisting of input variables $IX := IDPrt(c)$, output variables $OX := ODPrt(c)$ and local variables LX , which represent the data subcomponents of c ,
- the initial valuation $v_0 \in V_X$, which describes the default values of X in c ,
- the mode constraints $\chi : M \rightarrow (V_{LX} \rightarrow \mathbb{B})$ as defined in the mode invariants,
- the trajectory equations $\varphi : M \rightarrow (LX \rightarrow \mathbb{R})$ as defined in the mode invariants,
- the set of events E consisting of input events IE and output events OE where IE represents all incoming event ports $IEPrt(c)$ and all outgoing event ports of subcomponents of c and OE analogous the ports $OEPrt(c)$ and incoming event ports of subcomponents,

- the transitions $\longrightarrow \subseteq M \times E_\tau \times (V_X \rightarrow \mathbb{B}) \times (V_X \rightarrow V_X) \times M$ where $E_\tau := E \cup \{\tau\}$. For a transition $(m, e, g, f, m') \in \longrightarrow$, m is the source mode, m' the target mode, e the trigger, g the guard and f the effect. f does not change the values of IX , while OX and LX can be changed via assignments. Additionally, $d \in LX$ is reset to its default value when it is reactivated.

V_X is the set of valuations, meaning all partial functions assigning values to variables in X . HASDEL additionally has **@activation** transitions, which are executed when a component is reactivated. Since they are not part of the translation, they were not included in this formal definition.

The semantics of this EDA are given by a transition system $(Cnf, \kappa_0, L, \longrightarrow)$, where $Cnf := M \times V_X$ are configurations of the EDA, $\kappa_0 := (m_0, v_0)$ is the initial configuration, and $L := \mathbb{R}_{>0} \cup E_\tau$ are labels for the transitions. A transition in this transition system can either be a time transition indicating that $t \in \mathbb{R}_{>0}$ time has passed and all continuous variables are updated accordingly, or it is an internal or event transition as defined in the EDA, which happens instantaneously. The former can only occur if the constraints on the current mode stay valid. The latter can only occur if the current valuation satisfies the guard, and the valuation after applying the effect satisfies the constraints of the target mode.

This is mapped to a Stateflow chart, which could be formalised in a similar way as $\mathfrak{C} = (S, s_0, X', E', \longrightarrow')$ with

- the set of states S ,
- the initial state s_0 indicated by a default transition,
- the set of data elements X' consisting of input data IX' , output data OX' and references to data stores in the Simulink model LX' ,
- the set of events E' consisting of input events IE' and output events OE' ,
- the transition relation $\longrightarrow \subseteq S \times \mathbb{N} \times IE'_\tau \times (V_{X'} \rightarrow \mathbb{R}) \times (V_{X'} \rightarrow V_{X'}) \times 2^{OE'} \times S$. A transition (s, n, e, g, f, oe, s') consists of source and target state s and s' , an ordering number n , a triggering event e , a guard g , an effect f and a set of outgoing events oe .

A special transition is the default transition: While formally being a transition, all the parameters are empty except for the target state. Therefore, it is not seen as a transition here, instead its result (the definition of an initial state) is formalised. As the chart is configured to execute before the simulation (see Section 3.3.1), the system actually is in the initial state at the beginning of the simulation. Other features than the above mentioned such as state actions are not used in the translation and are hence omitted.

The translation could be seen as a mapping ξ from the EDA to a Simulink Subsystem with a Stateflow chart:

- Concerning the modes, it is a bijective function mapping SLIM modes to Stateflow states $\xi_M : M \rightarrow S$.
- The starting mode is mapped to the initial state: $\xi_M(m_0) = s_0$
- It is an injective mapping of data elements $\xi_X : X \rightarrow X'$, where $IX' = \{\xi_X(x) \mid x \in IX\}$, $LX' = \{\xi_X(x) \mid x \in LX\}$ and $OX' = \{\xi_X(x) \mid x \in OX\} \cup \{x_m\}$, with x_m being a new data element to output the current state of the chart. Its data type is an enumeration created from S . In the actual translation implementation, only the data elements which occur in mode transitions are included in the Stateflow for simplicity. But they still are part of the Subsystem in the form of ports and DataStoreMemories.

- ξ also defines an injective mapping of events $\xi_E : E \rightarrow E'$, where $IE' = \{\xi_E(e) \mid e \in IE\}$ and $OE' = \{\xi_E(e) \mid e \in OE\} \cup CE$, where CE contains one output event for each continuous variable in LX . Again, the Stateflow chart is only connected to those events it uses, but the **Subsystem** has all events.

- The transition relation needs to be adjusted: $\xi((m, e, g, f, m')) = (s, n, e', g', f', oe, s')$ where

- $s = \xi(m)$,
- $s' = \xi(m')$,
- for all $x \in IX$, $f'(v)(x) = v(x)$; for $x \in OX \cup LX$, it holds that

$$f'(v)(x) = \begin{cases} \llbracket a' \rrbracket(v) & \text{if } f \text{ contains assignment } x := a \\ s' & \text{if } x = x_m \\ v(x) & \text{else} \end{cases}$$

with a' being derived from the expression a as described in Section 3.1.2,

- $e' = \begin{cases} \xi(e) & \text{if } e \in IE \\ \tau & \text{else} \end{cases}$,
 - for all $d \in LX$ which have a continuous data type and for which f contains an assignment $d := a$, the corresponding $e_d \in CE$ is in oe ,
 - $\xi(e) \in oe$ if and only if $e \in OE$,
 - the guard is extended: $g' = g \wedge \chi(m')(v')$ with v' being the valuation of X' after applying f' .
 - n is a priority such that for any transitions $t_1, t_2 \in \longrightarrow'$ with the same source state $s_1 = s_2$, it holds for the corresponding values of n that $n_1 \neq n_2$.
- there is no equivalent for the initial valuation v_0 in Stateflow, but it is set in the Simulink model (see Sections 3.2.2 and 3.2.1),
 - there is no Stateflow equivalent, either, for the trajectory equations φ , which are implemented in Simulink as **Subsystems** with an **Integrator** (see Section 3.2.2).

The most important difference between the EDA and the Stateflow chart is the question when transitions are taken: In SLIM, time transition and internal and event transitions are interleaved, so that there can be an entire cascade of event transitions without passing time, or a long period of time without any events. This cannot be mimicked in Stateflow, because Simulink executes the chart in every simulation step. Stateflow cannot decide non-deterministically whether to take a transition, instead it always takes the first transition which is enabled (based on the transitions' priorities). Then it executes only that one transition in the chart, before it waits for another time step to pass. Therefore, the behaviour of SLIM and Simulink can differ greatly if the model relies on concentrated bursts of events with long cascades of transitions without much time passing in between (if any). In that case, the developer might consider turning on the Super Step Semantics of the Stateflow chart, which takes successive transitions as long as possible in each step (see the guide [18, p. 3-37f.]). Similarly, since Stateflow takes a transition in each time step if possible, SLIM models which expect long time to pass between non-deterministically taking a transition, are not represented appropriately, either. Stateflow always picks the first transition which is enabled in a state (according to an internal ordering which is formalised in the n of the transition, and graphically displayed in the chart). So models which rely on a non-deterministic choice between transitions with a certain distribution such as an equal rate for all transitions, are not mimicked properly.

Network of event-data automata

The behaviour of the entire SLIM model is formalised as a network of interacting EDA, which is a tuple $\mathfrak{N} = ((\mathfrak{A}_i)_{i=1}^n, \alpha, EC, DD)$ where

- each \mathfrak{A}_i is an EDA representing one SLIM component,
- α is the activation mapping which indicates for each global mode $m \in \prod_{i=1}^n M_i$ the set of all i that correspond to an active component in m ,
- EC is the set of all end-to-end event connections, which can represent a connection between two transition triggers across multiple event port connections,
- DD is the data dependency mapping representing data port connections and flows (only one step, not multi-hop like EC).

Its semantics are given by a transition system $(Cnf, \kappa_0, L, \Longrightarrow)$, which is derived from the transition systems of the constituting EDA, by

- combining their configurations: $Cnf := \prod_{i=1}^n Cnf_i$,
- inferring the global initial configuration $\kappa_0 := (\kappa_0^1, \dots, \kappa_0^n)$,
- using the same labelling as the first transition system: $L := L_1 = \mathbb{R}_{>0} \cup E_1 \cup \{\tau\}$,
- defining the global transition relation \Longrightarrow . Each transition can be
 - a time transition, which represents the passing of t time units by updating the continuous variables of all EDA,
 - an internal τ -transition within one EDA,
 - a multiway communication transition, broadcasting an outgoing event within the transition system from one EDA to all listening EDA, involving at least one non-passive event port connection,
 - a non-blocking event transition, broadcasting a non-blocking outgoing event within the transition system, only using passive connections,
 - an input transition, broadcasting an input event of the first EDA to at least one listening EDA, or
 - an output transition, sending an output event through an outgoing port of the first EDA to the environment.

One can see that the first transition system is considered the root system, and only the events of this system are visible from the outside. In each transition, all EDA involved take their own corresponding transition as defined above, updating their state and variables in the configuration accordingly. Afterwards, all EDA which get activated and have an **activation** mode are reset. Finally, the global valuation is adapted, taking the data dependencies into account: For each EDA i and each input or output variable $x \in IX_i \cup OX_i$, if there is a dependency active in the current mode, the value is set accordingly. If there is no data dependency active in the current mode, and there was an active one in the last mode, the value is reset to default. Otherwise, it remains unchanged.

