Formal verification of implementations of Stateflow charts

Alvaro Heiji Miyazawa

Submitted for the degree of Doctor of Philosophy

The University of York
Department of Computer Science

February 2012

To my mother.

Abstract

Simulink diagrams are widely used in industry for specifying control systems, and a particular type of block used in them is a Stateflow chart. Often, the systems specified are safety-critical ones. Therefore, the issue of correctness of implementations of these systems is relevant. We are interested in the verification of implementations of Stateflow charts.

In this thesis, we propose a formal model of Stateflow charts in the *Circus* notation. The proposed model makes a distinction between the general semantics of Stateflow charts and the specific aspects of each chart, and maintains the operational style used in the official informal description of the semantics of Stateflow. In this way, we support the comparison of our model to the informal description as an extra form of validation. Moreover, this separation allows us to obtain a translation from a Stateflow chart to a *Circus* model based mostly on the syntactic structure of the chart.

We formalise in Z a translation strategy that supports the generation of the chart specific model which is composed with the model of the semantics of Stateflow charts to formalise the execution of the chart. The translation strategy is implemented in a tool that supports the automatic generation of the complete model of a chart. The style in which the translation strategy is specified supports a very direct implementation, thus, minimising this potential source of error.

We identify an architecture of parallel implementations based on the sequential implementations automatically generated by a code generator, and propose a refinement strategy that applies the *Circus* refinement calculus to verify the correctness of the implementation with respect to the proposed formal model of Stateflow charts. The identification of the architecture allows us to specify the refinement strategy in a degree of detail that renders it suitable for formalisation in a tactical language, thus, potentially achieving a high degree of automation. Moreover, this strategy is a starting point for new strategies targeting different architectural patterns.

Contents

1 Introduction **1**

1.1 Objectives . . . 2

1.2 Thesis structure . . . 4

2 Literature review **5**

2.1 Stateflow . . . 6

2.1.1 Elements of the notations . . . 6

2.1.2 Informal semantics of Stateflow charts . . . 9

2.1.3 Early return logic . . . 13

2.2 Formal models . . . 14

2.2.1 Verification approaches . . . 15

2.2.2 Code generation approaches . . . 19

2.3 Specification languages . . . 20

2.4 Refinement calculus . . . 21

2.5 *Circus* . . . 21

2.6 Final considerations . . . 24

3 Formal Model **25**

3.1 Overview . . . 26

3.2 Process *Simulator* . . . 31

3.2.1 Step of execution . . . 32

3.2.2 Transition . . . 35

3.2.3 Entering a state . . . 42

3.2.4 Executing and exiting a state . . . 48

3.3 Chart process . . . 52

3.4 Validation . . . 59

3.5 Final considerations . . . 60

4 Translation strategy **63**

4.1 Syntax of Stateflow charts . . . 64

4.1.1 Names and Identifiers . . . 64

4.1.2 Expressions and Actions . . . 65

4.1.3 Stateflow objects . . . 66
4.2 Well-formedness conditions . . . 71
4.3 Translation strategy in Z . . . 72
4.3.1 Renaming functions . . . 73
4.3.2 Expression and Action functions . . . 74
4.3.3 Identifier and binding declaration functions . . . 77
4.3.4 Action, condition and process declaration functions . . . 78
4.4 Automation of the translation strategy . . . 82
4.4.1 Architecture . . . 82
4.4.2 Implementation of translation rules . . . 90
4.5 Evaluation . . . 93
4.6 Final considerations . . . 95

5 Refinement strategy 97
5.1 Implementations of Stateflow charts . . . 98
5.1.1 Architecture: data patterns . . . 99
5.1.2 Architecture: control flow . . . 102
5.2 *Circus* models of implementations . . . 106
5.3 Refinement strategy . . . 108
5.3.1 Data refinement . . . 109
5.3.2 Normalisation . . . 113
5.3.3 Structuring . . . 117
5.3.4 Parallelism introduction . . . 158
5.3.5 Action introduction . . . 166
5.4 Final considerations . . . 167

6 Conclusions 169
6.1 Thesis contributions . . . 169
6.2 Related Work . . . 172
6.3 Future work . . . 173

A Syntax of *Circus* 177

B *Circus* model of Stateflow semantics 179
B.1 Basic definitions . . . 179
B.2 Stateflow semantics . . . 180

C *Circus* models of Stateflow charts 197
C.1 *Circus* model of Shift Logic Chart . . . 197
C.2 *Circus* model of Air Controller Chart . . . 213

D *Circus* model of the implementation of Air Controller Chart 225

E Novel refinement laws **231**

List of Figures

2.1 Executing a Set of Transitions [98]. 10
2.2 Entering a State [98]. 11
2.3 Executing an active State [98]. 12
2.4 Exiting an active State [98]. 12
2.5 Early return logic example. 13
2.6 The *Buffer* process 22

3.1 Basic architecture of Stateflow models. 25
3.2 Example of a Stateflow Chart describing a car's shift logic. 27
3.3 Overview of the model 28
3.4 Structure of the *Simulator* process. 31
3.5 Executing a Set of Transitions [98] 43
3.6 Entering a State [98] 44
3.7 Executing an active State [98] 49
3.8 Exiting a State [98] 51
3.9 Structure of the *c_shift_logic* process. 53

4.1 Translation strategy: overview. 63
4.2 Architecture of s2c: main packages. 83
4.3 Architecture of the package `Parser`. 83
4.4 Structure of a *.mdl* file. 84
4.5 Excerpt from a *.mdl* file. 84
4.6 Architecture of the package `MDL Parser`. 85
4.7 State label 85
4.8 Transition label 85
4.9 Architecture of the package `Label Parser`. 86
4.10 Syntax of Stateflow charts: main packages. 86
4.11 Syntax of Stateflow charts: Objects. 87
4.12 Syntax of Stateflow charts: Actions. 88
4.13 Syntax of Stateflow charts: Expressions. 88
4.14 Syntax of *Circus*. 89
4.15 Architecture of s2c: the Translator package. 89

4.16 Implementations of the translation rule for variable expressions. 91
4.17 Implementations of the translation rule for the declaration of data. 92
4.18 Implementation of simulation instance rule 92

5.1 Overview of our refinement strategy. 97
5.2 Air controller chart: supplied with Stateflow. 98
5.3 Architecture of implementations of Stateflow charts. 99
5.4 Function `MDLInitialize`. 103
5.5 Structure of the procedure `calculate_output`. 104
5.6 Structure of `calculate_step` and interaction patterns with the servers. . . 104
5.7 Procedure `calculate_step` for our example. 105
5.8 Parallelism in the execution of the parallel states in our example. 106
5.9 Overview of the models of implementations of Stateflow charts. 107
5.10 Action *Client* in implementation models. 107
5.11 Refinement strategy: data refinement phase. 110
5.12 Schema D_Work_Air. 111
5.13 Total functional retrieve relation for our example. 112
5.14 Calculated total surjective functional retrieve relation: general form. 114
5.15 Refinement strategy: normalisation phase. 115
5.16 Normalisation – Example: Main action after Steps 1 and 2. 115
5.17 Normalisation – Example: Main action at the end. 116
5.18 Structuring starting point. 117
5.19 Structuring target. 119
5.20 Structuring target - chart execution. 119
5.21 Structuring target - writing the outputs. 119
5.22 Refinement strategy: structuring phase 121
5.23 Structuring: body of the outmost recursion in the main action after Step 2. 121
5.24 Structuring: part of the main action after Step 2. 122
5.25 Structuring: main action after Step 2 - writing outputs. 122
5.26 Structuring: body of the outmost recursion in the main action after Step 3. 123
5.27 Refinement strategy: structuring phase - input-event-var-introduction 125
5.28 parallelism-resolution: prefixing over channel in the synchronisation set on the right-hand side. 125
5.29 parallelism-resolution: steps for prefixing over channel in the synchronisation set on the right-hand side. 126
5.30 parallelism-resolution: result for prefixing over channel in the synchronisation set on the right-hand side. 126
5.31 parallelism-resolution: prefixing over channel not in the synchronisation set on either side. 128
5.32 parallelism-resolution: leading interleaving on the left-hand side. 129
5.33 parallelism-resolution: result for leading interleaving on the left-hand side. . . 129

5.34 parallelism-resolution: alternation followed by sequence, on either side. 130
5.35 parallelism-resolution: steps for alternation followed by sequence, on either side. 131
5.36 parallelism-resolution: possible results for alternation followed by sequence, on either side. 131
5.37 parallelism-resolution: steps for call action on either side. 132
5.38 parallelism-resolution: leading schema operation on the left-hand side. 133
5.39 parallelism-resolution: result for leading schema operation on the left-hand side. 133
5.40 parallelism-resolution: explicit recursion on the right-hand side. 133
5.41 parallelism-resolution: result for explicit recursion on the right-hand side. . . . 133
5.42 parallelism-resolution: steps for leading prefixing over channel in the synchronisation set on the left-hand side. 134
5.43 parallelism-resolution: local event broadcast. 135
5.44 parallelism-resolution: steps for local event broadcast. 135
5.45 parallelism-resolution: result for local event broadcast. 136
5.46 Procedure copy 141
5.47 Refinement strategy:structuring phase - recursion-introduction. 143
5.48 Refinement strategy: structuring phase - assignment-introduction 145
5.49 update-output starting point. 146
5.50 Refinement strategy: structuring phase - update-output 147
5.51 Refinement strategy: structuring phase - simplification 149
5.52 Refinement strategy: structuring phase - early-return-simplification 151
5.53 Procedure assumption-distribution. 152
5.54 Refinement strategy: structuring phase - parallel-state-simplification 153
5.55 parallel-state-simplification: example after step 2. 155
5.56 parallel-state-simplification: example after step 3(a). 155
5.57 parallel-state-simplification: example after step 3(b). 156
5.58 parallel-state-simplification: example after step 3(c)-i. 156
5.59 parallel-state-simplification: example after step 3(c)-ii. 157
5.60 Procedure sequential-state-simplification 157
5.61 Functions implemented the server for our example 159
5.62 Parallelism introduction - example of execution of parallel states. 159
5.63 Parallelism introduction target - execution of parallel states example. 160
5.64 Parallelism introduction target - server example. 161
5.65 Refinement strategy: parallelism introduction phase 161
5.66 Parallelism introduction - example: portion of the main action after Step 1. 163
5.67 Refinement strategy: action introduction phase. 166

Acknowledgements

I would like to, firstly, thank my supervisor, Ana Cavalcanti, for all her support and guidance. Without her encouragement and advice this thesis would not have been possible.

I would like to express my gratitude to my examiners, Professor Richard Paige and Professor Michael Butler, whose comments and suggestions were invaluable to this thesis.

I would also like to thank my colleagues at the Department of Computer Science, and the members of the *Circus* group; in particular, I am grateful to Frank Zeyda for many discussions throughout my time in York, and Leo Freitas who helped me understand *Circus* and the CZT framework.

Thanks to my family and friends whose support and friendship helped me maintain my sanity. Special thanks to my partner, Oleg Lisagor, who read and commented on various drafts of my thesis and gave me the all support I needed to complete this thesis, my friends in York, André Freire, Luiza Dias, Jennifer Winter and David Efird whose friendship and ad hoc counselling sessions in York's many pubs were indispensable, and my friends and former colleagues in Brazil – Ana Melo, Paulo Salem, Renata Matteoni, Jony Arrais, Patrícia Viana, Márcio Medeiros and Renato Massaro – who have provided me with support that, whilst difficult to pinpoint, was indispensable.

Most importantly, I am forever indebted to my mother, who throughout my life encouraged me to further my studies, and always supported me in all my endeavours. I could never possibly thank her enough. To her I dedicate this Thesis.

Author's declaration

I hereby declare that the contents of this thesis are the result of my own original contribution, except where otherwise stated. The material in chapters 3 and 4 has previously been published in [68, 69, 70].

[68] A. Miyazawa and A. L. C. Cavalcanti. Towards the formal verification of implementations of Stateflow Diagrams. Tech. Rep. YCS-2010-449, University of York, 2010.

[69] A. Miyazawa and A. L. C. Cavalcanti. A formal semantics of Stateflow charts. Tech. Rep. YCS-2011-461, University of York, 2011.

[70] A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 2011. doi:10.1016/j.scico.2011.07.007.

An initial version of chapter 5 has previously been published in [71], and the current version is in the process of being submitted for publication [72].

[71] A. Miyazawa and A. L. C. Cavalcanti. Refinement-based verification of sequential implementations of Stateflow charts. In *Proceedings 15th International Refinement Workshop*, volume 55, pages 65–83. Electronic Proceedings in Theoretical Computer Science, 2011. doi:10.4204/EPTCS.55.5.

[72] A. Miyazawa and A. L. C. Cavalcanti. Refinement-based verifications of implementations of Stateflow charts. 2012. (to be submitted).

Chapter 1

Introduction

Stateflow is part of the MATLAB Simulink tool [99] and consists of a statechart notation used to define charts used as blocks in control law diagrams. Control law diagrams are a popular notation for specifying control systems, and are widely used in the avionics and automotive industries.

While control law diagrams tackle the aspects of a system that are usually specified by differential equations relating inputs and outputs graphically, Stateflow charts are used to describe the aspects that are better modelled by finite state machines. For example, in a system with redundant subsystems, a Stateflow chart can be used to specify an automated reconfiguration procedure based on individual subsystems' health monitoring statuses. Another example is a switch-over between modes of operation of an aircraft system based on the phase of flight (e.g. take-off, climb, cruise, etc.) as commanded by the pilots and/or indicated by aircraft sensors (e.g. weight on wheels and ground speed indicators).

Some of the systems specified using Simulink diagrams and Stateflow charts are safety-critical systems. Such systems may cause death, injury, significant environmental damage or other material loss. The necessary rigour of verification and validation of software used in safety critical applications is often expressed in terms of Safety (or Software) Integrity Levels (SILs) which are determined based on the potential severity of the unsafe system-level conditions (hazards) that the software may contribute to and the extent of the contribution. For example, a software controller may sometimes be assigned a lower SIL if its outputs are checked by an independent sub-system (e.g. a monitor or an interlock) than if the controller has full authority over the system.

Various international standards provide guidance on which design, verification and validation techniques should be used for different SILs. For instance, the international standard IEC 61508 [47] recommends the use of formal methods for the specification of safety requirements as well as for the design and development of software for SILs 2 and 3, while highly recommending those techniques for SIL4; the standard also recommends the use of formal proof for the verification of software. Similar recommendations are reflected in domain-specific adaptations of IEC 61508 such as CENELEC 50128 [25] (for railway control systems) and the recently published ISO 20262 [48] (for automotive applications).

In civil aerospace, the applicable standard DO-178b [83], while not mandating the use of formal methods, explicitly recognises them as an alternative method. Finally, the currently superseded UK Military Standard DEF STAN 00-55 [62] makes the use of formal methods mandatory for SILs 3 and 4; it also requires explicit justification to be provided if formal methods are not used for lower SILs.

These requirements and recommendations contained in standards demonstrate that formal techniques that are capable of dealing with modelling languages and notations used in industry (such as MATLAB Simulink and Stateflow) are useful, if not necessary.

There are several approaches to the formal analysis of state diagram notations such as Stateflow. However, most of these approaches focus on the verification of properties, not on the verification of implementations; the verification of implementations with respect to a specification can be seen as a particular type of property verification in which the property in question is that the implementation is a refinement of the specification. By restricting ourselves to the verification of implementations we can take advantage of specific techniques (such as the refinement calculus) that are more adequate for this task.

One of the main approaches used for the verification of implementations consists of verifying a code generator [102]. This approach makes the implementation immutable because only generated code is correct by construction; however, in many cases, code tailored to specific situations is necessary, which makes such an approach not applicable. The verification of implementations, instead of the code generator, can overcome this problem, but can potentially increase the complexity of the task.

Arthan et al. [4] and Adams and Clayton [3] describe ClawZ, a tool for translating Simulink diagrams into Z [109] in order to formally verify implementations in Ada. This approach does not cover the Stateflow notation and can only deal with sequential implementations; to overcome the latter limitation, concurrent aspects are specified in CSP [85] and analysed through the model checker `FDR2` [30].

Cavalcanti and Clayton [17] define the semantics of control law diagrams in the *Circus* notation [79], which is a refinement language that combines Z, CSP, Dijkstra's language of guarded commands [22], and Morgan's specification statement [73]. This semantics reuses ClawZ and the CSP approach to concurrency, and extends them to cover a larger subset of the notation, but it still does not cover the Stateflow notation. A *Circus* model of Stateflow charts is a natural extension of previous work, allowing for the verification of a broader variety of control law diagrams.

1.1 Objectives

We are concerned with the verification of implementations of Stateflow charts, i.e., we want to verify whether a program correctly implements a Stateflow chart or not. We focus this work in the Stateflow variety of state diagrams because it is part of the widely used MATLAB Simulink tool.

In order to achieve the objective of verifying implementations of Stateflow charts, we

need, first, a formal semantics of these charts and, second, techniques suitable for the verification of implementations with respect to the proposed semantics.

The formal semantics of Stateflow charts needs to be suitable for formal reasoning about the correctness of implementations; it must be written in a way that facilitates validation and be subject to integration with models of Simulink diagrams. This last requirement is due to the fact that Stateflow charts are part of Simulink diagrams, and, in order to verify a complete system, we must be able to cover both notations.

The verification techniques must allow us to verify code and must also be consistent with the techniques used for Simulink diagrams, so that Simulink and Stateflow blocks within the same diagram can be verified in a uniform manner.

By using *Circus* as a specification language, we are able to tackle these desired properties: ability to formally reason about the model, to verify code and to integrate the model with the existing models of Simulink diagrams. Since *Circus* is a formal specification language, we can define properties of models and mathematically prove whether they hold or not.

The verification of code is carried out by proving that the code is (or is not) a refinement of a specification, and the integration with existing models of Simulink diagrams is possible because these models were previously specified in *Circus* [19].

Due to the informal nature of the Stateflow semantics, we need to define a formal model based on the informal description contained in the *Stateflow User's Guide* [98]. Since there is no official formal semantics with which we can formally compare our model, we must validate it through alternative approaches.

We distinguish three main possibilities: one is based on inspection of the informal description, the second is based on testing and is achieved by comparing a particular chart and its model by means of simulation, and the third consists of applying the refinement calculus to obtain an implementation of the chart.

In order to allow for the first form of validation, the model presented in Chapter 3 is defined in a way that facilitates the comparison to the informal description, which is given is steps in the *Stateflow User's Guide* for each of the main semantic rules that describe the behaviour of states and transitions. The comparison is eased by establishing a correspondence between the steps and elements of the specification.

The second form of validation can be achieved by simulation of the model and comparison of the traces to the results of the simulation of charts; we can improve this validation by using techniques for the selection of test cases of charts that yield better coverage.

Finally, by carefully applying the refinement calculus to models of Stateflow charts, we are able to validate the interaction between the different parts of the model, and spot any deviations of the expected behaviour of the chart, which might otherwise be overlooked. In this way, we allow for three different approaches for the validation of the model.

1.2 Thesis structure

This section describes the structure of this thesis. Chapter 2 reviews some of the variants of state diagram notations and formal specification languages that can be used to formalise the semantics of such notations. This chapter presents a brief introduction to Stateflow charts and *Circus*, and further reviews the existing approaches to modelling and analysis of state diagrams.

Chapter 3 presents an operational model of Stateflow charts by means of a small example; it discusses the rationale behind the particular structure of our models, and describes the formalisation of the semantics of Stateflow charts.

Chapter 4 presents the formalisation of the translation rules necessary to support the derivation of Stateflow models as presented in Chapter 3, and discusses the implementation of these translation rules in the tool *s2c* that supports the automatic generation of *Circus* models of Stateflow charts.

Chapter 5 identifies an architecture of implementations of Stateflow charts, and proposes a refinement based verification strategy that supports the verification of implementation conforming to the identified architecture with respect to the models discussed in Chapters 3 and 4. The architecture described in this chapter is based on the implementations generated by the Simulink/Stateflow code generator [100, 98] and extended to support parallel implementations of Stateflow charts. The verification strategy takes advantage of the architectural patterns described to provide a detailed step by step refinement strategy that can potentially lead to a high degree of automation.

Chapter 6 concludes with a discussion of the main contributions and limitations of our work, potential solutions to some of the limitations, and future lines of research.

Chapter 2

Literature review

This chapter sets the scene for the remainder of the thesis by presenting a review of the previous work and languages related to our main objective - verification of implementations of Stateflow charts.

Stateflow charts are a variant of Harel's statecharts [36], which are an extension of state diagrams. These are, basically, directed graphs, where nodes represent states and edges represent transitions [46]; the latter connect states and can be guarded by conditions. We identify three main variants of the state diagram notations that are obtained by adding new features, changing existing ones or modifying the semantics: Harel's statecharts, UML statecharts and Stateflow charts.

Statecharts [36] extend state diagrams to improve the expressive power of the basic notation, allowing for the specification of complex reactive systems, concurrent systems, communication protocols, etc. This extension is achieved by introducing concepts such as hierarchy of states (states within states), concurrency (parallel states) and communication (local events).

UML statecharts [81] are an object-oriented version of statecharts [53]. They mainly differ from Harel's statecharts with respect to the semantics, but also present syntactic differences, such as entry and exit actions in states [107].

Stateflow charts [98] extend statecharts by adding, among other features, flow charts, temporal logic triggers and different types of actions (during actions, on event actions, transition actions).

While both Stateflow charts and UML statecharts are widely used in industry, according to Crane and Dingel [21], even among close variants of statecharts, such as UML statecharts, Classical statecharts and Rhapsody statecharts, there are several syntactic and semantic differences. Moreover, Fecher et al. [27] identify 29 problems in the definition of the semantics of UML statecharts such as inconsistencies and omitted restrictions. Consequently, formal models of one variant cannot be easily reused for other variants.

The objective of this chapter is to present the context and the "baseline" of the research reported in this thesis. The remainder of the chapter is organised as follows. Firstly, we describe the Stateflow notation based on the description contained in the User's Guide [98];

we comment on some inconsistencies of this authoritative source and, where possible, correct the description of the semantics. Section 2.2 then reviews approaches for modelling and analysis of state diagram notations. The third section presents an overview of some of the languages that could be used to model Stateflow charts (Section 2.3). Section 2.5 presents Circus - the language we have selected for modelling the charts. The chapter concludes with some final observations and remarks (Section 2.6).

2.1 Stateflow

In this section, we give an brief overview of Stateflow charts based on the User's Guide [98].

Stateflow defines a new type of Simulink block, namely a Stateflow chart, that is used in a Simulink diagram. A Simulink diagram consists of blocks and wires connecting the inputs and outputs of the blocks. The execution of a Simulink diagram is done in steps, in which each of its blocks is executed in a particular order determined by the wiring.

The Stateflow chart is the root for the execution of the Stateflow model; whenever an event is directed at a chart, the chart is either entered or executed, depending on whether it was previously active or not. A chart comprises mainly objects of the following types: events, data, actions, states, junctions, transitions and function blocks, of which events and data can be used to communicate with other blocks of the Simulink diagram.

An event can also be used internally to trigger the execution of the chart (or part of it). This use potentially leads to recursive executions, which may lead the chart to a configuration where further execution leads to an inconsistent state. To avoid this, Stateflow uses early return logic to decide when it is safe to continue the execution, and when part of it must be interrupted. Early return logic is discussed in Section 2.1.3.

2.1.1 Elements of the notations

Events Events are objects that trigger the execution of a Simulink block, e.g. a chart. They can be distinguished by trigger type and scope. With respect to the trigger type, an event can be edge-triggered or function-call.

The basic difference between edge-triggered and function-call events is the time step in which outputs of the triggered block are available. If a block is triggered by an edge-triggered event, it is executed in the same time step as the Stateflow chart, but its outputs are only available in the next time step of the Stateflow chart. If a block is triggered by a function-call event, it is executed in the same time step as the Stateflow chart, and its outputs are available in that same time step, that is, a function-call triggered block is executed in interleave with the block that produced the function-call event.

The scope characterises an event as input, output, local, exported or imported. An input event is one generated in a different block in the Simulink diagram and processed by the chart. Charts can be triggered by a single function-call event, or by a sequence of edge-triggered events. In the latter case, the order is determined statically in the definition

of the chart, and is important because at each step the chart is executed once for each input event that has occurred.

An output event is generated by the chart and processed by some block in the Simulink diagram. Edge-triggered output events are only communicated to the Simulink models in the end of the step of execution (along with the output data), and present a queuing behaviour; if a chart broadcasts the same event more than once in an execution step, it queues the broadcasts and releases one per execution step. Function-call output events trigger the execution of a Simulink block immediately, and any outputs of the block (that are connected to the input ports of the chart) become available to the chart in the same time step. A local event is generated inside the chart and processed by the same chart; as previously mentioned, this can potentially lead to recursive behaviour and inconsistency.

An exported event is generated by the chart and processed by a module external to the chart and Simulink diagram; an imported event is generated by an external module and processed by the chart.

Data Another type of object is called data; it consists of variables that record values used by the chart. It can be used to record internal information or to communicate with the Simulink diagram.

Similar to events, data can be distinguished according to scope. Local data are defined in a particular state (or chart) and are available for their parent and children; input data are provided to the chart by the Simulink model through input ports; output data are provided by the chart to the Simulink model through output ports; data store memory are global variables of the Simulink model available to all blocks. Similarly to events, exported and imported data are used to share data with other Stateflow models or external code.

States A state can be active or inactive, and this status can be changed by entering and exiting it; it can also have sub-states. This creates a hierarchy of states inside a chart and because of this hierarchy, every state has a parent, including the top level states whose parent is the chart itself.

A state has a property called decomposition, which determines which sub-states (if any) can be active at any given time. The two types of decomposition are sequential (*CLUSTER*) and parallel (*SET*). A state can also have associated actions: *entry*, *during*, *exit* and *on* actions (refer to the description of actions below).

States with sequential decomposition can only have sub-states of type sequential (*OR*) and there always must be at most one active sub-state. States with parallel decomposition can only have sub-states of type parallel (*AND*) and either all the sub-states are inactive or all of them are active at the same time.

Junctions Junctions are connective nodes used to define decision points in a chart; they can be used, for example, to define `if-then-else` and `for` statements. Another type of junction is called history junction; it records information about the most recent active

sub-state of a state that contains the history junction. A history junction can stand by itself inside a state with sequential decomposition, or it can be reached by a transition.

Transitions A transition is usually a connection between two nodes, these nodes can be either states or junctions; a transition starts in a source node and ends in a destination node. A transition that starts or ends in a junction is called a transition segment, and a sequence of transition segments that starts and ends in states is called a transition path. A transition can connect nodes on different levels of the hierarchy; such a transition is called an inter-level transition.

A transition can have a trigger that consists of a set of events (possibly empty), a condition and two types of actions: *condition* and *transition* actions (refer to the description of actions below). A transition is considered to be valid if the event being processed by the chart triggers it and if its condition evaluates to true.

There are three types of transitions. Default transitions (or transition paths) determine which sub-state must be entered when entering a state; they have no source node. Outer transitions (or transition paths) connect a state to an external state. Finally, inner transitions (or transition paths) connect a state to one of its sub-states.

Actions Actions are objects that allow one to specify how to modify a particular variable, when to broadcast an event, etc. For example, an action can require that when certain conditions are met, local variable a is incremented and output event B is broadcast. Actions are always sequentially executed.

There are a series of actions that depend on the type of object that contains them and the scenario that triggers their execution. State actions are defined in the label of states and can be of type: *entry*, *exit*, *during*, *on* or *bind*.

- *entry* actions are executed when a state is entered;
- *exit* actions are executed when a state is exited;
- *during* actions are executed when an active state is executed and is not left through an outer transition;
- *on* actions are executed in the same situation as *during* actions with the additional restriction that the state is processing a specific event;
- *bind* actions make an event or data bound to a state, so that only the state and its children can broadcast the event or modify the data.

On actions can also be used with temporal operator that specify scenarios such as "after an event occurs n times".

Transitions can have two types of actions: *condition* and *transition* actions. A *condition* action is executed when a transition is deemed valid and a *transition* action is executed when a transition path is successful.

Function blocks A function block is an object that allow for the definition of functions that take input values and return output values. There are two types of function blocks that can be embedded in a Stateflow chart: graphical functions and Simulink functions. The former are defined using Stateflow objects and the action language, whereas the latter are defined by Simulink diagrams.

2.1.2 Informal semantics of Stateflow charts

The semantics of Stateflow is given primarily by simulation. However, it is also described in the User's Guide, scattered across object's descriptions and examples, and in a more coherent way, in Chapter 3 "Stateflow Chart Semantics". In this section, we will focus on how to execute a transition, and how to enter, execute and exit a state.

The description presented in Chapter 3 of the User's Guide contains some inconsistencies that were partially addressed in an appendix called *Semantic Rules Summary*. Although this appendix fixes some of the problems found in the main body of the User's Guide, some existing problems are not reviewed and new ones are introduced.

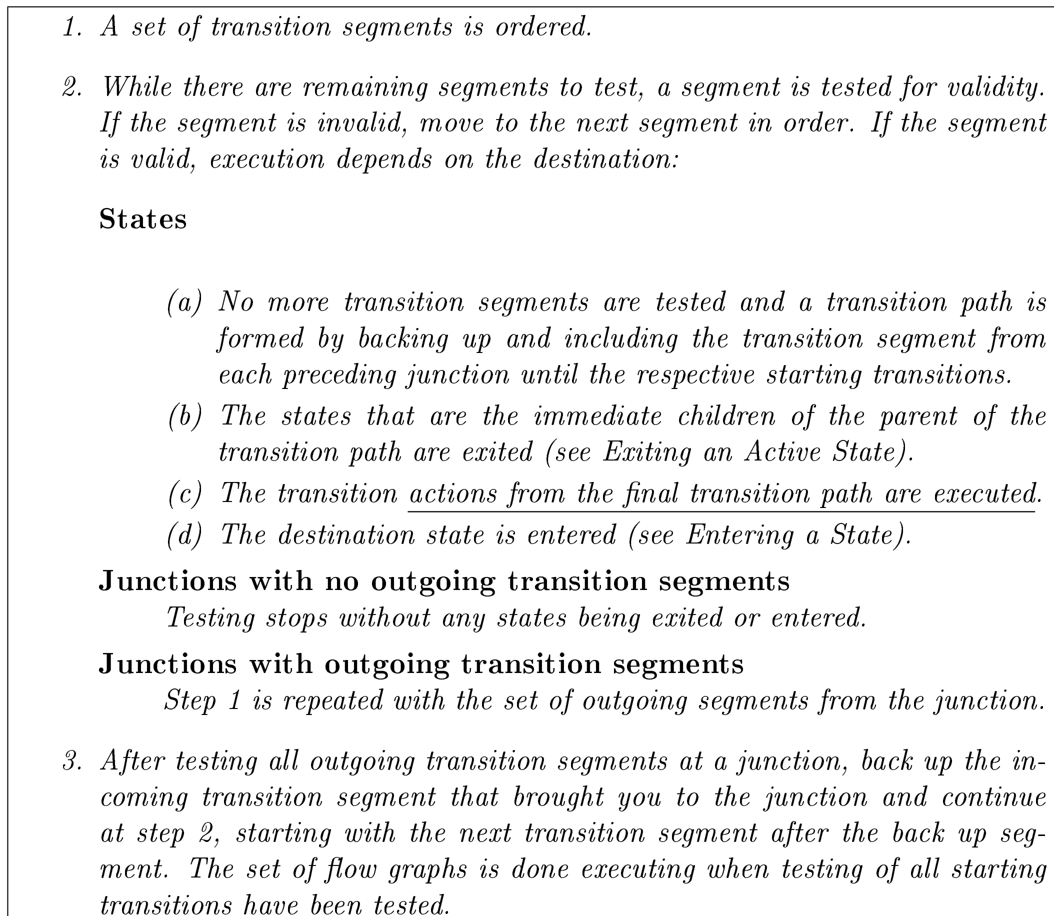
In this section, we present modified versions of the semantic rules that determine how states and transitions are to be interpreted. Except for the underlined parts, the wording in Figures 2.1, 2.2, 2.3 and 2.4 is mostly that of Chapter 3 and Appendix A of [98].

Executing transitions. Figure 2.1 presents the steps for the execution of a set of transitions. These steps specify that if a transition is invalid, the next one must be executed, otherwise it defines the execution of the transition according to the destination node. If the destination is a state, the necessary states are exited, the transition path is executed and the destination state is entered. If the destination is a junction, the behaviour depends on the outgoing transitions of the junctions. If there are no outgoing transitions, the execution of the transitions stops, otherwise the outgoing transitions of the junction are executed.

It is worth emphasising that when executing a transition path, if a state has been successfully reached, the source state must be exited. However, if the transition path crosses boundaries, that is, it contains interlevel transitions, it may be necessary to exit other states in addition to the source. The exact states that must be exited are the ancestors of the source state up to (and including) the one at the same level of the destination state.

The treatment of interlevel transitions is described in Figure 2.1 by step 2.b. This step requires that the substates of the parent of the transition path are exited. The parent of the transition path is the state that is an ancestor of the source and destination states, and that has no substate that is also an ancestor of both states. Since this state is necessarily sequential, at most on substate is active, therefore existing all substates only exits the active one. In this case, the active one is an ancestor of the source of the transition path.

As previously mentioned, the description of the semantics of Stateflow in Chapter 3 and Appendix A of [98] are inconsistent with each other, and with the behaviour observed in



1. *A set of transition segments is ordered.*
2. *While there are remaining segments to test, a segment is tested for validity. If the segment is invalid, move to the next segment in order. If the segment is valid, execution depends on the destination:*

   States

   (a) *No more transition segments are tested and a transition path is formed by backing up and including the transition segment from each preceding junction until the respective starting transitions.*
   (b) *The states that are the immediate children of the parent of the transition path are exited (see Exiting an Active State).*
   (c) *The transition actions from the final transition path are executed.*
   (d) *The destination state is entered (see Entering a State).*

   Junctions with no outgoing transition segments
   Testing stops without any states being exited or entered.

   Junctions with outgoing transition segments
   Step 1 is repeated with the set of outgoing segments from the junction.
3. *After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the back up segment. The set of flow graphs is done executing when testing of all starting transitions have been tested.*

Figure 2.1: Executing a Set of Transitions [98].

the Stateflow tool. For instance, the Stateflow User's Guide requires that after the source state (and any relevant ancestors) is exited, only *"the transition action of the final transition segment of the full transition path is executed"* [98]. This, however, is not observed in the simulation tool; the simulation of all examples we tested shows that all the transition actions in the transition path are executed. The property *ignoreUnsafeTransitionActions* cited in [68] can still be set to 0 or 1, but there is no change in the execution of transition actions. Furthermore, we were unable to find any reference to this property in the documentation supplied by *Mathworks*.

Entering a state. Figure 2.2 presents a modified version of the steps for entering a state. It merges the two descriptions in the Stateflow User's Guide, adopts a consistent terminology, and defines the correct steps that are recursively executed. Before a state is entered, some conditions must hold: the parent of the state and the left sibling of the state (if the state is parallel) must be active. Once these conditions are established, the entry action is executed and the children (if any) are entered. After a state is entered, if it is parallel, its right sibling must be entered.

The range of entry steps that are executed in certain situations is a common problem.

1. *If the parent of the state is not active, perform steps 1-4 for the parent.*
2. *If this is a parallel state, check the immediate sibling with a higher (i.e., earlier) entry order is active. If not, perform entry steps 1-5 for this state first.*
3. *Mark the state active.*
4. *Perform any entry actions.*
5. *Enter children, if needed:*
   (a) *If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialisation, perform entry steps 1-5 for that child. Otherwise, execute the default flow paths for the state.*
   (b) *If this state has parallel decomposition, i.e., has children that are parallel states, perform entry steps 1-5 for each state according to its entry order.*
6. *If this is a parallel state, perform all entry steps for the sibling state next in entry order if one exists.*
7. *Else, if the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.*

Figure 2.2: Entering a State [98].

For instance, when entering a parallel state, the User's Guide states *"check that all siblings with a higher (i.e., earlier) entry order are active. If not, perform all entry steps for these states first"* [98]. In fact, only the immediate left sibling is checked and entered if necessary. Moreover, not all entry steps are executed, only the steps from 1 to 5 are executed for that state. This fact has been confirmed independently in [20].

Both descriptions in the User's Guide require that if the history junction in a state points to one particular substate, the entry action of that state is executed. This would imply that the substates of that child state are not entered because the entry steps are not executed on it. This, however, is not the observed behaviour of the simulator. In this case, we observe that, in fact, the entry *steps* from 1 to 5 are executed on that child. Step 6 can be ignored because the state is sequential, and step 7 is not executed because the immediate parent of the child is already active, as it triggered the entering of this state.

This type of inconsistency can also be found in the description of the process of exiting a state in the main body of the User's Guide; however, it was corrected in the appendix.

Our experiments suggest that step 7 is only executed if the condition of step 6 fails, that is, if the state is not a parallel state, or if it does not have a right sibling. Since step 6 requires the execution of all entry steps to the right sibling, the step 7 is accumulated for each parallel state entered, and is therefore executed multiple times. It is our understanding that step 7 should only be executed once in a sequence of parallel states. If we require that step 7 is only executed when step 6 fails, it should be executed exactly when the state is

sequential, or when (it is parallel and) the last sibling has been entered.

Executing an active state. The steps for executing a state are described in Figure 2.3. These steps specify that the outer transitions of the state must be executed first. If the state is not exited, the during action is executed, followed by the inner transitions. Finally, if the inner transitions do not lead to a state transition, the active children are executed.

1. *The set of outer flow graphs execute (see Executing a Set of Flow Graphs). If this action causes a state transition, execution stops. (Note that this step never occurs for parallel states.)*
2. *During actions and valid on-event actions are performed.*
3. *The set of inner flow graphs execute. If this action does not cause a state transition, the active children execute, starting at step 1. Parallel states execute in the same order that they become active.*

Figure 2.3: Executing an active State [98].

Exiting an active state. There are four steps for exiting a state (Figure 2.4). Before a state is exited, its active right siblings (if the state is parallel) and active children must be exited. Next, the exit actions of the state are executed. The process concludes by marking the state as inactive.

1. *If this is a parallel state, check that the immediate sibling that became active after this state have already become inactive. Otherwise, perform all exiting steps on that sibling state.*
2. *If there are any active children, perform the exit steps on these states in the reverse order that they became active.*
3. *Perform any exit actions.*
4. *Mark the state as inactive.*

Figure 2.4: Exiting an active State [98].

This section has presented the semantic rules for executing transitions as well as for entering, executing and exiting a state in Stateflow. While the rules are largely similar to those published by Mathworks [98], we have commented on a number of inconsistencies and omissions found in the User's Guide and corrected those in our presentation above. Additionally, it is also worth pointing out that the Guide does not clearly define the difference between steps and actions. For example, the term "entry action" is used in the text ambiguously to denote both the steps of entering the state as well as the action that must be executed when a state is entered. Taken together, the various inconsistencies, omissions and errors of the Mathworks official documentation pose significant challenges for application of formal modelling and analysis techniques in the context of Stateflow

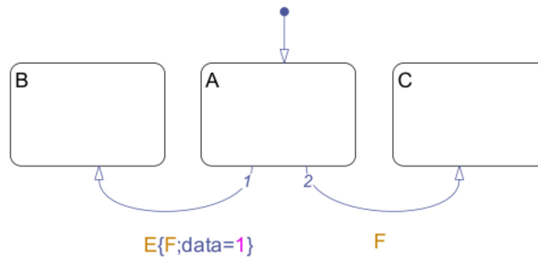


Figure 2.5: Early return logic example.

diagram; their detection and resolution can therefore be seen as one of the contributions of this thesis.

2.1.3 Early return logic

Early return logic occurs when a recursive execution (triggered by a local event broadcast) activates (or deactivates) states that should not be active (or, respectively, inactive) after the event broadcast. It interrupts part of the execution of the chart to avoid reaching an inconsistent state (e.g. a state with sequential decomposition and two active substates).

It is worth mentioning that early return logic does not necessarily interrupt the execution of the whole chart. For example, if the chart has two parallel states and the execution of the substates of the first parallel state is interrupted (in a consistent state) by early return logic, the execution of the second parallel state may continue, as the inconsistency was avoided.

Local event broadcasts may occur in entry, during, exit, condition, transition and *on* actions. Bind actions do not lead to local event broadcasts. The User's Guide [98] specifies the early return conditions for each type of action (on actions are considered as during actions).

For local event broadcast originating from the execution of the entry actions of state s, if s is inactive after the broadcast, the process of entering state s is interrupted. The instructions for during and exit actions are similar, but the processes of executing and existing, respectively, are interrupted. For condition actions, if the source of the transition path is inactive after the local event broadcast, the execution of the transitions is interrupted.

The case of local event broadcasts from transition actions is slightly different. For all the previous cases, some state that is active before the broadcast, must be active after it. Since transition actions are executed after the source state (and any necessary ancestor state) is exited, and before the destination state is entered, all substates of the parent of the transition path must be inactive before and after the broadcast. If any of them is active after the broadcast, the execution of the transitions is interrupted.

By way of illustration, we consider the simple chart in Figure 2.5 adapted from an

example in [98]. This example shows a situation where an event (E) triggers the execution of a transition, which raises a different event (F). This event then triggers a different transition, and since the first one was not completed before the event F was raised, it is abandoned and the simulation proceeds with the second transition. The first time this chart receives an input event E, the state A is entered. When the second event E is received, the first outer transition from A is attempted. Since its trigger is true, the transition is valid, and consequently the condition action is executed. This broadcasts the local event F, which triggers the reexecution of the chart under F. The reexecution considers the outer transitions of the active state, A. The first outer transition is not possible because the trigger does not contain F, but the second transition is possible. It is taken, A is exited, and the state C is entered. The reexecution finishes, but the execution of the chart cannot proceed because the transition from A to B can no longer be executed, since A is not active anymore. In this case, the original execution is interrupted, the assignment `data=1` is not executed, and the step of execution finishes.

The User's Guide [98] is not clear about exactly which portions of the execution must be interrupted. For example, in the case of entry actions, it simply says that *"any remaining steps in entering a state are not performed"*. We have verified using the simulation tool that when a local event is broadcast from the entry action of a state s, in certain cases only the remaining steps for entering s are interrupted, while in other cases the steps for entering some of the parents are also interrupted.