The configurations of each \mathfrak{A}_i in the NEDA corresponds to the setting of the respective **Subsystem** and its **Stateflow** chart: The chart determines the state, which exactly represents the mode of the EDA, and the **DataStoreMemories** and ports define the valuation of the data elements. Combining this information from every **Subsystem** thus represents the global configurations Cnf in the NEDA. The initial configuration κ_0 defines all starting modes and the initial valuation of all data elements. The **DataStoreMemories** are configured to the same initial values, and ensure for the data ports' **Subsystems** that they have

the correct initial value. Since the corresponding states are also the initial states in the chart, Simulink emulates the NEDA. The only difference is that the initial valuation in the NEDA is only a partial function and can leave some variables unassigned, while Simulink does not support undefined values and θ is used then. To the environment, Simulink provides the same event interface as the SLIM root component, so the labelling of the NEDA also describes what is visible to the outside of the Simulink model. Although time transitions do not exist in that explicit form in Simulink / Stateflow, the root system returns its internal clock to the environment, so the situation compares to the labelling L .

A time transition of the NEDA means all active components update their continuous variables according to the amount of time that passed, while inactive EDA remain untouched. This corresponds to one step of the Simulink simulation. In that step, for all active **Subsystems**, the updaters for all continuous components which are active in that state are executed. They increase the value in the corresponding data store according to the amount of time passed and the specified rate. Thereby, they mimic the EDA. But SLIM allows for arbitrarily small or large amounts of time to pass, while the Simulink model takes a step of a pre-set size. Although variable step size is activated in the model, it still tries to make a leap as large as possible, only decreasing the step size if it seems necessary (which is decided internally from the data values, see the guide [17, p. 3-21]). That might lead to differences in the timing of events in SLIM and Simulink, which can influence the general result of the simulation. But the Simulink execution represents one valid run of the SLIM model.

In each simulation step, the charts are reevaluated. All charts that can take a transition do so, so all these transitions happen at the same time in the simulation. If a chart has multiple transitions enabled, the first one according to the internal ordering is taken, which differs from the non-deterministic choice in SLIM. The taken transition might represent an internal transition in the NEDA. This means that one EDA takes a transition, and afterwards the others are updated by possibly activating or deactivating them and updating all data components. The Stateflow transition connects the same states like the SLIM transition, and has similar effects and guards (see above). If the state changes, this results in a change of the “*modes*” output of the chart, which leads to activating the subcomponents accordingly. If a reactivated subcomponent has an **activation** mode, the respective **Subsystem** is reset upon reactivation. Therefore, the activation process of SLIM is emulated accurately. The valuation update in SLIM distinguishes three possibilities for each input or output variable x :

1. In the target mode, a data dependency is active. This decides the value of x . In Simulink, the according connection between blocks is active as well, so x is updated accordingly.
2. The transition deactivated a data dependency and does not activate any other for x . Then the value of x is reset to its default. Simulink also deactivates the corresponding connection, and provides the port with its default value.
3. There is and was no dependency active for x . Then the value of the port stays the same. If there is no active change to the `DataStoreMemory` associated with the data port, it also stays constant.

Thus, Simulink mimics the transition accurately.

The taken transition might also be a communication transition. Because the translator assumes all ports to be non-blocking, this corresponds to multiway communication transitions of SLIM as well as non-blocking event transitions. In those cases, one EDA takes a transition which triggers an event, and all listening EDA take a corresponding transition. In Simulink, the transition of the sender causes an event to be sent. This is connected to other Stateflow charts via the event port connections specified in the SLIM

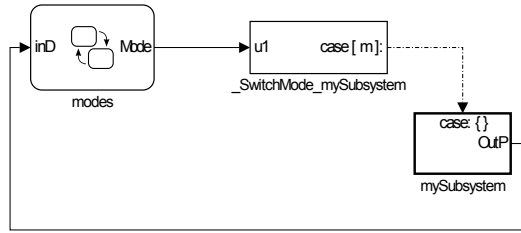


Figure 3.3: Circular dependency between atomic subsystem and Stateflow chart.

model. Since the event connections *EC* are deduced from the same port connections, the same components listen like in the NEDA. They process the incoming event and take the first matching transition. That differs from the non-deterministic choice in SLIM. But the transition in Stateflow represents a SLIM transition, so the behaviour is a valid one. As with non-blocking events in SLIM, it is possible that no chart can currently receive that event, and only the sender is updated. Afterwards, the activation and valuation updates apply as described above.

Input transitions from the environment are treated similarly. In the NEDA, all components connected to that port take a transition, while in Simulink, all charts which are connected to the **Inport** are updated. If an EDA sends an event out to the world, only that EDA takes a transition and activation and valuation are updated. In the semantics [20], this is a separate transition from the communication transitions. So the same outgoing event of one EDA can either be received by other components, or by the environment, but not both at once. Simulink makes no such distinction. Since this is a non-deterministic choice, Simulink cannot mimic it and sends the event to both internal and external receivers.

Execution order and event buffering

As already mentioned in the introduction to this chapter, Simulink orders the blocks and executes them successively. It honours dependencies, so the dependent system is executed after the source of the signal to ensure data consistency. If Simulink suspects circular dependencies, it aborts the build process with an error, as it cannot determine a correct execution order for the blocks. Therefore, the translator inserts buffering data stores whenever it suspects a possible dependency cycle. Because to Simulink, there is no direct dependency between the reading and the writing system of a data store. Although circular data and event dependencies are not allowed in SLIM, either, Simulink is more strict in its definition of a dependency, so correct SLIM models can result in a dependency loop. A common example are two subcomponents, which interact bidirectionally via events. Assume there is no causal connection between the events in the respective subcomponents, so SLIM recognises no dependency cycle. If the subcomponents are both active only in certain modes, they are wrapped in a `SwitchCaseSubsystem` to activate and deactivate them. Such systems are considered by Simulink as one atomic unit, rather than a collection of blocks. Therefore, it cannot execute only the blocks of a subsystem which the other one relies on, but it has to execute the entire subsystem in one step. This leads to a circular dependency, as depicted in Figure 3.3. Stateflow charts are treated as atomic subsystems as well. So if the chart interacts with a subcomponent bidirectionally, and the subcomponent is only active in certain modes, there is a circular dependency as well. To break that, all outputs of Stateflow charts are buffered. This includes the *mode* output, as the subsystem uses it in its `SwitchCase` block and therefore depends on the chart. All outgoing events which represent SLIM events are buffered as well. But due to a dummy

simulation step used in event broadcasts (see below), this buffering does not impede the event connection, which is still instantaneous. Currently, outgoing data is buffered as well. This is necessary in case the data is supplied to an atomic `Subsystem` and produces a dependency. For out data ports of the component itself, it would not be needed, since they already have a data store for their default value. But currently, such a nuanced distinction remains as future work.

To implement the buffering, the translator introduces data stores into certain port connections and flows (see Sections 3.2.3, 3.2.4). That way, Simulink can read the old value from one `Subsystem` without executing it, calculate the output of the other one depending on that, and then evaluating the first `Subsystem` and update the data store. This introduces a delay of one simulation step between one incident and its effect. Since events are to be executed instantaneously, this would heavily impact the semantics of the model if time could pass between the triggering of an event and its reception. As a result, the simulation is stopped whenever an event occurs, and only resumed when all receivers were executed. Since there is no direct way to call for an immediate recalculation of the model, the simulation has to be paused for an entire time step. This is done with a three-state lock `GLOBAL_EVENT_PHASE` which is defined in the `rootSystem` next to `slimModel`. As long as its value is zero, the simulation can proceed. Whenever a SLIM event occurs, the phase is set to one and no further transitions are allowed except for those receiving that exact event. At the end of the step's calculation, the phase is set to two. In the next simulation step, again only transitions which are receiving that event are allowed. At the end of that step, the phase is reset to zero and the simulation can continue. Since every chart processes each event only once, the charts which already received the event while the lock was one will not receive it again while it is two. In particular, the lock is integrated into the guards of the transitions: All transitions except those triggered by an incoming event can only execute during phase zero. If a transition sends out an event, it sets the phase to one in its effect. A dedicated updater chart in the `rootSystem` raises the value to two after the `slimModel` is executed. Transitions with an incoming event as trigger can always happen: If the event was triggered earlier in the same step, the lock has the value one. If the event was triggered after the receiver was already calculated, it is received in the next step when the phase is two. If the event was triggered externally and provided through an incoming port of the root component, the lock was not set and therefore, it is still zero.

The updaters for clocks and continuous data subcomponents are wrapped inside an enabled subsystem. This type of subsystem only executes when the value on its `EnablePort` is greater than zero. Here, this is the case if the phase is either one or zero. Therefore, clock updaters are wrapped in two subsystems (one for the mode activation and one for the phase). Trajectory equations are active in exactly one mode, so this check can be merged with the phase test and only one subsystem is needed. Activating the updater when the phase is one allows timed values to change after the transition was triggered, which stands in conflict with the SLIM definition. But if they were only allowed to update when the lock is zero, then clocks which stand before the sending component in the execution order were already updated, while those behind it would not be. This leads to inconsistencies between clocks, which would be even worse. Therefore, all continuous data is updated even after the event fired. Since continuous data can influence ports only in mode transitions, it cannot propagate between the event trigger and its arrival, so this difference does not influence the models behaviour greatly: To the sender, it seems like the update was done after all receivers were executed, and to the receiver, the update might as well have happened before the event. During the dummy step when the phase is two, the updaters are paused, so it appears to the system as if no such step happened.

Thus, the translator follows the following procedure in each time step:

1. Try to execute the updaters for continuous data components, as they have the highest

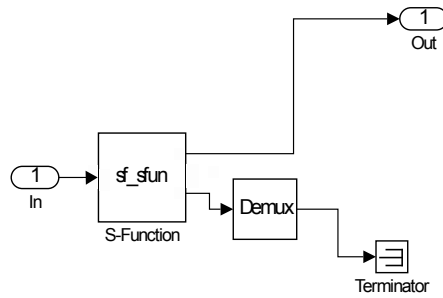


Figure 3.4: How a Stateflow chart looks beyond the mask: a S-Function with inputs for data inputs, outputs for data outputs and no function calls used, so that signal is terminated

priority. This corresponds to a time transition in SLIM. Yet, since Simulink rates the dependency relation higher than the priority, the updaters are not always executed first.

2. Execute the components one after the other according to their dependencies. The Stateflow chart of each component is executed, based on the configuration of the model at that point. This includes data values from the last step for components that were not executed in this step yet, and updated data for the rest. As long as no event was emitted, each chart makes a transition if one is enabled.
3. As soon as a chart triggers an event, it sets the phase to one and no further internal transitions are allowed. Continuous components can continue their updates, and charts which were not executed yet can receive the event.
4. At the end of the step, the phase is set to two.
5. At the beginning of the next step, the continuous updaters are not executed, because they are disabled by the lock.
6. The charts are executed, but can only take transitions which receive the event, and only if they have not done so in the previous step already. Again, they use the current state of the model (possibly influenced by the receiving transitions already taken).
7. At the end of the step, the lock is reset to zero, the event broadcast transition of the system is completed and execution can continue.

It becomes obvious that the first component in the sorted order which can send an event always determines the simulation. So if the model contains a component which fires many events, and which is placed high in the execution order, this component can block the entire model from taking any other transitions. Again, this unfair behaviour is one of the possible executions of SLIM. Therefore, the translation is especially well suited for models which do not have many event broadcasts, or where the broadcasts are intended to originate in only a few components, when the missing fairness does not have great influence.

3.3.3 Simulink Blocks Involved

The Stateflow Chart block is internally treated by Simulink as a Subsystem. Thus, all the data elements it uses, such as values from an Inport, must be provided to the chart via ports (except data stores, see below), like with any other Subsystem. If the chart uses events to trigger transitions, they have to be connected via ports on the Chart block as well. To the user, this chart appears as one block in the Simulink model, and the

implementation details are hidden. But the translation produces the model file, where the actual block composition must be defined (see Figure 3.4): Inside the **Subsystem** which represents the chart, there is an **S-Function** block (see Section 2.2). In this case, the Stateflow machine is defined within the Simulink model file after the Simulink model in the syntax described in Section 3.3.1. The executable file for the **S-Function** block is derived from that automatically.

For every port that is read in a mode transition in SLIM, an **Inport** is added to the chart **Subsystem** and connected to the **S-Function**. Similarly, **Outports** are created for all ports to which a transition assigns a value. Another **Outport** is added for the state output (see Section 3.3.1). As described in Section 3.2.2, a **Subsystem** has access to the data stores of its ancestors, so no ports need to be added for the SLIM data subcomponents mentioned in mode transitions. All triggering events of a **Chart** have to be bundled into a vector based signal. A **TriggerPort** for that vector is created in the **Subsystem**, which is connected to the **S-Function**. The Stateflow chart can trigger events itself. They are either of type *Function Call* or of type *Either Edge*. As the former is very restrictive, the second option is used here, which creates a boolean signal that has an edge whenever an event occurs. For these signals, an **Outport** is created within the **Subsystem** of the chart, and connected to the **S-Function** block. The **S-Function** block also has one port to output a vector of all the function call events generated by the Stateflow. Since the translator always uses edge events, that port is connected to a **Terminator**, which is a block used to tie up loose signals that would otherwise generate a warning about being unconnected.

The chart has three possible update methods: The *Discrete* method samples the chart at a fixed rate specified by the user, the *Continuous* one updates the chart in every Simulink simulation step, and the *Inherited* method derives the update interval from the inputs: If input events are specified, the chart is only updated when an input event occurs. If no input events are specified, the chart is updated at a rate so that it fits the sample rate of all data inputs. For our purposes, the chart needs to be updated in every simulation step. Unfortunately, the *Continuous* type does not allow any input events (see the guide [18, p. 18-25]). Since SLIM allows mode transitions to be triggered by events, this behaviour has to be mimicked by the Stateflow chart. Without input events, the incoming events would have to be provided as data signals, and the chart would have to detect their edges. That is not easy and would become confusing if a large number of input events needs to be processed. Therefore, the update method is set to *Inherited*. However, the chart must still be executed as often as possible, to allow for transitions to happen that are not triggered externally. Hence, another Stateflow chart is added to the system named “*modes.trigger*” (again implemented with a **Subsystem** containing an **S-Function** block). It contains only one (initial) state and one transition looping on that state, which has no trigger, no condition and its only effect is to emit an event (see Appendix A). Apart from this output event, it has no outputs and also no inputs, which is why it is updated in each simulation step. Thus, it generates an event in each simulation step. This tick is fed into the “*modes*” chart, which does not process it, but the chart is triggered by it and thus evaluated in each simulation step. That is a considerable overhead, as a second chart is added to the model for every chart that has input events, so future work might be to improve that mechanism.

Dynamic reconfiguration

The Stateflow chart has an **Outport** whose type is an enumeration of all the states in the chart and which reports the currently active state (see above). This information is used to implement the activation and deactivation of **Subsystems** and connections as specified by the SLIM `in modes` parameter. This **Subsystem** either represents a non-data subcomponent, the updater for clocks, the reset for data subcomponents (see Section 3.2.2), it contains a simple line for connections (see Section 3.2.3) or all the Simulink blocks

which constitute a flow expression (see Section 3.2.4). The activation and deactivation of these Subsystems is done via a SwitchCase block (see [15, p. 1-1962ff.]). These blocks have one data input and possibly multiple outputs. To each output of the block there is a case condition attached, which has the form of a MATLAB vector listing all values that are covered by this case. Optionally, a *default* case can be listed below the user specified cases. Simulink evaluates the case conditions successively, until it finds the first one covering the input value. Then an action signal is sent through the corresponding port of the SwitchCase. That has to be connected to a special port of the Subsystem: Within the Subsystem, this port is defined by an ActionPort block, whose *ActionType* is set to *case*. That way, the Subsystem is active whenever the SwitchCase selected the corresponding case, and inactive otherwise.

For our purposes, the single input of the SwitchCase is connected to the state output of the Stateflow chart. There is only one case condition defined that lists all states from the SLIM *in modes* list (or the complement for data subcomponent resets). The default case is disabled, so there is only one output of the SwitchCase. This can generate a warning in Simulink that the SwitchCase does not cover all possible cases. That warning can be ignored here. Alternatively, the default case could be enabled and connected to a Terminator, but that would add another block to the model, which can become confusing if there are many Subsystems depending on the current state. If the Subsystem represents a non-data subcomponent which has an activation mode, the ActionPort has its *Initialize States* option set to *reset*, like the SLIM component is reset upon reactivation. Otherwise, the option is not specified and therefore the default *held* applies, which keeps the values from the last active phase like the mode history in SLIM.

3.4 Error Model

SLIM models can contain error models to model the behaviour of the system if a fault occurs. They are defined as follows (from the documentation [21]):

```

ErrorType ::= error model identifier
                [features ErrorFeatures]
                end identifier;
ErrorFeatures ::= ErrorFeature+;
ErrorFeature ::= ErrorTypeState | ErrorPropagation
ErrorTypeState ::= identifier: StateType state;
                StateType ::= initial | activation | error
ErrorPropagation ::= InPropagation | OutPropagation
                InPropagation ::= identifier: in error propagation;
                OutPropagation ::= identifier: out error propagation;
ErrorImplementation ::= error model implementation Name
                [events ErrorEvents]
                [clocks ErrorClocks]
                [states ErrorStates]
                [transitions ErrorTransitions]
                end Name;
ErrorEvents ::= ErrorEvent+
                ErrorEvent ::= identifier: error event [Occurrence];
                Occurrence ::= occurrence Distribution
                Distribution ::= poisson number [per TimeUnit]

```

```

ErrorClocks ::= ErrorClock+
ErrorClock ::= identifier: data clock [TimeUnit];
ErrorStates ::= ErrorState+
ErrorState ::= identifier: StateTypestate [urgent in Delay] [while Invariant];
ErrorTransitions ::= ErrorTransition+
ErrorTransition ::= SourceState -[ErrorTrigger [when ModeGuard]
                               [within Delay to Delay] [then ModeEffect]]-> identifier;
SourceState ::= identifier | *
ErrorTrigger ::= identifier | reset |@activation
Delay ::= number [TimeUnit]

```

Before the transformation described in this chapter, the COMPASS tool-set integrates the error behaviour into the nominal model as described in the semantics [20], based on the model extension [4]. Since the error model and the nominal model can be specified independently, the user needs to specify *fault injections* to connect each error model component to a nominal component. They are assignments, which override the values of data elements of the component as they are specified by the nominal model with new values, depending on the error state the error component is in. One example would be to override a data port with a constant 0 to model a stuck-at-zero fault caused by a loose wire. The error component is integrated into the corresponding nominal component by adding a subcomponent. That subcomponent has the error states of the error component as modes, and the error events and error propagations as event ports. Additionally, the nominal component is informed through a data port of the error state in which the error component currently is. The nominal component is extended with additional outgoing event ports, which are connected to other components and their error components to implement error propagation between error components. Each mode transition in the nominal component has its effect altered to allow for fault injections to override them. Additionally, transitions with **reset** as a trigger cause the respective event in the error component to happen.