For example, Step 1 in Figure 2.2 activates the parent state. It is expected therefore that Step 2 can only continue if the parent is active. Assume that the parent p has a sequential decomposition, if its entry action exits p (for instance, by executing an outer transition), Step 3 would mark the substate active, and we would end up with an active state whose parent is inactive, which is an inconsistency. We believe that early return conditions should be checked not only after local event broadcasts, but after each step with respect to the appropriate state (the parent state in the case of Step 1). This is the approach we took to model the semantics of Stateflow charts as presented in Chapter 3 of this thesis.

2.2 Formal models of state diagram notations

In order to achieve the objective of this thesis, we need some formal account of the notation. In the literature on verification and analysis of state diagrams, we can find different approaches that can be divided into two types according to how the formalisation is carried out: formal semantics and translation strategies to a formal notation. Another distinction can be made with respect to the objectives of this formalisation: verification (of properties and implementations) and automatic code generation.

Works that establish the formal semantics of a notation allow for the analysis and verification of diagrams and can be used to define simulation and compilation procedures. Those that define a translation strategy can achieve the same results, but are able to re-use

existing technologies for the formal notation to which the state diagrams are translated.

In what follows, we discuss some of the formal approaches to state diagram notations. We divide them into two groups according to the objectives: verification (Section 2.2.1) and automatic code generation (Section 2.2.2). Within these groups, we will find works that can be classified according to the formalisation techniques.

2.2.1 Verification approaches

A formal semantics of statecharts was first introduced in [40] in an operational style using a notion of micro-steps to define a step of execution of the chart. Pnueli and Shalev [82] revise the operational semantics described in [40], and propose a declarative semantics consistent with the revised operational semantics. It relies on a notion of global consistency in order to show that both the declarative and operational semantics are consistent with each other. The execution step of statecharts is defined by the set of enabled transitions at a particular configuration, which are execute to calculate the next configuration. This semantics limits the number of times a transition may be executed in a step, and differs quite significantly from Stateflow in the treatment of event broadcasts.

In statecharts, events can be generated only by the transitions, and the transitions that are enabled by events generated by enabled transitions are restricted to be consistent with each other, therefore a transition cannot enable, for instance, another transition originating in the same source state. In Stateflow, local events can be broadcast both from transitions and states, thus the calculation of the set of enabled transitions would depend on the states being entered, executed and exited as well as the possible transitions. In addition, the transitions that can be triggered by a local event broadcast in Stateflow are not restricted as in statecharts. This generates the possibility of inconsistency, which is treated by early return logic. Since the consistency check is performed after the treatment of the local event broadcast, some actions (that would not occur otherwise) take place. For this reason, the approach in [82] cannot be applied directly to the semantics of Stateflow.

In [38], the semantics implemented in the STATEMATE system [39] is presented, but again in an informal fashion. Mikk et al. [60] give a formal account of the simulation of the semantics implemented in STATEMATE; it uses the Z notation to define the semantics. Given the syntactic and semantic differences of STATEMATE statecharts and Stateflow charts, it is not possible to use these results to verify implementations of Stateflow charts, and although Z has a refinement calculus, it is not clear how this could be used to verify parallel implementations of such charts. One possibility would be to specify the reactive behaviour of the chart in Z as described in [26].

The semantics of Stateflow charts is given in two forms: an informal description contained in the User's Guide [98] and a simulation semantics implemented in the Stateflow simulator. In [35], an operational semantics for Stateflow charts is proposed; however, it does not cover some features of the notation, e.g. history junctions, and also imposes restrictions on the use of local events and transitions. It is not clear, however, how to use

this semantics in the context of program verification, and how to integrate this semantics with a semantics of Simulink diagrams. Hamon [34] proposes a denotational semantics that overcomes some of the limitations of [35]. It claims to models the *"the full local event mechanism"* [34], but there is no discussion about the issue of inconsistent states arising from local event broadcasts, which is treated by early return logic.

The denotational and operational semantics proposed in [35] and [34] can be used to verify properties and implementations, and automatically generate code; however, because of the formalism in which the semantics is given, new theories and tools would need to be developed in order to support such goals.

Whalen [108] proposes a structural operational semantics for three dialects of statecharts (Stateflow, UML Statecharts and Rhapsody [37]) based on [34]. It lifts some of the restrictions previously imposed, and corrects some aspects of the semantics. However, history junctions are not covered, and the treatment of local event broadcasts is not clear. The formal rules for the treatment of local events seem to correctly model the semantics of local event broadcast, except that early return logic is not treated.

Chen [20] specifies the semantics of Stateflow in a version of CSP accepted by the model checker PAT [96]. Some of the problems with the informal semantics of Stateflow discussed in [68] were independently observed and corrected. The proposed treatment of interlevel transitions requires exiting "*the highest superstate (in terms of hierarchy) of the source state*", but this requirement is incomplete because the highest substate may be the parent of the source and destination states, and this state should not be exited. The User's Guide [98] description requires that the substates of the parent of the transition path are exited. Our model of Stateflow charts defines this state as the least upper bound of the source and destination states with respect to an ancestry relation. The treatment of input and local events is briefly mentioned in [20]. Multiple input events are treated using a notion of priority of transitions, when, in fact, Stateflow executes the chart for each input event that occurs in the same time step. Local event broadcasts are modelled, but early return logic is not mentioned. It is not clear how the models are obtained, and what is necessary to support automatic generation.

Boström and Morel [10] propose an approach to support the application of mode-automata [58] in an industrial setting, while maintaining its formal aspect. The approach identifies the subset of Simulink/Stateflow necessary to define mode-automata, and gives a formal semantics to this subset. This semantics is to be used as the basis for the verification of properties of the models, but the verification aspect is not developed further. The restrictions imposed on Simulink/Stateflow are so strong, that the proposed semantics cannot be used as a basis for the verification of implementations of Stateflow charts. Moreover, these restrictions yield a very simple semantics that cannot be easily extended to include the excluded features of the notation.

Sekerinski and Zurob [90] translate statecharts to the B notation, but impose some limitations on the structure of valid diagrams. The translation strategy is implemented in the *iState* tool that can also generate code in other languages. The choice of the B

notation as a target language is due to its support of non-determinism and suitability for safety analysis. Although the B notation supports verification of implementations, this aspect is not mentioned by the authors. Moreover, using this approach as a basis for our work would be rather difficult, given the syntactic and semantic differences between statecharts and Stateflow charts.

Latella et al. [53] propose an operational semantics of extended hierarchical automata and a translation strategy from UML state machines to this formalism. However, the subset of the language that is formalised is small. This work is significantly extended in [107]. The extensions include a treatment of the history mechanism, and entry and exit actions. Lilius and Paltor [55] translate UML state machines to PROMELA [44] and use the model checker SPIN [45] to analyse the diagrams. None of these works contemplate the verification of implementations.

Ng and Butler [75] define a translation from UML state machines to CSP specifications by translating UML states to CSP processes and UML events to CSP events. This work presents an elegant model of UML state machines, but data aspects are not covered. Furthermore, aspects of statecharts that make the semantics of Stateflow charts challenging (e.g. inter-level transitions) are not discussed. CSP is used primarily for its tool support for the verification of properties and refinement of UML state machines.

The approach presented in [84] extends that of Ng and Butler [75] by translating UML statecharts to *Circus* specifications, covering more aspects of the notation. A state is translated into a *Circus* action, and as in the previous work, a UML event is mapped to a *Circus* event. None of these works formalise aspects that render the semantics of Stateflow charts challenging, e.g. local event broadcast, connective and history junctions, etc. Therefore, formalisations of UML statecharts do not shed much light into the formalisation of Stateflow charts.

Snook et al. [95] propose a strategy for the verification of properties of UML models. The strategy is based on a translation from UML to UML-B [94] (a graphical notation similar to UML based on Event-B [2]) and applies the tools and techniques associated with UML-B and Event-B to verify both the internal consistency of the UML model and the target properties. This work differs from our approach both in the notation covered (UML) and in the objectives.

Banphawatthanarak and Krogh [6] propose a translation from Stateflow charts to the input notation of the SMV symbolic model checker, thus allowing for the verification of properties of the charts [7]; they impose restrictions on the types of input signals, number of transitions reaching a junctions, output signals and transition actions. Tiwari [101] translates Stateflow charts to a formalism called communicating push-down automata and uses the Symbolic Analysis Laboratory (SAL) framework to analyse the models; this work contemplates the main aspects of the Stateflow notation (parallel and sequential states, connective and history junctions, transitions, etc), but it is not clear what elements are not treated.

Scaife et al. [87] translate Stateflow models to the synchronous language Lustre [33],

allowing for the model checking of the models; the subset treated imposes some limitations on features such as inter-level transitions. This work extends previous work that defines a translation strategy from discrete-time Simulink to Lustre [12]. Chen [20], Banphawatthanarak et al. [7], Tiwari [101] and Scaife et al. [87] focus on verification of properties, and the latter approach could be used for automatic code generation; the verification of implementations of Stateflow charts is not supported by these works or by the notations used to formalise Stateflow charts.

In [104], Stateflow charts are formalised in the Z notation. Assumptions that record the requirements on the states of the chart are then combined with the chart using the Practical Formal Specification method [59] producing a set of healthiness conditions. These conditions are verified by the Simulink/Stateflow Analyser [103] in order to validate the Stateflow model. While we are interested in verifying that a program correctly implements a Stateflow chart, Toyn and Galloway [104] are interested on whether the chart is the intended model of the system. Moreover, the Simulink/Stateflow Analyser does not support some of the features of the Stateflow notation, such as parallel states and junctions.

Cavalcanti et al. [19] present the most similar work with respect to our objectives; it describes an approach for translating Simulink diagrams to *Circus* specifications. It uses extended versions of the ClawZ [4] and ClaSP tools developed by QinetiQ to translate control law diagrams to Z and CSP, respectively, and to generate a *Circus* specification. This allows the verification of functional and concurrent aspects in an integrated manner, as well as the verification of implementations of control law diagrams.

ClawZ comprises a library of block definitions and a translation strategy that maps a diagram into a Z schema that declares all the input and output signals, and constants; the predicate of the schema establishes the relationship between inputs and outputs. The translation strategy also includes in the specification the schemas corresponding to the blocks used in the diagram.

ClaSP does not produce CSP specifications of the control law diagrams, but rather generates a set of pairs that relate inputs and outputs. For each block A in the diagram, the pair (x, y), where x is the set of inputs of block A, and y is the sequence of outputs of block A, is included in this set.

ClawZ and ClaSP are extended to produce more information about the diagrams, so that it is possible to merge both specifications (from ClawZ and ClaSP) into one *Circus* specification [19]. For that purpose, ClawZ is extended to include action and enabled subsystems, as well as merge blocks (i.e, block that output the last input received), whereas ClaSP is modified so that it includes, for a particular diagram, its name, inputs, outputs and blocks; the blocks are characterised by their sequence of inputs, sequence of outputs and flows of execution.

A translation strategy is defined to merge these extended specifications. It defines the signals as channels and defines a channel called end_cycle. The strategy includes the ClawZ library and translates the diagram into a *Circus* process called $clasp.spec$ that consists of the parallel execution of all the blocks in the diagram. The parallel execution of two blocks

synchronises over the intersection of their alphabets, i.e. the set that contains a block's input and output channels, determining the order of execution of the blocks. The blocks also synchronise over the channel *end_cycle*, determining the end of the execution of each block and marking the end of the execution of a cycle of the diagram.

In [16], an initial *Circus* semantics of Stateflow charts is proposed using a denotational style, and, while many of the most interesting aspects of the semantics of Stateflow are discussed, they are not formalised. Also, the denotational style used in this work proved difficult to extend for the most complex aspects of Stateflow (e.g. early return logic), and hard to validate with respect to the informal semantics.

2.2.2 Code generation approaches

Caspi et al. [13] propose a code generation approach for obtaining embedded software implemented in a distributed architecture called Time-Triggered Architecture (TTA) [106]. This approach consists of using Simulink for designing control systems, SCADE/Lustre [24] for designing software and TTA as the distributed platform for the generated code. The three tools are widely used in avionics and automotive domains.

The decision to use SCADE/Lustre as an intermediate level between Simulink and TTA is due to a series of features of SCADE/Lustre. The latter has an automatic code generator qualified to SIL-A of the DO-178b standard and is suitable for the development of high integrity applications; it is supported by analysis tools (model checkers and test case generators) and presents some features that distinguish it as a programming language, rather then a simulation tool such as Simulink.

The approach consists, first, of translating a Simulink diagram to SCADE/Lustre [24], and then producing an implementation with the aid of the certified automatic code generator. This work extends [12] focusing on automatic code generation, but it does not cover the Stateflow notation. However, the fact that the work in [12] was extended to contemplate the Stateflow notation [87] suggests that this approach can also be extended.

Toom et al. [102] and Rugina et al. [86] report on work done in the context of the Gene-Auto ITEA European project. The goals of this project are to develop a code generator for Simulink/Stateflow and Scicos and to qualify the code generator through formal approaches. Izerrouken et al. [49] account for the validation and verification of the code generator. Since this work is based on automatic code generation, the only implementations that can be deemed correct are those generated by the tool. For this reason, the implementations cannot be modified. With an approach based on the verification of implementations, modified implementations can be individually assessed; thus allowing, for example, for manual code optimisation.

2.3 Specification languages

In order to verify implementations of Stateflow charts, we need to formalise them in a suitable notation. In this section, we survey specification languages that allow for this kind of verification and could be used for this formalisation.

Two aspects that can be used to distinguish specification languages are the capability of representing state on the one hand, and communication on the other. State based notations, such as Z [109], VDM [51] and B [1], are used mainly to specify sequential systems with complex data structures. In contrast, process algebras, such as CCS [61], CSP [85, 88, 41], ACP [9] and LOTOS [23], can specify reactive and concurrent systems.

Arthan et al. [4] and Adams and Clayton [3] use Z to verify implementations of Simulink diagrams. However, Z is not sufficient to describe all the aspects of such diagrams and programs. For that reason, CSP is used to describe the concurrent aspects. Given that Stateflow charts can have complex data types, for example arrays and matrices [98], and complex reactive behaviour, it is convenient to use a specification language that can tackle both of those aspects.

Fischer [28] and Smith [91] provide an integration of CSP and Object-Z [92], an object-oriented extension of the Z notation, by defining a failures-divergence semantics for Object-Z classes. Mahony and Dong [56] and Hoenicke and Olderog [43] also integrate CSP and Object-Z, but add a temporal aspect. Mahony and Dong [56] use Timed-CSP [89] for this, whereas Hoenicke and Olderog [43] use the duration calculus [112].

Butler and Leuschel [11] and Treharne and Schneider [105] use CSP and the B notation to provide an integrated notation. The former approach [11] identifies B operations and CSP channels. Treharne and Schneider [105] define a compatibility criteria between a B machine and a CSP specification that allows the CSP specification to direct the B machine execution.

Taguchi and Araki [97] integrate Z and CCS [61] by providing a state-based semantics of CCS, and then composing it with the semantics of Z in terms of labelled transition systems. It also introduces a logic for the specification of properties of systems described in this formalism.

Woodcock and Cavalcanti [110] define *Circus* as a combination of Z, CSP, the refinement calculus and Dijkstra's language of guarded commands. The semantics of *Circus* [79] is defined using the UTP [42], and its main advantage is the existence of a refinement calculus [78] that supports the verification of implementations of *Circus* specifications.

In the next section, we present a brief description of *Circus*. We chose *Circus* as the basis for our formalisation of Stateflow charts because of the existing work on formalisation of Simulink diagrams and on verification of implementations of such diagrams, as well as because *Circus* supports the specification of both the static and dynamic behaviour and provides a refinement calculus that supports the verification of implementations of specifications.

While other combinations of state base notations and process algebra (e.g, $CSP \parallel B$,

CSP-OZ, Event-B) have similar expressive power as *Circus* to the best of our knowledge, *Circus* is the only one that provides a refinement calculus that supports the verification of implementations in a calculational fashion. The latter aspects is particularly relevant to us because the potential automation allowed by the calculational style is fundamental for the successful adoption of our technique in industrial settings.

2.4 Refinement calculus

In this thesis, we are concerned with refinement based verification of implementations of Stateflow charts. In particular, we are interested in the approach known as the refinement calculus [5, 73, 74]. In the refinement calculus an abstract specification is transformed into a (possibly more concrete) specification by means of sound refinement laws.

In our approach, we favour the refinement calculus as it supports a high degree of automation. The automation derives from the fact that once a refinement law is selected, its applicability can, in general, be checked by simple provisos (mostly syntactic). This allows us to focus on strategies for the selection of refinement laws that support the verification of a specific implementation.

As previously mentioned, *Circus* combines the refinement calculus with Z, CSP and guarded commands. Besides supporting step-wise development (and verification), the refinement calculus in the context of *Circus* also support the reasoning about concurrency. In particular, it allows us to derive (and verify) a concurrent implementation from a centralised specification. Verification of implementations with respect to specifications is carried out by refining the specification into the implementation.

The soundness of the development (or verification) derives from the soundness of the refinement laws, which must be proved correct with respect to the semantics of *Circus* [79].

2.5 *Circus*

In this section, we present some of the *Circus* features using a simple *Circus* process (Figure 2.6) that models a systems of image transmitters and receivers as an example[1]. A detailed presentation of *Circus* can be found in Oliveira et al. [79].

A *Circus* specification is a sequence of paragraphs: Z paragraphs (axiomatic definitions, schemas, and so on), channel and channel set declarations, and process definitions (Appendix A presents the syntax of *Circus*). The first few paragraphs of our example (Figure 2.6) define the horizontal coordinates as the set $HORIZONTAL$ (of numbers from 1 to 800), the vertical coordinates as the set $VERTICAL$, the set of colours $COLOUR$, and the set of images $IMAGE$. An image is defined as a total function from the horizontal and vertical coordinates to colours. Next, an axiomatic definition declares the maximum number ($maxbuff$) of images that can be held in the transmitters and receivers as a constant.

[1] This example has been previously published in [70] and extends the example presented in [18]

$HORIZONTAL == 1..800$
$VERTICAL == 1..600$
$COLOUR == 0..255$
$IMAGE == HORIZONTAL \times VERTICAL \rightarrow COLOUR$

$| \quad maxbuff : \mathbb{N}_1$

channel $read, write : IMAGE$

process $Buffer \mathrel{\widehat=}$ **begin**
  state $S == [image : \operatorname{seq} IMAGE \mid \# image < maxbuff]$
  $InitState == [S' \mid image' = \langle\rangle]$
  $Read \mathrel{\widehat=} (\# image < maxbuff) \,\&\, read?x \longrightarrow image := image \frown \langle x \rangle$
  $Write \mathrel{\widehat=} (\# image > 0) \,\&\, write!(\operatorname{head} image) \longrightarrow image := tail\ image$
  $\bullet\ InitState\ ;\ (\mu X \bullet (Read \mathrel{\Box} Write)\ ;\ X)$
end

Figure 2.6: The *Buffer* process

A channel declaration introduces channel names and the types of the values that they communicate. A channel with no type does not communicate any values; it is used for synchronisation only. Our model declares two channels *read* and *write* of type *IMAGE*.

A basic process definition provides the name of a process, its state, local actions, and a main action. The state is defined by a schema expression. In Figure 2.6, we define a process *Buffer* whose state is given by the schema S. The state contains a single component: the sequence *image* of the elements stored in the buffer. The state invariant defines that the maximum size of the buffer is given by *maxbuff*.

A *Circus* action can freely combine schema expressions, CSP constructs, guarded commands, and specification statements. The buffer must be initialised before it is used; for that, we specify the action *InitState* as an operation schema that establishes that *image* is the empty sequence.

The *Buffer* can read new information only if there is space to store it. Thus the action *Read* has the condition $\# image < maxbuff$ as a guard. This requires the size of the image buffer to be smaller than *maxbuff*. If the guard is true, *Buffer* can receive a value through the *read* channel, and store it in *image* by concatenating it to the end of this sequence. Similarly, writing is only enabled if there is some value stored in the buffer. If the guard $\# image > 0$ is true, the action *Write* can send the first value of the sequence ($\operatorname{head} buffer$) through the channel *write*, and remove it from the *buffer* by redefining it as the rest of the sequence ($\operatorname{tail} buffer$).

The main action defines the behaviour of the process. In the case of *Buffer*, it initialises the state, and recursively offers the (external) choice ($\Box$) of *Read* or *Write*. A recursion is defined in the form $\mu X \bullet A(X)$ where $A(X)$ is an action that contains recursive calls X. The state of a process is local and encapsulated. Interaction with a process is only possible via communication through the channels that it uses.

Processes can also be defined through process operators. For example, we can define a new process by composing two other processes in parallel ($[\![\,]\!]$). Other process operators are interleave ($|||$), internal choice ($\sqcap$), external choice ($\Box$), and sequential composition ($;$).

Like in CSP, processes can be parametrised, have their components renamed, have their channels hidden, and so on. For instance, we can specialise the *Buffer* process as a transmitter that reads images from an *antenna* and sends them through a radio-frequency channel *rfchannel*. For that, we define a new process *Transmitter* by renaming the channels *read* and *write* of the process *Buffer* to reflect this change:

$$\mathbf{process}\ Transmitter \mathrel{\widehat=} Buffer[read, write := antenna, rfchannel]$$

The new channels *antenna* and *rfchannel* are implicitly declared by the renaming to have the same type as the corresponding channels *read* and *write*. We can define receivers in the same fashion:

$$\mathbf{process}\ Receiver1 \mathrel{\widehat=} Buffer[read, write := rfchannel, tv]$$
$$\mathbf{process}\ Receiver2 \mathrel{\widehat=} Buffer[read, write := rfchannel, vcr]$$

Both receivers run in parallel and share the reception through *rfchannel*. We specify this by composing them with synchronisation set $\{\!|\ rfchannel\ |\!\}$ to define a process called *Receivers* as shown below.

$$\mathbf{process}\ Receivers \mathrel{\widehat=} Receiver1 \ [\![\ \{\!|\ rfchannel\ |\!\}\ ]\!]\ Receiver2$$

Finally, the system is defined by composing the receivers and the transmitter in parallel. They communicate through *rfchannel*, which is hidden. Thus, interactions with the system use only *tv*, *vcr* and *antenna*.

$$\mathbf{process}\ System \mathrel{\widehat=} Receivers \ [\![\ \{\!|\ rfchannel\ |\!\}\ ]\!]\ Transmitter \setminus \{\!|\ rfchannel\ |\!\}$$

Actions can also be composed in parallel. In this case, not only the synchronisation channel set must be explicit, but also the sets of state components (and local variables in scope) to which each action writes. Interleaving does not require a synchronisation channel, but needs the sets of components that are modified. Actions can also be composed in sequence or in internal choice.

We present a much larger example of a *Circus* specification and the refinement calculus, as we describe our Stateflow models in Chapter 3, and our refinement strategy in Chapter 5.

Finally, *Circus* offers some tool support. A parser and a type checker have been developed and incorporated in the CZT toolkit [57]. An encoding of *Circus* in the theorem prover ProofPower is available [111], and a refinement tool called CRefine [80] are available. The latter, however, requires further development to fully support the application of refinement strategies such as the one proposed in this thesis. A translator from *Circus* to Java has been developed for an early version of *Circus* [31], but is currently not compatible with the version of *Circus* used in this thesis. A prototype model checker [32] was developed, but not made publicly available.

2.6 Final considerations

None of the works on formal modelling and verification of state diagram notations cover all desired features we discussed in Section 1.1. The work in [87] deals in an integrated way with models of Stateflow charts and Simulink diagrams, but does not cover verification of implementations because the code generated is supposedly correct by construction.

Most of the works on translation of Stateflow charts to formal notations focus on verification of properties, except, perhaps, [75] and [84], which translate UML state machines into CSP and *Circus*, languages that provide means for the verification of implementations. However, these works are concerned with UML state machines, which do not present the same complexities as Stateflow charts. Also, these works deal only with a well-behaved subset of UML state machines. For that reason, they do not give much insight into the formal treatment of more complicated aspects of statechart-like notations.

One approach that can be used in order to formalise Stateflow charts is the extension of the work presented in [19]. This is especially suitable because, in order to model Simulink function, we need a model of Simulink diagrams, and Stateflow charts are always part of a Simulink diagram.

Although most of the combined notations mentioned in section 2.3 present the features necessary to formalise Stateflow charts, since we are interested in the verification of implementations of such charts, we also need a notation that has a theory that allows us to verify implementations (for instance, a refinement theory). Smith and Derrick [93] discuss the refinement of specifications written in the combination proposed in [91], and Oliveira et al. [78] propose a refinement calculus for *Circus*. Both combinations have refinement theories, but *Circus* is the only one known to support the verification of executable code in a calculational style [17]. Moreover, *Circus* is unique in that it supports the specification of data-rich complex distributed systems, and the refinement calculus supports the definition of refinement strategies that can potentially reach a high degree of automation.

The semantics of Stateflow presented in this thesis stems from this initial work done in [16]. However, we took a completely different approach, favouring an operational style of specification, and a higher degree of connection with the informal semantics to facilitate the validation.

Chapter 3

A formal model of Stateflow Charts

In this chapter, we introduce an operational model of Stateflow charts described in *Circus*. The model is used as a basis for the verification of implementations of Stateflow charts. Since the established semantics of Stateflow is only available through simulation or in an informal description in the *Stateflow User's Guide* [98], it is not possible to prove that one particular model is correct (without access to the simulator's code). However, it is possible to develop a model close enough to the informal description of the behaviour of such charts, so that its validity can be argued on the basis of inspection with some degree of confidence.

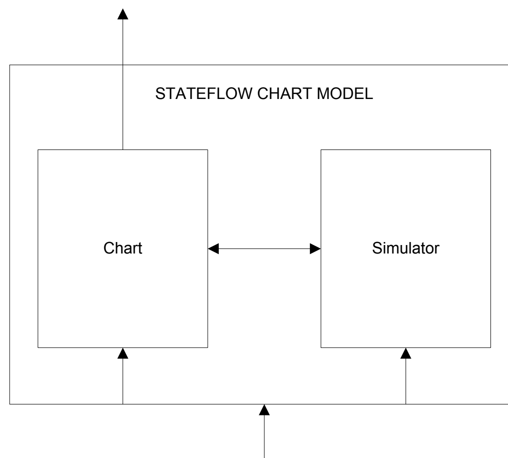


Figure 3.1: Basic architecture of Stateflow models.

Figure 3.1 gives an overview of the architecture of the formal semantics of Stateflow charts as encoded in our models. The model of a particular chart is partitioned into two components that capture separately the structure of the particular chart and the semantics of Stateflow. The two components interact with each other to carry the execution of the chart, and their composition (outer box) interacts with the environment by taking inputs and relaying them to the appropriate components (input events are directed to *Simulator*, and input data are relayed to *Chart*), and by communicating outputs. Outputs are

communicated by the component *Chart*.

The decomposition of the model into two components allows us to isolate the semantics of Stateflow from chart dependent aspects such as its structure and actions. Since the semantics is independent of a particular chart, only the component *Chart* must be generated to obtain a complete model. Moreover, changes to certain parts of the semantics (e.g., transition execution) can be contained within the component *Simulator*.

It is worth mentioning that this decomposition is not commonly used when formalising the semantics of a language with the goal of verification of implementations. Traditionally, a more denotational approach is favoured. However, a denotational approach embeds in the translation rules assumptions and simplifications that are not necessarily correct and must be verified. In our approach, no simplification is performed during the translation process, and the simplification is tackled by the verification strategy. Besides eliminating the (often unverified) simplifications from the translation process, our operational style yields simpler translation rules that can be more easily validated against the informal description of the semantics of Stateflow chart, which is given in an operational fashion.

One drawback of our choice of semantic style is that the generated models can be bulky and contain unnecessary complexity. Additionally, the verification of implementations is not as directs as in a more denotational setting. These hindrances are formally treated by our verification strategy which eliminates the unnecessary complexity and transforms the model into a format more amenable to verification. Our verification strategy is presented in Chapter 5.

In Section 3.1, we provide an overview of the model, its main components and how they interact. Section 3.2 presents the formal specification of the simulator process based on the informal description previously shown in Section 2.1.2. Section 3.3 explains the part of the model that is specific to a particular chart. Finally, Section 3.4 discusses the validation of our models, and Section 3.5 summarises and discusses our results.

3.1 Overview of the model

We describe and illustrate our models using the chart in Figure 3.2 which was adapted from a Stateflow model called "Modelling an Automatic Transmission Controller" [98].

This chart contains two parallel states `gear_state` and `selection_state`; each of them has a set of sequential substates. The state `gear_state` comprises sequential states `first`, `second`, `third` and `fourth`; the transitions between these states are controlled by local events (`UP` and `DOWN`) that are broadcast by the state `selection_state`. The choice of which event to broadcast, if any at all, is made according to the relation between the input variable `speed` and the local variables `up_th` and `down_th`. These local variables are updated by the Simulink function `calc_th` that takes variables `gear` and `speed` as parameters every time the state `selection_state` is executed. The chart has a single output variable `gear`.

Our models of Stateflow charts consist of two *Circus* processes in parallel: the simula-

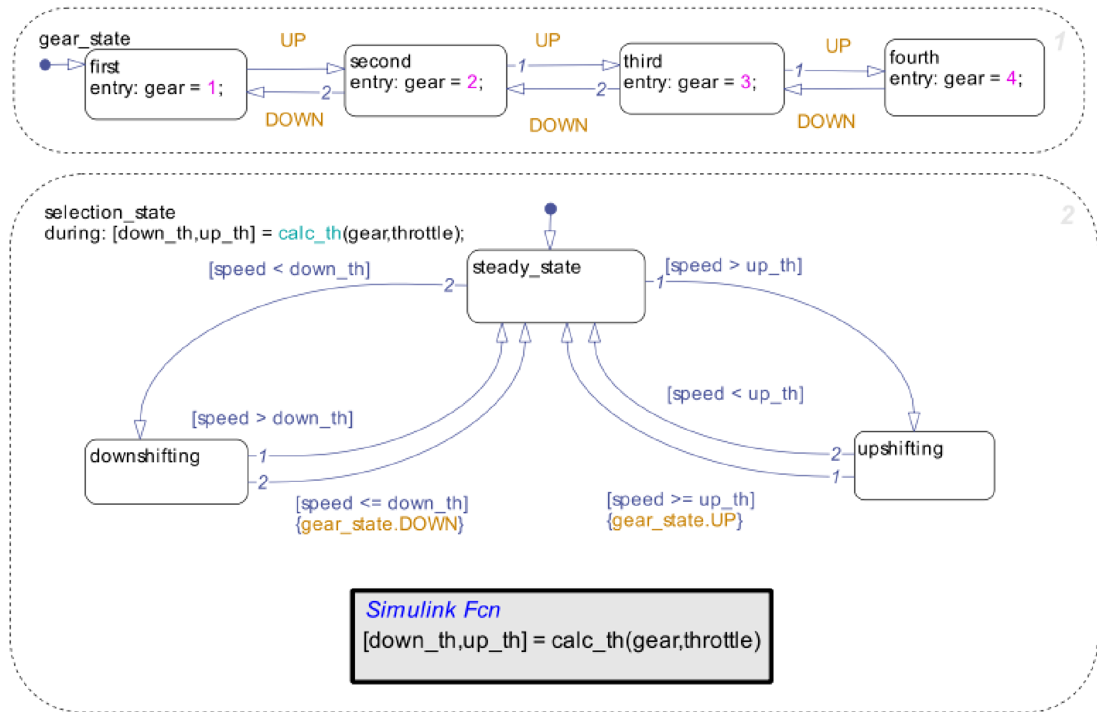


Figure 3.2: Example of a Stateflow Chart describing a car's shift logic.

tor process and the chart process. These processes are combined in parallel to model the execution of a particular chart. For example, the model of the chart shift logic shown in Figure 3.2 is given by the *Circus* process *Shift_logic* that combines the processes *c_shift_logic* and *Simulator* in parallel. They interact via a set *interface* of internal channels plus the channel *end_cycle*.

$$\textbf{process}\ Shift_logic \mathrel{\widehat=} (c_shift_logic \llbracket interface \cup \{\!| end_cycle |\!\} \rrbracket Simulator) \setminus interface$$

The process *Simulator* models the semantics of Stateflow charts independently of a particular chart, and the process *c_shift_logic* models the particular structure of the chart, including the actions defined in states and transitions. The channels in *interface* allow the process *Simulator* to obtain information from *c_shift_logic*, and request the execution of state and transition actions. These channels are hidden, and, thus, are local to the model. Figure 3.3 depicts the way in which the chart process is obtained, and the patterns of communication between the two processes and the environment.

The process *c_shift_logic* is generated from the concrete representation of the chart as provided by the MATLAB Simulink environment (that is, a *.mdl* file) by parsing it into an abstract syntax tree of the chart, and then translating it into a *Circus* model. The translation strategy that supports the generation of such models is detailed in Chapter 4 of this thesis.

The chart and simulator processes interact with each other and with the environment

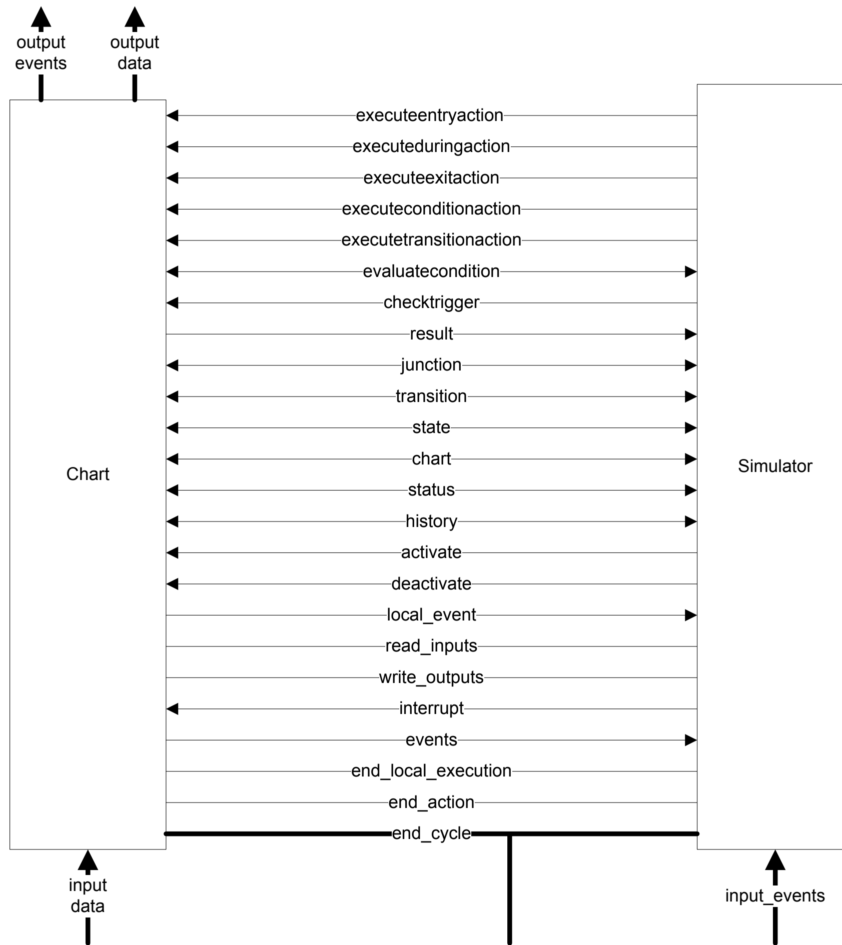


Figure 3.3: Overview of the model. Solid arrows indicate communication, solid lines indicate synchronisation, and bold lines indicate external interactions.

through a number of channels. The first group of channels is related to the execution of chart actions (*entry*, *during*, *exit*, *condition* and *transition* actions). The first five channels request the execution of a particular corresponding action; they communicate the identifier of the state or transition (as appropriate) associated with the request. Additionally, the channel *executeduringaction* communicates a value of type $EVENT$ because, in our models, a *during* action may include actions that are only executed when a particular event is being handled (*on* actions). The sixth channel of this group, *end_action*, is used to indicate the end of a chart action. These six channels are defined as follows.

$$\begin{array}{l}
\textbf{channel}\, \mathit{executeentryaction}, \mathit{executeexitaction} : \mathit{SID} \\
\textbf{channel}\, \mathit{executeduringaction} : \mathit{SID} \times \mathit{EVENT} \\
\textbf{channel}\, \mathit{executeconditionaction}, \mathit{executetransitionaction} : \mathit{TID} \\
\textbf{channel}\, \mathit{end_action}
\end{array}$$

The second group consists of channels that are used to check whether a transition is valid

or not. The channel *evaluatecondition*, used to evaluate the conditions of a transitions, communicates a transition identifier and a boolean value. The channels *checktrigger* and *result* are used to check whether a transition has been triggered. The former communicates a transition identifier and an event, whereas the latter communicates the same information plus a boolean value.

$$\textbf{channel}\, evaluatecondition : TID \times \mathbb{B}$$
$$\textbf{channel}\, checktrigger : TID \times EVENT$$
$$\textbf{channel}\, result : TID \times EVENT \times \mathbb{B}$$

The third group contains channels used to recover instances of elements contained in the chart from their identifiers.

$$\textbf{channel}\, junction : JID \times Junction$$
$$\textbf{channel}\, transition : TID \times Transition$$
$$\textbf{channel}\, state : SID \times State$$
$$\textbf{channel}\, chart : State$$

In general, these channels communicate an identifier (of one of the types: state, junction or transition) and an element of the same type. For example, the communication $state!sid?s$ is used to recover the state whose identifier is *sid*. The channel *chart* simply communicates the binding that represents the chart; it does not require an identifier because the identifier of a chart is unique.

The channels in the fourth group are used to request information about the status and history of particular states, and to request the activation and deactivation of states. They all communicate a state identifier. The *status* channel also communicates a boolean value that represents whether the state is active or not. Finally, the *history* channel communicates the additional state identifier of the last activated substate.

$$\textbf{channel}\, status : SID \times \mathbb{B}$$
$$\textbf{channel}\, history : SID \times SID$$
$$\textbf{channel}\, activate, deactivate : SID$$

The fifth group consists of channels related to local event broadcasts. The channel *local_event* communicates the event being broadcast along with the state that is the target of the broadcast. The local event broadcast triggers a local execution (of the chart or of a state) whose end is marked by a synchronisation on the channel *end_local_execution*. Finally, the channel *interrupt* communicates a boolean value, and indicates whether or not an early return logic condition has arisen; it controls the execution of the rest of a chart action after

a local event broadcast.

$$\mathbf{channel}\ local_event : EVENT \times State$$
$$\mathbf{channel}\ end_local_execution$$
$$\mathbf{channel}\ interrupt : \mathbb{B}$$

The sixth group comprises channels used to communicate data and events. The channel *events* communicates the sequence of input events declared by a chart. The channel *input_event* communicate a sequence of boolean values that identify which input events have occurred in a step of execution. The channels *read_inputs* and *write_outputs* are used to request the chart process to, respectively, read the inputs and write the outputs.

$$\mathbf{channel}\ events : \mathrm{seq}\,\mathbb{B}$$
$$\mathbf{channel}\ input_event : \mathrm{seq}\,EVENT$$
$$\mathbf{channel}\ read_inputs, write_outputs$$

Finally, the channel *end_cycle* indicates the end of a cycle of execution of the chart.

$$\mathbf{channel}\ end_cycle$$

We identify below the set of channels that are used exclusively between the chart and simulator processes, and are hidden from the environment; we call this set *interface*.

$$\mathbf{channelset}\ interface == \{\!|\ executeentryaction, executeexitaction, executeduringaction, executeconditionaction, executetransitionaction, end_action, evaluatecondition, checktrigger, result, junction, transition, state, chart, status, history, activate, deactivate, local_event, end_local_execution, interrupt, events, read_inputs, write_outputs\ |\!\}$$

The external channels of our Stateflow models include one channel for each input and output data, one channel for each output event, *input_event* and *end_cycle*. In our example in Figure 3.2, we have *o_gear*, used for output, and *i_speed* and *i_throttle*, used for input. There are no output events in our example.

The interaction of models of well formed diagrams and the simulator are deadlock-free and divergence-free. Deadlock occurs when a state inconsistency arises, and divergence occurs when a transition loop or local event broadcast lead an infinite loop. We assume that the chart under consideration has been analysed using the Stateflow tool, thus revealing issues in the chart that may lead to such situations. Nevertheless, by explicitly addressing these situations, our models potentially support some reasoning about state inconsistency and infinite loops in Stateflow charts.

$$
\begin{array}{l}
\textbf{channel}\ stsuccess, stfail \\
\textbf{channelset}\ statetransition == \{\!|\ stsuccess, stfail\ |\!\} \\
\textbf{process}\ Simulator \mathrel{\widehat=} \textbf{begin} \\
\quad entryActionCheck \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{res}\ b : \mathbb{B} \bullet \ldots \\
\quad \ldots \\
\quad enterState15Check \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{res}\ b : \mathbb{B} \bullet \ldots \\
\quad ExecuteTransition \mathrel{\widehat=} tid : TID;\ path : \mathrm{seq}\ TID;\ source : State;\ ce : EVENT \bullet \ldots \\
\quad CheckValidity \mathrel{\widehat=} tid : TID;\ path : \mathrm{seq}\ TID;\ source : State;\ ce : EVENT \bullet \ldots \\
\quad Proceed \mathrel{\widehat=} tid : TID;\ path : \mathrm{seq}\ TID;\ source : State;\ ce : EVENT \bullet \ldots \\
\quad proceedToState \mathrel{\widehat=} src, dest : State;\ path : \mathrm{seq}\ TID;\ ce : EVENT \bullet \ldots \\
\quad executePath \mathrel{\widehat=} path : \mathrm{seq}\ TID;\ src, dest : State;\ ce : EVENT \bullet \ldots \\
\quad proceedToJunction \mathrel{\widehat=} tid : TID;\ path : \mathrm{seq}\ TID;\ source : State;\ ce : EVENT \bullet \ldots \\
\quad executeJunction \mathrel{\widehat=} j : Junction;\ path : \mathrm{seq}\ TID;\ source : State;\ ce : EVENT \bullet \ldots \\
\quad ExecuteDefaultTransition \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \ldots \\
\quad EnterState \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \ldots \\
\quad EnterState1 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \ldots \\
\quad \ldots \\
\quad EnterState7 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \ldots \\
\quad ExecuteState \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \ldots \\
\quad AlternativeExecution \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \ldots \\
\quad \ldots \\
\quad ExecuteSequentialStates \mathrel{\widehat=} ss : \mathrm{seq}\ SID;\ ce : EVENT \bullet \ldots \\
\quad ExitState \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \ldots \\
\quad ExitStates \mathrel{\widehat=} ss : \mathrm{seq}\ SID;\ ce : EVENT \bullet \ldots \\
\quad ExecuteChart \mathrel{\widehat=} ce : EVENT \bullet \ldots \\
\quad EnterChart \mathrel{\widehat=} c : State;\ ce : EVENT \bullet \ldots \\
\quad ExecuteActiveChart \mathrel{\widehat=} c : State;\ ce : EVENT \bullet \ldots \\
\quad LocalEventEntry \mathrel{\widehat=} sid : SID \bullet \ldots \\
\quad \ldots \\
\quad LocalEventTransition \mathrel{\widehat=} sid : SID \bullet \ldots \\
\quad TreatLocalEvent \mathrel{\widehat=} e : EVENT;\ s : State \bullet \ldots \\
\quad ExecuteEvent \mathrel{\widehat=} e : EVENT;\ v : \mathbb{B} \bullet \\
\quad ExecuteEvents \mathrel{\widehat=} es : \mathrm{seq}\ EVENT;\ vs : \mathrm{seq}\ \mathbb{B} \bullet \\
\quad Step \mathrel{\widehat=} \ldots \\
\quad \bullet\ \mu X \bullet Step\ ;\ end_cycle \longrightarrow X \\
\textbf{end}
\end{array}
$$

Figure 3.4: Structure of the *Simulator* process.

3.2 Process *Simulator*

The process *Simulator* provides the main communication interface with a Simulink model; it accepts a communication that allow input events to trigger the execution of the chart, and the communication of the end of the chart's cycle of execution. An overview of the structure of the process *Simulator* is shown in Figure 3.4. It declares 57 actions that are used to build the process' main action.

3.2.1 Step of execution

The main action of *Simulator* recursively executes the *Circus* action *Step* and synchronises on the channel *end_cycle*. The action *Step* requests, using the channel *events*, the sequence *es* of input events that the chart accepts. The order of the events is important because the chart is executed once for each active event in the order the events are defined in the chart. Next, it reads through the channel *input_event* a sequence *vs* of boolean values (of the same size as *es*) that indicate which input events have occurred. It then requests (using the channel *read_inputs*) the chart process to read the input data, executes the chart for the input events (*es*) and their associated values (*vs*) using the action *ExecuteEvents*, and finally requests (using the channel *write_outputs*) the chart to communicate the output data and events.

$$Step \mathrel{\widehat=} \left(\begin{array}{l} events?es \longrightarrow input_event?vs : (\# vs = \# es) \longrightarrow read_inputs \longrightarrow \\ ExecuteEvents(es, vs) \; ; \; write_outputs \longrightarrow \mathbf{Skip} \end{array} \right)$$

The particular structure of the step of execution of our model supports the future integration of the models of Stateflow with the models of Simulink diagrams. The pattern on reading input events and data, executing the block and writing the outputs is particularly important, along with the signalling of the end of the cycle through the channel $end_c ycle$.

The execution of the chart for the input events is specified by the action *ExecuteEvents*. It takes a sequence of events and a sequence of boolean values as parameters and executes the chart for each event and the associated value.

$$ExecuteEvents \mathrel{\widehat=} es : \operatorname{seq} EVENT; \; vs : \operatorname{seq} \mathbb{B} \bullet (\mathbin{\raisebox{1ex}{\scalebox{.7}{$\fatsemi$}}} \; i : id(\operatorname{dom} \; es) \bullet ExecuteEvent(es(i), vs(i)))$$

Note that this action uses the iterated sequential operator ($\fatsemi$) to traverse the list of indices of the sequence *es*, and, for each such index i, calls *ExecuteEvent* with parameters $es(i)$ and $vs(i)$. The set of events *EVENT* used above is defined as a given set; for each chart, the values of this set are explicitly declared to contain the input and local events declared in the chart. The sequence of indices of *es* is defined as the identity function of its domain.

The action *ExecuteEvent* takes an event and a boolean value, and executes the chart for that event if the boolean value is true. Otherwise, it does nothing.

$$ExecuteEvent \mathrel{\widehat=} e : EVENT; \; v : \mathbb{B} \bullet \left(\begin{array}{l} \mathbf{if} \; v = \mathbf{True} \longrightarrow ExecuteChart(e) \\ [\!] \;\; v = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right)$$

As captured in the definition of *Step* above, the execution of a chart is driven by a Simulink model that communicates a set of events through *input_event*. It is, however, possible to define charts that are not driven by input events. This is accommodated in our model by defining a null event (*ENULL*) as the input event in the chart, and allowing the Simulink model to trigger the execution of the chart by communicating $\langle \mathbf{True} \rangle$.

The execution of a chart depends on whether it is active or inactive; it is modelled by the action *ExecuteChart*.

$$ExecuteChart \mathrel{\widehat=} ce : EVENT \bullet chart?c \longrightarrow status!(c.identifier)?active \longrightarrow \left(\begin{array}{l} \mathbf{if}\ active = \mathbf{True} \longrightarrow ExecuteActiveChart(c, ce) \\ [\!]\ active = \mathbf{False} \longrightarrow ExecuteInactiveChart(c, ce) \\ \mathbf{fi} \end{array}\right)$$

This action receives an event as a parameter, and, firstly, requests the chart c through channel *chart*. It then requests the status of the chart by communicating the identifier of the chart ($c.identifier$) through channel *status* and receiving the value in *active*. Finally, if the chart is active ($active = \mathbf{True}$), *ExecuteChart* calls the action *ExecuteActiveChart*; otherwise, it calls the action *ExecuteInactiveChart*.

$$ExecuteActiveChart \mathrel{\widehat=} c : State;\ ce : EVENT \bullet \left(\begin{array}{l} \mathbf{if}\ c.substates = \varnothing \longrightarrow ExecuteInactiveChart(c, ce) \\ [\!]\ c.substates \neq \varnothing \longrightarrow ExecuteSubstates(c, ce) \\ \mathbf{fi} \end{array}\right)$$

If the chart has substates, *ExecuteActiveChart* calls the action *ExecuteSubstates* to execute them. Otherwise, it executes the chart as if it were inactive. *ExecuteActiveChart* receives a parameter whose type is the schema *State*.

State

$identifier : SID$
$default, inner, outer : TID$
$parent, left, right : SID$
$substates : \operatorname{seq} SID$
$decomposition : DECOMPOSITION$
$type : TYPE;\ history : \mathbb{B}$

This schema records the identifier of the state, the identifiers of its first (if any) default, inner and outer transitions, the identifier of its parent state (or chart), left and right siblings, the sequence of the identifiers of its substates, its decomposition type, its type, and whether or not it has a history junction. Identifiers for states and transitions are drawn from the sets *SID* and *TID* (respectively), which together with the set *JID* of junction identifiers, are defined as disjoint given sets. The sets *DECOMPOSITION* and *TYPE* are defined by free types. In Stateflow, substates in both parallel and sequential decompositions are ordered. In the case of parallel decomposition, this order determines the order in which the states are entered, executed and exited. In the case of sequential decomposition, the order establishes the order in which the states are queried for their

statuses.

$TYPE ::= AND \mid OR \mid CHART$
$DECOMPOSITION ::= SET \mid CLUSTER$

A state must be of type *AND* or *OR*, and have decomposition of type *SET* or *CLUSTER*. Only a chart (represented as a state in our models) has type *CHART*.

$$
\begin{array}{l}
ExecuteInactiveChart \mathrel{\widehat=} c : State;\ ce : EVENT \bullet activate!(c.identifier) \longrightarrow \\
\left(\begin{array}{l}
\textbf{if}\ c.default \neq nulltransition.identifier \lor c.decomposition = CLUSTER \longrightarrow \\
\quad ExecuteDefaultTransition(c, c, ce) \\
[\!]\ c.default = nulltransition.identifier \land c.decomposition = SET \longrightarrow \\
\left(\begin{array}{l}
\textbf{if}\ c.substates = \langle\rangle \longrightarrow \textbf{Skip} \\
[\!]\ c.substates \neq \langle\rangle \longrightarrow state!(head(c.substates))?first \longrightarrow EnterState(first, c, ce) \\
\textbf{fi}
\end{array}\right) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

The execution of an inactive chart c triggers the activation of the state denoted by $c.identifier$, which represents the chart as a whole. *ExecuteInactiveChart* uses the channel *activate* to request that the chart process carry out this activation. Afterwards, if the chart has a default transition ($c.default \neq nulltransition.identifier$) or if it has a sequential decomposition ($c.decomposition = CLUSTER$), *ExecuteInactiveChart* executes the default transition using *ExecuteDefaultTransition*. If the chart has a parallel decomposition and no default transitions, *ExecuteInactiveChart* checks if the chart has any substates. If it does, *ExecuteInactiveChart* recovers the binding of the state whose identifier is the first substate using channel *state*, and enters it using action *EnterState*.

$$
\begin{array}{l}
ExecuteDefaultTransition \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \\
\quad \left(\begin{array}{l}
\textbf{if}\ s.default \neq nulltransition.identifier \longrightarrow \\
\quad \left(\begin{array}{l}
\textbf{var}\ success : \mathbb{B} \bullet ExecuteTransition(s.default, \langle\rangle, s, ce, success); \\
(\textbf{if}\ success = \textbf{True} \longrightarrow \textbf{Skip}\ [\!]\ success = \textbf{False} \longrightarrow \textbf{Stop}\ \textbf{fi})
\end{array}\right) \\
[\!]\ s.default = nulltransition.identifier \longrightarrow \\
\quad \left(\begin{array}{l}
\textbf{if}\ \# s.substates = 0 \longrightarrow \textbf{Skip} \\
[\!]\ \# s.substates = 1 \longrightarrow \left(\begin{array}{l} state!(head\ s.substates)?saux \longrightarrow \\ EnterState(saux, tpp, ce) \end{array}\right) \\
[\!]\ \# s.substates > 1 \longrightarrow \textbf{Stop} \\
\textbf{fi}
\end{array}\right) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

The action *ExecuteDefaultTransition* receives two states (s and tpp) and an event (ce) as parameters. The parameter s corresponds to the state whose default transitions are to be executed, tpp is the parent of the transition path being executed, and ce is the current event. The parent of the transition path is the state lowest in the hierarchy that contains

both ends of a transition path. This information is necessary for the proper treatment of interlevel transitions (discussed in Section 3.2.2).

If the state has a default transition, we declare a local variable *success* of type boolean and call action *ExecuteTransition* on the default transitions passing the variable *success* as a value-result parameter. This variable is used to monitor the success or failure of the execution of the default transitions. If the execution is successful, *ExecuteDefaultTransition* terminates. Otherwise, it deadlocks (**Stop**), indicating a chart error. This error occurs because default transitions cannot fail to lead to a state being entered. Chart errors that lead to deadlock are detected by the Stateflow tool, and are modelled in our semantics for completeness.

If the state does not have a default transition, *ExecuteDefaultTransition* must identify the unique state that can be entered (if any). This is only possible if the state has exactly one substate. If there are no substates, the action terminates successfully. If there is more than one substate, the action deadlocks. As previously mentioned, the model of a well formed Stateflow chart should never lead to a deadlock.

3.2.2 Transition

The execution of a transition is one of the most complicated aspects of the semantics of Stateflow charts. It is modelled by the action *ExecuteTransition* that attempts to execute a sequence of transitions. This sequence is formed by all the transitions that must be tried. For example, when executing the state `steady_state` (Figure 3.2), there are two outer transitions to be executed; they are ordered and their execution consists of attempting to follow the first, and, if that fails, trying the second.

ExecuteTransition takes as parameters the identifier *tid* of the first transition, the sequence *path* of identifiers of the transitions that have been successfully executed, the *source* state of the transition path, the current event *ce*, and a value-result parameter *success*. The parameter *path* is needed in cases where there are junctions in the path of the transitions and backtracking occurs, and *success* is used to indicate whether or not the execution of the transition caused a state to be entered.

$$
\begin{array}{l}
\mathit{ExecuteTransition} \mathrel{\widehat=} \\
\mathbf{val}\, \mathit{tid} : \mathit{TID};\ \mathbf{val}\, \mathit{path} : \mathrm{seq}\, \mathit{TID};\ \mathbf{val}\, \mathit{source} : \mathit{State};\ \mathbf{val}\, \mathit{ce} : \mathit{EVENT};\ \mathbf{vres}\, \mathit{success} : \mathbb{B} \bullet \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{tid} = \mathit{nulltransition.identifier} \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{path} = \varnothing \longrightarrow \mathit{success} := \mathbf{False} \\
[\!]\ \mathit{path} \neq \varnothing \longrightarrow \left(\begin{array}{l}
\mathit{transition}!(\mathit{last}\ \mathit{path})?\mathit{lt} \longrightarrow \\
\mathit{ExecuteTransition}(\mathit{lt.next}, (\mathit{front}\ \mathit{path}), \mathit{source}, \mathit{ce}, \mathit{success})
\end{array}\right) \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ \mathit{tid} \neq \mathit{nulltransition.identifier} \longrightarrow \\
\mathit{CheckValidity}(\mathit{tid}, \mathit{path}, \mathit{source}, \mathit{ce}, \mathit{success}) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

The backtracking mechanism of transitions has been modelled by means of continuations

[34], but *Circus* does not support this feature. Thus, we use the parameter *path* that contains the necessary information to model the backtracking without the use of explicit continuations.

ExecuteTransition checks if there are any transitions to execute by comparing *tid* to the identifier of the null transition (*nulltransition.identifier*). If they are equal, then *ExecuteTransition* evaluates *path*. If it is empty ($path = \varnothing$), no transition has been followed, and, since $tid = nulltransition.identifier$, there are no new transitions to try. In this case, the execution fails; this is indicated by assigning **False** to *success*. If *path* is not empty, *ExecuteTransition* uses the identifier of the last transition successfully followed (*last path*) to obtain the corresponding transition *lt* through the channel *transition*. It then executes the transition that follows *lt* (*lt.next*) on a reduced path excluding the last transition followed (*front path*), and the original *source* state. If *tid* does not correspond to the null transition, *ExecuteTransition* calls the action *CheckValidiy* on *tid*, the path, the source state, the current event, and the parameter *success*.

Transitions, such as *lt*, are defined by the schema *Transition*, which records the identifier of a transition, the identifiers of its source and destination nodes, the identifier of the next transition (if any), and the parent state (or chart) of that transition.

Transition

$identifier : TID$
$source, destination : NID$
$next : TID$
$parent : SID$

A sequence of transitions is represented by a structure similar to a linked list, where each transition points to the next (possibly null) transition. The information necessary to construct this sequence is directly obtained from the textual representation of a chart. The set *NID* contains node identifiers, which are taken from the sets *SID* and *JID*.

$$NID ::= snode\langle\langle SID\rangle\rangle \mid jnode\langle\langle JID\rangle\rangle$$

Returning to the execution of a transition, the action *CheckValidity* receives the identifier *tid* of a transition, a sequence *path* of transition identifiers, a state *source*, an event *ce*, and a boolean variable *success*.

CheckValidity requests the chart to evaluate the trigger of the transition by communicating *tid* and *ce* through the channel *checktrigger*. It then takes the boolean response *e* through the channel *result*. Next, it requests the evaluation of the condition of the transition through the channel *evaluatecondition* and stores the received value in *c*. If the trigger and the condition are true ($e = \textbf{True} \land c = \textbf{True}$), then *CheckValidity* requests the execution of the condition action through *executeconditionaction*, and calls the action *LocalEventCondition* to treat any local event broadcast. It then declares a local variable *b*

of type boolean, calls the action *conditionActionCheck* with b as a value-result parameter to check the early return logic conditions for condition actions. Consequently, it decides whether to proceed (if $b = \mathbf{False}$) with the execution of the destination node, or the interrupt the execution of the transition (if $b = \mathbf{True}$). In the latter case, *CheckValidity* assigns **True** to *success* indicating that the transitions execution was successful. The execution of the transition is considered successful because the interruption indicates that the local event broadcast (from the condition action) caused another transition to execute and a state to be entered.

$$
\begin{array}{l}
CheckValidity \mathrel{\widehat=} \\
\mathbf{val}\, tid : TID;\ \mathbf{val}\, path : \mathrm{seq}\, TID;\ \mathbf{val}\, source : State;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet \\
\quad checktrigger!tid!ce \longrightarrow result!tid!ce?e \longrightarrow evaluatecondition!tid?c \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ e = \mathbf{True} \land c = \mathbf{True} \longrightarrow \\
\left(\begin{array}{l}
executeconditionaction!tid \longrightarrow LocalEventCondition(source.identifier); \\
\mathbf{var}\, b : \mathbb{B} \bullet \left(\begin{array}{l}
conditionActionCheck(source.identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow success := \mathbf{True} \\
[\!]\ b = \mathbf{False} \longrightarrow Proceed(tid, path \frown \langle tid \rangle, source, ce, success) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
[\!]\ \neg\,(e = \mathbf{True} \land c = \mathbf{True}) \longrightarrow \left(\begin{array}{l}
transition!tid?t \longrightarrow \\
ExecuteTransition(t.next, path, source, ce, success)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

The execution of the destination node is carried out by calling the action *Proceed* on the path extended by the transition identifier ($path \frown \langle tid \rangle$). (Since the transition was successfully followed, it is added to *path*). If the trigger or the condition is false, then the transition is invalid: the transition t that corresponds to *tid* is obtained through the channel *transition*, and the next transition ($t.next$) is executed on the same path, source state, and event.

$$
\begin{array}{l}
LocalEventCondition \mathrel{\widehat=} sid : SID \bullet \mu X \bullet \\
\left(\begin{array}{l}
local_event?e?s \longrightarrow \left(\begin{array}{l}
TreatLocalEvent(e, s); \\
\mathbf{var}\, b : \mathbb{B} \bullet \\
\left(\begin{array}{l}
conditionActionCheck(sid, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \left(\begin{array}{l}
interrupt.\mathbf{True} \longrightarrow \\
end_action \longrightarrow \mathbf{Skip}
\end{array}\right) \\
[\!] b = \mathbf{False} \longrightarrow interrupt.\mathbf{False} \longrightarrow X \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
end_action \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

Whenever the process *Simulator* requests the execution of a chart action, there is a possibility that local events are broadcast. When this action is a condition action, local event broadcasts are treated by the action *LocalEventCondition* that recursively offers a choice between treating an event and waiting for other local events, or terminating. The first op-

tion is modelled by an action that waits for a communication on the channel *local_event*. If it occurs, the action *TreatLocalEvent* is called, and the action *conditionActionCheck* is called to check the appropriate early return logic conditions. If the early return logic conditions are true, the action interrupts the chart action by communicating *true* through the *interrupt* channel and waits for the end of the chart action on the channel *end_action*. In contrast, if the early return logic conditions are false, *LocalEventCondition* indicates that the chart action can continue by communicating **False** through *interrupt*, and recurses. The second option waits for a synchronisation on *end_action*; the chart process agrees on that when a state or transition action terminates. In this case, *LocalEventCondition* terminates.

The action *TreatLocalEvent* takes an event e and a destination state s. If s has type *CHART* ($s.type = CHART$), it executes the chart, as defined by *ExecuteChart*, on e. Otherwise, *TreatLocalEvent* executes s using *ExecuteState* again on the new event. Finally, it signals the end of the local execution on the channel *end_local_execution*, thus, allowing the broadcasting action of the chart (discussed in the next section) to terminate.

$$\begin{array}{l} TreatLocalEvent \mathrel{\widehat=} e : EVENT;\ s : State \bullet \\ \quad \left(\begin{array}{l} \mathbf{if}\ s.type = CHART \longrightarrow ExecuteChart(e) \\ [\!]\ s.type \neq CHART \longrightarrow ExecuteState(s, e) \\ \mathbf{fi} \end{array} \right);\ end_local_execution \longrightarrow \mathbf{Skip} \end{array}$$

This treatment of local events unifies the notions of event broadcast, directed and qualified event broadcasts. The form of broadcast modelled is the directed form. A simple broadcasting is a directed broadcast to the chart. A qualified broadcast is a directed broadcast to the qualifying state.

Whenever a local event broadcast occurs, the appropriate early return logic conditions must be checked to avoid the possibility of reaching an inconsistent state [1]. In the action *LocalEventCondition*, this is achieved by a call to *conditionActionCheck*, which checks whether the source of the transition path (*source*) is still active after the execution of the condition action. If it is, the execution proceeds as expected. Otherwise, it is halted. In *CheckValidity*, the check takes place after the condition action has been completely executed. In contrast, in *LocalEventCondition*, *conditionActionCheck* checks the early return logic conditions after each local event broadcast (because the same action might have more than one broadcast).

Local event broadcasts and early return logic are two aspects of Stateflow charts that complicate the semantics. They are particularly difficult because they are not well documented. Additionally, in our case, these aspects complicate the model because they do not respect the separation between chart structure and the simulator as enforced by our modelling approach. Our solution models a communication protocol that supports the recursive execution of chart (or states) and the interruption of specific parts of the execution

[1] An inconsistency occurs when, for instance, two sequential states are active at the same time, or the parent of an active state is inactive.

as required by the early return logic mechanism.

As previously mentioned, a valid transition may lead to the execution of the destination node by a call to the action *Proceed*. This action requests the transition identified by *tid* through the channel *transition*. It then determines the type of the destination node identified by $t.destination$. If it is a state, $t.destination$ is in the range of the function *snode*, which produces node identifiers from state identifiers. Otherwise, the destination node is a junction (and $t.destination$ is in the range of the function *jnode*, which associates a node identifier to a junction identifier). If the destination is a state *dest*, *Proceed* recovers it through the channel *state*, and then calls the action *proceedToState*. The state identifier is obtained from the node identifier by applying the inverse of the function *snode* ($(snode^{\sim})t.destination$). If the identifier of the destination node corresponds to a junction identifier, the action *proceedToJunction* is called.

$$
\begin{array}{l}
Proceed \mathrel{\widehat=} \\
\mathbf{val}\, tid : TID;\ \mathbf{val}\, path : \operatorname{seq} TID;\ \mathbf{val}\, source : State;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet \\
transition!tid?t \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ t.destination \in \operatorname{ran} snode \longrightarrow \left(\begin{array}{l} state!((snode^{\sim})\ t.destination)?dest \longrightarrow \\ proceedToState(source, dest, path, ce, success) \end{array}\right) \\
[]\ t.destination \in \operatorname{ran} jnode \longrightarrow proceedToJunction(tid, path, source, ce, success) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

The definition of *proceedToState* is based on the closest common parent, $la(src, dest)$, of the source and destination states *src* and *dest* (that is the state that is an ancestor of both *src* and *dest*, and that has no substate that is also an ancestor of both these states).

$$
\begin{array}{l}
proceedToState \mathrel{\widehat=} \mathbf{val}\, src, dest : State;\ \mathbf{val}\, path : \operatorname{seq} TID;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet \\
\left(\begin{array}{l}
ExitStates((la(src, dest)).substates, ce); \\
\mathbf{var}\, b : \mathbb{B} \bullet \left(\begin{array}{l}
exitStatesCheck((la(src, dest)).identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[]\ b = \mathbf{False} \longrightarrow \\
\left(\begin{array}{l}
executePath(path, src, dest, ce); \\
transitionActionCheck((la(src, dest)).identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[]\ b = \mathbf{False} \longrightarrow EnterState(dest, la(src, dest), ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right); \\
success := \mathbf{True}
\end{array}\right)
\end{array}
$$

Firstly, the action *proceedToState* exits all the active substates of the common ancestor of the source and destination states ($(la(src, dest)).substates$). Next, it checks the early return logic conditions for exiting states (*exitStatesCheck*), and, if they are true ($b = \mathbf{True}$), interrupts the execution. Otherwise, it executes *path* as defined by *executePath*, and checks the early return logic conditions for transition actions (*transitionActionCheck*). If these

conditions are true, *proceedToState* interrupts the execution. If they are false, it enters *dest* using *EnterState*. Finally, independent of whether the early return logic conditions interrupt the execution, *proceedToState* indicates the success of the transition execution by assigning **True** to *success*. As before, the halting of the execution due to early return logic is treated as a successful execution.

The call to *EnterState* takes the closest ancestor of *src* and *dest* as the first parameter. The closest ancestor is given by the function *la* that takes two states and calculates the least upper bound of the two states with respect to an ancestry relation.

$$la : (State \times State) \rightarrow State$$

$$\forall s_1, s_2 : State \bullet la(s_1, s_2) = \mu\, x : (ancestors(s_1) \cap ancestors(s_2)) \mid \\ (\forall y : (ancestors(s_1) \cap ancestors(s_2)) \bullet x = y \lor y \in ancestors(x)) \bullet x$$

The function *la* is defined for states s_1 and s_2 in a chart by applying the Z definite description operator to select the sole common ancestor of s_1 and s_2 that has no substates that are also common ancestors of s_1 and s_2.

$$ancestors : State \rightarrow \mathbb{P}\, State$$

$$\forall s : State \bullet ancestors(s) = (parent^{+}) (\!|\, \{s\} \,|\!) \setminus \{nullstate\}$$

The set of ancestors of a state s is defined as the set of all states that are related to s by the transitive closure of the relation *parent* ($parent^{+}$) except for the null state. The relation *parent* is defined as the set of pairs of states such that the component *parent* of the first is equal to the component *identifier* of the second.

$$executePath \mathrel{\widehat=} path : \mathrm{seq}\, TID;\ src, dest : State;\ ce : EVENT \bullet \\ \left(\begin{array}{l} \mathbf{if}\ \#\, path = 0 \longrightarrow \mathbf{Skip} \\ [\!]\ \#\, path > 0 \longrightarrow \left(\begin{array}{l} executetransitionaction!(head\ path) \longrightarrow \\ LocalEventTransition((la(src, dest)).identifier); \\ \mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l} transitionActionCheck((la(src, dest)).identifier, b); \\ \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ b = \mathbf{False} \longrightarrow executePath(tail\ path, src, dest, ce) \\ \mathbf{fi} \end{array}\right) \end{array}\right) \end{array}\right) \\ \mathbf{fi} \end{array}\right)$$

The execution of the path executes each of its transition actions; this is defined by the action *executePath*. If the path is empty ($\#\, path = 0$), then *executePath* does nothing. Otherwise, it requests the execution of the transition action of the first transition in the path by communicating its identifier (head *path*) through the channel *executetransitionaction*. It then calls *LocalEventTransition* to deal with any local event broadcast by the transition action, checks the early return logic conditions for transition actions, and, if the conditions do not require the interruption of the execution, recurses over the rest of the path (tail *path*).

The definition of *LocalEventTransition* is similar to that of *LocalEventCondition* (above), except that the early return logic conditions checked are those for transition actions. Action *transitionActionCheck* checks that the substates of the closest common ancestor, $la(src, dest)$, are not active, as they had just been exited before the execution of the path [2].

If the destination of the transition is a junction, *proceedToJunction* is called. It obtains the transition t with identifier *tid* through the channel *transition*, and then the destination junction dj of t by communicating its identifier $(jnode^{\sim})\ t.destination$ through *junction*. If dj is not a history junction, it calls *executeJunction* on dj, the sequence *path*, the state *source*, and the event *ce*. Otherwise, it obtains the state identifier *lsid* that is stored in the history junction by communicating the identifier of the parent of the junction $(dj.parent)$ through *history*. If *lsid* is the identifier of the null state $(lsid = nullstate.identifier)$, the default transitions of the state are executed using *ExecuteDefaultTransition*, and *success* is assigned **True**. Otherwise, *proceedToJunction* recovers the state *ls* identified by *lsid* through *state*, and calls *proceedToState* with *ls* as the destination state.

$$
\begin{array}{l}
proceedToJunction \mathrel{\widehat=} \\
\mathbf{val}\ tid : TID;\ \mathbf{val}\ path : \mathrm{seq}\ TID;\ \mathbf{val}\ source : State;\ \mathbf{val}\ ce : EVENT;\ \mathbf{vres}\ success : \mathbb{B} \bullet \\
transition!tid?t \longrightarrow junction!((jnode^{\sim})\ t.destination)?dj \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ dj.history = \mathbf{False} \longrightarrow executeJunction(dj, path, source, ce, success) \\
[]\ dj.history = \mathbf{True} \longrightarrow history!(dj.parent)?lsid \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ lsid = nullstate.identifier \longrightarrow \left(\begin{array}{l} state!(dj.parent)?s \longrightarrow \\ ExecuteDefaultTransition(s, s, ce); \\ success := \mathbf{True} \end{array}\right) \\
[]\ lsid \neq nullstate.identifier \longrightarrow \left(\begin{array}{l} state!lsid?ls \longrightarrow \\ proceedToState(source, ls, path, ce, success) \end{array}\right) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

The Stateflow User's Guide discusses inner transitions to history junctions, but outer and default transitions are not mentioned. Our experiments show that outer transitions to history junctions have a behaviour similar to that of inner transitions. On the other hand, default transitions to history junctions may lead to inconsistencies. In our model, this may lead to divergence: the first time a default transition to a history junction is followed, it observes *nullstate.identifier*, and the default transitions are attempted again in a potentially infinite loop. Moreover, inner transitions to history junctions can lead to an attempt to enter an already active state; in this case, the state is exited, and reentered. This is captured in *proceedToState*. As previously explained, this action calls *ExitStates* on the substates of the closest common parent of the source and destination states. In this case, they are the substates of the state with the inner transition. At least one of them is active, and is deactivated before any attempt at entering is made.

A junction is defined by the schema *Junction*, which records the identifier of a junc-

[2] The definition of *LocalEventTransition* and *transitionActionCheck* can be found in Appendix B.

tion, the identifier of the first (if any) transition leaving it, the identifier of its parent state (or chart), and a boolean component indicating whether or not the junction is a history junction.

$$\begin{array}{|l} \textit{Junction} \\ \hline \textit{identifier} : \textit{JID};\ \textit{transition} : \textit{TID};\ \textit{parent} : \textit{SID};\ \textit{history} : \mathbb{B} \\ \hline \end{array}$$

The execution of a junction consists of executing its outgoing transitions. If there are none, the transition path fails. The action *executeJunction* compares the identifier of the first outgoing transition ($j.transition$) to the identifier of the null transition. If they are the same, there are no transitions out of the junction, and the execution failure is indicated by assigning **False** to *success*. Otherwise, *executeJunction* calls *ExecuteTransition*.

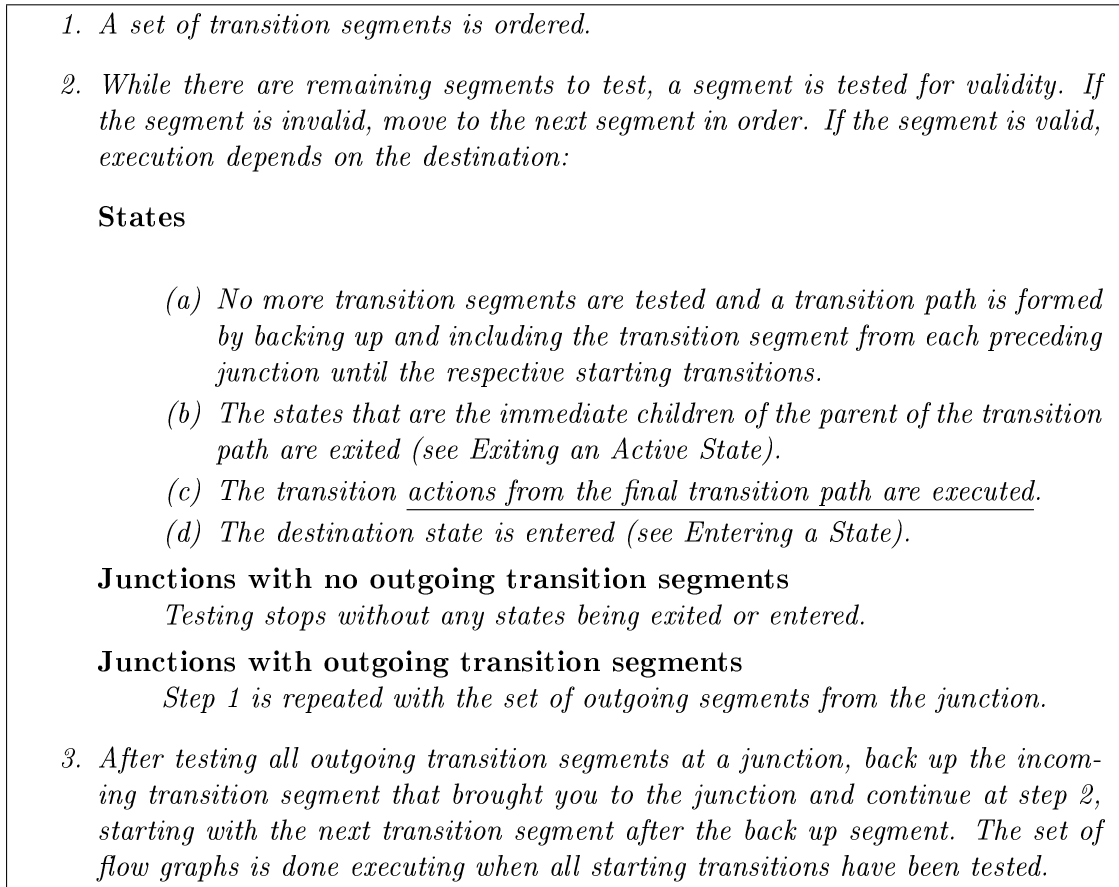
$$\begin{array}{l} \textit{executeJunction} \mathrel{\widehat{=}} \\ \mathbf{val}\, j : \textit{Junction};\ \mathbf{val}\, \textit{path} : \mathrm{seq}\, \textit{TID};\ \mathbf{val}\, \textit{source} : \textit{State};\ \mathbf{val}\, \textit{ce} : \textit{EVENT};\ \mathbf{vres}\, \textit{success} : \mathbb{B} \bullet \\ \left(\begin{array}{l} \mathbf{if}\, j.\textit{transition} = \textit{nulltransition.identifier} \longrightarrow \textit{success} := \mathbf{False} \\ [\!]\ j.\textit{transition} \neq \textit{nulltransition.identifier} \longrightarrow \\ \qquad \textit{ExecuteTransition}(j.\textit{transition}, \textit{path}, \textit{source}, \textit{ce}, \textit{success}) \\ \mathbf{fi} \end{array} \right) \end{array}$$

The key actions that model the execution of transitions are *ExecuteTransition*, which we presented above, and *ExecuteDefaultTransition*. *ExecuteDefaultTransition* extends *ExecuteTransition* by treating the possibility that there are no default transitions to follow, but the choice of which state to enter is deterministic. This is the case when, for example, there is only one substate.

The informal description of the execution of a set of transitions is shown in Figure 2.1, which is repeated in Figure 3.5 for convenience. These steps describe the execution of a set of transitions as an iterative procedure, where each iteration starts with step 2 and ends with step 3. *ExecuteTransition* corresponds to the first sentence of step 2 in conjunction with step 3. *CheckValidity* corresponds to the rest of step 2, *Proceed* to the selection between state and junction destinations specified in the end of step 2, *proceedToState* to the four steps under the label **States**, and *executeJunction* to the steps under the labels **Junctions with no outgoing transition segments** and **Junctions with outgoing transition segments**. The action *proceedToJunction* does not correspond to any step in the informal description; it extends the execution of transition by modelling the case when a transition leads to a history junction.

3.2.3 Entering a state

To keep close to the informal description previously shown in Figure 2.2, we preserve the structure and granularity of the steps in the actions that model the simulation semantics. For convenience, we repeat in Figure 3.6 the steps for entering a state.



1. *A set of transition segments is ordered.*
2. *While there are remaining segments to test, a segment is tested for validity. If the segment is invalid, move to the next segment in order. If the segment is valid, execution depends on the destination:*

   States

   (a) *No more transition segments are tested and a transition path is formed by backing up and including the transition segment from each preceding junction until the respective starting transitions.*
   (b) *The states that are the immediate children of the parent of the transition path are exited (see Exiting an Active State).*
   (c) *The transition actions from the final transition path are executed.*
   (d) *The destination state is entered (see Entering a State).*

   Junctions with no outgoing transition segments
   Testing stops without any states being exited or entered.

   Junctions with outgoing transition segments
   Step 1 is repeated with the set of outgoing segments from the junction.

3. *After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the back up segment. The set of flow graphs is done executing when all starting transitions have been tested.*

Figure 3.5: Executing a Set of Transitions [98]

The procedure for entering a state s is defined by the action *EnterState* below.

$$EnterState \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState16(s, tpp, ce)$$

For each step in Figure 2.2, we define an action whose name is *EnterState* postfixed by the number of the associated step. For instance, step 1 is formalised by the action *EnterState*1. Similarly, the name of actions that correspond to the execution of a range of steps is postfixed by the numbers of the first and last steps.

For example, the execution of steps 1-6 is modelled by *EnterState*16. Since some of the steps can result in a local event broadcast, we must check the early return logic conditions after each such step, and possibly interrupt part of the execution. For this reason, the combination of these steps is not straightforward.

$$\begin{array}{l} EnterState16 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState1(s, tpp, ce); \\ \mathbf{var}\ b : \mathbb{B} \bullet enterState1Check(s.identifier, b)\ ; \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ []\, b = \mathbf{False} \longrightarrow EnterState26(s, tpp, ce) \\ \mathbf{fi} \end{array}\right) \end{array}$$

*EnterState*16 executes step 1 by calling action *EnterState*1, declares a local variable b, checks the appropriate early return logic condition by calling *enterState*1*Check* with b as

1. *If the parent of the state is not active, perform steps 1-4 for the parent.*
2. *If this is a parallel state, check if the immediate sibling with a higher (i.e., earlier) entry order is active. If not, perform entry steps 1-5 for this state first.*
3. *Mark the state active.*
4. *Perform any entry actions.*
5. *Enter children, if needed:*
   (a) *If the state contains a history junction and there was an active child of this state at some point after the most recent chart initialisation, perform entry steps 1-5 for that child. Otherwise, execute the default flow paths for the state.*
   (b) *If this state has parallel decomposition, i.e., has children that are parallel states, perform entry steps 1-5 for each state according to its entry order.*
6. *If this is a parallel state, perform all entry steps for the sibling state next in entry order if one exists.*
7. *Else, if the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.*

Figure 3.6: Entering a State [98]

one of the parameters, and uses the value assigned by $enterState1Check$ to b, in order to decide whether to proceed with the execution of the remaining steps ($EnterState26$), or to terminate immediately.

Action $EnterState26$ is defined in a similar fashion. Its definition can be found in Appendix B as well as the definitions of the remaining actions that execute a range of steps while checking the appropriate early return logic conditions. As previously mentioned, the process of entering a state is modelled by the action $EnterState$, which executes the steps from 1 to 6. Step 7 is executed when 6 fails, and is called explicitly from Step 6.

The first step of execution requires the execution of steps 1-4 for the parent state. The action $EnterState1$ models this step; it takes as parameters the state s to be entered, the parent of the transition path tpp and the current event ce.

$$
\begin{array}{l}
EnterState1 \mathrel{\widehat{=}} s, tpp : State;\ ce : EVENT \bullet status!(s.parent)?active \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{False} \longrightarrow state!(s.parent)?p \longrightarrow EnterState14(p, tpp, ce) \\
[\!]\, active = \mathbf{True} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$EnterState1$ first checks whether the parent of the state is active by communicating its identifier ($s.parent$) through the channel $status$. If it is active, it does nothing. Otherwise, it obtains the parent state by communicating its identifier through the channel $state$ and calls action $EnterState14$ on it, thus executing steps 1-4 for the parent state.

After this step is carried out, the parent must be active. However, a local event broadcast that occurs while this step is being executed may deactivate the parent state. Therefore, the early return logic condition for step 1 is that the parent is still active after the step has been executed; this check is modelled by action $enterState1Check$.

$$enterState1Check \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet state!sid?s \longrightarrow entryActionCheck(s.parent, b)$$

This action receives a state identifier sid and a value-result parameter b; it obtains the state s whose identifier is sid and checks that its parent is still active. This is exactly the early return logic condition of entry actions applied to the parent of s. We perform this check by calling the action $entryActionCheck$ on the parent ($s.parent$) and b.

$$entryActionCheck \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet status!sid?active \longrightarrow b := not(active)$$

The action $entryActionCheck$ checks the early return logic conditions associated with the execution of entry actions. After the execution of an entry action if the state whose entry action has been executed is no longer active ($not(, active)$), the execution halts. The action $entryActionCheck$ takes the same parameters as $enterState1Check$; it checks the status $active$ of the state whose identifier is sid and assigns the negation of this value to b.

The action $EnterState2$ models the second step of the procedure for entering a state; it takes the same parameters as $EnterState1$. It first checks whether the state is parallel ($s.type = AND$) or not. If it is not, it does nothing. Otherwise, it checks if the state has a left sibling. Again, if it does not ($s.left = nullstate.identifier$), it terminates. Otherwise, it checks if the left sibling is active by communicating its identifier ($s.left$) through the channel $status$. If it is, there is nothing left to do. If the left sibling is not active, this action obtains the state (ls) and executes steps 1-5 for it by calling the action $EnterState15$.

$$
\begin{array}{l}
EnterState2 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \\
\left(\begin{array}{l}
\mathbf{if}\ s.type = AND \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ s.left \neq nullstate.identifier \longrightarrow status!(s.left)?active \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ active \neq \mathbf{True} \longrightarrow state!(s.left)?ls \longrightarrow EnterState15(ls, tpp, ce) \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ s.left = nullstate.identifier \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ s.type \neq AND \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

After step 2, it is expected that the left sibling, if any, is active. If a left sibling exists, and it is no longer active, execution must be halted. This is checked by action $enterState2Check$, which we omit here. The definition of $enterState2Check$ can be found in Appendix B.

The action $EnterState3$ simply requests the activation of the state being entered by communicating the identifier of the state through the channel $activate$. There are no

early return logic conditions associated with this action as it cannot trigger a local event broadcast.

$$EnterState3 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet activate!(s.identifier) \longrightarrow \mathbf{Skip}$$

The next action, *EnterState*4, requests the execution of the entry action of the state being entered by communicating its identifier through the channel *executeentryaction*.

$$\begin{array}{l} EnterState4 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \\ \quad executeentryaction!(s.identifier) \longrightarrow LocalEventEntry(s.identifier) \end{array}$$

Similarly to the execution of condition and transition actions, local event broadcasts are treated by a call to *LocalEventEntry*. This action is similar to *LocalEventCondition* and can be found in Appendix B. As previously mentioned, after the execution of an entry action, the state must still be active; otherwise, the execution must be halted.