Due to that process, which is called (*fault*) *model extension*, this translation sees the error model as a part of the nominal model. Error components are treated as any other control subcomponent, being translated into a *Subsystem* with a *Stateflow* chart to model the error states. Similar to the nominal model, activation error transitions are not supported by the translator. The chart either continues where it was before deactivation (in case of an **initial** state), or reset (in case of an **activation** state). Non-deterministic and probabilistic behaviour is not supported, either. Since the nature of errors usually shows such properties, this can render the Simulink version of the error behaviour less accurate or helpful than the SLIM counterpart.

Chapter 4

Case Study: a Lego Satellite

To demonstrate the translator introduced in this thesis, a case study was performed by constructing a model of a space telescope with a Lego Mindstorms EV3 robot. Its behaviour is modelled in SLIM using many features commonly used in SLIM models, such as nested subcomponents, mode transitions, events and continuous data. The SLIM model interacts with the satellite's motors and sensors via data ports of the root component. With the translator proposed in this thesis, it is then transformed into a Simulink model. That model is manually extended with special Simulink blocks to control the Lego motors and read the sensors as well as the motor positions, and then the model is deployed onto the EV3's control unit. Since the root component is wrapped in a `Subsystem` in the Simulink translation, these blocks can be added and connected to the generated model easily. In the process of deploying the model, Simulink automatically generates C code from it, which is used on the Lego platform.

Because of model extension, error models are integrated into the nominal model before the translation into Simulink. Additionally, error models lose a lot of their capabilities without non-determinism and probabilistic events. Therefore, it was omitted here to build an error model.

The robot offers three main features:

Orbit correction The Lego robot is placed on a pole, and slides downward on it to mimic a gravitational pull that the satellite experiences due to friction. If the robot sinks too far, it starts to push back up. This emulates the satellite firing its engines to return to its intended orbit.

Observation positions The robot represents a space telescope. It has two pre-set orientation to mimic the observation of different astronomical objects. It switches between those positions periodically.

Energy control An internal battery is modelled. If it is low, the robot uses a light sensor to automatically find the best orientation for its solar panels. Afterwards, the battery recharges with its speed depending on the measured light intensity. When the battery is fully loaded, the robot continues its observation.

4.1 The Lego Robot

Lego Mindstorms

The satellite was modelled using a Lego Mindstorms EV3 robot. *Lego Mindstorms* [10] is a platform to build robotic systems, developed by Lego [9], and *EV3* is its third generation. It is not one pre-built piece, but an assembly of various bricks and connectors with which the user can create a system of his own design. Everything is connected with plugs, so it can easily be disintegrated and reassembled. The user can employ several motors and

sensors to operate the robot. The main unit is a programmable brick which controls these motors and sensors and also supplies power to them. Lego provides a programming platform to develop software for the brick in an easy visual way by using function blocks and connecting them. But the EV3 software is open-source [11], and hence there is an increasing number of other developing platforms which can be used as well to program the brick. This includes a Simulink library that provides blocks for the sensors and motors, which is used here.

The EV3 brick can control up to four motors (labelled *A* to *D*) and use up to four sensors. There are several types of sensors available from Lego, including touch sensors, infrared sensors which can receive signals from a remote control, and gyroscopic sensors. In this thesis, the robot uses an ultrasonic sensor to measure distances and a light sensor. The latter can operate as a colour sensor or measure the intensity of either ambient light or reflected light originating from the sensor itself.

The space telescope

For this case study, a model of a space telescope was designed (see Figure 4.1). It is attached to a vertical pole of about 60 cm height with tothing on it, on which the robot can slide up and down. The robot itself has an outer frame which is approximately 20 cm × 15 cm × 15 cm large. In an upper corner, a light sensor is attached pointing upwards (in the Figure 4.1). It measures the intensity of the light to which the modelled solar panels are exposed. At the bottom side of the frame, the ultrasonic sensor is attached to measure the distance to the ground. As the frame pitches, so does the sensor. Therefore, it usually does not look straight down, but at an angle. But the robot cannot pitch far due to the central pole (only 10 to 20 degrees in each direction), so the measurement is not affected too gravely.

Three motors control its position: Motor *A* (see Figure 4.1) drives along the pole using a gear, raising and lowering the satellite to model orbit correction. It is attached to the upper part of a turntable, which is therefore the stationary part. The frame of the robot is attached to the lower part of the turntable, and can therefore rotate around the pole. To do so, motor *C* is used: It is fixed to motor *A* and thus stationary itself. It turns a gear, which uses the outer tothing of the turntable to rotate the latter. The turntable is placed around the pole, so the yaw axis is along the pole. Motor *C* is a small servo motor in contrast to the larger motors *A* and *B*. The axis of this motor's momentum is along its length (in contrast to the perpendicular axis of the other motors), and therefore it can be placed alongside the pole. Motor *B* is attached to the lower part of the turntable and is thereby rotated around the pole. It is used to pitch the robot. The motor uses two gears with a low gear ratio to turn the frame. Thereby, the motor has to make more turns to tilt the satellite compared to being attached directly to the frame. This increases the relative precision when measuring the motor's position, because differences of a few degrees do not matter as much.

The brick itself is not placed on the satellite. It is very heavy (mostly due to the batteries), and would therefore unbalance the robot. As the motors are not very strong, in particular the smaller motor *C*, the robot would have trouble positioning itself.

4.2 The SLIM and Simulink models

Due to their large size, the models are not included in the body of the thesis, but in its Appendices C.1 and C.2. The SLIM model of the satellite consists of 5 different components: The root component is the `system Sat` representing the satellite's main frame. It has the `Thruster` for orbit control as a subcomponent, together with a virtual battery, a `Turn` to turn into a certain position and `findLight` to find the best position for its solar panels. `findLight` has its own `Turn` subsystem to position the satellite, showing a case

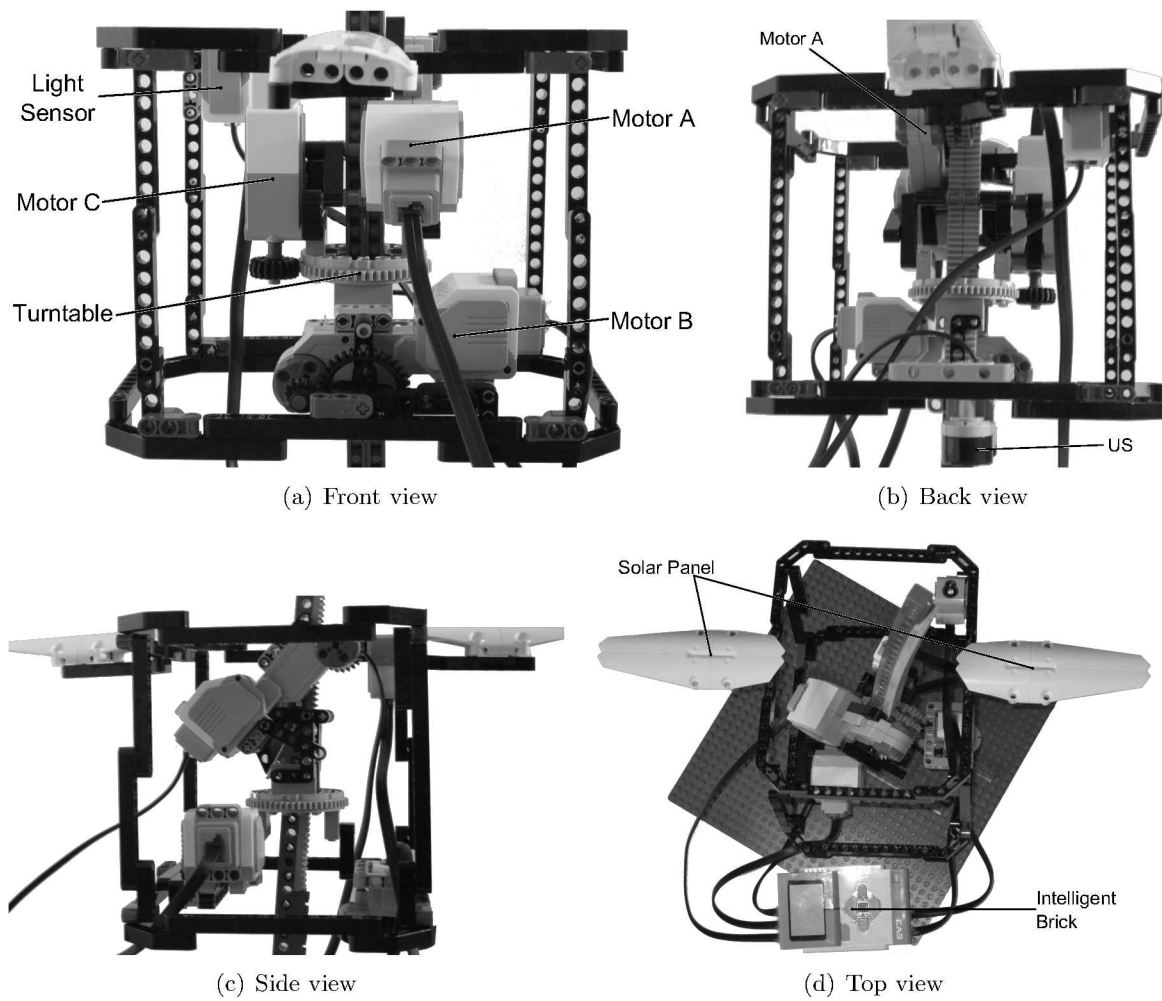


Figure 4.1: The Lego robot designed for this case study. *Motor A* uses the tothing of the pole to move the robot up and down. *Motor B* pitches the satellite model. *Motor C* uses the *Turntable* to yaw the robot. A *Light Sensor* on the top measures the light intensity on the *Solar Panel* models. An ultrasonic sensor (*US*) on the bottom measures the distance to the ground. The controlling *Intelligent Brick* is not fastened to the robot due to its weight.