The fifth step describes two cases to be treated. We model these cases as the actions *EnterState*5*a* and *EnterState*5*b*, which are executed in different situations. They are sequentially composed to form *EnterState*5.

$$EnterState5 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState5a(s, tpp, ce) \ ;\ EnterState5b(s, tpp, ce)$$

Action *EnterState*5*a* executes a history junction, if one exists. Otherwise, it executes the default transitions. An interesting consequence of this step is that if a state with a parallel decomposition has a default transition, instead of directly entering the substates, the default transition is executed and the substates are entered. This fact has been confirmed by experiments with the simulation tool.

$$\begin{array}{l} EnterState5a \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \\ \left(\begin{array}{l} \mathbf{if}\ s.history = \mathbf{True} \longrightarrow history!(s.identifier)?lsid \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{if}\ lsid \neq nullstate.identifier \longrightarrow state!lsid?ls \longrightarrow EnterState15(ls, tpp, ce) \\ [\!]\ lsid = nullstate.identifier \longrightarrow ExecuteDefaultTransition(s, tpp, ce) \\ \mathbf{fi} \end{array}\right) \\ [\!]\ s.history = \mathbf{False} \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{if}\ s.default \neq nulltransition.identifier \longrightarrow ExecuteDefaultTransition(s, tpp, ce) \\ [\!]\, s.default = nulltransition.identifier \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \\ \mathbf{fi} \end{array}\right) \end{array}$$

If s has a history junction ($s.history = \mathbf{True}$), then *EnterState*5*a* recovers the state identifier *lsid* in the history junction by communicating the state identifier ($s.identifier$) through the channel *history*. If *lsid* is the identifier of the null state ($lsid = nullstate.identifier$), *EnterState*5*a* calls *ExecuteDefaultTransition* on s. Otherwise, it obtains the state *ls* identified by *lsid* through *state*, and executes the steps 1 to 5. If s does not have a history junction, but has default transitions, these are executed.

If the state has a parallel decomposition, $EnterState5b$ executes steps 1-5 for each parallel substate. However, if there is a default transition, it must be executed instead. Since, this case is treated in action $EnterState5a$, we exclude this possibility in action $EnterState5b$ by conditioning the execution of $EnterStates15$ accordingly. $EnterStates15$ executes steps 1-5 for each parallel substate in order. Each execution of these steps may lead to a local event broadcasts, therefore the appropriate conditions are checked, and the execution is interrupted if necessary. The definition of this action can be found in Appendix B.

$$EnterState5b \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l} \mathbf{if}\ s.decomposition = SET \land s.default = nulltransition.identifier \longrightarrow \\ \qquad EnterStates15(s.substates, tpp, ce) \\ \Box\, s.decomposition \neq SET \lor s.default \neq nulltransition.identifier \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

The action $EnterState6$ checks if the state is a parallel state ($s.type = AND$) and if it has a right sibling ($s.right \neq nullstate.identifier$). If this is the case, it obtains the right sibling (rs) using channel $state$, and enters it using action $EnterState$. Otherwise, this action executes step 7 by calling $EnterState7$.

$$EnterState6 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l} \mathbf{if}\ s.type = AND \land s.right \neq nullstate.identifier \longrightarrow \\ \qquad state!(s.right)?rs \longrightarrow EnterState(rs, tpp, ce) \\ \Box\, s.type \neq AND \lor s.right = nullstate.identifier \longrightarrow EnterState7(s, tpp, ce) \\ \mathbf{fi} \end{array}\right)$$

The final step is modelled by the action $EnterState7$. It checks if the state is a chart or not ($s.type = CHART$). If it is, the action terminates immediately. Otherwise, it obtains the parent p by communicating the identifier of the parent of s ($s.parent$) through $state$, and compares p to the parent of the transition path. If they are the same, the action terminates. If the parent of the state is not tpp, the transition path that led to this state being entered contains interlevel transitions. In this case, the action calls $EnterState6$ on the parent (thus executing steps 6 and 7).

$$EnterState7 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l} \mathbf{if}\ s.type \neq CHART \longrightarrow state!(s.parent)?p \longrightarrow \left(\begin{array}{l} \mathbf{if}\ tpp \neq p \longrightarrow EnterState6(p, tpp, ce) \\ \Box\, tpp = p \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \\ \Box\, s.type = CHART \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

Note that, action $EnterState1$ guarantees that steps 1 through 4 are executed on all the inactive ancestors of a state. However, if any of them is a parallel state, it might be the case

that its right siblings are not entered. The action *EnterState*7 takes care of this scenario, by recursively applying the action *EnterState*6 to the necessary states.

3.2.4 Executing and exiting a state

The processes of executing and exiting active states are simple compared to entering a state or executing transitions. As before, we repeat in this section the informal description in Figures 3.7 and 3.8.

The action *ExecuteState* takes a state s and the current event as parameters.

$$
\begin{array}{l}
\mathit{ExecuteState} \mathrel{\widehat=} s : \mathit{State};\ ce : \mathit{EVENT} \bullet \mathit{status}!(s.\mathit{identifier})?\mathit{active} \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{active} = \mathbf{True} \longrightarrow \\
\left(\begin{array}{l}
\mathbf{var}\ \mathit{success} : \mathbb{B} \bullet \\
\mathit{ExecuteTransition}(s.\mathit{outer}, \langle\rangle, s, ce, \mathit{success}); \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{success} = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ \mathit{success} = \mathbf{False} \longrightarrow \\
\quad \left(\begin{array}{l}
\mathit{executeduringaction}!(s.\mathit{identifier})!ce \longrightarrow \\
\mathit{LocalEventDuring}(s.\mathit{identifier}); \\
\mathbf{var}\ b : \mathbb{B} \bullet \\
\left(\begin{array}{l}
\mathit{duringActionCheck}(s.\mathit{identifier}, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow \mathit{AlternativeExecution}(s, ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
[\!]\ \mathit{active} = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

ExecuteState first checks whether or not the state is active by communicating the identifier of the state through the channel *status*. If the state is not active, *ExecuteState* terminates immediately. Otherwise, *ExecuteState* declares a boolean variable *success* and calls the action *ExecuteTransition* on the outer transitions passing *success* as a parameter. The call to *ExecuteTransition* then updates *success* to indicate whether or not the execution of the outer transitions was successful. If it is ($\mathit{success} = \mathbf{True}$), then *ExecuteState* terminates immediately. Otherwise, it requests the execution of the during and on actions of the state by communicating the state identifier and the current event ce through *executeduringaction*.

Since the execution of a during action can lead to local event broadcasts, *ExecuteState* treats such broadcasts (*LocalEventDuring*), checks the appropriate early return logic conditions (*duringActionCheck*), and, if necessary, terminates immediately. Otherwise, it calls the action *AlternativeExecution* on the state to execute its inner transitions.

AlternativeExecution attempts to execute the inner transitions of the state s in the same fashion as *ExecuteState*. The local variable *success* is used to indicate the success or failure of the execution, and the alternation controls the remaining execution according to

> 1. *The set of outer flow graphs execute (see Executing a Set of Flow Graphs). If this action causes a state transition, execution stops. (Note that this step never occurs for parallel states.)*
> 2. *During actions and valid on-event actions are performed.*
> 3. *The set of inner flow graphs execute. If this action does not cause a state transition, the active children execute, starting at step 1. Parallel states execute in the same order that they become active.*

Figure 3.7: Executing an active State [98]

the value of *success*. If the execution is successful, *ExecuteState* terminates. Otherwise, the substates are executed.

$$AlternativeExecution \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \left(\begin{array}{l} \mathbf{var}\ success : \mathbb{B} \bullet ExecuteTransition(s.inner, \langle\rangle, s, ce, success); \\ \left(\begin{array}{l} \mathbf{if}\ success = \mathbf{True} \longrightarrow \mathbf{Skip} \\ []\, success = \mathbf{False} \longrightarrow ExecuteSubstates(s, ce) \\ \mathbf{fi} \end{array}\right) \end{array}\right)$$

The execution of substates must take into consideration the type of decomposition of the parent state. If the substates are in a parallel decomposition ($s.decomposition = SET$), then they are executed by action *ExecuteParallelStates*, and if they are in a sequential decomposition ($s.decomposition = CLUSTER$), they are executed by *ExecuteSequentialStates*.

$$ExecuteSubstates \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \left(\begin{array}{l} \mathbf{if}\ s.decomposition = SET \longrightarrow ExecuteParallelStates(s.substates, ce) \\ []\, s.decomposition = CLUSTER \longrightarrow ExecuteSequentialStates(s.substates, ce) \\ \mathbf{fi} \end{array}\right)$$

The action *ExecuteParallelStates* is defined as an implicit recursion that traverses the sequence of states, executing the active states. After each state is executed, the early return logic conditions are checked. If they are true, the execution is interrupted. Otherwise, the remaining states are executed.

$$ExecuteParallelStates \mathrel{\widehat=} ss : \mathrm{seq}\, SID;\ ce : EVENT \bullet \left(\begin{array}{l} \mathbf{if}\ \#\, ss = 0 \longrightarrow \mathbf{Skip} \\ []\ \#\, ss > 0 \longrightarrow state!(head\ ss)?first \longrightarrow ExecuteState(first, ce); \\ \quad \mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l} executeStateCheck(head\ ss, b); \\ \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ []\ b = \mathbf{False} \longrightarrow ExecuteParallelStates(tail\ ss, ce) \\ \mathbf{fi} \end{array}\right) \end{array}\right) \\ \mathbf{fi} \end{array}\right)$$

ExecuteSequentialStates is defined in a similar fashion to *ExecuteParallelStates*; it is a

recursive action that traverses the sequence of states while checking the status of each state, but as soon as an active state is found, it is executed and the recursion terminates.

$$
\begin{array}{l}
\mathit{ExecuteSequentialStates} \mathrel{\widehat=} ss : \operatorname{seq} \mathit{SID};\ ce : \mathit{EVENT} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{if}\ \# ss = 0 \longrightarrow \mathbf{Skip} \\
[\!]\ \# ss > 0 \longrightarrow \mathit{status}!(\mathit{head}\ ss)?\mathit{active} \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{active} = \mathbf{True} \longrightarrow \mathit{state}!(\mathit{head}\ ss)?\mathit{first} \longrightarrow \mathit{ExecuteState}(\mathit{first}, ce) \\
[\!]\ \mathit{active} = \mathbf{False} \longrightarrow \mathit{ExecuteSequentialStates}(\mathit{tail}\ ss, ce) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

Overall, the action *ExecuteState* models steps 1 and 2 in Figure 3.7, and *AlternativeExecution* models step 3. The action *ExecuteSubstates* distinguishes between sequential and parallel substates as required in step 3, and *ExecuteParallelStates* and *ExecuteSequentialStates* execute the appropriate substates in the right order.

The action *ExitState* takes a state and the current event as parameters. It first checks if the state has an active parallel right sibling through the channel *status* and, if it does, that state is exited. The action then exits all active substates by calling *ExitStates* on $s.substates$, requests the execution of the exit action through channel *executeexitaction*, treats any potential local event broadcasts (*LocalEventExit*), checks the appropriate early return logic conditions for that state (*exitActionCheck*) and – as appropriate – either terminates, or requests the deactivation of the state by communicating the identifier of the state through the channel *deactivate*.

$$
\begin{array}{l}
\mathit{ExitState} \mathrel{\widehat=} s : \mathit{State};\ ce : \mathit{EVENT} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{if}\ s.\mathit{right} \neq \mathit{nullstate.identifier} \longrightarrow \mathit{status}!(s.\mathit{right})?\mathit{active} \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ \mathit{active} = \mathbf{True} \longrightarrow \mathit{state}!(s.\mathit{right})?rs \longrightarrow \mathit{ExitState}(rs, ce) \\
[\!]\ \mathit{active} = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ s.\mathit{right} = \mathit{nullstate.identifier} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\mathit{ExitStates}(s.\mathit{substates}, ce); \\
\mathit{executeexitaction}!(s.\mathit{identifier}) \longrightarrow \mathit{LocalEventExit}(s.\mathit{identifier}); \\
\mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l}
\mathit{exitActionCheck}(s.\mathit{identifier}, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow \mathit{deactivate}!(s.\mathit{identifier}) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

The action *ExitState* is only called by itself and by the action *ExitStates* (see below). In both cases, its execution is conditioned on the state to be exited being active. This guarantees that no inactive states are ever exited.

A sequence of states is exited by calling the action *ExitStates*, which takes as parameters

1. *If this is a parallel state, check that the immediate sibling that became active after this state have already become inactive. Otherwise, perform all exiting steps on that sibling state.*
2. *If there are any active children, perform the exit steps on these states in the reverse order that they became active.*
3. *Perform any exit actions.*
4. *Mark the state as inactive.*

Figure 3.8: Exiting a State [98]

a sequence of state identifiers, and the current event. If the sequence is empty nothing is done. If the sequence is empty, *ExitStates* terminates. If the sequence has one or more state identifiers, *ExitStates* checks the status of the state that corresponds to the last identifier in the sequence, and, if the state is active, exits it. Next, *ExitStates* checks the early return logic conditions for exiting a state, and, if they are true, terminates. Otherwise, it recursively exits the remaining states.

$$
\begin{array}{l}
\mathit{ExitStates} \;\widehat{=}\; ss : \mathrm{seq}\, \mathit{SID};\; ce : \mathit{EVENT} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{if}\ \#\, ss = 0 \longrightarrow \mathbf{Skip} \\
[\!]\ \#\, ss > 0 \longrightarrow \mathit{status}!(\mathit{last}\ ss)?\mathit{active} \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathbf{if}\ \mathit{active} = \mathbf{True} \longrightarrow \mathit{state}!(\mathit{last}\ ss)?l \longrightarrow \mathit{ExitState}(l, ce) \\
[\!]\ \mathit{active} = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l}
\mathit{exitStateCheck}(\mathit{head}\ ss, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow \mathit{ExitStates}(\mathit{front}\ ss, ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

The action *ExitState* models steps 1, 3 and 4 of the process of exiting a state shown in Figure 3.8. Step 2 is modelled by a call to *ExitStates*.

This concludes the process *Simulator* that models the semantics of Stateflow charts independently of any particular instance of Stateflow charts. An advantage of this separation is that changes and extensions to the semantics of the main components of Stateflow are restricted to this process. Furthermore, since this process is the same for every chart, the effort require to generate the chart specific model is reduced significantly.

In the next section, we describe the process that represents the structure of a particular chart and complements the *Simulator* process in order to model the execution of the chart.

3.3 Chart process

The chart process records the structure of the chart (state, transitions, and junctions) using constants, as this information does not change throughout the execution of the chart. The process state records the data declared in the chart, and information particular to its execution. For instance, the chart process *c_shift_logic* (Figure 3.9) corresponds to our example in Figure 3.2.

A schema *StateflowChart* captures the general structure of a Stateflow chart. It is defined outside of the scope of the chart process, since it is the same for every chart.

StateflowChart

$$identifier : SID$$
$$states : SID \ffun State$$
$$transitions : TID \ffun Transition$$
$$junctions : JID \ffun Junction$$

$$nullstate \notin \operatorname{ran} states$$
$$nulltransition \notin \operatorname{ran} transitions$$
$$nulljunction \notin \operatorname{ran} junctions$$
$$\#\{s : \operatorname{ran} states \mid s.type = CHART\} = 1$$
$$(states(identifier)).type = CHART$$
$$\forall n : SID \mid n \in \operatorname{dom} states \bullet (states(n)).identifier = n$$
$$\forall n : JID \mid n \in \operatorname{dom} junctions \bullet (junctions(n)).identifier = n$$
$$\forall n : TID \mid n \in \operatorname{dom} transitions \bullet (transitions(n)).identifier = n$$

The component *identifier* identifies the state that represents the chart. The sets *State*, *Transition* and *Junction* are defined by the schemas described in the previous section. The components *states*, *transitions* and *junctions* are, therefore, finite functions from identifiers to bindings. The invariant defines basic structural constraints enforced by the Stateflow notation:

- each of the *states*, *transitions*, and *junctions* components must not contain the null object (of the appropriate type);
- the range of *states* must contain a single state of type *CHART*, that is indicated by *identifier*; and
- the application of each of the functions *states*, *transitions*, and *junctions* to an identifier n in its domain must yield a binding whose component *identifier* is equal to n.

In the chart process (see Figure 3.9), an axiomatic definition includes *StateflowChart*, promoting its components to process constants, and identifying their values. For our shift logic example, the axiomatic definition states that $identifier = c_shift_logic$ and that *states* includes the pairs $(s_downshifting, S_downshifting)$, $(s_gear_state, S_gear_state)$ and so

process $c_shift_logic \mathrel{\widehat=}$ **begin**

$$
\begin{array}{l}
\quad StateflowChart \\
\hline
\quad \ldots
\end{array}
$$

$SimulationData == [state_status : SID \twoheadrightarrow \mathbb{B};\ state_history : SID \twoheadrightarrow SID \mid \ldots]$
$InitSimulationData == [SimulationData\,' \mid \ldots]$
$SimulationInstance == [v_gear, v_speed, v_throttle, v_up_th, v_down_th : \mathbb{R}]$
$InitSimulationInstance == [SimulationInstance\,' \mid \ldots]$
$InitState == (InitSimulationInstance) \land (InitSimulationData)$
$\ldots$
$Activate == (ActivateWithHistory \lor ActivateNoHistory) \land \Xi SimulationInstance$
$\ldots$
$Deactivate == [\Xi SimulationInstance;\ \Delta SimulationData \mid \ldots]$
state $shift_logic_state == SimulationInstance \land SimulationData$
$entryaction_downshifting \mathrel{\widehat=} executeentryaction.(s_downshifting) \longrightarrow$ **Skip**
$\ldots$
$entryactions \mathrel{\widehat=} entryaction_downshifting \,\Box \ldots \Box\, entryaction_steady_state$
$duringaction_downshifting \mathrel{\widehat=} executeduringaction.(s_downshifting)?ce \longrightarrow$ **Skip**
$\ldots$
$duringactions \mathrel{\widehat=} duringaction_downshifting \,\Box \ldots \Box\, duringaction_steady_state$
$exitaction_downshifting \mathrel{\widehat=} executeexitaction.(s_downshifting) \longrightarrow$ **Skip**
$\ldots$
$exitactions \mathrel{\widehat=} exitaction_downshifting \,\Box \ldots \Box\, exitaction_steady_state$
$conditionaction_third_fourth \mathrel{\widehat=} executeconditionaction.(t_third_fourth) \longrightarrow$ **Skip**
$\ldots$
$conditionactions \mathrel{\widehat=} conditionaction_third_fourth \,\Box \ldots \Box\, conditionaction_default_steady_state$
$transitionaction_third_fourth \mathrel{\widehat=} executetransitionaction.(t_third_fourth) \longrightarrow$ **Skip**
$\ldots$
$transitionactions \mathrel{\widehat=} transitionaction_third_fourth \,\Box \ldots transitionaction_steady_state_upshifting$
$condition_third_fourth \mathrel{\widehat=} evaluatecondition.(t_third_fourth)!(\mathbf{True}) \longrightarrow$ **Skip**
$\ldots$
$conditions \mathrel{\widehat=} condition_third_fourth \,\Box \ldots condition_steady_state_upshifting$
$trigger_default_first \mathrel{\widehat=} checktrigger.(t_default_first)?e \longrightarrow result.(t_default_first).(e)!(\mathbf{True}) \longrightarrow$ **Skip**
$\ldots$
$triggers \mathrel{\widehat=} trigger_third_fourth \,\Box \ldots \Box\, trigger_upshifting_steady_state26$
$getevents \mathrel{\widehat=} events!(\langle ENULL \rangle) \longrightarrow$ **Skip**
$getstate \mathrel{\widehat=} state?x : (x \in \mathrm{dom}(states))!(states(x)) \longrightarrow$ **Skip**
$getjunction \mathrel{\widehat=} junction?x : (x \in \mathrm{dom}(junctions))!(junctions(x)) \longrightarrow$ **Skip**
$gettransition \mathrel{\widehat=} transition?x : (x \in \mathrm{dom}(transitions))!(transitions(x)) \longrightarrow$ **Skip**
$getchart \mathrel{\widehat=} chart!(states(identifier)) \longrightarrow$ **Skip**
$status \mathrel{\widehat=} status?x : (x \in \mathrm{dom}(state_status))!(state_status(x)) \longrightarrow$ **Skip**
$history \mathrel{\widehat=} history?x : (x \in \mathrm{dom}(state_history))!(state_history(x)) \longrightarrow$ **Skip**
$activation \mathrel{\widehat=} activate?x \longrightarrow Activate$
$deactivation \mathrel{\widehat=} deactivate?x \longrightarrow Deactivate$
$ChartActions \mathrel{\widehat=} \ldots$
$InterfaceActions \mathrel{\widehat=} \ldots$
$Inputs \mathrel{\widehat=} \ldots$
$Outputs \mathrel{\widehat=} \ldots$
$AllActions \mathrel{\widehat=} \ldots$
$broadcast \mathrel{\widehat=} e : EVENT;\ dest : SID \bullet \ldots$
$check \mathrel{\widehat=}$ **res** $erl : \mathbb{B} \bullet \ldots$

$$\bullet\ InitState\ ;\ \mu X \bullet \left(\mu Y \left(\begin{array}{l} AllActions\ ;\ \ Y \\ \Box \\ end_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) ;\ X$$

end

Figure 3.9: Structure of the c_shift_logic process.

on, where $s_downshifting$ and s_gear_state are state identifiers, and $S_downshifting$ and S_gear_state are bindings of the schema *State*.

These identifiers and binding are constants that are defined outside the process and model the main objects of the Stateflow notation: states, transitions, junctions and events. For instance, for each state in the diagram, we define a constant of type *State* and a constant

of type SID; the name of the first is the name of the state prefixed by a capital S, and the name of the second is created in the same fashion, but prefixed by a lower case s. For state $gear_state$, the constant S_gear_state has type $State$ and the value of its components are defined by the following binding.

$$S_gear_state : State$$

$$\begin{aligned}
S_gear_state = \langle\!| & identifier == s_gear_state, default == default_first, \\
& inner == nulltransition.identifier, outer == nulltransition.identifier, \\
& parent == s_shift_logic, left == nullstate.identifier, \\
& right == s_selection_state, substates == \langle s_first, s_second, s_third, s_fourth \rangle, \\
& decomposition == CLUSTER, type == AND, history == \mathbf{False} |\!\rangle
\end{aligned}$$

This state's identifier is s_gear_state. It has a default transition whose identifier is $default_first$. It has no inner or outer transitions, no left sibling, but has a right sibling that has identifier $s_selection_state$. It has a sequence of substates and sequential decomposition ($CLUSTER$). It is a parallel (AND) state and has no history junction.

Identifiers and binding of transitions and junctions are defined in a similar way. The remaining identifier and binding definitions for our example can be found in Appendix C.1. Input and local events are defined as constants of type $EVENT$. For our example, the event identifiers are defined as follows.

$$e_UP, e_DOWN, ENULL : EVENT$$

As discussed in the previous section, at each cycle, the process *Simulator* executes the chart once for each input event that has occurred. If the chart does not have any input events, we declare the dummy event $ENULL$ to allow the *Simulator* to execute the chart.

The definition of the schema *SimulationData* is independent of a particular chart. It declares the state components of the chart process that record the status of each state in the chart ($state_status$), and the last active substate for those with a history junction ($state_history$). If $state_status\, n$ is true, the state n is active; $state_history$ is a function from state identifiers to state identifiers.

SimulationData

$$\begin{aligned}
& state_status : SID \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{B} \\
& state_history : SID \mathrel{+\!\!\!\!\twoheadrightarrow} SID
\end{aligned}$$

$$\begin{aligned}
& \mathrm{dom}\, state_status = \mathrm{dom}\, states \\
& \mathrm{dom}\, state_history = \{ j : \mathrm{ran}\, junctions \mid j.history = \mathbf{True} \bullet j.parent \} \\
& \forall s : \mathrm{ran}\, states \mid s.decomposition = CLUSTER \bullet \\
& \quad \#\{ ss : \mathrm{ran}\, s.substates \mid state_status(ss) = \mathbf{True} \} \leq 1
\end{aligned}$$

The predicate of *SimulationData* states that the domain of $state_status$ contains the iden-

tifiers of all states, that the domain of *state_history* contains the identifiers of all parent states of history junctions, and that for all states with a sequential decomposition, at most one substate can be active at any given time. The (omitted) initialisation operation *InitSimulationData* for *SimulationData* marks all states as inactive, and associates all states in the domain of *state_history* with the identifier of the null state.

The schema *SimulationInstance* (in Figure 3.9) declares the data and output events defined in the chart. For each data, it declares a variable of the same type whose name is the name of the data prefixed by *v_*. For each output event, the schema declares a variable of type $\mathbb{N}$ whose name is the name of the event prefixed by *counter_*.

For our example, there are no output events, and the declared data are *v_gear*, *v_speed*, *v_throttle*, *v_up_th*, *v_down_th*. The initialisation action *InitSimulationInstance* assigns to them the default values of their types; in this case, it sets them to 0.

In summary, the state of a chart process includes all the components declared in *SimulationInstance*, which defines the data declared in the chart, and in *SimulationData*, which records the status of the states and history junctions.

The operations that initialise the state of the process, and activate and deactivate Stateflow states are defined by the schemas *InitState*, *Activate* and *Deactivate*. *InitState* is defined as the conjunction of the operations, *InitSimulationInstance* and *InitSimulationData*, that initialise the two schemas that define the state of the process. *Activate* marks a state as active, and is defined as the schema operation that does not change the components of *SimulationInstance* (Ξ*SimulationInstance*), and activates the state differently according to whether or not it has a history junction (*ActivateWithHistory* $\lor$ *ActivateNoHistory*). The schema *Deactivate* that marks a state as inactive is defined similarly. Their definitions are available in Appendix C.

The main action of a chart process initialises its state and recursively offers a choice of actions that are selected by the process *Simulator* through the channels in *interface*. The choice of actions available is encoded in the *Circus* action *AllActions*. As shown below, *AllActions* is defined as the external choice of actions called *conditions*, *triggers*, *Inputs*, *Outputs*, *ChartActions*, and *InterfaceActions*.

$$AllActions \mathrel{\widehat=} (conditions \mathrel{\Box} triggers \mathrel{\Box} Inputs \mathrel{\Box} Outputs \mathrel{\Box} ChartActions \mathrel{\Box} InterfaceActions)$$

The *Circus* action *conditions* offers the choice of all the actions that encode the evaluation of a condition in a transition. The *Circus* action *triggers* similarly offers the actions encoding the evaluation of a trigger. The *Circus* actions *Inputs* and *Outputs* offer the possibility of reading the inputs or writing the outputs of the chart. The *Circus* action *ChartActions* offers the choice of all the state and transition actions, and once the selected action is completed, it synchronises on *end_action* signalling that the chart action has finished; this is necessary because the simulator waits for a local event broadcast or the end of the chart action. Finally, *InterfaceActions* offers the actions used by the *Simulator* to obtain information about the chart.

The evaluation of a condition is specified as a *Circus* action that communicates true through the channel *evaluatecondition* if the condition holds, and false otherwise. For example, the condition of the transition between the states steady_state and downshifting (of Figure 3.2) is defined as follows.

$$
\begin{array}{l}
condition_steady_state_downshifting \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
\mathbf{if}\ (v_speed <_{\mathcal{A}} v_down_th) \neq 0 \longrightarrow \\
\qquad evaluatecondition.t_steady_state_downshifting!\mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ \neg\ ((v_speed <_{\mathcal{A}} v_down_th) \neq 0) \longrightarrow \\
\qquad evaluatecondition.t_steady_state_downshifting!\mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

First, the predicate $(v_speed <_{\mathcal{A}} v_down_th) \neq 0$ is evaluated; if it is true, the action offers a synchronisation on *evaluatecondition* with the identifier of the condition's transition ($t_steady_state_downshifting$) and the boolean value **True**. Otherwise, it offers a communication with **False**. The operation $<_{\mathcal{A}}$ returns 0 if the first operand is not less than the second operand, otherwise, it returns a number different than 0.

The evaluation of a transition trigger is defined by a *Circus* action that waits for the communication of an event e along with the transition identifier through *checktrigger*, and then evaluates the trigger with respect to e. Afterwards, it communicates the identifier, the event, and the boolean result of the evaluation through *result*. For the trigger of the transition between the states first and second, we have the following action.

$$
\begin{array}{l}
trigger_first_second \mathrel{\widehat=} checktrigger.t_first_second?e \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ e = e_UP \longrightarrow result.t_first_second.e!\mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ \neg\ (e = e_UP) \longrightarrow result.t_first_second.e!\mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

This action communicates **True** if the event received through the channel *checktrigger* is e_UP, and **False** otherwise.

The *Circus* action *Inputs* waits for a synchronisation on *read_inputs* and reads the values of the input variables in interleaving. A channel of the appropriate type is declared for each input and output data, as well as for each output event. In our example, there are two input variables, *v_speed* and *v_throttle*, and their values are communicated through the channels *i_speed* and *i_throttle*.

$$
Inputs \mathrel{\widehat=} read_inputs \longrightarrow \left(\begin{array}{c}
i_speed?x \longrightarrow v_speed := x \\
|\!|\![\{v_speed\} \mid \{v_throttle\}]\!|\!| \\
i_throttle?x \longrightarrow v_throttle := x
\end{array}\right)
$$

The interleaving requires the partitioning of the state components that are written to by each parallel action to avoid race conditions. This interleaving terminates when all the input variables are read and recorded in the appropriate state component.

The *Circus* action *Outputs* is also defined by a communication followed by an interleaving (if necessary). Each interleaved action communicates an output variable or an output event. For each output event, the action that communicates it first checks the counter associated to that event. If the counter is positive, the actions decrements it, and communicates **True** through the appropriate channel. If the counter is zero, the value **False** is communicated. For our example, there are no output events, and the only output variable in the example is v_gear; therefore there is no need for an interleaving. The action $Output$, for our example, is as follows.

$$Outputs \mathrel{\widehat=} write_outputs \longrightarrow o_gear!(v_gear) \longrightarrow \mathbf{Skip}$$

The action $ChartActions$ offers a choice between the actions $entryactions$, $duringactions$, $exitactions$, $conditionactions$, and $transitionactions$, which are all defined as the external choice of all the actions of the appropriate type. For example, $entryactions$ is defined by an external choice of all the *Circus* actions that model the execution of an entry action. In our example, the state first, for instance, has an entry action that consists of assigning the value 1 to the variable gear; this is modelled by the following *Circus* action.

$$entryaction_first \mathrel{\widehat=} executeentryaction.(s_first) \longrightarrow v_gear := 1$$

This action waits for the communication of the state's identifier on $executeentryaction$, and assigns the value 1 to the state component v_gear. As previously discussed, the execution of during actions requires information about the current event. This is supplied to the chart process as an additional value communicated by the channel $executeduringaction$.

The same approach is taken for the other types of actions. For example, the condition action of the first transition from upshifting to steady_state is modelled as follows.

$$\begin{array}{l} conditionaction_upshifting_steady_state26 \mathrel{\widehat=} \\ \quad executeconditionaction.(t_upshifting_steady_state26) \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{var}\ b : \mathbb{B} \bullet broadcast(e_UP, s_gear_state)\ ;\ check(b); \\ \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ \neg\ (b = \mathbf{True}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right) \end{array} \right) \end{array}$$

The number postfixed to the transition name uniquely identifies it, as there are two transitions with the same source and destination states. The action waits for the communication of the transition identifier through $executeconditionaction$, broadcasts the event e_UP to the state s_gear_state, and waits for the simulator to check the appropriate early return logic conditions by calling the action $check$. The local variable b is used to store the result of the early return logic condition. Finally, an alternation decides to proceed ($b = \mathbf{False}$) or terminate ($b = \mathbf{True}$). In our example, there are no actions after the broadcast, and the alternation immediately terminates in both cases. The action $check$, therefore, must

always be called after a call to *broadcast* as it offers the process *Simulator* the information necessary to perform the verification of the early return logic conditions.

A local event broadcast can be directed at the whole chart or specific states. In the shift logic example, the local event broadcasts are used to trigger a change of state within the parallel state gear_state.

The *Circus* action *broadcast* that specifies the broadcasting mechanism takes two parameters: the local event e and the identifier of the destination state $dest$.

$$broadcast \mathrel{\widehat=} e : EVENT;\ dest : SID \bullet$$
$$local_event!(e, states(dest)) \longrightarrow \mu X \bullet \left(\begin{array}{l}(AllActions\ ;\ X)\\ \Box\\ (end_local_execution \longrightarrow \mathbf{Skip})\end{array}\right)$$

First, the action *broadcast* communicates e and the destination state $states(dest)$ through the channel *local_event*. Next, it recursively offers a choice between *AllActions* followed by a recursive call to X, and a terminating action that waits for a synchronisation on the channel *end_local_execution*, and terminates the recursion.

Every call to *broadcast* is followed by a call to the action *check* that takes a result parameter *erl* of type boolean, and recursively offers the option either to execute *InterfaceActions* and recurse, or to terminate the recursion by reading a value on *interrupt* and assigning it to *erl*. The actions in *InterfaceActions* are offered to allow the simulator to inspect the state of the chart and decide whether the early return logic conditions hold or not. Once the simulator has finished checking the conditions, it communicates **True** or **False** on the channel *interrupt*, thus, terminating the recursion.

$$check \mathrel{\widehat=} \mathbf{res}\ erl : \mathbb{B} \bullet \mu X \bullet (InterfaceAction\ ;\ X \Box interrupt?x \longrightarrow erl := x)$$

Whenever the local event broadcast is followed by another action, an alternation decides whether to proceed with the execution or not based on the value assigned by *check* to a local variable b.

The only during action in the chart in Figure 3.2 involves the use of the Simulink function *calc_th*. This is specified as the application of a Z function with the same name whose definition must be provided in a separate Z library. The *Circus* action that models these during action waits for the communication of the state identifier, *s_selection_state*, through *executeduringaction* and executes the assignment.

$$duringaction_selection_state \mathrel{\widehat=} executeduringaction.(s_selection_state)?ce \longrightarrow$$
$$\mathbf{var}\ _aux : \mathbb{R} \times \mathbb{R} \bullet$$
$$(_aux := calc_th(v_gear, v_throttle)\ ;\ v_down_th := _aux.1\ ;\ v_up_th := _aux.2)$$

Since *calc_th* returns a pair whose elements are assigned to a vector formed by two components of the process state, we define an auxiliary variable *_aux*, assign the pair to it, and then assign the values in *_aux* to the appropriate state components. More precisely, the

first value of $_aux$, that is, $_aux.1$ is assigned to v_down_th, and the second value ($_aux.2$) is assigned to v_up_th.

The action *InterfaceActions* offers a choice of actions that are the same for every chart.

$$InterfaceActions \mathrel{\widehat=} \left(\begin{array}{l} getevents \mathrel{\Box} getchart \mathrel{\Box} getstate \mathrel{\Box} getjunction \mathrel{\Box} gettransition \\ \Box status \mathrel{\Box} history \mathrel{\Box} activation \mathrel{\Box} deactivation \end{array} \right)$$

These *Circus* actions allow the *Simulator* to recover the bindings that specify objects of the chart using their identifiers, obtain information about the status of a state, or about the history junction of a state, and activate and deactivate a state.

The action *getstate*, for example, receives a state identifier and outputs the corresponding state, and the action *getevents* outputs the sequence of input events of the chart. The order of the input events is important because the simulator executes the chart once for each input event that has occurred in a time step, in the order specified in the chart and recorded in the textual representation of the chart.

Finally, the main action of the chart process consists of two nested recursions. The inner recursion repeatedly offers a choice between *AllActions* and a synchronisation on end_cycle, which terminates the inner recursion, and marks the end of an execution step.

In this section, we presented a brief overview of the model of the chart process of the shift logic example. The complete model of this example can be found in Appendix C.1.

3.4 Validation

As previously mentioned in Chapter 1, we identify three main avenues for evaluating and validating our models: inspection, testing and refinement.

The model of the simulator presented in Section 3.2 was developed with evaluation by inspection in mind. We maintained in our specification the granularity of the informal description, associating the *Circus* actions of the process *Simulator* to steps or clearly identified parts of steps in the informal description by means of naming conventions. For instance, in the specification of the process of entering a state, this correspondence is made explicit by postfixing the name of the action with the number of the step it models.

The second form of validation was carried by simulating our models. Since there is no simulation tool for *Circus* we have translated the *Circus* models of Stateflow charts to CSP, and used tools such as `ProBE` [29], `ProB` [54] and `FDR2` [30] to simulate and analyse the models.

State components are modelled in CSP as one one-place buffers: every assignment is converted to a communication, and every use of a state component must be preceded by a communication that reads the value of the state component and puts it in scope. This significantly increases the amount of communication in the model, restricting further the complexity of models that can be simulated using the available tools.

While the translation of simple models is straightforward, it does not scale for more

complex models, and the (informal) link between the specification and the informal semantics is lost in the translation. Because of this and due to the support of verification of implementations, as well as the availability of a refinement calculus, *Circus* is still a better choice for modelling Stateflow charts. Nevertheless, better tool support is necessary.

We used `ProBE` and `ProB` to simulate the models and compare the results of the simulation to the execution of the chart in the Stateflow toolbox; `FDR2` and `ProB` were used to model check our models for interesting properties such as deadlock-freedom, divergence-freedom and determinism. It is worth mentioning, that for larger examples, `ProB` performed significantly better than `ProBE` and `FDR2` both when executing the model and verifying properties.

Finally, the third form of validation was performed by applying the refinement calculus to the verification of the implementation of small examples, and describing a refinement strategy for the verification of parallel implementation of Stateflow charts (presented in Chapter 5). With the support of appropriate tools[3] this validation can be strengthened by applying our refinement strategy to larger examples and industrial case-studies.

The evaluation of our models (and similar models) can be performed using either of the three approaches mentioned. However, the easiest method is the inspections of the models. For this reason, it is important that any formalisation that follows our approach take this into consideration and document the link between the formal and informal descriptions.

3.5 Final considerations

In this chapter, we gave an overview of our models of Stateflow chart, and discussed in further details the two main components of these models: the simulator process and the chart process. These processes are composed in parallel to obtain a process that represents the simulation of the given chart. The main action of the chart process offers the structure and the actions of the chart, and the main action of the process *Simulator* waits for an event and executes a step of the simulation.

As previously mentioned, the external channels of our Stateflow models include one channel for each input and output data, one channel for each output event, *input_event* and *end_cycle*. The channel *input_event* triggers the execution of a cycle of the chart, and the channel *end_cycle* marks the end of a cycle.

The separation between the execution of charts, and the structure of a particular chart (including its actions) leads to a simpler set of translation rules since the complex semantics of Stateflow charts is almost completely isolated in the process *Simulator*. Moreover, when updating our models to reflect changes in the semantics of Stateflow, these changes can potentially be restricted to the process *Simulator* and have minimal impact on the translation rules.

[3]We have developed a (incomplete) prototype tool to support the application of basic refinement laws. While this was helpful in the beginning of the validation process, it soon became clear that for any practical application, a more complete and robust tool is necessary.

Since it is not possible to formally verify our models with respect to the simulator embedded in Stateflow, we have to rely partially on inspection and peer-reviewing to validate the models. In this context, it is important to manage the complexity of our models to facilitate the validation. The separation between chart and simulator is intuitive and supports a divide-and-conquer approach to validation.

The model described in this chapter extends those presented in [68, 69, 70]. In particular, it provides a more complete account for the possibility of early return logic in local event broadcasts. This model covers aspect that were left untreated on other works, such as history junctions, local events and inter-level transitions. Other aspects, in particular the interaction with Simulink blocks, are not yet formalised. The complete specification of the process *Simulator* and the supporting definitions can be found in Appendix B. The complete model of the chart in Figure 3.2 can be found in Appendix C.1.

In the next chapter, we discuss the formalisation of a translation strategy that automatically generates models of Stateflow charts, such as the one presented in Section 3.3.

Chapter 4

Translation strategy: from Stateflow charts to *Circus*

Having presented a formal model of Stateflow charts in the previous chapter, we now turn to the formalisation and implementation of a translation strategy. Since the model presented in Chapter 3 segregates the chart-specific information from the general encoding of the semantics of Stateflow (modelled in the process *Simulator*), we only need to concern ourselves with the translation of the former. Furthermore, since most of the complexity of the semantics of Stateflow is isolated in the process *Simulator*, the translation of Stateflow chart can be described by means of simpler rules, which can be more easily validated. As a means for further validation, we have formalised the translation rules in the Z notation.

While the chart process is simpler than the process *Simulator*, the translation rules are by no means trivial. In the interest of conciseness, we present an overview of the translation strategy, and focus on the most interesting aspects of the formalisation. The complete formalisation of the translation rules can be found in [67].

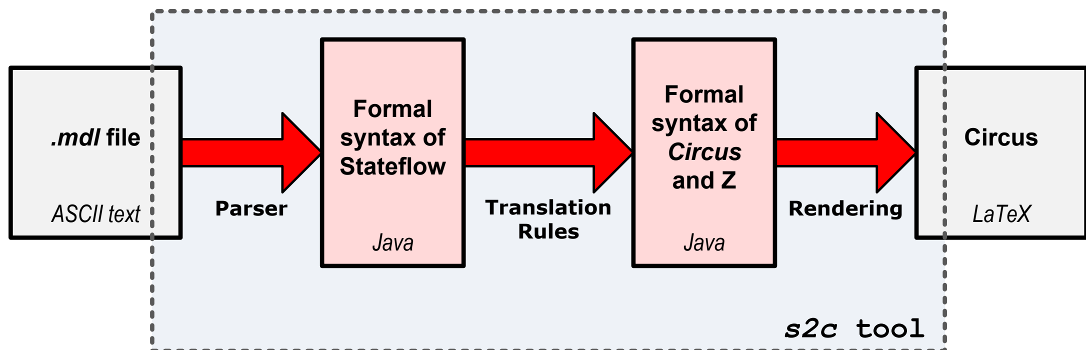


Figure 4.1: Translation strategy: overview.

In this chapter, we explain and formalise the algebraic strategy devised in order to translate a Stateflow chart into a *Circus* model. The translation process is implemented by the *s2c* tool, which reflects the steps of the formalised strategy and is depicted in

Figure 4.1. The translation starts with a textual representation of the chart (in the form of a *.mdl* file), which is parsed to produce an intermediate representation of the chart (based on our formalisation of the syntax of Stateflow). Next, our formal translation rules (as implemented in the tool) are applied to the intermediate representation to produce a *Circus* model (encoded in our formalisation of the syntax of *Circus* and Z). Finally, the *Circus* model is rendered in LaTeX by direct translation.

The choice of Z as the notations for the specification of the translation rules stems from our familiarity with the notation and the availability of tools that support parsing, type checking and verification. While we did not perform any verification beyond type checking, the formal nature of the specification opens the possibility for the verification of properties of the verification strategy. It is worth noting that other (perhaps more) suitable notations exist, but for the purposes of this work, we believe any benefits would be overshadowed by the effort of re-specification and re-validation.

This chapter is organised as follows. The first three sections discuss the formalisation of the translation strategy: Section 4.1 introduces the formal syntax of Stateflow charts that is used as a basis for formalisation of the translation rules, Section 4.2 discusses the well-formedness conditions that characterise valid charts and Section 4.3 presents the translation rules used to produce the *Circus* models of charts. Section 4.4 discusses the implementation of the translation rules previously presented, and in Section 4.5, we discuss the evaluation and validation of the translation rules and *s2c* tool (Section 4.5) before concluding the chapter with a discussion of results and limitations (Section 4.6).

4.1 Syntax of Stateflow charts

In this section, we explain the formalisation of the syntax of Stateflow charts that is used as a basis for the definition of the translation rules and implementation of the translation tool. The syntax is formalised in Z [109], and basic knowledge of the Z notation is assumed.

The syntax of Stateflow charts is captured by seven main groups of objects: charts, states, junctions, transitions, functions, events and data. Supporting the definition of some of these objects are identifiers, names, expressions and actions. Section 4.1.1 discusses names and identifiers, Section 4.1.2 presents the syntax of expressions and actions, and Section 4.1.3 describes the syntax of the main objects previously mentioned.

4.1.1 Names and Identifiers

Every Stateflow object contains an identifier that makes it unique among all objects defined in a *.mdl* file. Moreover, some objects contain names that are used as references in actions and expressions. The objects that are named are charts, states, functions, events and data.

We define the set $NAME$ of names (as a given set) as well as five subsets that correspond to each group of objects that are given names in Stateflow. The set of names of charts, states, functions, events and data are disjoint sets of names.

$$
\begin{array}{|l}
CNAME, SNAME, FNAME, ENAME, DNAME : \mathbb{P}\, NAME \\
\hline
\text{disjoint}\langle CNAME, SNAME, FNAME, ENAME, DNAME \rangle
\end{array}
$$

We define the sets of identifiers of each of the groups of objects in the same way we defined the sets of names. We must also include sets of identifiers of junctions and transitions (which, while having an identifier, have no name).

$$
\begin{array}{|l}
CID, SID, JID, TID, EID, DID, FID : \mathbb{P}\, \mathbb{N} \\
\hline
\text{disjoint}\langle CID, SID, JID, TID, EID, DID, FID \rangle
\end{array}
$$

Names and identifiers play an important role in allowing references to be made both in the encoding in the *.mdl* file and in the expressions and actions used within certain objects. For example, in a *.mdl* file, the chart associated with a state is referred to by its identifier, while a piece of data is referred to by its name in an action or expression.

4.1.2 Expressions and Actions

One of the most basic elements of the Stateflow notation are expressions, which are elements of the notation that are associated with a value. An expression can be a variable name, a scalar value, a vector, a matrix, a function application, an operator applied to one or more expressions, or a temporal expression.

The set *EXPR* of expressions is defined as a free type – a Z construct that supports the definition of a set by specifying its elements in terms of constants and constructor functions. Free types and schemas are the constructs of Z commonly used to model syntax. The free type definition of the set *EXPR* is partially shown below. The constructors *not*, *and*, and *or* are used to build logical expressions from other expressions or pairs of expressions.

$$EXPR \quad ::= \quad \ldots \mid not\langle\langle EXPR \rangle\rangle \mid and\langle\langle EXPR \times EXPR \rangle\rangle \mid or\langle\langle EXPR \times EXPR \rangle\rangle \mid \ldots$$

It is worth noticing that expressions which refer to Stateflow objects always do so by name, rather than by identifier. For example, the constructor *in* builds an expression from a state name which returns 1 if the state referred by the expression is active, and 0 otherwise.

$$in\langle\langle SNAME \rangle\rangle$$

There is a special set of expressions called temporal expressions; they are built by the following constructors:

$$
\begin{array}{l}
after\langle\langle EXPR \times ENAME \rangle\rangle \mid before\langle\langle EXPR \times ENAME \rangle\rangle \mid at\langle\langle EXPR \times ENAME \rangle\rangle \mid \\
every\langle\langle EXPR \times ENAME \rangle\rangle \mid tempCount\langle\langle ENAME \rangle\rangle \mid
\end{array}
$$

The set of temporal expressions is the range of the constructors of those expressions.

$$TEXPR == \operatorname{ran} after \cup \operatorname{ran} before \cup \operatorname{ran} at \cup \operatorname{ran} every \cup \operatorname{ran} tempCount$$

The complete definition of the set *EXPR* can be found in [65].

Actions, in general, alter the state of the system and do not have an associated value. A Stateflow action can be the broadcasting of an event, the execution of an expression, the assignment of an expression to a variable, or the assignment of an expression to a vector of variables.

$$\begin{aligned} ACTION \quad ::= \quad & bcast\langle\langle ENAME\rangle\rangle \mid \\ & expr\langle\langle EXPR\rangle\rangle \mid \\ & assign\langle\langle DNAME \times \operatorname{seq} EXPR \times EXPR\rangle\rangle \mid \\ & massign\langle\langle \operatorname{seq} DNAME \times EXPR\rangle\rangle \end{aligned}$$

Multiple assignment actions, represented above using the constructor *massign*, only occur when assigning the return value of a function with multiple output data.

In general, an expression being used as an action can be ignored, since it does not affect the state of the chart. However, the evaluation of certain expressions can trigger changes to the chart. In particular, a function can be defined to alter the value of data in the chart that contains the function. In this case, we cannot ignore the evaluation of the expression, as this could change the behaviour of the chart. Our current treatment of functions does not cover the modelling of functions which change data values because they are specified externally to the chart process, and therefore do not have access to its variables. In Chapter 6, we discuss how this limitation can be overcome.

A derived type of action is an *on* action, which associates an action to an event name.

$$ONACTION == ENAME \times \operatorname{seq} ACTION$$

As already mentioned, expressions and actions are used in the definition of the main objects of Stateflow, which are discussed next.

4.1.3 Stateflow objects

The main objects of the language of Stateflow charts are states, junctions, transitions, events, data, functions and charts. These are the objects used to build charts; the expressions and actions (previously defined) are used to complement the behaviour of some of these objects (states and transitions).

4.1.3.1 State

A state contains an identifier, a name, a reference to its parent, information about its type and decomposition, entry, during and exit actions, and a list of bound variables and events. While the decomposition of a state refers to the organisation of its sub-states, the type of a state refers to the way it is organised with respect to its siblings.

While the terms used to represent both type and decompositions are the same, they encode different facts. A state with a decomposition of type *SEQ* contains sequential substates, while a state with a type *SEQ* is the substate of a state with sequential decomposition.

$$\begin{aligned} DECOMP &::= SEQ \mid PAR \\ STYPE &== DECOMP \end{aligned}$$

The type *State* is defined as a Z schema that includes all the components needed to characterise a state as described above.

State

$identifier : SID;\ name : SNAME$
$parent : CID \cup SID$
$decomp : DECOMP$
$type : STYPE$
$entry, exit : \mathrm{seq}\, ACTION$
$during : \mathrm{seq}\, DACTION$
$binding : \mathrm{seq}(ENAME \cup DNAME)$
$history : \mathbb{B}$
$default, inner, outer : \ \mathrm{opt}\ TID$
$left, right : \ \mathrm{opt}\, SID$
$substates : \mathrm{seq}\, SID$

The parent of a state can be either a state or a chart, so the set of possible identifiers of a parent is $CID \cup SID$. Some of the information contained in a state are optional, e.g. entry and exit actions. Unlike entry and exit actions, a during action can be either a regular action or an *on* action. We define the type of during actions as follows.

$$DACTION ::= action\langle\langle \mathrm{seq}\, ACTION \rangle\rangle \mid on\langle\langle ONACTION \rangle\rangle$$

The components *entry* and *exit* of the schema *State* are sequences of actions, and the component *during* is a sequence of during actions. Notice that some of the components have an opt type; components of this type are optional components, that is, they may or may not have a value (of the appropriate set) associated with them. The opt type is defined as the set of all sequences of size one plus the empty sequence.

4.1.3.2 Junction

A junction has an identifier, a reference to its parent and the first outgoing transition, and an indication of whether or not it is a history junction.

Junction

$$
\begin{array}{l}
identifier : JID \\
parent : CID \cup SID \\
transition : \text{opt } TID \\
history : \mathbb{B}
\end{array}
$$

The schema *Junction* represents two types of junctions: connective junctions and history junctions; this difference is established by the component *history*. If the parent of a junction is the chart or a state with parallel decomposition, then it should not be a history junction. Well-formedness conditions like this are formalised in [65] and are discussed in Section 4.2.

4.1.3.3 Transition

A transition has a sequence of triggers, a condition, a condition action and a transition action.

Transition

$$
\begin{array}{l}
identifier : TID;\ parent : CID \cup SID \\
source : \text{opt } (SID \cup JID) \\
destination : SID \cup JID \\
trigger : \text{seq } TRIGGER \\
condition : \text{opt } EXPR \\
condact, transact : \text{seq } ACTION \\
next : \text{opt } TID
\end{array}
$$

Default transitions do not have source states, thus, the component *source* takes an optional value ranging over the set of state and junction identifiers. A trigger is either an event (which must be an input or local event) or a temporal expression.

$$TRIGGER ::= e\langle\langle ENAME \rangle\rangle \mid t\langle\langle TEXPR \rangle\rangle$$

As in the case of states, condition and transition actions are recorded as sequences of actions. Finally, the component *next* points to the next transition in a list of transitions.

4.1.3.4 Event

An event has a trigger type and scope. The former can be rising edge, falling edge, either edge or function call, whereas the latter can be local, input or output.

$$\begin{aligned} EVENTTRIGGER &::= RISINGEDGE \mid FALLINGEDGE \mid \\ &\quad EITHEREDGE \mid FUNCTIONCALL \\ EVENTSCOPE &::= LOCALEVENT \mid INPUTEVENT \mid OUTPUTEVENT \end{aligned}$$

The representation of an event consists of an identifier and a name, a reference to its parent, and information about its scope and trigger type.

Event

$identifier : EID;\ name : ENAME$
$parent : CID \cup SID$
$scope : EVENTSCOPE;\ trigger : EVENTTRIGGER$

4.1.3.5 Data

Data has a larger choice of scope; it can be local, input, output, constant, parameter, or data store memory. We restrict the type of the data to scalar types and multi-dimensional vectors of scalar types. Other possibilities are fixed-point and enumerated types.

$$\begin{aligned} DATASCOPE &::= LOCALDATA \mid INPUTDATA \mid OUTPUTDATA \mid \\ &\quad CONSTANTDATA \mid PARAMETERDATA \mid \\ &\quad DATASTOREMEMORYDATA \\ DATATYPE &::= scalar\langle\langle SCALAR\rangle\rangle \mid vector\langle\langle SCALAR \times \operatorname{seq} \mathbb{N}\rangle\rangle \\ SCALAR &::= BOOLEAN \mid DOUBLE \mid SINGLE \mid INT32 \mid INT16 \mid \\ &\quad INT8 \mid UINT32 \mid UINT16 \mid UINT8 \end{aligned}$$

A vector is characterised by a sequence of natural numbers; they encode the number of dimensions (size of the list) and the size of each dimension. The scalar types available are boolean, floating point numbers, signed integers and unsigned integers.

A piece of data consists of an identifier and a name, a reference to its parent, and information about its scope and type.

Data

$identifier : DID;\ name : DNAME$
$parent : CID \cup SID$
$scope : DATASCOPE;\ type : DATATYPE$

4.1.3.6 Functions

A Simulink function consists of an identifier and a name, a sequence of names of input variables, a sequence of names of output variables, a reference to its parent, and the name of the Simulink block that defines the function.

SimulinkFunction

$identifier : FID; \ name : FNAME$
$inputs, outputs : \operatorname{seq} DNAME$
$parent : CID \cup SID$
$block : NAME$

A graphical function is similar to a Simulink function, but instead of a block name, it includes an identifier of a chart as a component. The chart to which the function refers is obtained from the flowchart that defines the function. The schema *GraphicalFunction* can be found in [65].

4.1.3.7 Chart

The top object of a Stateflow model is a chart. It contains an identifier, a name, the identifier of the first default transition (if one exists), a sequence of state identifiers, its decomposition, and partial functions from identifier to the objects contained in the chart.

Chart

$identifier : CID$
$name : CNAME$
$default : \operatorname{opt} TID$
$substates : \operatorname{seq} SID$
$decomp : DECOMP$
$states : SID \pfun State$
$junctions : JID \pfun Junction$
$transitions : TID \pfun Transition$
$events : EID \pfun Event$
$data : DID \pfun Data$
$sfunctions : FID \pfun SimulinkFunction; \ gfunctions : FID \pfun GraphicalFunction$

The *default* component is a reference to the first default transition of the chart; its type is opt *TID* since this reference is optional because a chart with parallel decomposition or an unambiguous start point does not need to define a default transition.

There are two levels of representation in a chart: the chart as a container, and a chart as an object in itself. The latter perspective is associated with the first five components, whereas the former is associated with the remaining components.

4.2 Well-formedness conditions

In this section, we briefly discuss the well-formedness conditions for the syntax of the Stateflow notation. These conditions are characterised by the definition of sets of well formed expressions and objects, culminating in the definition of the set of well formed charts. The well-formedness conditions are by no mean complete; they simply characterise the minimal properties that must hold for the translation rules to applied successfully. The translation strategy assumes that the chart is accepted by the Stateflow tool as a valid chart.

The functions *snames*, *dnames*, *enames*, *sfnames* and *gfnames* are used to extract information about names of states, data, events, Simulink functions and graphical functions from a chart. They are defined in [65].

We further define three sets, *unary*, *binary* and *tempbinary*, of expressions that represent, respectively, unary expression, binary expressions, and temporal binary expressions. The well-formedness conditions over expressions are defined in terms of these sets in order to simplify the specification.

Well-formedness of expressions is characterised inductively over the constructors of expressions, and always depends on a chart. For instance, a name expression is well formed with respect to a chart if, and only if, its name component corresponds to a piece of data in the chart, and the expressions in the index sequence are well formed with respect to the chart.

$$WF_EXPR : Chart \leftrightarrow EXPR$$

$$\begin{array}{l}
\forall c : Chart;\ n : DNAME;\ s : \mathrm{seq}\, EXPR \bullet (c, name(n, s)) \in WF_EXPR \Leftrightarrow \\
\quad n \in dnames(c) \land (\forall e : \mathrm{ran}\, s \bullet (c, e) \in WF_EXPR) \\
\forall c : Chart;\ n : NAME;\ s : \mathrm{seq}\, EXPR \bullet (c, fun(n, s)) \in WF_EXPR \Leftrightarrow \\
\quad n \in sfnames(c) \cup gfnames(c) \land (\forall e : \mathrm{ran}\, s \bullet (c, e) \in WF_EXPR) \\
\forall c : Chart;\ a : \mathbb{A} \bullet (c, value(a)) \in WF_EXPR \\
\forall c : Chart;\ s : \mathrm{seq}_1\, \mathbb{A} \mid \#\, s > 1 \bullet (c, array(s)) \in WF_EXPR \\
\forall c : Chart;\ s : \mathrm{seq}_1\, \mathrm{seq}_1\, \mathbb{A} \mid \#\, s > 1 \bullet \\
\quad (c, matrix(s)) \in WF_EXPR \Leftrightarrow \exists\, n : \mathbb{N} \mid n > 1 \bullet \forall\, row : \mathrm{ran}\ s \bullet \#\, row = n \\
\forall c : Chart;\ e : EXPR;\ op : unary \bullet (c, op(e)) \in WF_EXPR \Leftrightarrow (c, e) \in WF_EXPR \\
\forall c : Chart;\ e_1, e_2 : EXPR;\ op : binary \bullet \\
\quad (c, op(e_1, e_2)) \in WF_EXPR \Leftrightarrow (c, e_1) \in WF_EXPR \land (c, e_2) \in WF_EXPR \\
\forall c : Chart;\ s : SNAME \bullet (c, in(s)) \in WF_EXPR \Leftrightarrow s \in snames(c) \\
\forall c : Chart;\ e : EXPR;\ n : ENAME;\ op : tempbinary \bullet \\
\quad (c, op(e, n)) \in WF_EXPR \Leftrightarrow (c, e) \in WF_EXPR \land n \in enames(c) \\
\forall c : Chart;\ n : ENAME \bullet (c, tempCount(n)) \in WF_EXPR \Leftrightarrow n \in enames(c) \forall
\end{array}$$

A function application is well formed if, and only if, the name corresponds to a Simulink or graphical function in the chart, and the expressions that form the sequence of indexes

are well formed. A value expression is always well formed, and an array of values is well formed if, and only if, its size is greater than one. A matrix is well formed if, and only if, all the rows have the same size and the number of rows and columns is greater than one. The application of a unary operator to an expression is well formed if, and only if, the expression is well formed; the application of a binary operator to two expressions is well formed if, and only if, both expressions are well formed. The application of a binary temporal operator to an expression and an event name is well formed if, and only if, the expression is well formed and the name corresponds to an event. An expression formed by applying *tempCount* to a name is well formed if, and only if, the name corresponds to an event.

Similarly to expressions, we define the well-formedness of actions inductively.

$$
\begin{array}{l}
WF_ACTION : Chart \leftrightarrow ACTION \\
\hline
\forall c : Chart;\ e : Event;\ d : (SID \cup CID) \bullet (c, bcast(e.name, d)) \in WF_ACTION \Leftrightarrow \\
\quad (e.scope \neq INPUTEVENT \land e \in \operatorname{ran} c.events \land \\
\quad\quad (d \in (\operatorname{dom} c.states) \lor d = c.identifier)) \\
\forall c : Chart;\ e : EXPR \bullet (c, expr(e)) \in WF_ACTION \Leftrightarrow (c, e) \in WF_EXPR \\
\forall c : Chart;\ n : DNAME;\ s : \operatorname{seq} EXPR;\ e : EXPR \bullet \\
\quad (c, assign(n, s, e)) \in WF_ACTION \Leftrightarrow \\
\quad (n \in dnames(c) \land (\forall e : \operatorname{ran} s \bullet (c, e) \in WF_EXPR) \land (c, e) \in WF_EXPR)
\end{array}
$$

A broadcast action is well formed if, and only if, the name corresponds to a local or output event, and the destination is either the identifier of the chart or the identifier of one of its states ($d = c.identifier \lor d \in \operatorname{dom} states$). An action formed by an expression is well formed if, and only if, the expression is well formed. An assignment is well formed if, and only if, the assigned name corresponds to a piece of data, the expressions that form the sequence of indexes are well formed, and the expression being assigned is also well formed.

Well-formedness conditions for the other syntactic categories are formalised in [65]. Like for expressions and actions, the definitions are relatively straight forward an excluded from this thesis in the interest of conciseness.

Finally, a chart is well formed if, and only if, each of its states, junctions, transitions, data, events, graphical functions and Simulink functions is well formed.

4.3 Translation strategy in Z

The translation strategy devised for obtaining *Circus* models of Stateflow charts is defined in a top-down fashion and is formalised as a set of translation rules. The translation process starts by the application of one translation rule which uses other translation rules to treat specific features, and these rules potentially rely on other translation rules, and so on. Here, we present and exemplify the main groups of translation rules (the complete formalisation can be found in [67]).

In order to support the formalisation of the translation rules, we formalised the syntax of *Circus* and Z[1] in a similar fashion to our formal syntax of Stateflow charts. The formalisation consists of a set of constructs which are used to build elements of the *Circus* notations, such as actions and processes, as well as elements of the Z notations, such as free types and schemas. Because the rules are fully formalised, we are able to parse and type check them. This, in turn, allows us to conclude that the application of *Circus* and Z constructors in the translation rules is correct with respect to the syntax of *Circus*.

A translation rule is an equation that contributes to the definition of a function mapping a well formed construct of a Stateflow chart, that is, an element of one of the sets defined in the previous section, to a *Circus* construct. We can divide the translation rules in four groups: renaming functions, expression and action functions, identifier and binding declaration functions, and action and process declaration functions. These groups are explained in the following sections.

4.3.1 Renaming functions

These functions are responsible for eliminating ambiguity in the names of Stateflow objects. For example, in Stateflow, two states can have the same name as long as they are not siblings, but in the *Circus* model this creates complications. How the ambiguity is eliminated is not specified, but the implementation uses prefixes and suffixes to eliminate ambiguity. In particular, numerical suffixes and type prefixes are attached to the names. For example, the prefix $S_$ is attached to the name of a state, and, in the case of two states with the same name, the unique identifier of the state is suffixed to each one of them. There are also renaming functions for the declaration of identifiers in the model, which behave like the renaming functions for names, but attach a lower case letter prefix to the name. For example, the identifiers of states are given the prefix $s_$ instead of $S_$.

There are over twenty such renaming functions; they take a well formed object (state, transition, junction, and so on) and return a name. For example, Rule 4.1 below declares the function used to rename states.

Rule 4.1. *Renaming function for states.*

$$statename : WF_STATE \rightarrowtail NAME$$

As stated above, we do not specify how unique names are to be constructed. The functions *chartname*, *statename*, *junctionname* and *transitionname* give names for the respective objects, whereas their identifier counterparts, *chartid*, *stateid*, *junctionid* and *transitionid* provide unique identifiers for these objects. The functions *dataname* and *eventname* do not have an identifier counterpart, because data and events are not represented as bindings of some schema; they are given names, that are determined by these functions. There is an additional renaming function for data, *datachannelname*, which gives the name of the

[1] This formalisations can be found in [64, 66].

channel used to communicate the values of the variables to the environment. The renaming functions for output events give the name of the counter variable associated with the event (*eventcountername*), and the name of the output channel for communicating such events (*outputeventchannelname*).

The functions *processname* and *processstatename* define, respectively, the name of the process that models the chart, and the name of the schema that represents that chart (and is part of the process' state). The functions *entryactionname*, *duringactionname* and *exitactionname* define the name of the action that models entry, during and exit actions, respectively. The functions *conditionactionname* and *transitionactionname* name condition and transition actions after the corresponding transition, whereas *conditionname* and *triggername* name the *Circus* actions that specify the conditions and triggers of a transition. The range of all the renaming functions is specified to be disjoint. The same holds for the functions that provide identifiers.

4.3.2 Expression and Action functions

The second group of functions translates expressions, actions, *on* actions, during actions, predicates, triggers, and sequences of these objects. It consists of $translate_{Expr}$, $translate_{Exprs}$, $translate_{Act}$, $translate_{Acts}$, $translate_{ONAct}$, $translate_{DAct}$, $translate_{DActs}$, $translate_{Pred}$, $translate_{Trigger}$ and $translate_{Triggers}$. The complete definition of these functions can be found in [67]; below, we discuss a selection of the rules that we consider most interesting.

Variable expression The translation of this type of expression is separated into two cases: simple variable and vector position. A simple variable is translated by obtaining the corresponding data and calculating the appropriate name through the renaming function *dataname*. A vector position requires us to first translate the expressions that are used as indexes, calculate the appropriate name of the variable (as in the case of simple variables) and apply this name to the translated expressions. Rule 4.2 (F) presents the formal translation rule for variable expressions.

Rule 4.2 (F). *Formal translation rule for variable expressions.*

$$\forall n : DNAME;\ es : \mathrm{seq}\, EXPR;\ c : WF_CHART \mid (c, name(n, es)) \in WF_EXPR \bullet$$

$$translate_{Expr}(c, name(n, es)) = \left(\begin{array}{l} \mathbf{if}\ \#\, es = 0 \\ \mathbf{then}\, reference(ref(dataname(c, getdatabyname(c, n)))) \\ \mathbf{else}\, functionapplication(\\ \quad app(dataname(c, getdatabyname(c, n)), \\ \quad translate_{Exprs}(c, es))) \end{array}\right)$$

As mentioned above, the domain of $translate_{Expr}$ is the set of well formed expressions. We apply the function to a well formed chart c, and a well formed variable expression formed by a name n and a sequence of vector indexes es. If the sequence es is empty

($\# es = 0$), then we recover the data to which the name refers, calculate its name with function *dataname*, and build a reference from this name using the Z constructors *reference* and *ref*. If the sequence *es* is not empty, we build a function application from the name of the data (obtained as in the previous case) and the sequence of translated expressions obtained by applying the function $translate_{Exprs}$ to *es*.

The formal translation rule can be hard to read because of the amount of constructor applications used to build the expressions; this is necessary to keep the syntactical and type information correct. Rule 4.2 shows the same translation rule as Rule 4.2 (F), but in a semi-formal fashion; instead of building the expression from the language constructors, we show how the expression would be rendered in *Circus*.

Rule 4.2. *Semi-formal translation rule for variable expressions.*

$$translate_{Expr}(name) = dataname(c, getdatabyname(c, name))$$
$$translate_{Expr}(name[e_1] \dots [e_n]) =$$
$$dataname(c, getdatabyname(c, name))(translate_{Expr}(e_1), \dots, translate_{Expr}(e_n))$$

All the translation rules take a well formed parameter, and all well formed parameters contain an instance of a chart. In the semi-formal translation rules, we leave the chart parameter c implicit (except where it is the only parameter). Rule 4.2 shows the two possible cases of variable expressions: simple variable and vector position. In the case of a simple variable, the translation returns the name of the data processed according to Section 4.3.1, otherwise a function application is build by applying the name of the data (as before) to the translated indexes. Henceforth, we only present the semi-formal versions of the rules. The complete formalisation of the translation rules can be found in [67].

Broadcast action The translation of broadcast actions depends on the type of event being broadcast. If it is an output event, the associated counter obtained through the function *eventcountername* is incremented. Otherwise, the broadcast is translated into a call to action *broadcast* followed by a call to action *check*. This is specified in Rule 4.3.

Rule 4.3. *Semi-formal translation rule for broadcast action.*

$$translate_{Act}(E) =$$
$$\left(\begin{array}{l} \textbf{if } (geteventbyname(E)).scope = OUTPUTEVENT \\ \textbf{then let } counter == eventcountername(c, geteventbyname(c, n)) \bullet \\ \quad eventcountername(counter) := eventcountername(counter) + 1 \\ \textbf{else } broadcast(eventname(c, geteventbyname(c, n)), chartid(c)) \ ; \ check(b) \end{array}\right)$$
$$translate_{Act}(send(E, dest) =$$
$$broadcast(eventname(c, geteventbyname(c, n)), stateid(c, getstatebyid(c, dest))) \ ; \ check(b)$$

The call to *broacast* takes the event and the destination as parameters, whereas the call to check takes a local variable b as parameter. Note that, well-formedness conditions require that the event is not an input event.

Assignment action The translation of assignment is also separated into the two cases depending on whether the variable being assigned is a simple variable or a vector position. The simple variable case is trivial, and consists of renaming the variable (as previously discussed), and building the *Circus* assignment expression. The vector position case $n[e_1] \ldots [e_m]$ is more interesting. Namely, because multi-dimensional vectors are specified as finite functions, the assignment of a value to one position needs to alter the function only in that position. This is achieved by overriding ($\oplus$) the function with the mapping $(indexes) \mapsto value$ and assigning the result to the function variable. Rule 4.4 formalises this translation rule.

Rule 4.4. *Semi-formal translation rule for assignment actions.*

$$\begin{array}{l}
translate_{Act}(name = e) = \\
\quad dataname(getdatabyname(name)) := translate_{Expr}(e) \\
translate_{Act}(name[e_1] \ldots [e_n] = e) = \\
\quad dataname(getdatabyname(name)) := \\
\quad\quad dataname(getdatabyname(name)) \oplus \\
\quad\quad \{(translate_{Expr}(e_1), \ldots, translate_{Expr}(e_n)) \mapsto translate_{Expr}(e)\}
\end{array}$$

The translation rule is defined for the two cases previously mentioned. If the variable being assigned is simple ($name = e$), we recover the data to which $name$ refers, calculate its name with function $dataname$, and use this name together with the translation of the expression e to build an assignment. If the variable being assigned is a vector position, we first obtain the name of the data in the same way as in the previous case. We then build an assignment from this name and from an expression obtained by applying the override operator to the name of the variable and a mapping that specifies the change to the vector position. This mapping is composed by the translated indexes of the vector position and the translated expression that is being assigned.

On action These actions are conditionally executed according to the current event being processed. Rule 4.5 shows the semi-formal translation rule for on actions.

Rule 4.5. *Semi-formal translation rule for on action.*

$$translate_{ONAct}(onn : a_1, \ldots a_n) = \left(\begin{array}{l}
\mathbf{if}\ E = eventname(geteventbyname(n)) \longrightarrow \\
\quad translate_{Acts}(\langle a_1, \ldots, a_n \rangle) \\
[\!]\ E \neq eventname(geteventbyname(n)) \longrightarrow \\
\quad \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)$$

On actions are translated to an alternation that executes the associated action if the current event is the one specified by the on action, otherwise it terminates immediately.

Sequence of actions The translation of a sequence of actions is relatively simple, except in the case where one of the actions is a local event broadcast. In this case, any remaining action after the broadcast is conditioned on the value of a local variable b. The local variable is declared by the translation rules that generate chart action declarations. Rule 4.6 shows the translation of a sequence of actions.

Rule 4.6. *Semi-formal translation rule for a sequence of actions.*

$$translate_{Acts}(\langle A \rangle \frown AS) =$$
$$\left(\begin{array}{l} \textbf{if } A \textit{ is a local event broadcast} \\ \textbf{then } translate_{Act}(A)\ ;\ \left(\begin{array}{l} \textbf{if } b = \textbf{True} \longrightarrow \textbf{Skip} \\ [\!]\, b = \textbf{False} \longrightarrow translate_{Acts}(AS) \\ \textbf{fi} \end{array}\right) \\ \textbf{else } translate_{Act}(A)\ ;\ translate_{Acts}(AS) \end{array}\right)$$

The case of a sequence of during actions is similar, but the conditioning of the remaining actions is performed in two levels: inside an on action, which has multiple actions, and immediately after an on action that contains a local event broadcast.

4.3.3 Identifier and binding declaration functions

Every chart, state, junction and transition in a model has a unique identifier. We define these identifiers through axiomatic definitions and the identifier renaming functions. Rule 4.7 shows the translation function that produces the declaration of all state identifiers belonging to a well formed chart.

Rule 4.7. *Semi-formal translation rule for the declaration of state identifiers.*

$$StateIdentifierDeclaration(c) =$$

$$\big|\ stateid(c.states(1)), \ldots, stateid(c.states(\#\, c.states)), chartid(c) : STATEID$$

This rule takes a well formed chart, calculates the sequence of processed identifiers (as discussed in Section 4.3.1) of the states in the chart (using the function *stateid*), creates a sequence containing the processed identifier of the chart (obtained by applying the function *chartid*), and returns an axiomatic description that declares these names as belonging to the set *SID*. The rules for declaring junction and transition identifiers are similar [67].

Once the identifiers of the chart, states, junctions and transitions are declared, we need to declare the bindings (of the appropriate type) that represent them in the *Circus* model. Rule 4.8 presents the rule that declares a junction; it takes a well formed junction j and returns an axiomatic description that declares a variable of type *JUNCTION* and equates it to the appropriate binding.

Rule 4.8. *Semi-formal translation rule for the declaration of junctions.*

$JunctionDeclaration(j) =$

$$
\begin{array}{|l}
junctionname(j) : JUNCTION \\
\hline
junctionname(j) = \\
\quad \langle\!| junction_identifier(j), junction_transition(j), junction_parent(j), \\
\quad junction_history(j) |\!\rangle
\end{array}
$$

This rule uses four functions to specify the binding extension: *junction_identifier*, *junction_transition*, *junction_parent* and *junction_history*. Each of these functions takes a well formed junction and returns a pair (n, e) where n is a name and e is an expression. This pair correspond to the element $n == e$ in a binding. The same approach is taken for declaring states, transitions and charts. Notice that charts are declared both as states (function *ChartAsStateDeclaration*) and as proper charts (function *ChartDeclaration*).

The local and input events are declared as members of the set $EVENT$; they do not have any additional information in their declaration. This can be done because the relevant information about an event is either treated separately or ignored. For example, we are not treating the distinction between events triggered by rising edge, falling edge and both-edges; the scope is treated at the level of the translation by treating input, output and local events appropriately. The function that declares the events is similar to the functions that declare identifiers [67]. The translation of output events differs in that we do not declare them as elements of type $EVENT$. Instead, the translation of output events generates event counters and communication channels as specified by the translation rules *OutputEventDeclaration* and *ChannelsDeclaration* in [67].

Event counters and data are declared in the schema *SimulationInstance* generated by the translation rule *SimulationInstanceDeclaration*. The rule *ProcessStateDeclaration* produces the state of the chart process by conjoining *SimulationInstance* and the schema *SimulationData* defined in the `stateflow_toolkit` library (see Appendix B).

4.3.4 Action, condition and process declaration functions

There are five types of action declaration rules, corresponding to five translation functions: *EntryActionDeclaration*, *DuringActionDeclaration*, *ExitActionDeclaration*, *ConditionActionDeclaration*, and *TransitionActionDeclaration*. The first three take a well formed state, and the remaining ones take a well formed transition as parameter; all of them return a *Circus* action that encodes the corresponding action. Rule 4.9 generates an entry action declaration.

Rule 4.9. *Semi-formal translation rule for the declaration of entry actions.*

$$
\begin{array}{l}
\mathit{EntryActionDeclaration}(s) = \\
\left(\begin{array}{l}
\textbf{if } s.entry \textit{ has a local event broadcast} \\
\textbf{then } \mathit{EntryActionName}(s) \mathrel{\widehat=} \left(\begin{array}{l} executeentryaction.stateid(s) \longrightarrow \\ \textbf{var } b : \mathbb{B} \bullet (translate_{Acts}(s.entry)) \end{array}\right) \\
\textbf{else } \mathit{EntryActionName}(s) \mathrel{\widehat=} \left(\begin{array}{l} executeentryaction.stateid(s) \longrightarrow \\ translate_{Acts}(s.entry) \end{array}\right)
\end{array}\right)
\end{array}
$$

This function takes a state, and returns an action whose name is calculated based on the name of the state and is defined as a prefixed action. The prefix is the communication through channel *executeentryaction* of the identifier of the state, while the action is the translation of the sequence of entry actions (which yields the **Skip** action if the sequence is empty). The translation of the sequence of actions is enclosed in a local variable block if the actions contains a local event broadcast. This local variable block declares variable b of type $\mathbb{B}$, which is used to decide whether or not to stop the execution of the sequence of actions after a local event broadcast. The value of this variable is updated by the action *check*, which is always called after a local event broadcast, as discussed in Section 4.3.2.

After the entry actions of all states are declared, they are joined in an action called *entryactions* that offers them in an external choice. This action is generated by the rule *EntryActionsDeclaration* shown in Rule 4.10. The same approach is adopted for the declaration of during, exit, condition and transition actions.

Rule 4.10. *Semi-formal translation rule for the declaration of the external choice of all entry actions.*

$$
\begin{array}{l}
\mathit{EntryActionsDeclaration}(c) = \\
\qquad entryactions \mathrel{\widehat=} \left(\begin{array}{l}
\mathit{EntryActionName}(c.states(1)) \\
\Box \\
\ldots \\
\Box \\
\mathit{EntryActionName}(c.states(\# c.states))
\end{array}\right)
\end{array}
$$

The declaration of conditions (Rule 4.11) is similar to the declaration of actions, but involves the use of an alternation; the action communicates the truth value of the condition through channel *evaluatecondition*. All the conditions are joined in an action in the same fashion as in Rule 4.10 above.

Rule 4.11. *Semi-formal translation rule for the declaration of conditions.*

$$\begin{array}{l}
\textbf{if}\ \ \#\,t.condition = 0\\
\textbf{then}\ \ ConditionDeclaration(t) =\\
\qquad ConditionName(t) \mathrel{\widehat=} evaluatecondition.transitionid(t).\textbf{True} \longrightarrow \textbf{Skip}\\
\textbf{else}\ \ ConditionDeclaration(t) =\\
\qquad ConditionName(t) \mathrel{\widehat=} \left(\begin{array}{l}
\textbf{if}\ translate_{Pred}(head\ t.condition) \longrightarrow\\
\qquad evaluatecondition.transitionid(t).\textbf{True} \longrightarrow \textbf{Skip}\\
[\!]\,\neg\ translate_{Pred}(head\ t.condition) \longrightarrow\\
\qquad evaluatecondition.transitionid(t).\textbf{False} \longrightarrow \textbf{Skip}\\
\textbf{fi}
\end{array}\right)
\end{array}$$

The condition declaration translation rule takes a transition as parameter and builds a *Circus* action, named after the transition, through the function *ConditionName*. Since a condition is optional, it is recorded in the transition t as a sequence of size at most 1. If there is no condition, this action is defined as a communication over the channel *evaluatecondition* of the identifier of the transition and the value **True**. If there is a condition ($head\ t.condition$), this action is defined as an alternation. The first guard checks if the translated condition is different than zero; the associated action communicates the identifier of the transition and the value **True** through channel *evaluatecondition*. The second guard checks if the translated condition is equal to zero; the associated action then communicates the identifier of the transition and the value **False** through the same channel as in the first case. Note that Stateflow conditions yield integer values; zero corresponds to false, and values different than zero correspond to true.

Trigger declarations are specified to evaluate the triggers of a transition; they use two communication channels *checktrigger* and *result* to, respectively, prompt the evaluation of the trigger and to obtain the result of the evaluation. The declaration consists of two cases: if there is no trigger, the action returns **True** as a default value, otherwise, it calculates the truth value and returns it. Rule 4.12 specifies the declaration of triggers.

Rule 4.12. *Semi-formal translation rule for the declaration of triggers.*

$$\begin{array}{l}
\textbf{if}\ \#\,t.trigger = 0\ \textbf{then}\ \ TriggerDeclaration(t) =\\
\qquad (TriggerName(t) \mathrel{\widehat=} checktrigger.transitionid(t)?E \longrightarrow\\
\qquad\qquad result.transitionid(t).E!\textbf{True} \longrightarrow \textbf{Skip})\\
\textbf{else}\ TriggerDeclaration(t) = TriggerName(t) \mathrel{\widehat=} checktrigger.transitionid(t)?E \longrightarrow\\
\qquad \left(\begin{array}{l}
\textbf{if}\ translate_{Triggers}(t.trigger) \longrightarrow result.transitionid(t).E!\textbf{True} \longrightarrow \textbf{Skip}\\
[\!]\,\neg\ translate_{Triggers}(t.trigger) \longrightarrow result.transitionid(t).E!\textbf{False} \longrightarrow \textbf{Skip}\\
\textbf{fi}
\end{array}\right)
\end{array}$$

This translation rule takes a transition as parameter and generates a *Circus* action whose name is the result of the function *TriggerName*, and whose behaviour depends on whether

the transitions has a trigger or not. If there is no trigger, the action uses channel *checktrigger* to synchronise on the transition identifier and receive the current event E. It then uses channel *result* to synchronise on the transition identifier and event, and output the value **True**. If there is a trigger, the action communicates through channel *checktrigger* in the same way as before, and uses an if-statement to offer two different communications. If the translation of the trigger is true, the channel *result* is used to synchronise on the transition identifier and current event, and output the value **True**; otherwise, instead of **True**, the value **False** is communicated through the channel *result*.

The process declaration rule takes a well formed chart and produces a *Circus* process that encloses the appropriate paragraphs produced by other functions; it is shown in Rule 4.13. Because of the size of the rule, some parts are omitted for clarity.

Rule 4.13. *Semi-formal translation rule for the declaration of chart processes.*

$$
\begin{array}{l}
ProcessDeclaration(c) = \\
\quad \mathbf{process}\ processname(c) \mathrel{\widehat=} \mathbf{begin} \\
\quad ChartDeclaration(c)) \\
\quad SimulationInstanceDeclaration(c) \\
\quad InitSimulationInstanceDeclaration(c) \\
\quad ProcessStateDeclaration(c) \\
\quad EntryActionDeclaration(s_1) \\
\quad \ldots \\
\quad EntryActionDeclaration(s_n) \\
\quad EntryActionsDeclaration(c) \\
\quad \ldots \\
\quad ConditionActionsDeclaration(c) \\
\quad \ldots \\
\quad ConditionsDeclaration(c) \ldots \\
\quad TriggersDeclaration(c) \\
\quad GetStateDeclaration \\
\quad \ldots \\
\quad \bullet\ MainActionDeclaration \\
\quad \mathbf{end}
\end{array}
$$

This rule takes a chart c and returns a *Circus* process whose name is the result of applying *processname* to c. The body of the process consists of a series of schema and action declarations which are obtained by applying the appropriate translation functions. For instance, the axiomatic definition that represents the chart is part of the process, this is specified in the rule by the application of the translation rule *ChartDeclaration*.

Rule 4.13 includes in the body of the process the declarations of the chart, the schemas *SimulationInstance* and *SimulationData*, the initialisation and operation schemas, the state

of the process, and all the actions. It completes the process by declaring the main action through the function *MainActionDeclaration*.

The root translation rule, *translate*, takes a well formed chart and outputs a *Circus* specification. This rule is structured in the same way as the process declaration rule. It declares the paragraphs external to the process (e.g. identifier declarations) and the process through previously defined translation rules.

4.4 Automation of the translation strategy

The manual translation process is extremely error prone, a mechanisation of such process is clearly beneficial. Moreover, when the translation of very simple charts (6 states and 7 transitions) generates more than ten pages of specification, the translation of larger charts is prohibitively costly. For example, one of our experiments on an industrial case study yielded a 180-page specification.

A mechanisation of the translation process can reduce the cost of the translation, thus increasing the viability of any task that depends on a *Circus* model. In addition, mechanising the translation provides extra validation of the model (and translation rules), uncovering potential errors and inconsistencies. Since the *Circus* models are meant to be used in the formal verification of implementations of Stateflow charts, the mechanical translation increases the number and size of charts that can be tackled, and minimises the possibility that translation errors occur.