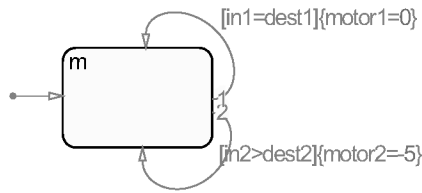


Figure 4.2: Controlling two motors, stopping one and turning the other using non-determinism

of nested subcomponents. In the main component, a clock clk runs during observation periods, and causes the system to switch between the two configurations every 15 seconds. It demonstrates how the *updater Subsystem* for clocks in Simulink is used.

The **system** *Thruster* controls the orbit correction mechanism. It evaluates the current distance to the ground, provided via the incoming port US . This port is connected to the ultrasonic sensor of the robot. Depending its value, the controller either causes motor A to lift the robot up (setting the motor's speed to -40), or to let it sink down slowly (with a speed of 2). This part of the robot is independent of the remaining features.

The *Battery* is a system which mainly consists of one continuous data subcomponent *level* to model the current charge of the battery. In the mode *discharge*, the satellite is in use and consumes energy. If the level drops too low, the battery sends a signal *low* to the main component to initiate charging. In this simplification, it immediately starts charging. The speed of charging depends on the light intensity on the solar panels, provided by the light sensor through a data port. As soon as the battery is fully charged, the event *full* tells the robot to proceed with its task. Charging and discharging the battery is done via trajectory equations in the mode invariant. One can see how for each trajectory equation, a separate updater is created to drive the data component in the respective mode.

The **system** *Turn* points the satellite into a specified position, which is a helper to several features. It takes the current position of the motors B and C and the desired value for each of them. Depending on how far they are apart, the controller turns the satellite into the right direction with different speeds. The destination only has to be reached within a certain margin, because the Lego motors are not very precise and might therefore oscillate around the intended position for some time before reaching it exactly. To shorten the oscillation, the motors turn with different speeds in the different directions when they are near the target. The alignment starts by pitching the satellite into position. When the motor's rotational position matches the intended one, the system switches to turning yawing if needed. When this motor has reached its destination as well (or if it already was in that position), *Turn* switches to a check mode. Only if a second measurement still returns the intended values, the system triggers the event *turn_ready* to indicate to the calling system that it can proceed. This filters out the situation that the motor overshoots, and returns being ready when in fact, it is not in the correct position anymore. In that case, the system returns to turning the motor which is out of place. A normal SLIM system could turn the two motors at the same time (see Figure 4.2). But because Simulink cannot pick its next transition non-deterministically, it is not possible to switch off one motor when it is finished, and let another find its mark in the same mode (one of these actions would be preferred by Stateflow, preventing the other). Therefore, turning the two motors is separated. This is an example of how the restrictions of Simulink have to be taken into account in the design of the SLIM model. If the satellite receives the event *low* from the battery, it interrupts the current observation and turns on the *findLight* subcomponent. This subcomponent rotates the satellite into an initial position with a *Turn* subcomponent. When this is done (event *turn_ready*), *findLight* starts to pitch as

long as the light intensity increases. As soon as it starts to decrease, the motor is stopped, assuming this is the best position. Afterwards, the same procedure is continued with yawing. When both motors are in their respective locally maximal position, the robot remains there until fully charged.

On the one hand, *findLight* sends a signal to the main component, creating a dependency between the Stateflow chart of the main component and the `Subsystem`. On the other hand, *findLight* is active only in certain modes, and therefore itself dependent on the main components Stateflow chart. This is an example of the dependency cycle which can occur in Simulink, because certain blocks like the dependent subsystem are treated as one unit and not broken apart (see Section 3.3.2). For these cases, the buffering systems and dummy simulation steps are necessary. Their overhead is immense:

4.3 Remaining manual tasks

Most of the transformation from SLIM to executable code for the EV3 brick is done automatically. The major task which the user still has to do manually is placing the EV3 control blocks around the system. The blocks for the ultrasonic distance meter and the light sensor are added and their `Port` parameter set to the number of the physical port they connect to. To control the robot, the outgoing ports of the model are connected to motor blocks representing the three motors used in the robot (distinguished via the `Port` setting). Lego allows the user to read the total amount of degrees which a motor has turned since starting the program. The SLIM model assumes the robot to be in a neutral position at the beginning of the execution, and uses the motor position to deduce the current orientation of the robot. Therefore, the `Encoder` blocks are added and set to the corresponding motor port. In the future, this might be replaced with a gyroscopic sensor and a compass sensor to become independent of the motor position. That would allow for greater portability, because changes such as replacing a gear would not effect the measurement any longer.

Some of the Simulink blocks representing the Lego sensors produce values of a type that is not supported by SLIM. Thus, `DataTypeConversion` blocks are added to those signals. Their incoming and outgoing type are not specified explicitly. Instead, Simulink deduces them automatically from the connected signals.

Usage After the model is completed in this way, it can be deployed onto the EV3 brick. To do so, it has to be connected via network to the computer which is running Simulink. In Simulink, the user has to enter the IP address of the brick within the network. Afterwards, the model can be deployed onto the robot. This might take some time, especially if Stateflow charts are involved, which take time to be converted into executable code.

Chapter 5

Conclusion

This thesis defines a translation from SLIM models to Simulink models. SLIM is a dialect of AADL, which is a language for model-based engineering, and it is used by the COMPASS tool-set. It describes the modelled system in the form of components, each of them modelling a software or hardware entity in the system, such as a thread. Components can contain other components, creating a hierarchical tree of subcomponents. The components can interact through ports and their connections. This structure can be dynamically re-configured, enabling different setups in different modes. To switch modes, the system uses transitions, which can be synchronised using events.

Simulink is an extension to MATLAB, using a block-oriented view to model a system. The model consists of function blocks to perform operations, ranging from simple arithmetics to complex integration. To organise the model, there are subsystem blocks, which can hide away large parts of the model in one block and execute conditionally. Simulink can simulate the model and its behaviour in small discrete time steps. In each step, all blocks are updated successively, ordered according to their data dependencies. To introduce discrete behaviour into this quasi continuous system, there is the Stateflow block library which defines a finite state machine. Separate charts can synchronise using events, but the chain of event connections must not be circular.

This thesis provided a translation, mapping the aspects of SLIM to Simulink blocks. It uses the COMPASS toolset to instantiate the SLIM model. Many of their features could be mapped easily, such as making `Inport` and `Outport` blocks for SLIM ports, and `Subsystems` for the components. With the exception of the case-operator, all SLIM expressions can be transformed into equivalent Simulink representations, using Simulink blocks as well as MATLAB expressions. However, many features needed more elaborate techniques: To update continuous variables over time, an explicit updater is needed, which uses an integrator to implement the respective linear equation. This creates a considerable overhead if many trajectory equations are defined. If a port connection is active only in particular modes, an intermediate system is needed to model this. Together with the controlling `SwitchCase` block, this increases the number of blocks in the model considerably, rendering even small models like the case study from the last chapter unclear. The modes and their transitions have to be expressed via Stateflow charts, using a separate mechanism. All events are considered non-blocking, being able to trigger even if no component can receive them. This is a large change regarding the behaviour of the model. But since there is no direct possibility to check whether a component is listening, this would require a considerable effort to implement.

Most of the overhead introduced by the translator is necessary due to the difference between the execution process defined in SLIM and the Simulink procedure. First, Stateflow charts are only recalculated if an event comes in. To allow for internal transitions as well, a separate chart is added to trigger the Stateflow chart at every time step. This increases the size of the model and might impede the simulation time, but should mimic

the behaviour of SLIM accurately. Second, Simulink executes the blocks in a time step successively, with one calculation using the result of the previous one. This differs from the SLIM semantics, where all components which synchronise through an event update at once, only depending on the value of the last evaluation. Therefore, synchronised transitions which use the same data element concurrently might produce different results in Simulink than in SLIM. To avoid that, data would have to be buffered during the execution. Such intermediate data stores are already used to model the default values for ports. If the reading system is executed before the writing one, this may lead to a delay in the connection of one time step. This is avoided by giving the write operation a higher priority than the reading. However, this might not always resolve in the desired execution order: Simulink determines the execution order of the blocks depending on their data and event dependencies. Such a relation can override the assigned priorities if they are conflicting.

Since Simulink is more strict in its definition of dependencies than SLIM, it might detect dependency loops in a correct SLIM model. While `Subsystems` are generally regarded to be virtual, so the execution of blocks in the subsystem can be interleaved with blocks from the parent system, they can create circular dependencies: Especially subsystems that are activated conditionally are treated as an atomic unit, and cannot be broken apart and interleaved with other systems. This can lead to Simulink detecting a loop, even if there is no actual data dependency. To break these loops, the translator inserts data stores into signals which might be involved in such a cycle. Because this can introduce a delay into the connection, priorities are used again. Delays must not happen in event connections, which are defined to be synchronous in SLIM. Thus, the simulation is paused whenever an event occurs, until all components had the possibility to process it. To achieve that, a global lock is defined, which is checked in the transition guards, disabling every transition except receiving ones while an event broadcast is in progress. The lock also stops continuous data subcomponents to change during the break.

Finally, one large difference is that Simulink updates all blocks in each time step, resulting in a transition in every Stateflow chart if possible. In contrast, the semantics of SLIM define a non-deterministic choice between taking an arbitrarily large time step and one or more mode transitions. So while the more regular succession of one time step of a certain size followed by a group of transitions is one valid run in the SLIM definition, it does not reflect the possibilities of SLIM. This result relates to other resolutions of non-determinism as well: Simulink orders the transitions internally, always taking the first one enabled. This is one of the many valid executions defined by a non-deterministic SLIM model.