In this section, we describe the tool *s2c* which implements the translation rules discussed in the previous sections. *s2c* takes a *.mdl* file and produces a *Circus* specification containing the processes that model the charts contained in the file.

The translation of Stateflow charts into *Circus* processes is defined in a manner that makes it suitable for mechanisation; namely, by using formal and yet concrete translation rules. Since the implementation of *s2c* resembles the formal specification, it can be inspected and compared to the formalisation.

The implementation of *s2c* is guided by one main principle: to keep a close relationship between the implementation and the formalisation of the translation rules. The advantage of an implementation based on this principle is that it is easy to identify how components of the code map to the formal rules, thus, facilitating the validation of the implementation with respect to the formalisation.

Section 4.4.1 discusses the architecture of the tool, and Section 4.4.2 discusses the implementation of the translation rules.

4.4.1 Architecture

The architecture of *s2c* is organised in five main packages: `parser`, `stateflow abstract syntax`, *Circus* `syntax`, `Z syntax` and `translator` (Figure 4.2). The `translator` package is the main one, and uses all the other packages to perform the translation. We now discuss

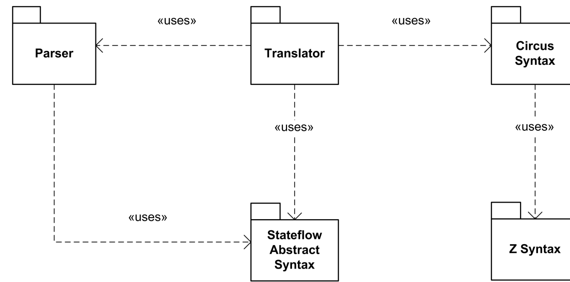


Figure 4.2: Architecture of s2c: main packages.

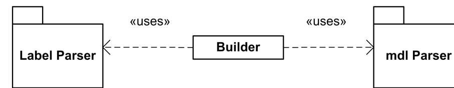


Figure 4.3: Architecture of the package `Parser`.

each package in more detail.

The `parser` package. This package contains two sub-packages: `MDL parser` and `label parser` (Figure 4.3). The `MDL parser` provides classes that support taking a *.mdl* file and converting it into a tree structure that facilitates the manipulation of data. The `label parser` provides classes that build states and transitions from labels that encode these objects in the *.mdl* file. The Java class `Builder` uses both packages to define a method to build the abstract syntax tree of the chart. In the following, we give more details about these packages, and explain how the class `Builder` interacts with them in order to build the abstract syntax tree.

The `MDL parser` package. This package provides a class `MDLParser` that takes a *.mdl* file that has a particular structure and builds a data structure that facilitates the querying and manipulation of the data contained in the file. As depicted in Figure 4.4, a *.mdl* file contains groups of attribute-value pairs, which may contains other groups. The hierarchy established by a *.mdl* file makes it simple to extract some of the information needed to build an abstract syntax tree, but not all. Namely, the representation of states and transitions contains a pair whose attribute is called `labelString`, and whose value encodes, for states, the name and the actions of the state, and, for transitions, the trigger, the condition, and the actions of the transition. This encoding is processed separately by the `label parser`, as we explain later.

Of the top groups in a *.mdl* file (that is, groups without a parent group), the one that is relevant to Stateflow charts is called `Stateflow`. This group contains other groups that describe charts, states, transitions, junctions, data and events. Each one of these groups has attribute-value pairs (e.g. the pair `"id 2"` associates the value 2 to the attribute `id`) and may contain other groups (for example, a `transition` group contains a `src` group and a `dst` group that represent, respectively, the source and the destination of the transition). Figure 4.5 shows an excerpt from a *.mdl* file.

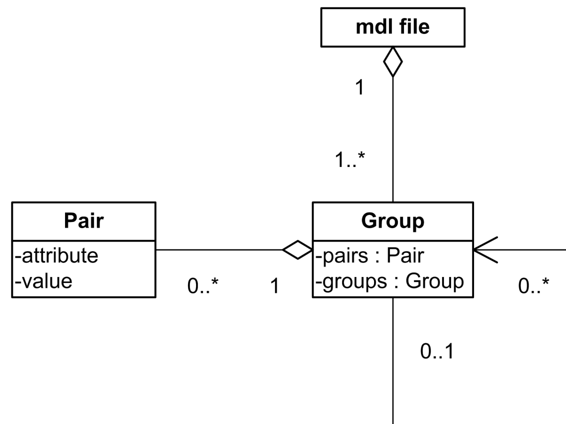


Figure 4.4: Structure of a *.mdl* file.

```
Stateflow {
   chart {
      id     2
      name     "Chart"
      decomposition     CLUSTER_CHART
      ...
   }
   state {
      id     3
      labelString     "State1"
      type     OR_STATE
      decomposition     CLUSTER_STATE
      ...
   }
  transition {
      id     9
      labelString     "{Data = Data+1;}"
      src {
      }
      dst {
         id     3
      }
      chart     2
      ...
   }
   ...
}
```

Figure 4.5: Excerpt from a *.mdl* file.

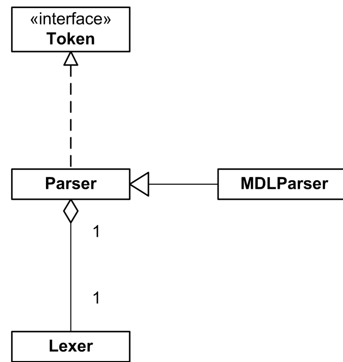


Figure 4.6: Architecture of the package `MDL Parser`.

state_name
  en: action
  du: action
  ex: action
  on event_name: action
  on event_name: action
  ...

Figure 4.7: State label

trigger[expression]action/action

Figure 4.8: Transition label

We observe that, as we mentioned before, the actions, conditions and triggers used in a chart are encoded in the label (`labelString` attribute) of the appropriate object. For example, in Figure 4.5, in the group `transition`, the attribute `labelString` is associated with a string that encodes the condition action of the transition.

The `MDL parser` processes a *.mdl* file and structures it in a way that facilitates the manipulation of the groups and pairs. From this structure, a method in the class called `Builder` builds the charts, states, transitions, junctions, data and events. In order to process the labels of states and transitions (that contain the actions, conditions and triggers), the `Builder` class uses the `label parser`. The class `MDLParser` extends the class `Parser` which is automatically generated by the tool *jacc* [52], and uses the class `Lexer`, which is automatically generated by the tool *jflex* [50], as shown in Figure 4.6.

The `label parser` package contains two parsers: state parser and transition parser. These parsers are contained in the same package because, although state and transition labels have different structures, they have common units of information; namely, expressions and actions. The syntax of expressions and actions constitutes most of the syntax of labels. Therefore we reuse the parser of expressions and actions in both parsers. Figures 4.7 and 4.8 show the structure of state and transition labels.

The state parser takes the value of the attribute `labelString` of a `state` group, and returns a `State` object with the appropriate actions assigned to it. The rest of the information that is not available from the label is filled in by a method in the `Builder` class. The same process is carried out by the transition parser, with the difference that the object returned is of type `Transition`. As with the `MDL parser`, both the state parser

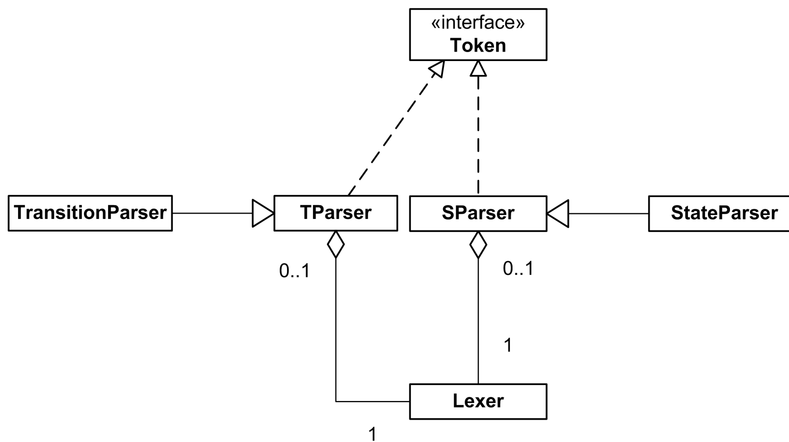


Figure 4.9: Architecture of the package `Label Parser`.

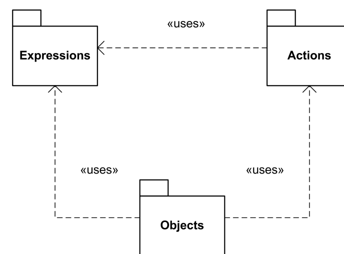


Figure 4.10: Syntax of Stateflow charts: main packages.

and the transition parser were automatically generated by the tool *jacc*, and the class Lexer, which is common to both parsers, was generated automatically by the tool *jflex*. The classes `TransitionParser` and `StateParser` respectively extend the automatically generated classes `TParser` and `SParser`, as shown in Figure 4.9.

Both state and transition objects are defined in the abstract syntax of Stateflow, which defines the main Stateflow objects, i.e. chart, state, transition, junction, data and event, as well as the action language used in the labels. The stateflow abstract syntax is an implementation of the formal syntax of Stateflow charts discussed in the Section 4.1. We now discuss its implementation.

The `stateflow abstract syntax` package is organised in three packages: `Objects`, `Actions` and `Expressions`, as shown in Figure 4.10. The package `Objects` contains the classes that represent the main objects of the language. The package `Actions` contains the class model of the action language, which uses the model of expressions contained in the package `Expressions`.

As can be seen in Figure 4.11, data, event and junction are simple objects, while state, transition and chart contribute most of the complexity to the chart structure. A chart object contains sets of state, transition, junction, data and event objects. A state object contains two sequences of actions, *entry* and *exit*, and a sequence of *during* actions. A transition object has two sequences of actions, condition action (*condAction*) and transition action (*transAction*), an optional expression *condition*, and a sequence of triggers *trigger*.

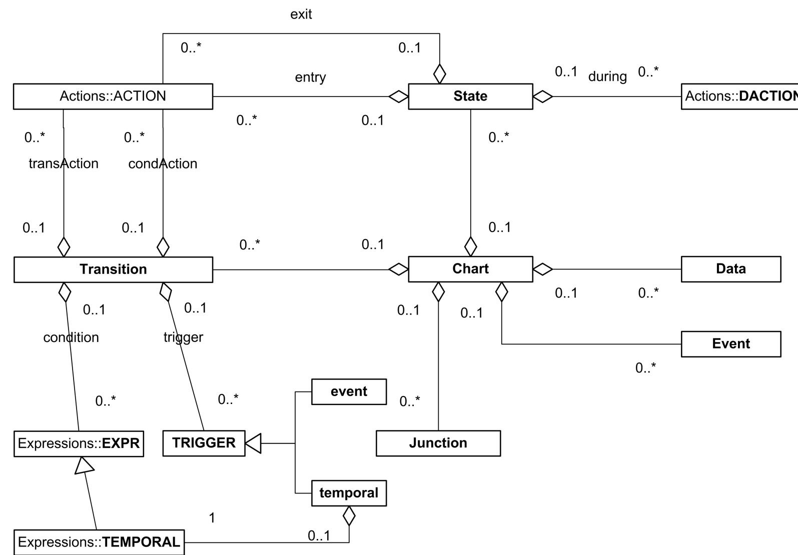


Figure 4.11: Syntax of Stateflow charts: Objects.

Figure 4.12 depicts the representation of the syntax of actions. There are four subclasses representing four types of actions: `massign`, `assign`, `bcast` and `expr`. Objects of the class `massign` represent multiple assignments in which the left-hand side of the assignment is an array of variables. Objects of the class `assign` represent simple assignments and objects of the class `bcast` represent broadcasting of events. Object of the class `expr` represent expressions used as actions. Objects of the class `ONACTION` represent actions that are executed only if a certain trigger occurs. Objects of the more abstract class `DACTION` represent during actions and can belong either to `ACTION` or `ONACTION` class.

The model for the syntax of expressions is the largest, but it is fairly simple. The only noteworthy type of expression is the variable expression, which can take a sequence of expressions as indexes, in the case where the variable represents an array. The class diagram is shown in Figure 4.13.

Each free type constructor defined in the formal definition of the Stateflow syntax in Section 4.1 is now represented by a class. For example, for the free type ACTION, we have the constructors *bcast*, *expr*, *assign*, and *massign*, which correspond to the classes `bcast`, `expr`, `assign`, and `massign`. The type of the constructor functions determines the type of the constructor methods of the corresponding classes. For example, the constructor *bcast* takes a value of type *ENAME* and a state name, both of which are implemented as objects of type `String`; therefore, the method `bcast` takes `String` objects as parameter. The type of the constructor is not represented in the diagrams, but is specified in the formal syntax and implemented in the tool.

The *Circus* syntax package implements the *Circus* syntax differently from the way the Stateflow abstract syntax is implemented. Instead of defining classes for each element of the syntax, we define functions that return the *Circus* element encoded in some form. We adopt this approach because we do not need to manipulate *Circus* syntax trees in the

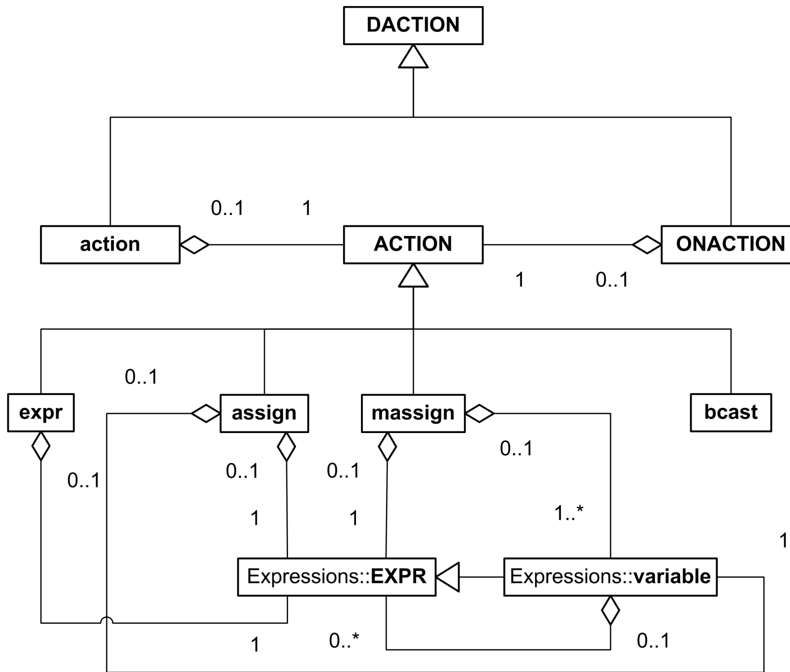


Figure 4.12: Syntax of Stateflow charts: Actions.

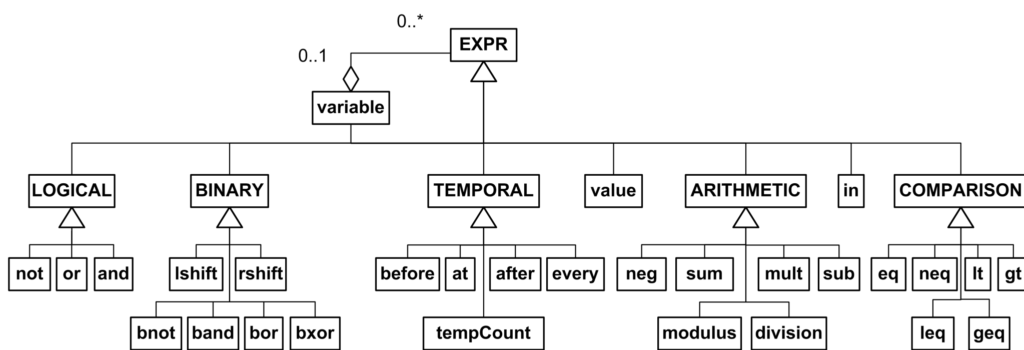


Figure 4.13: Syntax of Stateflow charts: Expressions.

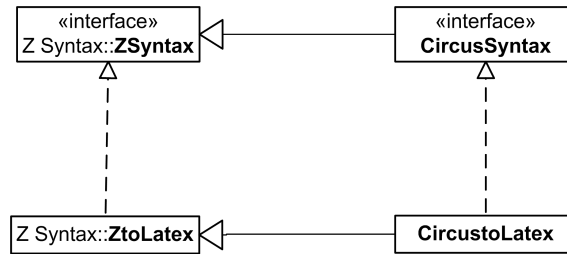


Figure 4.14: Syntax of *Circus*.

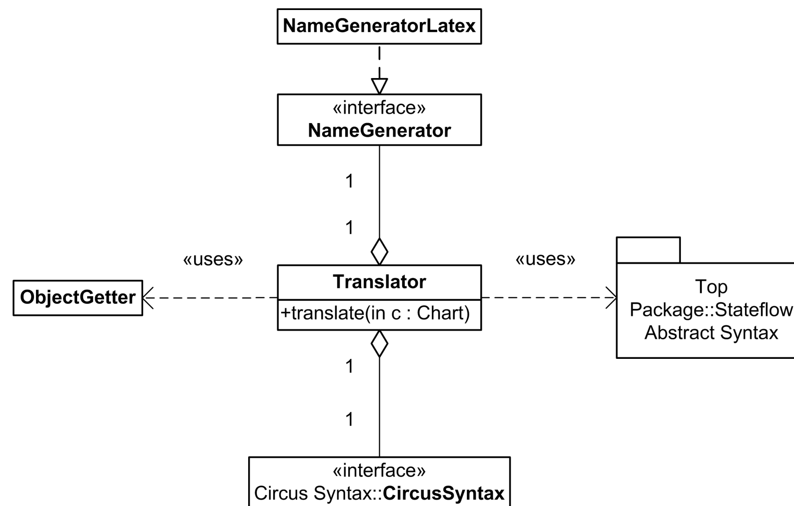


Figure 4.15: Architecture of s2c: the Translator package.

tool. Moreover, the use of functions to build *Circus* specifications makes the resulting code resemble the formal specification more closely. We define an interface that contains the functions that can be used to build *Circus* statements and provide an implementation of this interface that produces LaTeX code. Note that it is simple to implement the interface to generate specifications suitable for other tools.

The Z syntax package is structured in exactly the same fashion. The *Circus* syntax interface and implementation extend, respectively, the interface and implementation of the Z syntax, as shown in Figure 4.14.

Finally, the `Translator` class contains a method `translate` that takes a Stateflow chart represented in the abstract syntax and produces the *Circus* specification of the chart. The translator uses an implementation of the *Circus* syntax interface and a name generator to produce the specifications. The name generator is responsible for taking Stateflow objects and returning their names in an appropriate format. For example, we disambiguate state names (when needed) by suffixing the unique identifier of the state. State identifiers are obtained by prefixing the (disambiguated) name of the state with a lower case "s", while state names are obtained by prefixing the name with a capital "S". Figure 4.15 shows the architecture of the translator package.

4.4.2 Implementation of translation rules

We illustrate the strategy used for implementing the translation rules through an example. For convenience, Rule 4.14 reproduces the formal specification of the translation function for variable expressions. This function takes a well formed expression (the set of well formed expressions is defined in [65]) and produces a Z and *Circus* expression (Z and *Circus* expressions are defined in [66, 64]). The resulting Z expression depends on whether the variable expression represents a simple variable or a position in a vector. If the expression represents a simple variable, the function returns a reference to the appropriate data. We observe that the name of the piece of data being treated needs to be processed in order to avoid duplicated names and attach the variable prefix $v_$. This is the objective of the function *dataname*. If the variable is a position in a vector, the function returns the application of the name of the appropriate data to the translated expressions contained in the index sequence; multi-dimensional arrays are modelled as functions from a set of indexes to the appropriate type.

Rule 4.14. *Formal translation rule for variable expressions.*

$$translate_{Expr} : WF_EXPR \rightarrow Expression$$

$$\begin{array}{l}
\forall n : DNAME;\ es : \mathrm{seq}\, EXPR;\ c : WF_CHART \mid \\
(c, variable(n, es)) \in WF_EXPR) \bullet \\
translate_{Expr}(c, variable(n, es)) = \\
\left(\begin{array}{l}
\textbf{if}\ \#\, es = 0 \\
\textbf{then}\ reference(ref(dataname(c, getdatabyname(c, n)))) \\
\textbf{else}\ functionapplication(app \left(\begin{array}{l} dataname(c, getdatabyname(c, n)), \\ translate_{Exprs}(c, es)) \end{array}\right)
\end{array}\right)
\end{array}$$

Figure 4.16 shows the implementation of the Rule 4.14. The $translate_{Expr}$ function is translated into a Java method called `translate_expr`. The local variables introduced by the **let** expression in the formal rule are translated into local variables of the method. The function *getdatabyname* is mapped to a static function of the same name contained in the class `ObjectGetters`, and the function *dataname* is mapped to a function from the object `_n` of a class that implements the interface `NameGenerator`. The constructor functions are used exactly in the same way as in the formal specification. Each of them is mapped to a method of the same name contained in the object `_c` of a class that implements the interface `CircusSyntax`. Calls to other translation functions are mapped to methods of the `Translator` class with equivalent names. For example, the function $translate_{Exprs}$ is mapped to the method `translate_exprs`.

The implementation of the example shown in Rule 4.14 is straightforward once we provide an embedding of the syntax of Stateflow charts, of *Circus*, and of Z operations used in the formal specification. The implementation of some of the translation rules, however, impose a challenge; this is due mainly to the use of sequence extensions of the form $\langle a, b, c\rangle$, and set comprehensions.

```
public String translate_expr(Chart c, EXPR e) {
  if (e instanceof variable) {
    String n = ((variable)e).dname;
    List<EXPR> es = ((variable)e).es;
    Data d = ObjectGetters.getdatabyname(c, n);
    if (es.size() == 0) {
      return _c.reference(_c.ref(_n.dataname(c,d)));
    }
    else {
      return _c.functionapplication(_c.app(
        _n.dataname(c, d), translate_exprs(c,es)));
    }
  }
  ...
}
```

Figure 4.16: Implementations of the translation rule for variable expressions.

Sequence extension Rule 4.15 shows the translation rule that takes a well formed data and produces the declaration associated with that data. The constructor *variable* requires a sequence of declared names (built through the application of the constructor *decl* to a name) and an expression that defines the type of the variables being declared. In this case, the sequence contains only the declared name of the data being declared.

Rule 4.15. *Formal translation rule for the declaration of data.*

$$DataDeclaration : WF_DATA \rightarrow Declaration$$

$$\forall\, c : WF_CHART;\ d : Data \mid (c, d) \in WF_DATA \bullet DataDeclaration(c, d) = variable(\langle decl(dataname(c, d))\rangle, translate_{Type}(d.type))$$

In order to translate the *DataDeclaration* function, we define an instance method called `DataDeclaration` that takes a chart and a piece of data, and returns a `String` that contains the declaration. The use of a sequence extension as an argument to the constructor *variable* is tackled by introducing an auxiliary variable (in this case, called `declarations`) of type `List<String>`, instantiating it, and adding the appropriate element to the list. Finally, the auxiliary variable is used in place of the sequence extension. The implementation of the function shown in Rule 4.15 is presented in Figure 4.17.

Although the structure of the implementation of this function is different than the specification, by establishing how the translation of sequence and set constructors is done, we provide a simple way to assess its correspondence. In the next example, we examine the implementation of functions that contain set comprehensions of the form $\{x : S \mid p(x) \bullet f(x)\}$, where x is a variable, S is a set, p is a predicate, and f is a function.

```
public String DataDeclaration(Chart c, Data d) {
    List<String> declarations = new LinkedList<String>();
    declarations.add(_c.decl(_n.dataname(c, d)));
    return _c.variable(declarations,translate_type(d.type));
}
```

Figure 4.17: Implementations of the translation rule for the declaration of data.

```
public String SimulationInstanceDeclaration(Chart c) {
    List<String> variables = new LinkedList<String>();
    for (Integer i: c.data.keySet()) {
        variables.add(DataDeclaration(c, c.data.get(i)));
    }
    List<String> outputevents = new LinkedList<String>();
    for (Integer i: c.events.keySet()) {
        if (c.events.get(i).scope == EVENTSCOPE.OUTPUTEVENT) {
            outputevents.add(OutputEventDeclaration(c, c.events.get(i)));
        }
    }
    variables.addAll(outputevents);
    String decl = _c.declpart_vertical(variables);
    return _c.schemadefinition(
        _n.SIMULATIONINSTANCE(),
        _c.schematext_vertical(
            new Opt<String>(decl),
            new LinkedList<String>()));
}
```

Figure 4.18: Implementations of the translation rule for the declaration of the simulation instance.

Set comprehension Rule 4.16 shows a function that declares a schema used as part of the state of the chart process. The function takes a well formed chart and produces a Z paragraph. Notice that the local variable *variables* is defined as a sequence by applying the function *squash* to a set comprehension. The set comprehension takes the indexes of the sequence *c.data* and produces a partial function from natural numbers to data declarations by applying a translation function to the data contained in *c.data*.

Rule 4.16. *Formal translation rules for the declaration of the simulation instance.*

$$SimulationInstanceDeclaration : WF_CHART \rightarrow Paragraph$$

$$\begin{array}{l}
\forall c : WF_CHART \bullet SimulationInstanceDeclaration(c) = \mathbf{let} \\
\quad variables == squash\{i : \mathrm{dom}\, c.data \bullet i \mapsto DataDeclaration(c, c.data(i))\}; \\
\quad outputevents == squash\{i : \mathrm{dom}\, c.events \mid (c.events(i)).scope = OUTPUTEVENT \bullet \\
\qquad\qquad i \mapsto OutputEventDeclaration(c, c.events(i))\} \bullet \mathbf{let} \\
\qquad decl == declpart(variables \frown outputevents) \bullet \\
\qquad\qquad schemadefinition(SimulationInstance, schematext(\langle decl \rangle, \langle\rangle))
\end{array}$$

As shown in Figure 4.18, the set comprehension is translated into the instantiation of a list

(because the function *squash* applied to the set comprehension, in this example, defines a list) which is assigned to the variable *variables*. The list is then filled in with all the data declarations obtained from the data in the list `c.data`. The same is done for the list of output events, and the two lists are joined together before the declaration is built. Note that we use the class `Opt` which implements the type *Opt* defined in the specification to obtain, more easily, lists of size zero or one.

Set comprehensions could be implemented by the Java class `Set`, but, in most cases, the sets defined in the specification are transformed into sequences. Since a sequence can be seen as an ordered set, we do not lose information by using sequences instead of sets.

In some of the rules, we use a function *set2seq* which takes a set and produces a sequence with the same objects. The definition of this function is underspecified, stating only that the range of the sequence obtained from the application of the function to a set S must be equal to S. Since the order of the elements in the list is clearly not important, we add them directly to a list, instead of a set, thus rendering the use of the function *set2seq* unnecessary in the implementation.

The implementation of *s2c*[2] has approximately 19000 lines of code, 75% of which implements the various parsers and is automatically generated. In the next section, we will discuss the usage and evaluation of the tool *s2c*.

4.5 Evaluation

The tool *s2c* takes a *.mdl* file that contains the definition of a number of charts, and produces a *Circus* process for each chart; each process and associated definitions are enclosed in a Z section. The tool ignores definitions of Simulink and graphical functions, truth tables, as well as Simulink blocks. The existence of functions implies that there are data definitions of the input and output variables of the function; these are also ignored.

Since Simulink and graphical functions are not translated, but are widely used in charts along with *matlab* and *C* functions, we adopt a strategy for dealing with functions whereby the user is required to supply the formal definition of the functions being used in the form of libraries. Each library is specified in a separate file within a named Z section which is inherited by the section that contains the model of the chart that uses the library.

Our evaluation strategy consists of testing the tool on three distinct sets of examples: basic charts, complete examples, and industrial case studies. The group of basic charts contains models that exercise specific features of the Stateflow notation, for example, for each type of action a, we have a chart formed by a single state whose entry action is a. The group of complete examples is formed by those charts (or collections of charts) that model a simplified system (usually "toy-examples"), but whose complexity is larger. For example, we have used the *shift_logic* chart of the *Automatic Transmission Control* example supplied with Stateflow and used in Chapter 3. The third group consists of models supplied by industrial collaborators. These models usually describe the whole system. For example,

[2]The tool can be downloaded from `http://www-users.cs.york.ac.uk/~alvarohm/s2c/tool/`

one of the case studies with which we have tested *s2c* was supplied by Airbus Operations and describes a generic Fuel Management Control System.

By testing basic charts, we can easily track any errors in the tool to the specific feature being tested. Moreover, since these models are very small, it is possible to check the generated specification and notice any potential errors. With this approach, while validating the tool, we also validate the translation strategy and the model presented in Chapter 3. On the other hand, the complete examples allow us to test the interaction between different features of the notation, and while their size makes it difficult to manually check the generated specification, it is still feasible to track errors in the translation process to features of the Stateflow model. The specification generated by industrial case-studies can be extremely large, and, therefore, not tractable manually, and, while it is possible to track translation errors to pieces of the model, it is a demanding task. Nevertheless, testing industrial case-studies allows us to evaluate the correctness, robustness and efficiency of the tool.

We now turn to the results of testing the tool with all three groups. Due to the number of examples in the first group, we only give an overview of how they were built, and discuss the main findings.

We have tested the tool with the basic charts in different stages of development, and errors found during the development were corrected. The main problems found during testing were concerned with the handling of elements not treated by the translation strategy, and simple programming errors (such as index out of bound errors, un-initialised variables etc). The latter errors were easily fixed, while the former required us either to treat the features in the translation strategy, or find a way to overcome the features when they appear in a chart.

The testing of the tool has highlighted two features that were not treated, one known and the other unknown. The former is the treatment of functions, and the latter is the use of the time symbol t.

The treatment of functions is not a simple issue, because the functions that can be used in charts are not restricted to the types of functions that can be defined within a chart – MATLAB and even C function can be used. The complete treatment of function would require us to formalise all the possible functions (including the MATLAB and C ones). Our approach to this issue was to assume that the user of the tool has a library of definition of the functions being used in the chart. It is worth mentioning that this approach is limited in the sense that it does not allow the specification of graphical functions with side-effects (i.e., graphical functions that change the state of the chart). However, if a separate translation strategy is defined for graphical functions, it can be easily incorporated to the tool, thus, eliminating this limitation.

The proper treatment of the time symbol t was delegated to the Simulink diagram and referenced by a process variable which is updated through a channel *time*. The reason why the treatment of t can be delegated to the Simulink diagram is that this symbol represents the "absolute time inherited from a Simulink signal" [98].

The two industrial case studies we use to test the tool were provided, respectively, by Embraer S.A and Airbus Operations. The example from Embraer is fairly simple and the translation, although laborious, is trivial. This example uses the *abs* function, which reinforced the need to reconsider our treatment of functions, and motivated the use of libraries for specifying functions.

The case study from Airbus is larger, and the translation of four charts found in one of the files resulted in a 180-page specification. It also required the treatment of functions and the symbol t. We observe that the charts contained in both case studies extensively used functions (MATLAB functions and graphical functions), and that the approach to the treatment of general function by use of Z libraries has proved successful so far.

In its current version, the tool translates, without errors, all the tested examples, and the specifications generated by the tool are all correctly parsed and type checked by the *Circus* CZT toolkit [57].

4.6 Final considerations

In this chapter, we discussed the syntax of Stateflow charts, the conditions that establish the well-formedness of charts, the translation rules that guide the translation process, and the automation of the translation strategy implemented in the tool *s2c*. The syntax, well-formedness conditions and translation rules, as well as their automation, cover a large portion of the Stateflow notation, including many aspects that, as far as we know, are not covered in other approaches presented in the literature.

To the best of our knowledge, no other work has treated history junctions, unrestricted transitions and early return logic. Moreover, as far as we know, no one has provided a partial (or complete) treatment of functions. The main features that are usually not treated by other approaches to verification of Stateflow charts are: during actions, inter-level transitions, local event broadcast, flow-charts, and in expressions.

It is worth noticing, however, that some features that are specified in the syntax are not treated by the translation rules. They are temporal expressions, bind actions, function-call events, and constant, parameter and data store memory data. Graphical and Simulink functions are only partially treated. Our models can be easily extended to cover such function by integrating it with the existing models of Simulink diagrams. The reason the syntax covers features not treated by the translation rules is that this will allow us to extend the translation rules to cover such features in the future.

Our translation has been evaluated through a number of case studies that ranged in scale and complexity (from small feature-focused examples to case studies supplied by industrial collaborators). These experiments not only validated the translation strategy and the rules implemented by the *s2c* tool, but also provided additional confidence in the correctness of our model of the semantics of Stateflow presented in Chapter 3.

Furthermore, the use of the models generated by the tool as the basis for the verification of implementations described in the next chapter provided an additional validation of both

the tool and the translation rules implemented in it.

It is worth noting that while the choice of Z as the basis for the formalisation of the translation strategy proved suitable, a notation that supports automatic code generation might be more appropriate. The main disadvantage of the approach we have taken is that the translation strategy has to be implemented separately. In retrospect, since the formality of Z was not fully explored, our choice of notation for the translation rules should have favoured automatic code generation more heavily.

Chapter 5

Refinement strategy

In this chapter, we propose a refinement strategy that supports the verification of parallel and sequential implementations of Stateflow charts with respect to the models discussed in the previous chapters. Our aim is to describe a refinement strategy in enough detail that it can be automated.

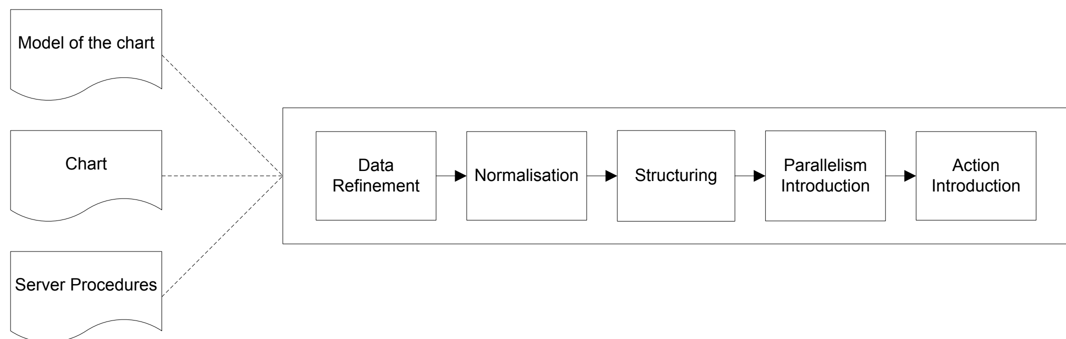


Figure 5.1: Overview of our refinement strategy.

Our refinement strategy aims at deciding whether an existing implementation of a Stateflow chart is a refinement of the model of the chart. The strategy takes as inputs that *Circus* model of the chart, the chart itself, and the procedures of the implementation that implement any parallel server. The refinement strategy is divided in five phases: data refinement, normalisation, structuring, parallelism introduction, and action introduction, and the inputs are used throughout these phases to support the application of the refinement laws. An overview is provided in Figure 5.1.

In the following, we first present an architecture with which we assume the implementations comply. Next we describe in details the refinement strategy for the verification of implementations that follow the described architecture. This architectural restriction is important because it allows us to increase both the level of detail in which the strategy can be specified, and the level of automation that can be achieved.

Our running example is shown in Figure 5.2; it is a chart (supplied with MATLAB

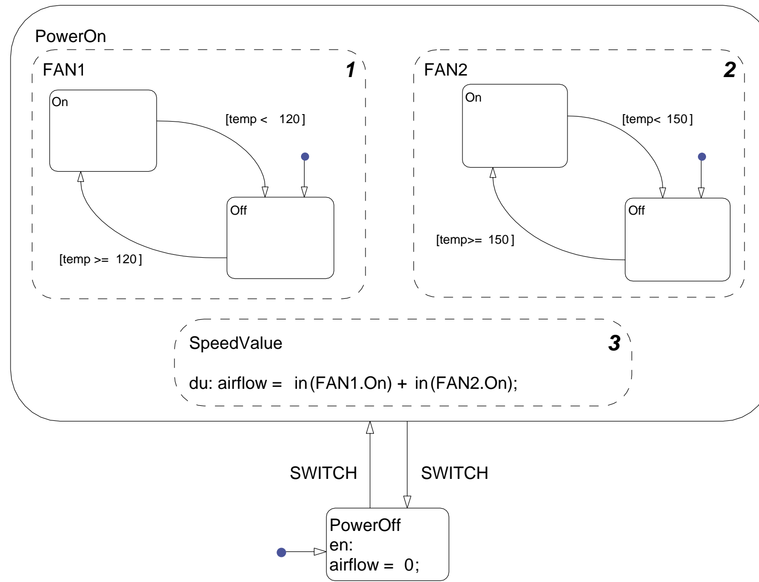


Figure 5.2: Air controller chart: supplied with Stateflow.

Stateflow) that models a temperature controller for a ventilation system. The chart has one input variable `temp`, one output variable `airflow`, and is triggered by two events: `SWITCH` and `CLOCK`. The event `CLOCK` is not shown in Figure 5.2; it is part of the Simulink diagram that includes the Stateflow block defined by this chart. In every step of the Simulink diagram, the Stateflow block is executed if `SWITCH` or `CLOCK` occurs. In the execution of the block, the chart is executed once for each of the events that occurred. This is recorded in the properties of the Stateflow block defined by the chart (even though it is not included in its diagrammatic description).

The *Circus* model of this example is obtained through the application of the translation rules described in the previous chapter and follows the structure detailed in Chapter 3. The complete model of this chart is in Appendix C.2

Section 5.1 describes the architecture of the implementations we consider, Section 5.2 discusses the *Circus* models of implementations, Section 5.3 presents the refinement strategy that supports the verification of implementations of Stateflow charts, and Section 5.4 summarises and discusses our results. In the sequel, we illustrate our approach using an implementation of the chart in Figure 5.2, and the *Circus* models of the chart and of the implementation. The implementation can be found in [63], and its model is in Appendix D.

5.1 Implementations of Stateflow charts

Our refinement strategy focuses on the implementations of Stateflow charts that may be generated by the Realtime Workshop [100] in association with the Stateflow Coder [98], but also covers programs that, perhaps, result from modifications of such implementations, but preserve specific architectural patterns. In particular, we are interested in parallel im-

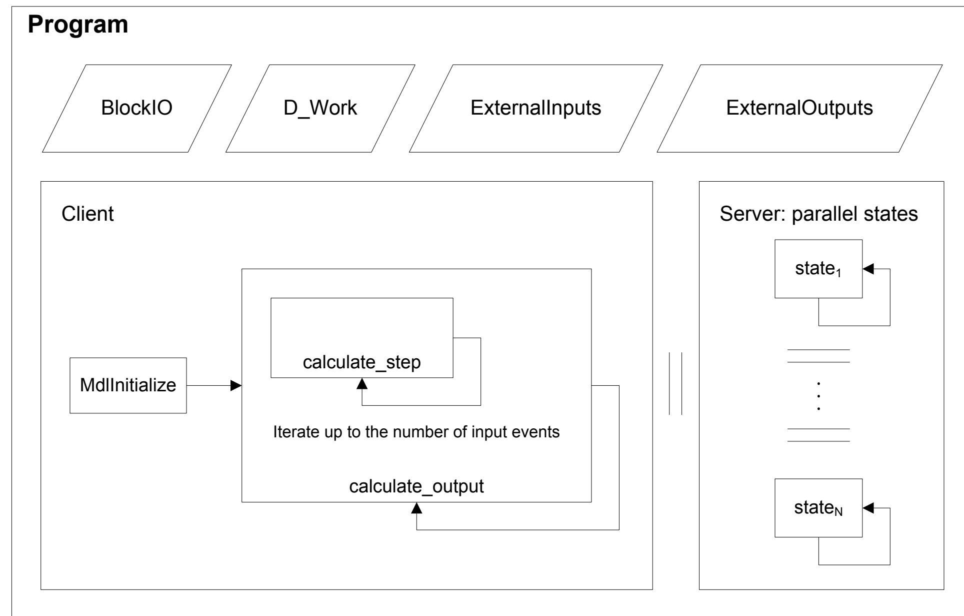


Figure 5.3: Architecture of implementations of Stateflow charts.

plementations of Stateflow charts, and while sequential implementations can be obtained automatically through the use of code generators, the same is not true for parallel ones. We propose a pattern for the introduction of parallelism in a sequential implementation, and equip our strategy with the means for verifying both sequential and parallel implementations.

The programming language used by Stateflow Coder, and that we adopt in examples, is a subset of C. Our strategy, however, is in no way restricted to C, but to the architectural pattern described here, which can be realised by programs written in other languages, like SPARK Ada [8], for instance.

Figure 5.3 gives an overview of the pattern. We distinguish two major aspects of the architecture: data types (represented by slanted boxes in Figure 5.3) and control flow (rectangular boxes). The first determines how information regarding the status of the states, history junctions, input, output and local data, and events are represented. The second defines how states and transitions are executed, and includes patterns for parallel execution of states. Section 5.1.1 discusses the data patterns, and Section 5.1.2 the execution patterns.

5.1.1 Architecture: data patterns

An implementation of a Stateflow chart uses a number of variables to record input, output and local data, input, output and local events, and execution data used to determine the state of the chart. The variables that store these values are grouped in records represented by the slanted boxes in Figure 5.3. They are used as types of global variables used to control

the execution of the chart. We also have an extra global variable that records the event under which the chart is being executed in a particular step. It is called `_sfEvent_C_`, where `C` is the name of the chart. For our example, the variable is `_sfEvent_Air`.

The variables of type `ExternalInputs` and `ExternalOutputs` are shared. They are used to communicate with the implementation of other blocks of the Simulink diagram. For instance, in the end of the cycle, the values of the output data and events are written to the `ExternalOutputs` record to make them available.

The `BlockIO` record groups the variables that store output data and events; it is used to construct their final values. For each output data variable `od`, a variable of the same name and type is included in `BlockIO`. For each output event `oe`, a variable of the same name and type Boolean is included; it records whether the event has occurred or not. The pattern adopted in the declarations of this record is as follows.

```
typedef struct { <type od;>*  <boolean_T oe;>* } BlockIO_C;
```

The corresponding record for the chart in Figure 5.2 is shown below; it contains only one variable that records the value of the output data variable `airflow`. (Its type, `uint8_T`, is defined by the code generator for the unsigned integers of 8 bits.) There are no output events in this example.

```
typedef struct { uint8_T airflow; } BlockIO_Air;
```

The second record, `D_Work`, contains the variables that model local data, variables that record the status of the states and history junctions, and output event counters. Its general structure is as follows.

```
typedef struct {
  <type ld;>*
  <uint8_T is_active_S1;>*
  <uint8_T is_S2;>*
  <uint8_T was_S3;>*
  <uint32_T oeEventCounter;>*
} D_Work_C;
```

For each chart local data variable `ld`, a corresponding variable of the appropriate `type` is declared. For each parallel state `S1`, that is, for each state `S1` in a parallel decomposition, a variable whose name is prefixed by `is_active_` is declared with type integer. Such a variable is also defined for the chart. For each state `S2` with a sequential decomposition, an integer variable whose name is prefixed by `is` is declared. For each state `S3` with a history junction, an integer variable whose name is prefixed by `was` is declared. For each output event `oe`, a variable of type integer whose name is postfixed by `EventCounter` is declared.

The encoding of the status of the individual states in `D_Work` relies on two groups of variables. The variables prefixed by `is_active_` record the status of parallel states and the chart. The variables prefixed by just `is_` record the status of sequential states: those

contained in a sequential decomposition. This encoding relies on the fact that at most one substate in a sequential decomposition is active at any given time. While all variables have type `uint8_T`, those in the first group are used as boolean variables, and the ones in the second store values that indicate which substate is active or that no substate is active.

The record `D_Work` in the implementation of the chart in Figure 5.2 is shown below. Since the chart has no local data, output events or history junctions, this record encodes only the status of states.

```
typedef struct {
  uint8_T is_active_c1_Air;
  uint8_T is_c1_Air;
  uint8_T is_active_FAN1, is_active_FAN2;
  uint8_T is_FAN1, is_FAN2;
  uint8_T is_active_SpeedValue;
} D_Work_Air;
```

The variable `is_active_c1_Air` records the status of the chart, and `is_active_FAN1`, `is_active_FAN2` and `is_active_SpeedValue` record, respectively, the status of the parallel states `FAN1`, `FAN2` and `SpeedValue`. The variable `is_c1_Air` records the status of the substates of the chart, and `is_FAN1` and `is_FAN2` record the statuses of the substates of, respectively, `FAN1` and `FAN2`.

The values of the `is_` variables are defined as constants. For our running example, they are shown below.

```
#define Air_IN_NO_ACTIVE_CHILD    (0U)
#define Air_IN_Off                (1U)
#define Air_IN_On                 (2U)
#define Air_IN_PowerOff           (1U)
#define Air_IN_PowerOn            (2U)
```

A `was_` variable, corresponding to a history junction, records the constant for the last active substates.

As already said, the third record, `ExternalInputs`, is used to share values of the input variables and events. The input events are represented by an array of real values. The size of the array corresponds to the number of input events, and the real values indicate whether the corresponding event occurred or not. The order in which the events are represented in the array is determined by an implicit ordering in the chart, record by Stateflow (although not shown in the diagram). The general structure of this record is as follows.

```
typedef struct {
  <type id;>*
  real_T inputevents[<nev>];
} ExternalInputs_C;
```

For each input data `id`, a variable of the same name and appropriate type is declared, and an array `inputevents` of real numbers (type `real_T`). Below, we show the corresponding record for our implementation. The variable `temp` corresponds to the input data of the same name, and the array `inputevents` is used to store values associated to each of the two events of the chart, `SWITCH` and `CLOCK`.

```
typedef struct {
  real_T temp;
  real_T inputevents[2];
} ExternalInputs_Air;
```

Finally, as already mentioned, the fourth record, `ExternalOutputs`, is used to communicate, at the end of the cycle, the values associated to the output data and events. Unlike input events, outputs events are communicated individually, not through an array. The pattern for this kind of records is shown below.

```
typedef struct {
  <type od;>*
  <boolean_T oe;>*
} ExternalOutputs_C;
```

For each output data `od`, a variable of the appropriate type and same name is declared, and for each output event `oe`, a boolean variable of the same name is declared. This record for our example is shown below.

```
typedef struct { uint8_T airflow; } ExternalOutputs_Air;
```

In the next section, we identify the patterns used to implement the chart's control flow.

5.1.2 Architecture: control flow

With respect to the execution flow of the chart, the relevant procedures of the program are depicted in Figure 5.3: `MdlInitialize`, `calculate_output`, `calculate_step`, and the procedures that correspond to the execution of the states implemented in parallel (generally denoted $state_1$ through $state_N$, in the figure).

The control flow is organised in a client-server pattern, where there is one client that iteratively calculates the outputs, and a number of servers that carry out the execution of particular states. If no states are executed in parallel, the implementation has no servers: the client fully implements the execution of the chart.

Each server consists of an iteration that waits for a request from the client, executes some code, and signals to the client the completion of the execution. In our implementation, these synchronisations are realised as barriers, and each server is run in a separate thread of execution.

The client initialises the execution of the chart by calling the procedure `MdlInitialize`, and iteratively calculates the outputs by calling the procedure `calculate_output`. The

calculation of the outputs depends on the procedure calculate_step, which implements the execution step of the chart.

The procedure MDLInitialize initialises the components of the records of type D_Work and BlockIO. For our example, MDLInitialize is sketched in Figure 5.4. It initialises the components of the variable Air_DWork of type D_Work_Air that represent state status to 0 (value 0U that represents an unsigned 0), indicating that every state is inactive. It also initialises the single component of the variable Air_B of type BlockIO_Air; the initial value of output data is defined in the chart. We omit in Figure 5.4 the commands that relate to aspects of the program that we do not model, like time control.

```
void MdlInitialize(void)
{
  ...
  Air_DWork.is_active_c1_Air = 0U;
  Air_DWork.is_c1_Air = 0U;
  Air_DWork.is_active_FAN1 = 0U;
  Air_DWork.is_active_FAN2 = 0U;
  Air_DWork.is_FAN1 = 0U;
  Air_DWork.is_FAN2 = 0U;
  Air_DWork.is_active_SpeedValue = 0U;
  Air_B.airflow = 0U;
}
```

Figure 5.4: Function MDLInitialize.

Figure 5.5 presents an overview of the procedure calculate_output. It processes the array inputevents, and calls calculate_step for each event that occurred. Once the chart is executed for all events, calculate_output shares the values recorded in the variable of type BlockIO, by copying them to the variable whose type is ExternalOutputs. Moreover, for each output event raised, it decrements the associated counter in the BlockIO record. In our example, this procedure is implemented by the function Air_output.

The procedure calculate_output generated by the Realtime Workshop [100] uses a local variable c_previousEvent to save the value of the global variable _sfEvent_C_, and restore it after each call to calculate_step. Our strategy uncovered that this is unnecessary. The only situation where the program is required to save and restore the value of _sfEvent_C_ is when there is a local event broadcast. For instance, if the broadcast is directed towards the chart, a variable c_previousEvent is used inside the procedure calculate_step. Our target implementations do not contain the redundant uses of this variable.

Figure 5.6 shows the structure of calculate_step, and how it uses the servers in a parallel implementation. The procedure calculate_step implements one complete execution of the chart. If the chart is triggered, this execution depends on the particular event being treated. In a parallel implementation, calculate_step is decomposed into other (server) procedures that encode the execution of particular states. Its structure con-

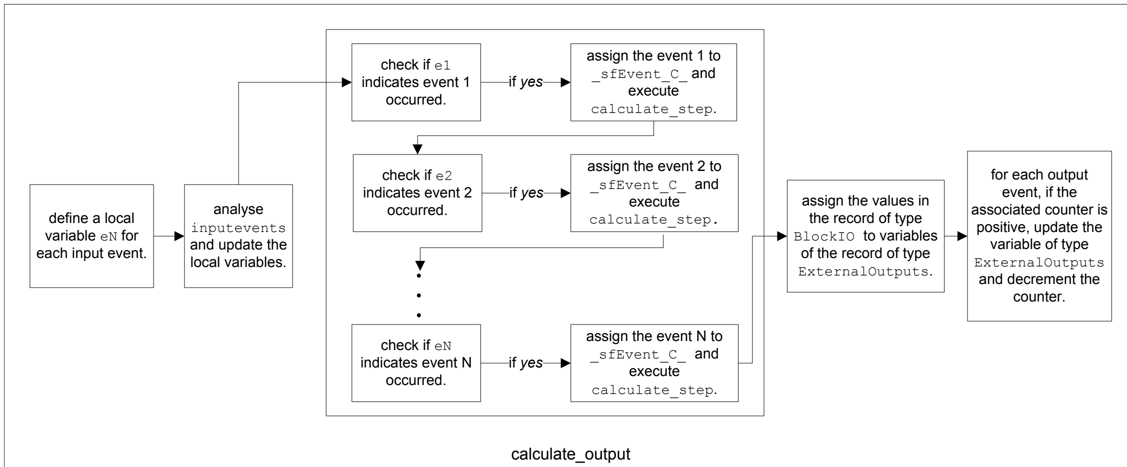


Figure 5.5: Structure of the procedure `calculate_output`.

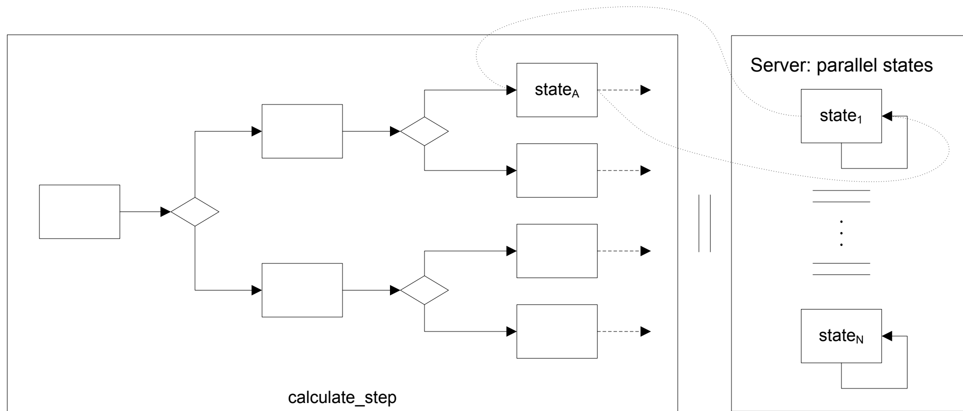


Figure 5.6: Structure of `calculate_step` and interaction patterns with the servers.

sists of a number of nested alternations that evaluate the status of the chart and states, and the transition guards, and proceeds according to the semantics embedded in the code generator.

In Figure 5.6, the rectangular boxes correspond to blocks of statements, and the diamond-shaped boxes to decision points. While, for clarity, our figure shows only binary decision points, they correspond in fact to both `if` and `switch` statements. The dotted lines indicate synchronisation points. The block $state_A$, for instance, represents a procedure that is to be executed in parallel with the server procedure $state_1$. In this case, before $state_A$ is executed, `calculate_step` synchronises with the appropriate server triggering the execution of $state_1$, executes $state_A$ in parallel, and waits for the triggered server to signal completion of its execution. At this point, the server iterates and waits for another request, and the client continues. This pattern generalises for any number of procedures implementing parallel states.

The decision points correspond to the evaluation of state (and chart) status, and transition guards. While the transition conditions can be rather complex, the evaluation of

status is extremely simple. In the case where the state `S` is in a parallel decomposition, an `if` statement with a condition `is_active_S == 0` is used. If `S` is in a sequential decomposition, a `switch` statement over the `is_` variable named after the parent state is used. Each `case` branch then treats one of the substates.

In our example, the procedure `calculate_step` is implemented by a function called `Air_chartstep_c1_Air`, an excerpt of which is shown in Figure 5.7. This function first checks whether or not the chart is active. If it is not active, it enters the state (omitted in the figure), otherwise it proceeds to check the status of the substates (`switch` statement). This illustrates the two forms a decision point can take.

```
static void Air_chartstep_c1_Air(void) {
  if (Air_DWork.is_active_c1_Air == 0) {
    ...
  } else {
    switch (Air_DWork.is_c1_Air) {
     case Air_IN_PowerOff:
      ...
      break;
     case Air_IN_PowerOn:
      ...
      break;
     default:
      ...
      break;
    }
  }
}
```

Figure 5.7: Procedure `calculate_step` for our example.

As mentioned before, the states in a parallel decomposition are executed in an order statically determined by the structure of the chart. In Figure 5.6, therefore, each decision (sub)tree with a block of statements (rectangular box) as a root may actually correspond to a sequence of decision trees: one for each parallel state.

Under certain circumstances, the order in which the parallel states are executed is not relevant (e.g. if they do not share variables). In this cases, they can be implemented in parallel using the client-server pattern previously described. For our example, the execution of the state `FAN1` is implemented in parallel with the execution of the state `FAN2`. Figure 5.8 shows the excerpt of the function `Air_chartstep_c1_Air` that executes the parallel states `FAN1`, `FAN2` and `SpeedValue`. The first call to `sychronise` prompts the server to start calculating, and the second acknowledges that it has finished its calculation. The `switch` statement enclosed between the two synchronisation points corresponds to the execution of the substates of `FAN2`. The last assignment executes the state *SpeedValue*.

The execution of a transition is carried out by an `if` statement whose condition is the conjunction of its trigger and condition. A group of transitions connected by junctions is

```
synchronise();
switch (Air_DWork.is_FAN2) {
 ...
}
synchronise();
Air_B.airflow = (uint8_T)((Air_DWork.is_FAN1 == Air_IN_On) +
  (Air_DWork.is_FAN2 == Air_IN_On));
```

Figure 5.8: Parallelism in the execution of the parallel states in our example.

executed by nested alternations executing each of the transitions. In this case, multiple branches of the alternations may signal the failure in the execution of the transitions, and initiate the execution of during actions and substates.

In general, we are not concerned with the particular structure of the decomposition of this procedure, except when it stems from the broadcast of local events. In this case, the whole chart or part of it is reexecuted under a new event. When the chart is reexecuted, the procedure `calculate_step` is called recursively. When the broadcast is directed at a particular state, only the portion of the execution that corresponds to the execution of the target state is reexecuted. In this case, this portion is decomposed into an auxiliary procedure, and a call to the new procedure is substituted for every instance of its body.

We do not treat programs that include mutual recursion, which might arise from the automatic generation of code from charts that have potentially nonterminating loops, or certain forms of event broadcast. Although, we can (easily) generate *Circus* models for them as explained in the next section, our refinement strategy needs to be generalised. This is not difficult, as pointed out in Section 5.4, but is left as future work. We observe that language subsets used in the safety-critical industry do not allow even the use of simple recursion.

5.2 *Circus* models of implementations

In this section, we discuss the *Circus* models of implementations that follow the architectural pattern presented in the previous section. The architecture of the models is close to that of the programs; Figure 5.9 gives an overview. The difference is that the *Circus* model has actions $read_inputs$ and $write_outputs$ that do not correspond to a program component; they encode the Stateflow block behaviour.

The *Circus* model is composed of a single process. Schemas $BlockIO_C$, D_Work_C, $ExternalInputs_C$, and $ExternalOutputs_C$, where as before C stand for the name of the chart, are used to model the record types of the program. The state of the process includes components C_B, C_DWork, C_U, and C_Y of these types, and a component $sfEvent_C$, all corresponding to global variables of the program. The (possibly parallel) main action reflects the client-server control pattern of the program.

The generation of *Circus* models of implementations involves two different aspects:

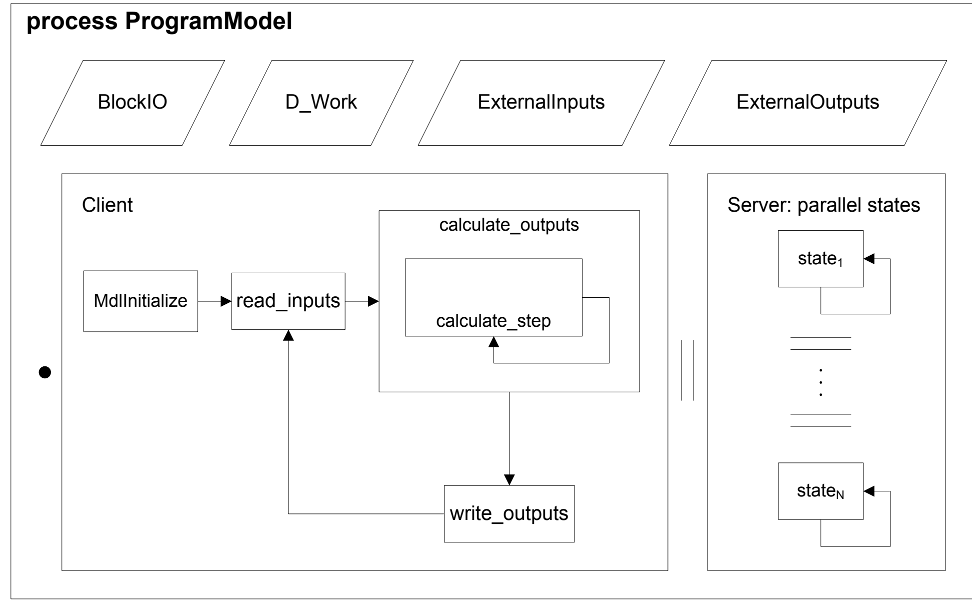


Figure 5.9: Overview of the models of implementations of Stateflow charts.

$$Client \mathrel{\widehat=} MdlInitialize \; ; \left(\mu X \bullet \left(\begin{array}{l} input_events?s \longrightarrow C_U.inputevents := s; \\ read_inputs \; ; \; calculate_output; \\ write_outputs \; ; \; end_cycle \longrightarrow X \end{array} \right) \right)$$

Figure 5.10: Action *Client* in implementation models.

translation and abstraction. The programming language statements are translated into *Circus* actions, and aspects of the program that are not covered by our models of Stateflow chart (for instance, time control) are abstracted.

The calculations of the time steps that determine the execution of a chart are abstracted by synchronisations over the channels *input_event* and *end_cycle* that mark, respectively, the start and end of a cycle (step). The way in which these channels are used to abstract time is shown in Figure 5.10, which presents the action *Client*. It initialises the state, and recursively reads input events, reads input data, calculates outputs, writes outputs, and signals the end of the cycle. (As an aside, we observe that the name of the action *calculate_output* is specific to each example; for our example, it is *Air_output*. Moreover, in the code generated by Realtime Workshop, the procedure `Air_output` has a parameter `tid`, probably for uniformity with programs generated for the Simulink diagram. Our strategy shows that this parameter is not needed, and we simplify the code and the model to avoid useless features. This is in line with guidelines adopted in the development of safety-critical systems.)

As illustrated in (the first two boxes of) Figure 5.5, the treatment of input events in the implementation involves calculations that identify which events occurred according

to values supplied by the Simulink model in an array `inputevents`. In our models, we abstract from this calculation by assuming that this is an array of booleans that indicate the occurrence of each event, not its associated value. As a consequence, we cannot reason about different types of input events (rising, falling, or either edge trigger).

Finally, sharing is modelled as communication. For instance, the use of a global shared variable of type `ExternalInputs` is modelled by an action that reads in interleaving the input variables and the array of events modelled as a sequence of boolean values, and writes them to the state component of type *ExternalInputs*. Similarly, the use of a global shared variable of type `ExternalOutputs` is modelled by an action that communicates the values of the components of the state component of type *ExternalOutputs* in interleaving. The channels used to read inputs and write outputs, and the channel used to communicate input and output events are the same channels used in the model of the chart being implemented. This ensures that the *Circus* models of the chart and of the program can be compared by refinement.

Another aspect of the implementations that involves sharing is the client-server pattern of parallel implementations. In our models, the different threads are modelled as *Circus* parallel actions *Client* and *Servers*. The state of the process is handled by *Client*, and *Servers* is itself the parallel composition of actions that model the procedures that execute parallel states. As for interleaving, the state needs to be partitioned between parallel actions to avoid racing conditions. We, therefore, model the initial synchronisation between *Client* and a server action as a communication that sends the complete state from the client to the server. The final synchronisations are modelled as a number of interleaved communications that send the components modified by each server to the client. Once all the values are received by *Client*, it updates the state.

Except for the aspects discussed above, the *Circus* models of the implementations are, in general, obtained by direct translation. *Circus* includes constructs that map directly into imperative programming languages. Records are translated into schemas, loops into recursive actions, `if` and `switch` statements are mapped to alternations, and procedures are mapped to named *Circus* actions.

5.3 Refinement strategy

In this section, we present a new refinement strategy suited for the verification of (parallel) implementations of Stateflow charts that follow the architectural pattern presented in Section 5.1. It is a tactic of refinement, which we define as a procedure for systematic application of refinement laws.

As previously mentioned, our refinement strategy is organised in five phases: data refinement, normalisation, structuring, parallelism introduction, and action introduction; an overview is provided in Figure 5.1. The main input is the model of the Stateflow chart, which as already explained, can be automatically generated. We also use the chart itself, for instance, to guide our calculation of a concrete state, and of a retrieve relation between

the concrete and abstract states. We also need to extract from the program information like the names of the states that are implemented in parallel, and the procedures that implement these states. (It is not difficult to automate the extraction of this information.) The strategy is a tactic that proves that the model of the chart is refined by that of the implementation by transforming the former into the latter using the algebraic laws of *Circus*.

In general terms, the data refinement phase introduces the data model of the implementation, the normalisation phase removes the structure of the abstract chart model, which reflects the Stateflow semantics, and the next three phases introduce the control aspects of the implementation architecture. The structuring phase introduces the sequential control pattern, the parallelism introduction phase introduces the client-server pattern, if present, and the last phase, action introduction, introduces the appropriate naming of actions as adopted in the program model ($read_inputs$, $calculate_output$, and so on).

In the sequel, we present each individual phase, and conclude with a discussion of the automation of this strategy, as well as the impact on our strategy of modifications to the architecture of programs.

5.3.1 Data refinement

The data refinement phase transforms the state of the chart process. The result is a process whose concrete state already includes many of the components of the implementation model. The exceptions are the components that correspond to output events in C_B, the component $inputevents$ of C_U, and the components C_Y and $sfEvent_C$, which are related to the treatment of input and output events, and output data, and are introduced later in the structuring phase. Figure 5.11 describes the steps of the data refinement phase. The laws for which we do not give a reference in this figure, and in others to follow, can be found in Appendix E.

We calculate the concrete state of the implementation model, and a retrieve relation that allows us to calculate a data refinement of the chart process using the *Circus* refinement calculus. It preserves the structure of the process, and transforms the assignments, operation schemas, and communications. In the sequel, we detail each of the steps. We use our running example for illustration.

Step 1. To support a higher degree of automation and enable certain simplifications later on (see Section 5.3.3.7), we calculate, besides the concrete state of the model of the implementation, properties of its components that become the concrete state invariant. Namely, we define three schemas: $BlockIO_C$, D_Work_C, and $ExternalInputs_C$, where, as before, C stands for the name of the chart, as follows.

$BlockIO_C$ For each output variable in the chart, we declare a component of the same name and type.

1. **Calculate the concrete state**. Use the provided templates.
2. **Calculate the retrieve relation**. Use the provided template.

Transform the chart process as follows.

3. **Introduce the abstract state invariant as an assumption after the initialisation.** Apply Law C.29 [76] to the initialisation schema in the main action, and distribute the assumption towards the assignments (apply Laws C.37-C.54 [76] exhaustively until all assignments are reached).
4. **Convert any actions of the form $\{inv\}\ ;\ v := e$.** Apply Law assign-schema-conv to all of them.
5. **Calculate the simulation.** Apply the *Circus* laws of action simulation to the chart process (Laws C.1-C.25 [76]).

Figure 5.11: Refinement strategy: data refinement phase.

D_Work_C To characterise the general form of D_Work_C, we consider the unique identifiers $LV_1, \ldots, LV_m$ for the chart local variables of types $T_1, \ldots, T_m$, the identifiers $PS_1, \ldots, PS_n$ for the parallel states and for the chart, the identifiers $SS_1, \ldots, SS_o$ of the states that have a sequential decomposition, possibly including the chart, and the identifiers $HS_1, \ldots, HS_p$ of the states that contain a history junction. Additionally, we consider the names $OE_1, \ldots, OE_q$ of the output events. To specify the invariant of D_Work_C, we also use *substates*, a function that associates (the identifier of) each state to the set of (identifiers of) its substates. With these, we can specify that D_Work_C is to be defined as follows.

$$
\begin{array}{l}
D_Work_C \\
\hline
LV_1 : T_1;\ \ldots;\ LV_m : T_m \\
is_active_PS_1, \ldots, is_active_PS_n : \mathbb{N} \\
is_SS_1, \ldots, is_SS_o : \mathbb{N} \\
was_HS_1, \ldots, was_HS_p : \mathbb{N} \\
OE_1EventCounter, \ldots, OE_qEventCounter : \mathbb{N} \\
\hline
\forall i : 1 \mathinner{\ldotp\ldotp} o \bullet is_SS_i \in \{C_IN_NO_ACTIVE_CHILD\} \cup \{s : substates(SS_i) \bullet C_IN_s\} \\
\forall i : 1 \mathinner{\ldotp\ldotp} p \bullet \\
\quad was_HS_i \in \{C_IN_NO_ACTIVE_CHILD\} \cup \{s : substates(HS_i) \bullet C_IN_s\}
\end{array}
$$

For each chart local variable, we declare a component of the same name and type. All the remaining components are of type $\mathbb{N}$. We also declare, for each parallel state S and the chart, a component is_active_S. For each S with a sequential decomposition, we declare is_S. For each history junction within a state S, we declare was_S, and for each

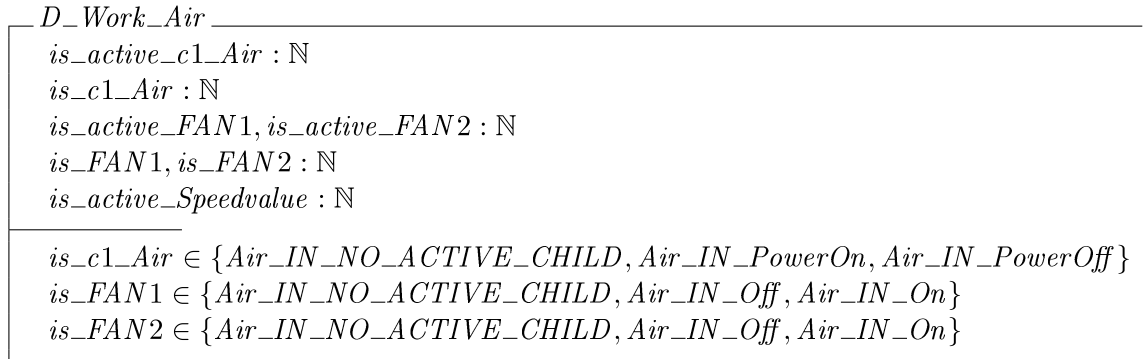


Figure 5.12: Schema *D_Work_Air*.

output event e, we declare a component *eEventCounter*. For each state S with a sequential decomposition, possibly including the chart, the invariant requires that the value of *is_S* is restricted to one of the identifiers of the substates of S, or *C_IN_NO_ACTIVE_CHILD*, when none of them are active. Similarly, the value of a *was_S* variable is restricted to the substates of S, or *C_IN_NO_ACTIVE_CHILD*, if none of them have been active yet. Figure 5.12 shows the *D_Work_C* schema for our example.

ExternalInputs_C For each input data, we declare a component of the same name and type. In the case of our example, we have a schema *ExternalInputs_Air* with a single component $v_temp : \mathbb{R}$.

The concrete state is defined by a schema *ConcreteState* with three components *C_B*, *C_DWork*, and *C_U*, whose types are the schemas defined above. In our example, we have *Air_B*, *Air_DWork*, and *Air_U*.

Step 2. The retrieve relation maps the components of *BlockIO_C* (that correspond to input and output variables) and the components of *D_Work_C* that correspond to local variables to components of *SimulationInstance*. The *eEventCounter* components of *D_Work_C* are mapped to the counters in the schema *SimulationInstance*. The components of *D_Work_C* that record the status of the states and the history junctions, precisely, those whose names are prefixed with *is_active_*, *is_* or *was_*, are mapped to the components *state_status* and *state_history* of *SimulationData*.

The correspondence between variables is trivial. It is obtained by equating each of the concrete variables to the corresponding abstract variable whose name is the same except for a prefix *v_*. For event counters, we also have a simple mapping: each *eEventCounter* component is equated to the corresponding *counter_e*.

The specification of the relation between the *is_active_* and *is_* prefixed variables and the function *state_status* in *SimulationData* is defined by equating *state_status* to a set comprehension that specifies the status of the states using the *is_active_* and *is_* variables.

RetrieveFunction

P_Air_S
$ConcreteState$

$$
\begin{array}{l}
v_airflow = Air_B.airflow \\
state_status = \{ s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet \\
\quad s = s_PowerOn \land active = \left(\begin{array}{l} \textbf{if } Air_DWork.is_Air = Air_IN_PowerOn \\ \textbf{then} \textbf{True} \\ \textbf{else False} \end{array} \right) \lor \\
\quad \ldots \\
\quad s = c_Air \land active = \textbf{if } Air_DWork.is_active_Air \neq 0 \textbf{ then True else False} \\
\} \\
state_history = \{\} \\
v_temp = Air_U.temp
\end{array}
$$

Figure 5.13: Total functional retrieve relation for our example.

Figure 5.13 gives the retrieve relation that we calculate for our example. The set comprehension is over state identifiers s, and booleans *active*. For each is_active_S variable, it requires $s = S \land active = (\textbf{if } C_DWork.is_active_S \neq 0 \textbf{ then True else False})$, where C_DWork is the concrete state component whose value is a binding of type D_Work_C. For each is_S variable and for each substate SS of S, the set comprehension requires $s = s_SS \land active = (\textbf{if } C_DWork.is_S = C_IN_SS \textbf{ then True else False})$. All these conditions are composed in a disjunction. In our example, the condition for the state *PowerOn* equates s to $s_PowerOn$, and *active* to **True** or **False** depending on whether the value of is_Air, which corresponds to the chart and records its active sequential state, is $Air_IN_PowerOn$ or not.

The general form of the retrieve relation that we calculate in this step is shown in Figure 5.14. It also specifies the relation between the *was_* prefixed variables and $state_history$ using a set comprehension. It simply uses Z (nested) conditional expressions to define, using the value of a was_S component, the value of the state identifier to be associated to s_S in $state_history$. Since our example does not contain history junctions, the `D_Work_Air` record in its implementation has no `was_` field. The model of the implementation, therefore, has no component that models the state component $state_history$ of the chart process. In this case, in the retrieve relation, we equate $state_history$ to the empty function.

In the definition in Figure 5.14, we use the following notation.

- Identifiers $PS_1, \ldots, PS_n$ for the parallel states and the chart.
- Identifiers $SS_1, \ldots, SS_o$ for the states that have a sequential decomposition, possibly including the chart.
- A function $substate(i, SS_j)$ that identifies the *i-th* substate of a state SS_j.
- Identifiers $HS_1, \ldots, HS_p$ for the states that contain a history junction.

- The number r_i of substates of SS_i.
- Identifiers $LV_1, \ldots, LV_m$ for the local variables.
- Identifiers $IV_1, \ldots, IV_s$ for the input variables, and $OV_1, \ldots, OV_t$ for the output variables.
- Names $OE_1, \ldots, OE_q$ of the output events.

The retrieve relation is always a total function because it is specified by a set of equations that defines each abstract state component as a total function of the concrete components.

Steps 3 and 4. Some of the assignments in the chart, and therefore, in the (abstract) chart process are implemented as assignments to components of records. In the implementation model, they become assignments to records themselves. For instance, the assignment $v_airflow := 0$ in the action *entryaction_PowerOff* in the chart process corresponds to an assignment $Air_B := \langle\!| airflow == 0 |\!\rangle$ in the model of the implementation. The simulation law for assignment, however, does not handle records directly, and therefore, in these steps we transform the assignments to schema operations.

For that, in Step 3, we introduce the abstract state invariant after the initialisation operation in the main action of the chart process, and distribute it to just before each of the assignments in *AllActions*. The application of the distribution laws raises no proof obligations because all data operations in the chart process preserve its state invariant. In Step 4, we convert the assumptions followed by the assignments into schema operations. For the assignment $v_airflow := 0$, we get the schema below.

*Assign*1

ΔP_Air_S

$$state_status' = state_status \land state_history' = state_history$$
$$v_temp' = v_temp \land v_airflow' = 0$$

Step 4 uses a novel, but very simple Law assign-schema-conv. It is defined in Appendix E.

Step 5. This is a standard data refinement step. Using the retrieve relation, we apply the *Circus* laws of simulation [76] to obtain a *Circus* process $C_P_Air_Controller$ by data refinement of the process $P_Air_Controller$. The state of the new process contains three components Air_B, Air_DWork, and Air_U.

5.3.2 Normalisation

The objective of this phase is to remove the top (parallel) structure of the chart model, which reflects the operational semantics of the Stateflow notation (see Chapter 3). This results in a model whose monolithic, but simple, process structure is adequate as a starting

RetrieveRelation

AbstractState
ConcreteState

$$
\begin{array}{l}
\forall i : 1 \mathrel{..} s \bullet v_IV_i = C_U.IV_i \\
state_status = \{ s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet \\
\quad s = s_PS_1 \wedge \\
\quad active = (\textbf{if } C_DWork.is_active_PS_1 \neq 0 \textbf{ then True else False}) \vee \\
\quad \ldots \vee \\
\quad s = s_PS_m \wedge \\
\quad active = (\textbf{if } C_DWork.is_active_PS_m \neq 0 \textbf{ then True else False}) \vee \\
\quad s = s_substate(1, S) \wedge \\
\quad active = (\textbf{if } C_DWork.is_SS_1 = C_IN_(substate(1, SS_1)) \textbf{ then True else False}) \vee \\
\quad \ldots \vee \\
\quad s = s_substate(r_1, SS_1) \wedge \\
\quad active = (\textbf{if } C_DWork.is_SS_1 = C_IN_(substate(r_1, SS_1)) \textbf{ then True else False}) \vee \\
\quad \ldots \vee \\
\quad s = s_substate(1, SS_n) \wedge \\
\quad active = (\textbf{if } C_DWork.is_SS_n = C_IN_(substate(1, SS_n)) \textbf{ then True else False}) \vee \\
\qquad \ldots \vee \\
\quad s = s_substate(r_n, SS_n) \wedge \\
\quad active = (\textbf{if } C_DWork.is_SS_n = C_IN_(substate(r_n, SS_n)) \textbf{ then True else False}) \\
\} \\
state_history = \{ s : \mathrm{dom}\, states;\ sub : \mathrm{dom}\, states \mid \\
\quad s = s_HS_1 \wedge \\
\quad sub = \left(\begin{array}{l} \textbf{if } C_DWork.was_HS_1 = C_IN_(substate(1, HS_1)) \\ \textbf{then } substate(1, HS_1) \\ \textbf{else} \left(\begin{array}{l} \ldots \\ \textbf{else} \left(\begin{array}{l} \textbf{if } C_DWork.was_HS_1 = C_IN_(substate(r_1, HS_1)) \\ \textbf{then } substate(r_1, HS_1) \\ \textbf{else } nullstate.identifier \end{array} \right) \end{array} \right) \end{array} \right) \\
\quad \vee \ldots \vee \\
\quad s = s_HS_p \wedge \\
sub = \left(\begin{array}{l} \textbf{if } C_DWork.was_HS_p = C_IN_(substate(1, HS_p)) \\ \textbf{then } substates(1, HS_p) \\ \textbf{else} \left(\begin{array}{l} \ldots \\ \textbf{else} \left(\begin{array}{l} \textbf{if } C_DWork.was_HS_p = C_IN_(substate(r_p, HS_p)) \\ \textbf{then } substates(r_p, HS_p) \\ \textbf{else } nullstate.identifier \end{array} \right) \end{array} \right) \end{array} \right) \\
\} \\
\forall i : 1 \mathrel{..} p \bullet v_LV_i = C_DWork.LV_i \\
\forall i : 1 \mathrel{..} q \bullet counter_OE_q = C_DWork.OE_q EventConter \\
\forall i : 1 \mathrel{..} t \bullet v_OV_i = C_B.OV_i
\end{array}
$$

Figure 5.14: Calculated total surjective functional retrieve relation: general form.

point for us to introduce the structure of the architectural pattern of implementations in the later phases of the refinement strategy.

1. **Remove the parallelism between the chart and the *Simulator* processes.** Apply the definition of process parallelism (Definition B.43 [76]).
2. **Move the hiding to the main action.** Apply the definition of process hiding (Definition B.45 [76]).

Transform the main action of the resulting process as follows.

3. **Isolate the initialisation operation.** Apply Law C.73 [76].
4. **Distribute the hiding.** Apply Law C.125 [76].
5. **Eliminate the hiding over the initialisation.** Apply Law C.120 [76].
6. **Evaluate the parallel recursions.** Apply Law rec-par-merge.
7. **Refine the initialisation.** Apply Law seq-assign-conv.

Figure 5.15: Refinement strategy: normalisation phase.

$$\bullet \left(\begin{array}{l} (CInitState \; ; \; \mu X \bullet (\mu Y \bullet (AllActions \; ; \; Y \;\Box\; end_cycle \longrightarrow \mathbf{Skip})) \; ; \; X) \\ \qquad \llbracket \{Air_B, Air_DWork, Air_U\} \mid interface \cup \{\!| end_cycle |\!\} \mid \{\} \rrbracket \\ (\mu X \bullet Step \; ; \; end_cycle \longrightarrow X) \end{array} \right) \setminus interface$$

Figure 5.16: Normalisation – Example: Main action after Steps 1 and 2.

The steps of this phase are described in Figure 5.15. We first eliminate the parallelism between the chart and *Simulator* processes, and then rewrite the main action of the resulting new process to a normal form: an initialisation action, followed by a recursive action that captures each step of execution of the chart.

Following the *Circus* definition of process parallelism [76], we construct the new process by taking the state of the chart process (since the *Simulator* process is stateless). For the main action, we combine those of the chart and *Simulator* processes in parallel in the same way the processes were combined. In a parallelism of *Circus* actions, however, to avoid race conditions, the variables in scope need to be partitioned among the interleaved actions. All actions have access to the value of all variables before the parallelism, but can write only to those in their own partition. In the parallelism created in the Step 1 of this phase, the name sets that define the partitions list the state components of the original processes.

We also use the definition of process hiding to move the hiding of the set *interface* of channels to the main action of the resulting process. For our example, it is shown in Figure 5.16. The parallel action $(\mu X \bullet Step \; ; \; end_cycle \longrightarrow X)$ is the main action of the *Simulator* process. As mentioned before, *Step* encodes the operational semantics of one step of execution of an arbitrary chart. The end of a step is marked by a synchronisation on the channel *end_cycle*.

The following steps, namely, Steps 2 to 7, transform the parallelism of recursions in the

$$\bullet \left(\begin{array}{l} Air_U := \langle\!| temp == 0 |\!\rangle \ ; \ Air_B := \langle\!| airflow == 0 |\!\rangle ; \\ Air_DWork := \langle\!| is_active_c1_Air == 0, \ldots |\!\rangle ; \\ \left(\mu X \bullet \left(\begin{array}{c} (\mu Y \bullet AllActions \ ; \ \ Y \Box end_cycle \longrightarrow \mathbf{Skip}) \\ [\![\ldots]\!] \\ Step; \ end_cycle \longrightarrow \mathbf{Skip} \end{array} \right) ; \ X \right) \setminus interface \end{array} \right)$$

Figure 5.17: Normalisation – Example: Main action at the end.

main action to obtain a single recursion; the result for our example is shown in Figure 5.17. For conciseness, we omit the name and synchronisation sets in the parallelism, which do not change.

The Steps 2 to 7 are very simple. The only interesting novelty is the specific Law rec-par-merge, presented below, which transforms a parallelism of recursions into a recursion of parallelisms. The channel *end* is instantiated to *end_cycle* in our strategy, and the actions A and B are instantiated to *AllActions* and *Step*.

Law[rec-par-merge]

$$(\mu X \bullet (\mu Y \bullet A \ ; \ \ Y \Box end \longrightarrow \mathbf{Skip}) \ ; \ \ X) [\![ns_1 \mid cs \mid ns_2]\!] (\mu X \bullet B \ ; \ \ end \longrightarrow X)$$

$$=$$

$$(\mu X \bullet ((\mu Y \bullet A \ ; \ \ Y \Box end \longrightarrow \mathbf{Skip}) [\![ns_1 \mid cs \mid ns_2]\!] B \ ; \ \ end \longrightarrow \mathbf{Skip}) \ ; \ \ X)$$

provided

- $end \in cs$; $end \notin usedC(A, B)$
- $wrtV(A) \cap usedV(B) = \varnothing$; $usedV(A) \cap wrtV(B) = \varnothing$

The two parallel recursions proceed in synchrony. This is enforced by the first proviso of the Law rec-par-merge, which states that the channel *end* is in the synchronisation set *cs* and is only used where explicitly shown. The syntactic function $usedC(A)$ gives the set of channels used in the definition of the action A [76]; also, we use $usedC(A, B)$ as an abbreviation for $usedC(A) \cup usedC(B)$.

In the parallel recursions, a communication over *end* terminates the inner recursion of the first parallel action, and, therefore, one step of its outer recursion, and one step of the recursion in the second parallel action. In the recursion of parallelisms, this synchronous behaviour is captured as a single recursion.

We use $usedV(A)$ to denote the set of variables used (read, but not written) by A, and $wrtV(A)$ the set of variables written by A. The second proviso of the Law rec-par-merge guarantees that each parallel recursion does not use the variables written by the other. This is necessary because, after each step of the recursion of parallelisms, the parallel actions have access to the new values of variables updated in the previous step. This is not the case in the parallel recursions, because the parallelism does not terminate.

In our verification, the application of Law rec-par-merge allows us to take advantage of

$$\left(\begin{array}{l} \mathit{MdlInitialize}; \\ \left(\mu X \bullet \left(\begin{array}{l} \mu Y \bullet \mathit{AllActions} \;;\; Y \,\Box\, \mathit{end_cycle} \longrightarrow \mathbf{Skip} \\ \qquad [\![ns_1 \mid \mathit{interface} \cup \{\!|\, \mathit{end_cycle} \,|\!\} \mid ns_2]\!] \\ \mathit{Step};\; \mathit{end_cycle} \longrightarrow \mathbf{Skip} \end{array}\right) ;\; X \right) \setminus \mathit{interface} \end{array}\right)$$

Figure 5.18: Structuring starting point.

the fact that, in both the chart and the *Simulator* process, each step of execution of the chart is marked by a synchronisation on *end_cycle*. For each of the steps of the simulator, an arbitrary number of executions of *AllActions* may be necessary to provide and update information about the chart components.