Non-determinism cannot be exactly reproduced at all by a simulation like Simulink. This is the largest deficiency in the translation, and is an inherent problem which cannot be overcome. At best, features like the arbitrary choice between enabled transitions can be mimicked probabilistically. Probabilistic behaviour of the model is not included in the translation, either. Despite these limitations, the case study in the last chapter shows that the translation is viable in typical applications. Although it introduces a large overhead, which will become probably even larger if executable code is generated from the Simulink model, the satellite performed its task successfully. Because issues like non-determinism were already avoided in the generation of the SLIM model, the Simulink model could be generated automatically, and only the environment had to be added manually by connecting the specific Lego Simulink blocks.

5.1 Future work

There are some aspects which could improve the translation in the future. First, introducing probabilistic behaviour would increase the application domain of the translation. Simulink offers blocks to generate normally or uniformly distributed random values. These

blocks could for instance be used by the Stateflow chart to enable transitions depending on those values. This could also be used to resolve non-deterministic choices. While in a non-deterministic system, all paths are equally valid, a probabilistic choice with a uniform distribution might be more similar to the user's expectations.

Second, the translation increases the size of the model considerably. At various places, additional blocks were added to mirror the behaviour of SLIM. This overhead can be reduced. For instance, the updaters for clocks are wrapped in two conditional subsystems, one for the activation depending on the system's state, and one to pause it during event broadcasts. While these conditions cannot be merged easily because there cannot be two control ports on the same subsystem and If blocks put heavy restrictions on their condition expression, it is desirable to avoid the nested subsystems. Another improvement would be to buffer signals only if necessary. Currently, all signals from the Stateflow chart and all *out-to-in* port connections are buffered in a data store. Advanced heuristics can reduce the number of buffers considerably, only using one if there actually is a dependency cycle.

Additionally, the above-mentioned restrictions on the execution step could be approached. This includes the recognition of receivers to enable synchronous event transitions. It also concerns the successive operations on concurrent data elements. While a global buffering for the entire step can lead to inconsistencies if several internal transitions depend on the value, a buffering at least during event broadcasts would be one step towards the semantics of SLIM.

Finally, the Simulink blocks could be arranged in a neat way. This paper only provides the semantics for the translation, describing which block to use in what configuration. To increase the usability of the generated model, the Simulink blocks should be positioned in a clear way, which places connected blocks next to each other and minimises the number of signal crossings. The placement is a hard problem to solve, and various external tools and algorithms might be used here.

Bibliography

- [1] A. Angermann, M. Beuschel, and M. Rau. *Matlab - Simulink - Stateflow. Grundlagen, Toolboxen, Beispiele*. Oldenbourg, 7 edition, 2011.
- [2] O. Beucher. *MATLAB und Simulink: grundlegende Einführung für Studenten und Ingenieure in der Praxis*, volume 10. Pearson Deutschland GmbH, 2008.
- [3] M. Bozzano, A. Cimatti, M. Roveri, J. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 121–130, 2009.
- [4] M. Bozzano and A. Villaflorita. The FSAP/NuSMV-SA safety analysis platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.
- [5] COMPASS. <http://compass.informatik.rwth-aachen.de/>. visited 2015-09-27.
- [6] J. Dormand and P. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980.
- [7] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.
- [8] HASDEL. <https://es-static.fbk.eu/projects/hasdel/>. visited 2015-09-27.
- [9] Lego. www.lego.com. visited 2015-09-27.
- [10] Lego Mindstorms. www.lego.com/en-gb/mindstorms/. visited 2015-09-27.
- [11] Lego Mindstorms Support. www.lego.com/en-gb/mindstorms/support/. visited 2015-09-27.
- [12] MathWorks. www.mathworks.com/. visited 2015-09-27.
- [13] MathWorks Inc. MATLAB. External Interfaces. 2015.
- [14] MathWorks Inc. Simulink. Developing S-Functions. 2015.
- [15] MathWorks Inc. Simulink. Reference. 2015.
- [16] MathWorks Inc. Simulink. Release Notes. 2015.
- [17] MathWorks Inc. Simulink. User’s Guide. 2015.
- [18] MathWorks Inc. Stateflow. User’s Guide. 2015.
- [19] MATLAB. www.mathworks.com/products/matlab/. visited 2015-09-27.
- [20] T. Noll. Semantics of the HASDEL System-Level Integrated Modeling (SLIM) Language. 2014.

- [21] RWTH Aachen University, Software Modeling and Verification Group. Specification of the HASDEL System-Level Integrated Modeling (SLIM) Language. 2014.
- [22] SAE International. <http://www.sae.org/>. visited 2015-09-27.
- [23] SAE International. Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex. SAE Standard AS5506/1. 2006.
- [24] SAE International. Architecture Analysis and Design Language (AADL) V2. SAE Draft Standard AS5506 V2. 2008.
- [25] Simulink. www.mathworks.com/products/simulink/. visited 2015-09-27.
- [26] Simulink Verification and Validation. www.mathworks.com/products/simverification/. visited 2015-09-27.
- [27] Stateflow. www.mathworks.com/products/stateflow/. visited 2015-09-27.

Appendix A

Auxiliary Systems

The following Subsystems are introduced by the helper:

`_checkMode_traj>d<_in_<m>` compares the current mode to a predefined mode m in which the trajectory equation for component d is active.

`_checkPhase_traj>d<_in_<m>` checks the current phase whether the updater for d should be executed.

`_clock` is an outgoing data port of the “*rootSystem*” to inform the environment how much time has passed in the simulation (adjusted for dummy steps).

`_Con<Nr>` is a system to activate a connection depending on the current mode. Contains only a single simple connection between *In* and *Out*.

`_default_<d>` writes the default value into the data subcomponent d while it is deactivated so that it is reset upon reactivation.

`_Flow<Nr>` is a system containing the Simulink blocks that constitute the flow expression if the flow is active depending on the mode.

`_Inport_<p>` is the Inport block for the incoming data port p . It has to be renamed to avoid clashes with subcomponent names.

`_merge_<p>` combines the values from different connections and flows targeting the same port in different modes by picking the one that is currently active.

`_Mode_Events` bundles all incoming events of the Stateflow chart into a vector based signal. Only that way, they can trigger the Stateflow chart.

`_modes_trigger` generates an event whenever executed. This is used to trigger the “*modes*” chart in every time step even if it has incoming events.

`_Outport_<p>` is the Outport block for the outgoing data port p . It has to be renamed to avoid clashes with subcomponent names.

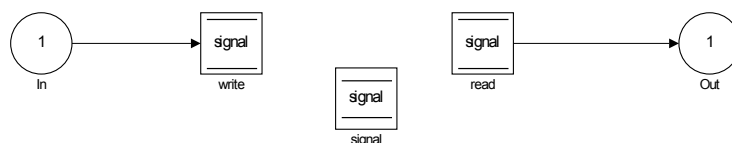


Figure A.1: Subsystem to define a port default value

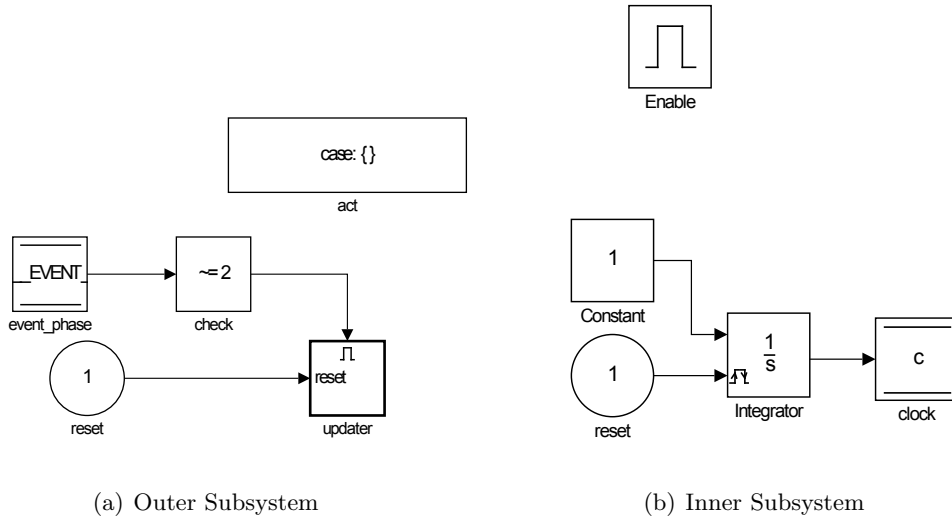


Figure A.2: Subsystems to update a clock (outer is activated depending on mode, inner depending on phase)

_phase<Nr> is a `DataStoreRead` to access the current phase of the simulation. Input for the `_checkPhase_traj>d<_in_<m>` block.

port<p> is the name for `Inport` and `Outport` blocks for event ports. Additionally, it is the name for the subsystem containing the data store for data ports. This is the block to which all signals from the system to the port are directed.

reset<d> is the name of the event to reset the updater of the continuous data subcomponent *d*. It is invoked whenever the subcomponent is set in a mode transition.

signal<s> is a system to buffer the signals that targets *s*. It contains a data store to break dependency cycles.

SwitchMode<name> activates the block *name* if the system is in one of the modes defined in this `SwitchCase` block.

traj<d>_in_<m> is the updater for the continuous data subcomponent *d* through a trajectory equation in mode *m*. It contains an `Integrator`.

update<d> updates the clock *d* when it is active (decided with a `SwitchCase`). Contains an enabled subsystem for the phase test.

Appendix B

Stateflow Syntax Example

```
Stateflow {
  machine {
    id 1
    name "instances"
    firstTarget 37
  }
  chart {
    id 2
    name "rootSystem/slimModel/p/t2/modes"
    firstTransition 4
    machine 1
    firstData 5
    updateMethod INHERITED
    executeAtInitialization 1
    disableImplicitCasting 1
    actionLanguage 1
    outputData 5
    activeStateOutput {
      useCustomName 1
      customName "Mode"
      useCustomEnumTypeName 1
      enumTypeName "Thr_Impl_ModeType"
    }
  }
  state {
    id 3
    labelString "DefaultInitialMode"
    position [100 75 90 60]
    chart 2
  }
  data {
    id 5
    name "Mode"
    linkNode [2 0 6]
    scope OUTPUT_DATA
    machine 1
    outputState 2
    dataType "Enum: Thr_Impl_ModeType"
  }
}
```

```

data {
  id      6
  name    "GLOBAL__EVENT__PHASE"
  linkNode [2 5 0]
  scope   DATA_STORE_MEMORY_DATA
  machine  1
  props {
    range {
      minimum "0"
      maximum "3"
    }
  }
  dataType "uint8"
}
transition {
  id      4
  labelString ""
  src {
    intersection [0 1 0 0 70 90 0 0]
  }
  dst {
    id      3
    intersection [4 -1 0 0.75 100 90 0 0]
  }
  midPoint [85 90]
  chart    2
  linkNode [2 0 0]
}
instance {
  id      7
  name    "rootSystem/slimModel/p/t2/modes"
  machine  1
  chart    2
}
target {
  id      8
  name    "sfun"
  machine  1
  linkNode [1 0 0]
}
}

```

Appendix C

Robot Model

C.1 SLIM

This is the complete SLIM model for the Lego robot in Chapter 4:

```
-- main component representing the entire robot satellite
system Sat
  features
    dist: in data port real;
    light: in data port int;
    motor_thrust: out data port [-100..100] default -40;
    motor_pitch: out data port [-100..100] default 0;
    motor_yaw: out data port [-100..100] default 0;
    in_pitch: in data port int;
    in_yaw: in data port int;
end Sat;

system implementation Sat.Imp
  subcomponents
    -- control components
    Thruster: system Thruster.Imp;
    battery: system Battery.Imp;
    turn: system Turn.Imp in modes (to_comm, to_obs);
    find_light: system FindLight.Imp in modes (init_charge, charge);
    -- used to switch periodically between comm and obs
    clk: data clock sec in modes(observe, comm);
    last_is_comm: data bool default false;
  connections
    -- Thruster needs distance to "earth" and motor for pole
    port dist -> Thruster.dist;
    port Thruster.motor -> motor_thrust;
    -- battery needs power of sun
    port light -> battery.sun;
    -- turn controls motors, positions them as indicated by "comp_"
    port turn.motor_pitch -> motor_pitch in modes (to_comm, to_obs);
    port turn.motor_yaw -> motor_yaw in modes (to_comm, to_obs);
    port in_pitch -> turn.in_pitch in modes (to_comm, to_obs);
    port in_yaw -> turn.in_yaw in modes (to_comm, to_obs);
    -- find_light needs control of motors and the light sensor
    port light -> find_light.light in modes (init_charge, charge);
    port find_light.motor_pitch -> motor_pitch in modes (charge);
```

```

    port find_light.motor_yaw -> motor_yaw in modes (charge);
    port in_yaw -> find_light.in_yaw in modes (init_charge, charge);
    port in_pitch -> find_light.in_pitch
        in modes (init_charge, charge);
flows
    -- constant positions which represent the states
        "observe" and "communicate"
    turn.comp_pitch := -20 in modes (to_obs);
    turn.comp_yaw := 90 in modes (to_obs);
    turn.comp_pitch := 15 in modes (to_comm);
    turn.comp_yaw := -100 in modes (to_comm);
modes
    -- turning into position and observing / communicating
    to_obs: initial mode;
    observe: mode;
    to_comm: mode;
    comm: mode;
    -- turn towards light and charge
    init_charge: mode;
    charge: mode;
transitions
    -- battery low -> charge
    observe -[battery.low then last_is_comm:=false]-> init_charge;
    to_obs -[battery.low then last_is_comm:=false]-> init_charge;
    to_comm -[battery.low then last_is_comm:=true]-> init_charge;
    comm -[battery.low then last_is_comm:=true]-> init_charge;
    -- intermediate state "init_charge" only needed
        to trigger event "rst"
    init_charge -[find_light.rst]-> charge;
    -- fully charged
    charge -[battery.full when last_is_comm]-> to_comm;
    charge -[battery.full when not last_is_comm]-> to_obs;
    -- reached position for observation/communication
    to_obs -[turn.ready then clk:=0]-> observe;
    to_comm -[turn.ready then clk:=0]-> comm;
    -- switch between observe and comm
    observe -[when clk>15sec]-> to_comm;
    comm -[when clk>15sec]-> to_obs;
end Sat.Imp;

-- orbit control mechanism
system Thruster
    features
        dist: in data port real;
        motor: out data port [-100..100] default -15;
    end Thruster;

system implementation Thruster.Imp
    modes
        up: initial mode;
        down: mode;

```

```

transitions
  -- negative values raise satellite , positive ones lower it
  up -[when dist>33 then motor:=2]-> down;
  down -[when dist<10 then motor:=-15]-> up;
end Thruster.Imp;

-- model of battery
system Battery
  features
    low: out event port;
    full: out event port;
    sun: in data port int;
end Battery;

system implementation Battery.Imp
  subcomponents
    level: data continuous default 100;
  modes
    discharge: initial mode while level' = -3 per sec;
    charge_slow: mode while level'=3 per sec;
    charge_fast: mode while level'=13 per sec;
  transitions
    -- charge
    discharge -[low when level<15]-> charge_slow;
    charge_slow -[when sun>=18]-> charge_fast;
    charge_fast -[when sun<18]-> charge_slow;
    -- fully charged
    charge_slow -[full when level>=100 then level:=100]-> discharge;
    charge_fast -[full when level>=100 then level:=100]-> discharge;
end Battery.Imp;

-- find the position with the highest light intensity
system FindLight
  features
    light: in data port int;
    motor_pitch: out data port [-100..100] default 0;
    motor_yaw: out data port [-100..100] default 0;
    in_yaw: in data port int;
    in_pitch: in data port int;
    rst: in event port;
end FindLight;

system implementation FindLight.Imp
  subcomponents
    best: data int default 0;
    turn: system Turn.Imp in modes (init);
  connections
    port in_pitch -> turn.in_pitch in modes (init);
    port in_yaw -> turn.in_yaw in modes (init);
  flows

```

```

-- target for turn: initial position to start search
turn.comp_pitch := 40 in modes (init);
turn.comp_yaw := 120 in modes (init);
-- motor_pitch either controlled by turn,
    or constant depending on mode
motor_pitch := turn.motor_pitch in modes (init);
motor_pitch := -18 in modes (find_pitch);
motor_pitch := 0 in modes (find_yaw, done);
-- motor_yaw either controlled by turn,
    or constant depending on mode
motor_yaw := turn.motor_yaw in modes (init);
motor_yaw := -40 in modes (find_yaw);
motor_yaw := 0 in modes (find_pitch, done);
modes
  init: initial mode;
  find_pitch: mode;
  find_yaw: mode;
  done: mode;
transitions
  -- reset system, moving into initial position
  find_pitch -[rst]-> init;
  find_yaw -[rst]-> init;
  done -[rst]-> init;
  -- initial position reached: start search with pitch
  init -[turn.ready then best:=light]-> find_pitch;
  find_pitch -[when light>=best then best:=light]-> find_pitch;
  -- pitching decreases light -> stop, yaw instead
  find_pitch -[when light<best-2 or in_pitch<-40]-> find_yaw;
  find_yaw -[when light>=best then best:=light]-> find_yaw;
  -- best position reached
  find_yaw -[when light<best-2 or in_yaw<-120]-> done;
end FindLight.Imp;

-- helper to turn satellite into given position
system Turn
  features
    in_pitch: in data port int;
    in_yaw: in data port int;
    motor_pitch: out data port [-100..100] default 0;
    motor_yaw: out data port [-100..100] default 0;
    -- destination position
    comp_pitch: in data port int;
    comp_yaw: in data port int;
    ready: out event port;
end Turn;

system implementation Turn.Imp
  modes
    pitch: initial mode;
    yaw: mode;
    check: mode;

```

```

transitions
  -- pitching robot until within margin of destination
  yaw -[when not(comp_pitch-5<=in_pitch and in_pitch<=comp_pitch+5)
        and (comp_yaw-9<=in_yaw and in_yaw<=comp_yaw+9)
        then motor_yaw:=0]-> pitch;
  check -[when not (comp_pitch-5<=in_pitch
                    and in_pitch<=comp_pitch+5)]-> pitch;
  pitch -[when in_pitch < comp_pitch-20
          then motor_pitch:=20]-> pitch;
  pitch -[when comp_pitch-20 <= in_pitch
          and in_pitch < comp_pitch-5
          then motor_pitch:=13]-> pitch;
  pitch -[when comp_pitch+5 < in_pitch and in_pitch<comp_pitch+20
          then motor_pitch:=-9]-> pitch;
  pitch -[when comp_pitch+20 <= in_pitch
          then motor_pitch:=-20]-> pitch;
  -- yawing robot
  pitch -[when (comp_pitch-5<=in_pitch and in_pitch<=comp_pitch+5)
          and not (comp_yaw-9<=in_yaw and in_yaw<=comp_yaw+9)
          then motor_pitch:=0]-> yaw;
  check -[when not (comp_yaw-9<=in_yaw
                    and in_yaw<=comp_yaw+9)]-> yaw;
  yaw -[when in_yaw < comp_yaw-35 then motor_yaw:=55]-> yaw;
  yaw -[when comp_yaw-35 <= in_yaw and in_yaw < comp_yaw-9
          then motor_yaw:=29]-> yaw;
  yaw -[when comp_yaw+9 < in_yaw and in_yaw < comp_yaw+35
          then motor_yaw:=-26]-> yaw;
  yaw -[when comp_yaw+35 <= in_yaw then motor_yaw:=-55]-> yaw;
  --finished
  yaw -[when (comp_pitch-5<=in_pitch and in_pitch<=comp_pitch+5)
          and (comp_yaw-9<=in_yaw and in_yaw<=comp_yaw+9)
          then motor_yaw:=0]-> check;
  pitch -[when (comp_pitch-5<=in_pitch and in_pitch<=comp_pitch+5)
          and (comp_yaw-9<=in_yaw and in_yaw<=comp_yaw+9)
          then motor_pitch:=0]-> check;
  check -[ready when (comp_pitch-5<=in_pitch
                     and in_pitch<=comp_pitch+5)
          and (comp_yaw-9<=in_yaw
              and in_yaw<=comp_yaw+9)]-> pitch;
end Turn.Imp;

```

C.2 Simulink

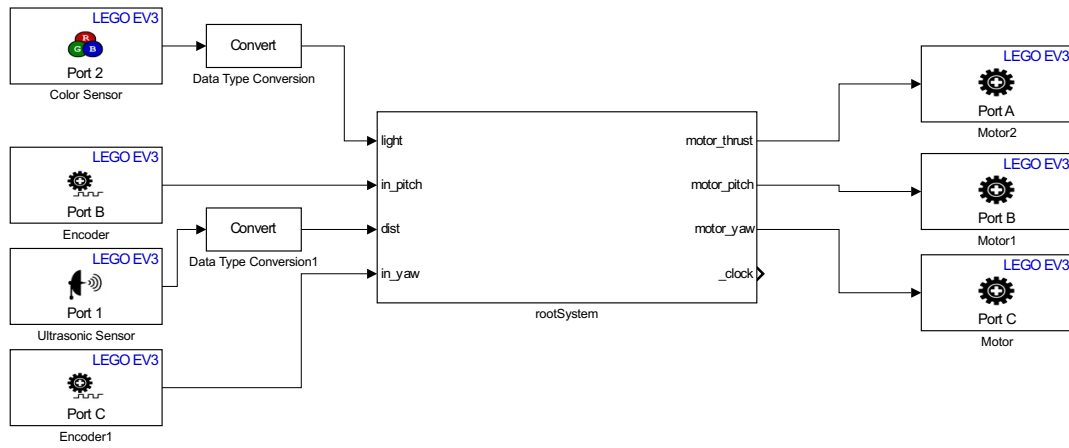


Figure C.1: outermost system

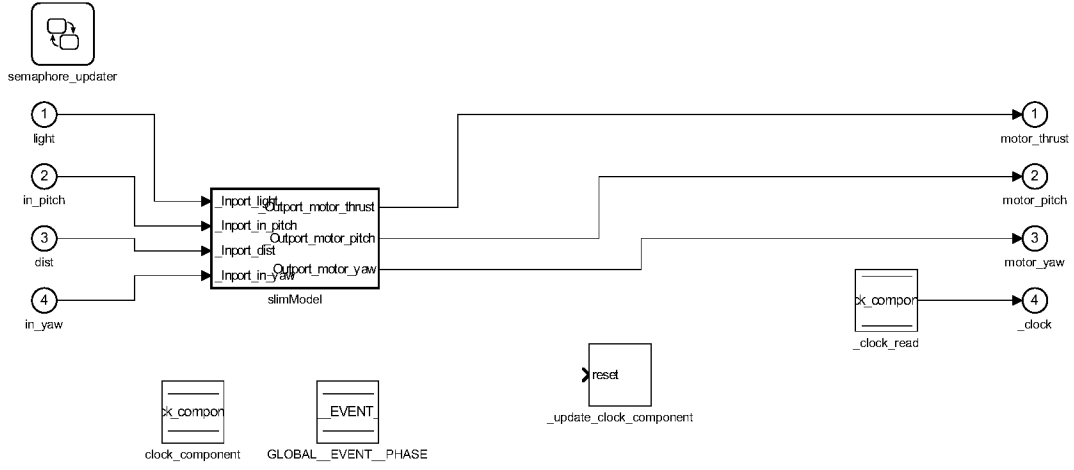


Figure C.2: inside "rootSystem"

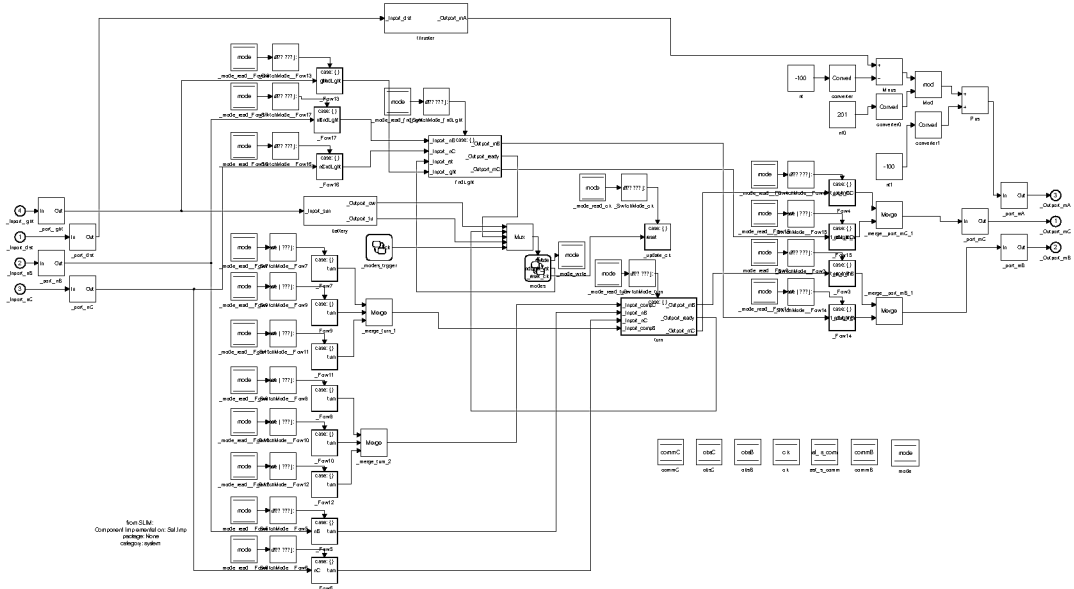


Figure C.3: inside "slimModel"

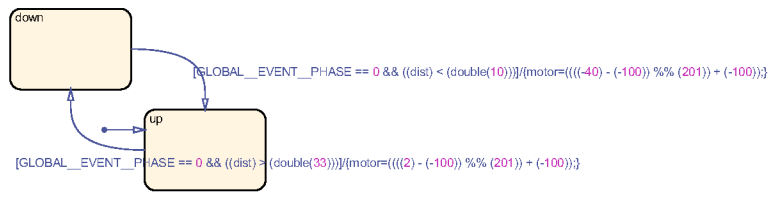
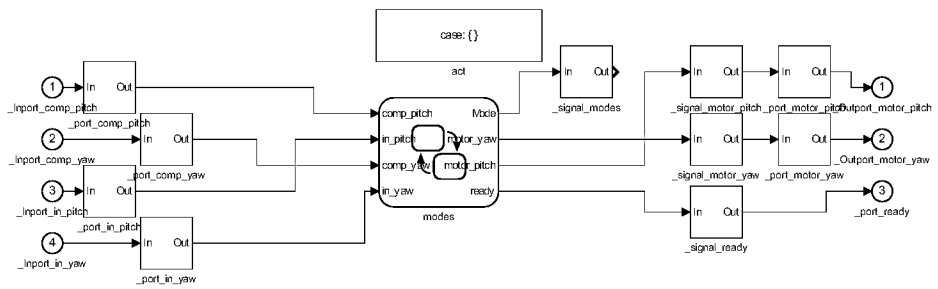


Figure C.6: modes of "Thrust"



from SLIM:
 Component Implementation: Turn.Imp
 package: None
 category: system

Figure C.7: inside "Turn"

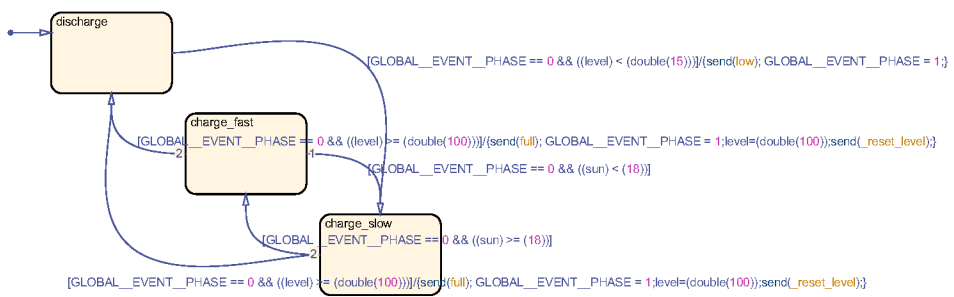


Figure C.10: modes of "Battery"