The other new law used in Figure 5.15 are very simple and are listed in Appendix E as usual.

5.3.3 Structuring

In this phase, we introduce the sequential control structure of the implementation: the structure of the client in Figure 5.9. For parallel implementations, the next phase (parallelism introduction) introduces the servers.

Starting point The steps of this phase are to be applied to the normalised process obtained in the previous phase. The general form of its main action is illustrated in Figure 5.17 and described in Figure 5.18, where *MdlInitialize* stands for a sequence of assignments that initialise the state. The action *Step* of the *Simulator* process is defined as shown and explained below. A detailed description is in Chapter 3.

$$\mathit{Step} \mathrel{\widehat=} \left(\begin{array}{l} \mathit{events}?\mathit{es} \longrightarrow \mathit{input_event}?\mathit{vs} : (\# \mathit{vs} = \# \mathit{es}) \longrightarrow \mathit{read_inputs} \longrightarrow \\ \mathit{ExecuteEvents}(\mathit{es}, \mathit{vs}); \\ \mathit{write_outputs} \longrightarrow \mathbf{Skip} \end{array}\right)$$

$$\mathit{ExecuteEvents} \mathrel{\widehat=} \mathit{es} : \mathrm{seq}\, \mathit{EVENT};\; \mathit{vs} : \mathrm{seq}\, \mathbb{B} \bullet (\mathrm{;}\;\; i : \mathit{id}(1 \,..\, \# \mathit{es}) \bullet \mathit{ExecuteEvent}(\mathit{es}(i), \mathit{vs}(i)))$$

$$\mathit{ExecuteEvent} \mathrel{\widehat=} e : \mathit{EVENT};\; v : \mathbb{B} \bullet \left(\begin{array}{l} \mathbf{if}\; v = \mathbf{True} \longrightarrow \mathit{ExecuteChart}(e) \\ [\!]\; v = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

Step requests from the chart a sequence *es* of input events, reads a sequence *vs* (of the same size) of boolean values associated to these events, requests the chart to read the input data, executes the chart for each event by calling another action $\mathit{ExecuteEvents}(es, vs)$, and requests the chart to write its outputs.

ExecuteEvents is defined by an iterated sequence of calls $\mathit{ExecuteEvent}(es(i), vs(i))$ that execute the chart for each of the events in *es* (and their associated values in *vs*). $\mathit{ExecuteEvent}(e, v)$ models the execution of the chart for the event e; the boolean parameter

v indicates whether it occurred or not. An alternation calls the action *ExecuteChart*(e) if e did occur, that is, $v =$ **True**. It is *ExecuteChart*(e) that models the execution of the chart for e. (Its definition is in Chapter 3).

Target In this phase, the state of the normalised process is extended to include the components *sfEvent_C*, which records the event being handled, and *C_Y*, which records the final value of the output data and events in each step, and to extend the record in *C_B*, to include components that keep the value of the output events as they are calculated during a step of execution, and the record in *C_U*, to include a component *inputevents* that keeps the input values associated with each event. Basically, the global variables and the components of the record-valued global variables of the program that are used in the treatment of inputs and outputs are introduced and allocated in the right record of the program data model.

The main action of the process resulting from this phase of refinement is completely sequential, and has no schema operations, just assignments. Its overall structure, which is depicted in Figure 5.9 and formalised in Figure 5.10, is described in more detail in Figures 5.19, 5.20, and 5.21: we also provide the structure of *calculate_output*, which corresponds to that of the model of the procedure `calculate_output` in Figure 5.5.

As already said, this main action initialises the state (*MdlInitialize*) and starts a recursion whose iterations model the implementation of the execution of one step of the chart. In each iteration, (1) the inputs are read using an action that we denote as *ReadInputs* in Figure 5.19; (2) some calculations are carried out, as defined by the sequence of alternations in Figure 5.19; (3) the outputs are written using an action whose pattern is defined in Figure 5.21; and (4) the end of the step is signalled using *end_cycle*.

In the calculations, for each input event represented by the i-th element of the sequence *C_U.inputevents*, if it occurred (*C_U.inputevents*$(i) =$ **True**), the action updates the value of the state component *sfEvent_C* to the corresponding event (E_i), and executes the chart. The order of the events is determined by the chart. The pattern of the actions that execute the chart is sketched in Figure 5.20.

If the chart is not active (*C_DWork.is_active_C* $= 0$), an action that executes the chart activates it by changing the value of *is_active_C* in the state component *C_DWork* to 1. This is modelled by an assignment to a binding component (or, more precisely, an assignment to a binding that changes the value of just one component). Afterwards, the action executes the tasks required to enter the states of a chart. This is omitted in Figure 5.20 and further discussed later. If the chart is active, the action checks which substates S_i are active (if any), and executes them. In Figure 5.20, we show the pattern for a chart with a sequential decomposition and substates S_1, S_2, and so on.

If a state contains a local event broadcast to the whole chart, the pattern of the execution of that state includes a recursive call in a pattern illustrated in Figure 5.20 as part of the execution of S_1. It uses a local variable *c_previousEvent* to record the current event, updates *sfEvent_C* to the broadcast event *LE*, and makes a recursive call to reexecute the

$$
\begin{array}{l}
\mathit{MdlInitialize};\\
\left(\mu X \bullet \left(\begin{array}{l}
\mathit{ReadInputs};\\
\left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{if } C_U.\mathit{inputevents}(1) = \textbf{True} \longrightarrow\\
\quad \mathit{sfEvent_C} := E_1\ ;\ \ldots \text{ (Figure 5.20)}\\
[]\ C_U.\mathit{inputevents}(1) = \textbf{False} \longrightarrow \textbf{Skip}\\
\textbf{fi}
\end{array}\right);\\
\ldots;\\
\left(\begin{array}{l}
\textbf{if } C_U.\mathit{inputevents}(n) = \textbf{True} \longrightarrow\\
\quad \mathit{sfEvent_C} := E_n\ ;\ \ldots \text{ (Figure 5.20)}\\
[]\ C_U.\mathit{inputevents}(n) = \textbf{False} \longrightarrow \textbf{Skip}\\
\textbf{fi}
\end{array}\right)
\end{array}\right);\\
\text{(Figure 5.21)}\\
;\ \mathit{end_cycle} \longrightarrow X
\end{array}\right)\right)
\end{array}
$$

Figure 5.19: Structuring target.

$$
\left(\mu Y \bullet \left(\begin{array}{l}
\textbf{var } \mathit{c_previousEvent} : \mathbb{N} \bullet\\
\left(\begin{array}{l}
\textbf{if } C_DWork.\mathit{is_active_C} = 0 \longrightarrow\\
\qquad C_DWork := \langle\!|\ \mathit{is_active_C} == 1, \ldots |\!\rangle\ ;\ \ldots\\
[]\ C_DWork.\mathit{is_active_C} > 0 \longrightarrow\\
\quad \left(\begin{array}{l}
\textbf{if } C_DWork.\mathit{is_C} = C_IN_S_1 \longrightarrow\\
\quad \ldots \left(\begin{array}{l}
\mathit{c_previousEvent} := \mathit{sfEvent_C};\\
\mathit{sfEvent_C} := LE;\\
Y;\\
\mathit{sfEvent_C} := \mathit{c_previousEvent};
\end{array}\right);\ \ldots\\
[]\ C_DWork.\mathit{is_C} = C_IN_S_2 \longrightarrow\\
\quad \ldots \left(\mu Z \bullet \left(\begin{array}{l}
\textbf{if } ct \longrightarrow \ldots Z\\
[]\ \neg\ ct \longrightarrow \textbf{Skip}\\
\textbf{fi}
\end{array}\right)\right);\ \ldots\\
[]\ \ldots\\
\textbf{fi}
\end{array}\right)\\
\textbf{fi}
\end{array}\right)\right)\right)
$$

Figure 5.20: Structuring target - chart execution.

$$
\begin{array}{l}
\left(\begin{array}{l}
\textbf{if } C_DWork.\mathit{counter_E} > 0 \longrightarrow\\
\qquad C_DWork := \langle\!|\ \mathit{counter_E} == (C_DWork.\mathit{counter_E} - 1), \ldots |\!\rangle\ ;\ C_B.E := \textbf{True}\\
[]\ C_DWork.\mathit{counter_E} = 0 \longrightarrow E := \textbf{False}\\
\textbf{fi}
\end{array}\right);\\
\ldots;\\
C_Y := C_B;\\
o_E!(C_Y.E) \longrightarrow \textbf{Skip} \mathbin{|||} \ldots \mathbin{|||} o_v!(C_Y.v) \longrightarrow \textbf{Skip} \mathbin{|||} \ldots
\end{array}
$$

Figure 5.21: Structuring target - writing the outputs.

chart under the new event. Afterwards, *sfEvent_C* is restored.

The pattern of the implementation of a transition loop (involving just junctions) is shown in Figure 5.20 as part of the execution of S_2. It introduces a recursion, where an alternation decides the termination of the loop. We use *ct* to stand for the check of the guard (condition and trigger) of the first transition in the loop. If the transition is to be followed, we eventually have a recursive call. If not, the loop is terminated.

As previously mentioned, Figure 5.21 describes the form of the action that communicates the output events and variables. Through a channel *o_E* corresponding to an event E, we communicate a boolean, indicating whether E occurred or not. This is determined by a preceding alternation that checks the counter for E in *C_DWork*, and stores the result of the check in a new component of C_B also named E. If the counter is positive, its value is decremented. The assignment $C_Y := C_B$ records in the C_Y component, which corresponds to a shared variable of the program, the calculated values of the output variables and events recorded in C_B. The interleaving of communications realises the sharing by outputting the value of the components of C_Y. Through *o_E* we communicate the value of E as stored in C_Y, and through a channel *o_v* we communicate the value of the variable v stored in C_Y.

Refinement steps Figure 5.22 shows the steps of the structuring phase; each of them is the application of a separate refinement procedure specified later in this section. The first step introduces local variables that later become part of the concrete state, the following four steps introduce different elements of the control structure of the implementation architecture, and the last step simplifies the resulting actions. In the sequel we give an overview of these steps and of the refinement procedures.

Step 1 The procedure input-event-var-introduction is presented in Section 5.3.3.1. It introduces *inputevents* and *sfEvent_C* as local variables in the parallel action *Step* (originally in the *Simulator* process), and then extends their scope. This gives *Step* control over the new state components in the parallelism: as a result of these steps, *inputevents* and *sfEvent_C* are added to the name set ns_2 of the parallelism (see Figure 5.18).

Step 2 In general terms, the procedure parallelism-resolution systematically applies, to the body of the outer recursion in the main action, step laws to resolve the parallelism between the recursion offering *AllActions*, and *Step*. As it does so, it unravels the structure embedded in *AllActions* and *Step*: alternations that check the status of chart states, the occurrence of events, and guards of transitions. The result, which we sketch in Figures 5.23, 5.24, and 5.25, is an action whose structure is closer to that of the implementation model, but may still retain some parallel actions. This arises if the chart has local event broadcasts or transition loops (involving just junctions). These parallelisms are the target of the next step of the structuring phase.

The overall structure (Figure 5.23) is a sequence, where *ReadInputs* is followed by

1. **Introduce input event variables.** Apply procedure input-event-var-introduction.
2. **Introduce alternations in *calculate_output*.** Apply the procedure parallelism-resolution to the body of the outer recursion in the main action.
3. **Introduce recursions that implement event broadcast and transition loops.** Apply the procedure recursion-introduction to the body of the outmost recursion in the main action.
4. **Introduce assignments.** Apply procedure assignment-introduction to the body of the outmost recursion in the main action.
5. **Introduce update of outputs.** Apply procedure update-output to the second action in the sequence that defines body of the outmost recursion in the main action.
6. **Simplify.** Apply the procedure simplification to the recursion in the main action.

Figure 5.22: Refinement strategy: structuring phase

$$\left(\begin{array}{l} \mathit{ReadInputs}; \\ \left(\begin{array}{l} \mathbf{if}\ C_U.\mathit{inputevents}(1) = \mathbf{True} \longrightarrow \mathit{sfEvent_C} := E_1\ ;\ \ldots;\ \text{(Figure 5.24)} \\ [\!]\ C_U.\mathit{inputevents}(1) = \mathbf{False} \longrightarrow \\ \quad \left(\begin{array}{l} \ldots \\ \left(\begin{array}{l} \mathbf{if}\ C_U.\mathit{inputevents}(n) = \mathbf{True} \longrightarrow \\ \quad \mathit{sfEvent_C} := E_n\ ;\ \ldots;\ \text{(Figure 5.24)} \\ [\!]\ C_U.\mathit{inputevents}(n) = \mathbf{False} \longrightarrow \mathit{end_cycle} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \end{array}\right) \\ \mathbf{fi} \end{array}\right) \end{array}\right)$$

Figure 5.23: Structuring: body of the outmost recursion in the main action after Step 2.

nested alternations that check the value of each of the input events in order. If the i-th event occurred ($C_U.\mathit{inputevents}(i) = \mathbf{True}$), then E_i is recorded as the current event ($\mathit{sfEvent_C} := E_i$), and the chart is executed. This involves internal communications (omitted in Figure 5.23 and eliminated in Step 6), followed by a variable block as in Figure 5.24.

The body of the variable block also contains internal communications followed by another group of nested alternations that check for the active states and valid transitions. It is as part of executing a state or following a transition that there may remain parallelisms. Those that are sketched in Figure 5.24 correspond to a local event broadcast (as part of executing S_1) and a transition loop involving just junctions (as part of executing S_2). Once all the checks and executions are carried out, further alternations (similar to those in Figure 5.23) check whether the remaining events, if any, have occurred. When all events have been considered, the outputs are produced, and a synchronisation on *end_cycle* signals the

$$
\begin{array}{l}
\mathbf{var}\ c_previousEvent : \mathbb{N} \bullet \ldots ; \\
\left(\begin{array}{l}
\mathbf{if}\ C_DWork.is_active_C = 0 \longrightarrow \ldots \\
[\!]\ C_DWork.is_active_C \neq 0 \longrightarrow \ldots \\
\quad \left(\begin{array}{l}
\mathbf{if}\ C_DWork.is_C = C_IN_S_1 \longrightarrow \\
\quad \ldots \left(\begin{array}{l}
c_previousEvent := sfEvent_C\ ;\ sfEvent_C := LE; \\
\left(\begin{array}{l}
\mu Y \bullet AllActions\ ;\ Y \Box end_local_execution \longrightarrow \mathbf{Skip} \\
\qquad [\![ns_1 \mid cs \mid ns_2 \cup \{sfEvent_C\}]\!] \\
ExecuteChart(sfEvent_C)\ ;\ end_local_execution \longrightarrow Skip
\end{array}\right) ; \\
sfEvent_C := c_previousEvent
\end{array}\right) ; \ldots \\
[\!]\ C_DWork.is_C \neq C_IN_S_1 \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ C_DWork.is_C = C_IN_S_2 \longrightarrow \\
\quad \ldots \left(\begin{array}{l}
\mathbf{if}\ ct \longrightarrow \\
\quad \ldots \left(\begin{array}{l}
(\mu Y \bullet AllActions\ ;\ Y \Box end_cycle \longrightarrow \mathbf{Skip}) \\
\qquad [\![ns_1 \mid cs \mid ns_2 \cup \{sfEvent_C\}]\!] \\
ExecuteTransition(t, p, s, sfEvent_C)
\end{array}\right) \\
[\!]\ \neg\ ct \longrightarrow \ldots \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ C_DWork.is_C \neq C_IN_S_2 \longrightarrow \ldots \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

Figure 5.24: Structuring: part of the main action after Step 2.

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{if}\ C_DWork.counter_E > 0 \longrightarrow \\
\quad C_DWork := \langle\!| counter_E == (C_DWork.counter_E - 1), \ldots |\!\rangle; \\
\quad o_E!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
[\!]\ C_DWork.counter_E = 0 \longrightarrow o_E!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
\qquad |\![\{C_DWork.counter_E\} \mid \ldots]\!| \\
\ldots
\end{array}\right) \\
\qquad |\![\ldots \mid \ldots]\!| \\
(o_v!(C_B.v) \longrightarrow \mathbf{Skip}\ |||\ \ldots)
\end{array}\right)
$$

Figure 5.25: Structuring: main action after Step 2 - writing outputs.

end of the step.

Figure 5.25 describes the action that communicates the outputs. It shows that the main action resulting from this step also retains interleavings. They arise from the parallel communications of output events and variables, and are tackled in Step 5 to produce an action in the target form defined in Figure 5.21. Like parallel actions, interleaved *Circus* actions are associated with name sets that partition the variables in scope to avoid race conditions. In fact, as defined in [76], parallelisms and interleavings do not accept partitions at the level of fields of individual record-valued components as in Figure 5.25. For this, we need to extend the definition of parallelism (and, consequently, interleaving). In the *Circus* semantics, the state after a parallel action is determined by a merge operation that takes

$$
\left(\begin{array}{l}
\mu Y \bullet \\
\quad \left(\begin{array}{l}
\mathbf{var}\ c_previousEvent : \mathbb{N} \bullet \ldots; \\
\quad \left(\begin{array}{l}
\mathbf{if}\ C_DWork.is_active_C = 0 \longrightarrow \ldots \\
[\!]\ C_DWork.is_active_C \neq 0 \longrightarrow \ldots \\
\quad \left(\begin{array}{l}
\mathbf{if}\ C_DWork.is_C = C_IN_S_1 \longrightarrow \\
\quad \ldots \left(\begin{array}{l}
c_previousEvent := sfEvent_C; \\
sfEvent_C := LE; \\
Y; \\
sfEvent_C := c_previousEvent
\end{array}\right); \ldots \\
[\!]\ C_DWork.is_C \neq C_IN_S_1 \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ C_DWork.is_C = C_IN_S_2 \longrightarrow \\
\quad \ldots \left(\mu Z \bullet \left(\begin{array}{l}
\mathbf{if}\ ct \longrightarrow \ldots Z \\
[\!]\ \neg\ ct \longrightarrow \ldots \\
\mathbf{fi}
\end{array}\right)\right) \\
[\!]\ C_DWork.is_C \neq C_IN_S_2 \longrightarrow \ldots \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right)
$$

Figure 5.26: Structuring: body of the outmost recursion in the main action after Step 3.

into account the partition. So it is enough to change this merge operation to support the kind of partitioning that we are using.

The details of the procedure parallelism-resolution are presented in Section 5.3.3.2. It is rather extensive, as it has to consider the several forms of parallelism that can arise from the application of the step laws.

Step 3 This deals with the parallelisms left in the previous step, if any. The procedure recursion-introduction defines new recursive actions and proves their equivalence to the existing parallel actions. The result of this step transforms the main action so that variable blocks like those in Figure 5.24 change: in the case illustrated, where we have a local event broadcast to the chart as a whole, the block itself becomes a recursive action as sketched in Figure 5.26. We present recursion-introduction in Section 5.3.3.4.

Step 4 This step refines the main action so that all calls to actions defined by schema operations are refined to assignments, by applying the procedure assignment-introduction. It is presented in Section 5.3.3.5. After this step, we have no abstract specification of data operations left, just assignments.

Step 5 As already mentioned, as a result of Step 2, output events and data are communicated in interleaving (see Figure 5.25). The objective of this step is to gather together the construction of the values to be output, before making all outputs available (still in

interleaving). As a result, we introduce the action described in Figure 5.21. The procedure update-output used for that is described in Section 5.3.3.6.

Step 6 To conclude the structuring phase, we have a final simplification step. The operational semantics of Stateflow charts, as specified in the *Simulator* process of the chart models, considers all possible paths of execution that might arise in the execution of an arbitrary chart. Typically, the semantics of a particular chart, as defined by the chart process, does not involve all these paths. We, therefore, in carrying out the Steps 2 and 4 above, which basically evaluate the semantics of the chart in a systematic way, may introduce unnecessary assignments, and alternations. They are eliminated in this step.

Unnecessary assignments arise, for example, when a transition loop leads to a path that exits and subsequently enters the same state. In this case, during this step we remove the sequence of two assignments that record the state as inactive and then active. Unnecessary alternations arise, for example, when absence of local event broadcast makes it unnecessary to check early return logic conditions. In this case, during this step we remove alternations whose conditions can be shown to be always true or false.

Additionally, in this step, we remove unnecessary local variables (that arise from the action that models early return logic checks). Finally, we also eliminate internal channels that are originally used for communication between the chart and *Simulator* processes. These communications are no longer necessary, since the parallelism between these processes has been eliminated.

The procedure simplification applied in this step is described in Section 5.3.3.7.

5.3.3.1 Procedure input-event-var-introduction

The steps of this procedure are presented in Figure 5.27. The first two steps are related to the introduction of *inputevents*, and the Steps 3 and 4 to the introduction of $sfEvent_C$. The final step promotes both variables to state components. The Law var-assign-intro used in Steps 1 and 3 to introduce both variable allows us to introduce both a local variable declaration and an assignment to initialise the new variable. The new, but simple, laws used in Steps 2 and 4 extend the scope of a variable declaration.

5.3.3.2 Procedure parallelism-resolution

This procedure takes as parameters a set $loopT$ of transitions that start loops, a set $treatedT$ of such transitions that have already been treated by the procedure, which is initially empty, and the parallel action A being refined. It is a recursive procedure defined in terms of the syntactic structure of A.

In the sequel, we describe how each form of parallelism is refined by this procedure. For each (non-trivial) case, we present up to three figures: the first defines a pattern of parallel actions, the second specifies the refinement steps to be applied to actions in this pattern, and the third gives an overview of the resulting refined action. The third figure

1. **Introduce variable *inputevents*.** Apply Law var-assign-intro to the action prefixed by the communication $input_event?vs : (\# vs = \# es)$. The type to be used for *inputevents* is $\text{seq}\,\mathbb{B}$, and the initialisation value is vs.
2. **Extend the scope of *inputevents* to the whole action.** Apply Law var-prefix-ext twice, apply Law C.138 [76] to the parallelism, Law var-tail-rec-ext to the recursion, and finally Law var-seq-ext-left to the outer sequence.
3. **Introduce variable *sfEvent_C* of type $\mathbb{N}$.** Apply Law var-assign-intro to the call to *ExecuteChart*. The type to be used for *sfEvent_C* is $\mathbb{N}$, and the initialisation value is $es(i)$.
4. **Extend the scope of *sfEvent_C* to the whole action.** Apply Law var-alt-dist to the alternation, apply Law var-iter-seq-ext to the iterated sequential composition, Law C.100 [76] to action prefixed by *read_inputs*, Law C.137 [76] to the right-hand side of the parallelism, Law C.138 [76] to the parallelism, Law var-tail-rec-ext to the recursion, and Law var-seq-ext-left to the outer sequence.
5. **Promote *sfEvent_C* and *inputevents* to a state component.** Apply Law A.5 [14] to the process twice.

Figure 5.27: Refinement strategy: structuring phase - input-event-var-introduction

$$(\mu Y \bullet AllActions \; ; \; Y \,\Box\, end_cycle \longrightarrow \mathbf{Skip}) \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; c \longrightarrow A$$

Figure 5.28: parallelism-resolution: prefixing over channel in the synchronisation set on the right-hand side.

also boxes the subactions of the result to which the procedure is applied recursively. The possible cases are determined by the form of the *AllActions* and *Step* actions of our model.

A. Parallel composition unity (base case) The rather trivial first case is the base case of our procedure: a parallelism $\mathbf{Skip} \, [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \, \mathbf{Skip}$. It requires the application of Law C.90 [76] to obtain **Skip**.

B. Prefixing over channel in the synchronisation set on the right-hand side This case covers the situation where the right-hand side (originally, the process *Simulator*) is requesting the left-hand side (originally the chart process) to execute an action. This is characterised in Figure 5.28, where c is in cs.

In our refinement strategy, for every model, this is the first case applicable, by the definition of *Step*, the action on the right-hand side of the target of the Step 2, which uses parallelism-resolution. As explained above, *Step* is a prefixing over a communication through *events*, which is in the synchronisation set.

The refinement has to evaluate the communication. Since, the synchronisation is offered by *AllActions*, we need to unfold the recursion. The precise steps are described in Figure 5.29.

1. **Unfold recursion.** Apply Law C.128 [76] to the left-hand side of the parallelism.
2. **Expand action calls.** Apply to the outermost occurrence of *AllActions* the copy rule exhaustively, except to calls to *broadcast*.
3. **Distribute sequence.** Apply Law C.112 [76].
4. **Distribute parallelism.** Apply Law C.87 [76] from right to left.
5. **Introduce deadlocks.** Apply Law C.92 [76] exhaustively.
6. **Eliminate deadlocks.** Apply Law C.114 [76] exhaustively.
7. **Recurse.** Apply the procedure parallelism-resolution.

Figure 5.29: parallelism-resolution: steps for prefixing over channel in the synchronisation set on the right-hand side.

$$\boxed{c \longrightarrow B \; ; \; (\mu Y \bullet \mathit{AllActions} \; ; \; Y \; \Box \; \mathit{end_cycle} \longrightarrow \mathbf{Skip}) \; [\![\; ns_1 \mid cs \mid ns_2 \;]\!] \; c \longrightarrow A}$$

Figure 5.30: parallelism-resolution: result for prefixing over channel in the synchronisation set on the right-hand side.

In the first step, we unfold the recursion on the left-hand side of the parallelism to obtain an external choice between *AllActions* and $\mathit{end_cycle} \longrightarrow \mathbf{Skip}$. In the second step, we expand the definition of *AllActions* and all its subactions (except for the chart action *broadcast*, since this is not needed and would lead to nontermination, due to recursive calls to *AllActions* itself). The result is of the following form.

$$(((c_1 \longrightarrow \ldots \Box \ldots \Box \; c_n \longrightarrow \ldots) \; ; \; (\mu Y \bullet \mathit{AllActions} \ldots)) \; \Box \; \mathit{end_cycle} \longrightarrow \mathit{Skip}) \; [\![\; ns_1 \mid cs \mid ns_2 \;]\!] \; c \longrightarrow A$$

It is an external choice, which is sequentially composed with the original recursion; this sequence is itself in choice with $\mathit{end_cycle} \longrightarrow \mathbf{Skip}$, and this external choice is in parallel with the right-hand actions. Steps 3 and 4 distribute the sequential composition over the external choice, and then the parallel composition over the resulting external choice, to obtain an external choice of parallel actions. Each parallel action so obtained is prefixed by a single communication. Step 5 evaluates the deadlocked parallelisms to **Stop**, and Step 6 eliminates them from the choice (using the unit law of external choice). Because for each possible request c there is necessarily a unique matching initial communication in *AllActions*, we are guaranteed to have exactly one non-deadlocked parallelism after Step 4. Finally, we recursively apply parallelism-resolution to the remaining parallel action. The parameters of the recursive call are *loopT* and *treatedT* unchanged, and the boxed action in Figure 5.30, which gives the form of the action resulting in this case.

For our example, when this case is applied, we obtain the action below (before recurs-

ing).

$$\left(\begin{array}{l} events!\langle e_SWITCH, e_CLOCK\rangle \longrightarrow \mathbf{Skip} \; ; \; (\mu Y \bullet AllActions \; ; \; Y \Box end_cycle \longrightarrow \mathbf{Skip}) \\ \quad [\![\{Air_B, Air_DWork, Air_U\} \mid interface \cup \{\!| end_cycle |\!\} \mid \{inputevents, sfEvent_Air\}]\!] \\ (events?es \longrightarrow input_event?vs : (\# vs = \# es) \longrightarrow inputevents := vs \; ; \; read_inputs \longrightarrow \ldots) \end{array}\right)$$

Since the first communication of *Step* is an input $events?es$, the matching output through *events* in *AllActions* is revealed. The assignment to *inputevents* now on the right-hand side of the parallelism was introduced as a result of the Step 1 of the structuring phase described previously.

C. Synchronisation This case occurs as a result of refinement carried out in the previous case; it evaluates the parallel prefixed actions that are unfolded (see Figure 5.30) using standard step laws. If we have a simple synchronisation, we apply Law C.105 [76]. If we have a pair of input and output communications $c?x$ and $c!e$, we apply Law C.107 [76] to obtain a(n internal) communication $c.e$ followed by a parallelism in which e is substituted for x on the input side (and to which we apply the procedure parallelism-resolution recursively, with parameters *loopT* and *treatedT* unchanged as in the previous case).

Proceeding with our example, we obtain the action below as a result of applying this case.

$$\begin{array}{l} events.\langle e_SWITCH, e_CLOCK\rangle \longrightarrow \\ \left(\begin{array}{l} \mathbf{Skip} \; ; \; (\mu Y \bullet AllActions \; ; \; Y \Box end_cycle \longrightarrow \mathbf{Skip}) \\ [\![\{Air_B, Air_DWork, Air_U\} \mid interface \cup \{\!| end_cycle |\!\} \mid \{inputevents, sfEvent_Air\}]\!] \\ (input_event?vs : (\# vs = 2) \longrightarrow inputevents := vs \; ; \; read_inputs \longrightarrow \ldots) \end{array}\right) \end{array}$$

The communication on *events* is evaluated and extracted from the parallelism. Additionally, in the input side of the parallelism, the value of e is determined by the output: $\# es$, for instance, can be resolved to 2. (In the case of es, this also allows us to remove the use of the iterated sequence operator in *ExecuteEvents*.)

D. Leading Skip on either side In this case we clear any leading **Skip** actions that are left over on either side of the parallelism, or more precisely, simplify actions $(\mathbf{Skip} \; ; \; A) \; [\![\; ns_1 \mid cs \mid ns_2 \;]\!] \; B$ for example. For that, we use a unit law of sequence (Law C.132 [76]), before recursing. The similar case where the assignment is on the right-hand side can be handled by a similar law (or by relying on commutativity of parallelism).

In the sequel, for every case, like this one, where the pattern of interest can be characterised by either parallel action, we explain what to do when it appears on the left-hand side. The refinement for the situation where it appears on the right-hand side can either rely on commutativity or on a set of similar laws.

Proceeding with our example, after this case, we remove the **Skip** before the recursion involving *AllActions*.

$$(c \longrightarrow A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B \text{ or } A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c \longrightarrow B)$$

Figure 5.31: parallelism-resolution: prefixing over channel not in the synchronisation set on either side.

E. Prefixing over channel not in the synchronisation set on either side This case deals with the communications with the environment. The relevant channels are the input and output channels (used in the left parallel action) and *input_event* (used in the right action). In general, this case covers parallelisms as shown in Figure 5.31. A simple step law prefix-parallelism-dist-2 carries out the refinement to extract the communication from the parallelism. We can show that its provisos always hold, since the structure of the process we are refining is quite restricted. For instance, it is easy to show that for all communications that are relevant to this case, the first possible communication on the opposite side of the parallel composition is over a channel in the synchronisation set. (On the left, we have communications on input and output channels; according to the definition of *Simulator*, at these points *Step* is always waiting for an internal communication. On the right, we have a communication on *input_events*, and the definition of *AllActions* shows that its initials only contains communications in the synchronisation set.)

Proceeding with our example, we obtain the action below.

$$\begin{array}{l} events.\langle e_SWITCH, e_CLOCK\rangle \longrightarrow input_event?vs : (\# vs = 2) \longrightarrow \\ \left(\begin{array}{l} \mu Y \bullet AllActions \,;\; Y \,\Box\, end_cycle \longrightarrow \mathbf{Skip} \\ \llbracket \{Air_B, Air_DWork, Air_U\} \mid interface \cup \{\!| end_cycle |\!\} \mid \{inputevents, sfEvent_Air\} \rrbracket \\ (inputevents := vs \,;\; read_inputs \longrightarrow \ldots) \end{array} \right) \end{array}$$

The communication with the environment on *input_events* is extracted from the parallelism.

F. Leading assignment on either side This case covers parallel actions of the form $(v := e \,;\, A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B$ or $A\llbracket ns_1 \mid cs \mid ns_2 \rrbracket (v := e \,;\, B)$. Basically, we use our simple law assign-par-dist to extract the assignment from the parallelism. Its provisos require that the assigned variables are in the name set of the parallelism, and are not used by the other parallel action. These always hold by construction, since assignments result from the refinement of schema operations in the normalisation phase, or from the introduction of local variables in the Step 1 of the structuring phase. In the first case, the assignments change state components, which are always used exclusively by the left-hand side of the parallelism, and whose names are in the appropriate name set by construction of the model. In the second case, the local variable and the assignment are introduced in one side of the parallelism, thus the other side does not use them, and when the scope of the local variable is expanded over the parallelism on the other side, the variable name is included in the appropriate name set. Therefore, in both cases, the provisos raised by this case are satisfied.

Proceeding with our example, the refinement in this case extracts the assignment

$$\left(\left(\begin{array}{c} A_1 \\ \lVert[ns_1 \mid \bigcup i : 2 \mathrel{..} n \bullet ns_i]\rVert \\ (\ldots \lVert[\, ns_{n-1} \mid ns_n \,]\rVert\, A_n) \end{array} \right) ;\ B \right) [\![\, ns_x \mid cs \mid ns_y \,]\!]\ C$$

Figure 5.32: parallelism-resolution: leading interleaving on the left-hand side.

$$\left(\begin{array}{c} A_1 \\ \lVert[ns_1 \mid \bigcup i : 2 \mathrel{..} n \bullet ns_i]\rVert \\ (\ldots \lVert[\, ns_{n-1} \mid ns_n \,]\rVert\, A_n) \end{array} \right) ;\ \boxed{(B\ [\![\, ns_x \mid cs \mid ns_y \,]\!]\ C)}$$

Figure 5.33: parallelism-resolution: result for leading interleaving on the left-hand side.

inputevents := *vs* from the parallelism. Moreover, further recursive applications of previous cases extract the internal communication *read_inputs*, and the external communication in *AllActions* over the channel *i_temp* corresponding to the input variable `temp`. In general, however, there may be several input variables and, consequently, several input communications in interleaving. This is the object of the next case.

G. Leading interleaving on the left-hand side This case covers actions characterised as shown in Figure 5.32. We have on the left an interleaving of actions $A_1, \ldots A_n$, followed by an action B, all in parallel with another action C. Each A_i is associated in the interleaving with a name set ns_i, so that the name set of the interleaving of actions A_i to A_n is the union of the sets ns_i to ns_n.

In this case, we simply apply the step law C.84 [76] to extract the interleaving from the parallelism (and recursively call the parallelism-resolution procedure). The result is shown in Figure 5.33.

The provisos of Law C.84 always hold by the construction of the model. This case only arises after the refinement related to communications over *read_inputs* and *write_outputs*. In the first case, the first communication on the right-hand side is over the channel *chart* in the synchronisation set, and in the second case, the first communication is over *end_cycle*, which is also in the synchronisation set. All the communications in the interleaving are over channels not in the synchronisation set because they communicate inputs and outputs of the chart. The variables written by the interleaving are not used by the right-hand side of the parallelism. In the case the interleaving preceded by *read_inputs*, the variables written by the interleaving are the input variables, and in the case of the interleaving preceded by *write_outputs*, the written variables correspond to event counters. In both cases, these variables are only written by the action on the left-hand side of the parallelism, which originates from the chart process, and are contained in its name set. Moreover, both parallel actions are divergence free, also by construction of the model and its refinement.

H. Alternation followed by sequence, on either side As already mentioned, the structure of *AllActions* and *Step* involves a number of alternations. This case covers their treatment, considering actions of the form shown in Figure 5.34 or the similar cases where

$$\left(\left(\begin{array}{l} \mathbf{if}\ g \longrightarrow A_1 \\ [\!] \neg\, g \longrightarrow A_2 \\ \mathbf{fi} \end{array} \right) ;\ B \right) [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ C$$

Figure 5.34: parallelism-resolution: alternation followed by sequence, on either side.

the alternation is on the right-hand side of the parallelism. All alternations have mutually exclusive guards g and $\neg\, g$.

Proceeding with our example, at this stage, we have the action below, where the alternations now shown on the right parallel action are originally part of *ExecuteEvent*.

$$\begin{array}{l} events.\langle e_SWITCH, e_CLOCK \rangle \longrightarrow input_event?vs : (\#\, vs = 2) \longrightarrow inputevents := vs; \\ read_inputs \longrightarrow \\ \left(\begin{array}{l} \mu Y \bullet AllActions\ ;\ Y \,\square\, end_cycle \longrightarrow \mathbf{Skip} \\ [\![\{Air_B, Air_DWork, Air_U\} \mid interface \cup \{\!|\, end_cycle \,|\!\} \mid \{inputevents, sfEvent_Air\}]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} \mathbf{if}\ inputevents(1) = \mathbf{True} \longrightarrow \\ \qquad sfEvent_Air := e_SWITCH\ ;\ ExecuteChart(sfEvent_Air) \\ [\!]\ inputevents(1) = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right) ; \\ \left(\begin{array}{l} \mathbf{if}\ inputevents(2) = \mathbf{True} \longrightarrow \\ \qquad sfEvent_Air := e_CLOCK\ ;\ ExecuteChart(sfEvent_Air) \\ [\!]\ inputevents(2) = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right) ; \\ write_outputs \longrightarrow end_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \end{array} \right) \end{array}$$

In general, the alternations can involve checks that depend on the current state of the chart, for instance, the verification of the guard of a transition, or can decide how to proceed based solely on the structure of the chart. These arise from the generality of the semantics encoded in the process *Simulator*. The two types of alternation can be easily distinguished by the names used in the guards. If a guards refers to state components, it is of the first type. If not, it is of the second type, and can be eliminated using the static information that defines the structure of the chart. This is achieved in this step. In our example above, the alternation refers to the state component *inputevents* and is not eliminated.

The refinement steps to be carried out in this case are shown in Figure 5.35, and the possible outcomes of this case are shown in Figure 5.36. We first distribute the sequential composition over the alternation, and then the parallelism using the fact that the guards of the alternation are mutually exclusive. Next, if the guard refers to state components, we recursively apply the procedure parallelism-resolution to the actions in each branch. If the guard does not refer to state components, we remove the alternation using the definitions of constants like *s_Off*3, *states*, and *transitions* (see Appendix C). Finally, we recursively apply this procedure to the remaining action with the remaining parameters unchanged.

The result depends on whether the alternation is eliminated or not: both option are

1. Apply Law alt-seq-dist.
2. **Distribute parallelism.** Apply Law alt-par-dist.
3. If g refers to a state component, recursively apply parallelism-resolution to each branch. Otherwise, apply Law alt-elim, before recursing.

Figure 5.35: parallelism-resolution: steps for alternation followed by sequence, on either side.

$$\left(\begin{array}{l} \textbf{if}\ g \longrightarrow \boxed{(A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_1)} \\ \square\ \neg\ g \longrightarrow \boxed{(A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2)} \\ \textbf{fi} \end{array}\right) \quad \text{or} \quad \boxed{(A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B)}$$

Figure 5.36: parallelism-resolution: possible results for alternation followed by sequence, on either side.

shown in Figure 5.36.

If we proceed with our example, we obtain the following action.

$$\begin{array}{l} events.\langle e_SWITCH, e_CLOCK\rangle \longrightarrow input_event?vs : (\# vs = 2) \longrightarrow inputevents := vs\ ;\ read_inputs \longrightarrow \\ \left(\begin{array}{l} \textbf{if}\ inputevents(1) = \textbf{True} \longrightarrow \\ \quad \left(\begin{array}{l} \mu Y \bullet AllActions\ ;\ Y \Box end_cycle \longrightarrow \textbf{Skip} \\ \quad \llbracket \ldots \rrbracket \\ \left(\begin{array}{l} sfEvent_Air := e_SWITCH\ ;\ ExecuteChart(sfEvent_Air); \\ \left(\begin{array}{l} \textbf{if}\ inputevents(2) = \textbf{True} \longrightarrow \\ \quad sfEvent_Air := e_CLOCK\ ;\ ExecuteChart(sfEvent_Air) \\ \square\ inputevents(2) = \textbf{False} \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right); \\ write_outputs \longrightarrow end_cycle \longrightarrow \textbf{Skip} \end{array}\right) \end{array}\right) \\ \square\ inputevents(1) = \textbf{False} \longrightarrow \\ \quad \left(\begin{array}{l} \mu Y \bullet AllActions\ ;\ Y \Box end_cycle \longrightarrow \textbf{Skip} \\ \quad \llbracket \ldots \rrbracket \\ \left(\begin{array}{l} \left(\begin{array}{l} \textbf{if}\ inputevents(2) = \textbf{True} \longrightarrow \\ \quad sfEvent_Air := e_CLOCK\ ;\ ExecuteChart(sfEvent_Air) \\ \square\ inputevents(2) = \textbf{False} \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right); \\ write_outputs \longrightarrow end_cycle \longrightarrow \textbf{Skip} \end{array}\right) \end{array}\right) \\ \textbf{fi} \end{array}\right) \end{array}$$

At this stage, recursive calls to parallelism-resolution lead to a number of applications of the previous cases, until we reach a parallelism whose right-hand parallel action is a call to *ExecuteChart*.

I. Call action on either side Whenever a call action is the leading action in one of the sides of the parallelism, and none of the other cases apply, we expand it using a procedure copy. This simply replaces a call to an action with its definition, with the appropriate parameters substituted. This procedure is described in Section 5.3.3.3. It basically applies

1. If action called is $ExecuteTransition(t, p, s, e, success)$ and $t \in loopT$.
   (a) If $t \notin treatedT$
      i. **Expand call.** Apply procedure copy to expand the definition.
      ii. **Recurse.** Recursively apply the procedure parallelism-resolution to the parallel action with parameters $loopT$ and $treatedT \cup \{t\}$.
   (b) else, do nothing.
2. else
   (a) **Expand call.** Apply the copy procedure.
   (b) **Recurse.** Apply the procedure parallelism-resolution with the original parameters.

Figure 5.37: parallelism-resolution: steps for call action on either side.

the copy rule, dealing with both value and value-result parameters, and eliminates the spurious local variables introduced as a consequence. With an application of this case, we can proceed with our example to expand the call to *ExecuteChart*.

If the action that is called is the *Simmulator* action *ExecuteTransition*, however, the refinement to be carried out in this case is different. Figure 5.37 shows the refinement steps for this case, where calls to *ExecuteTransition* are singled out. *ExecuteTransition* models the execution of a sequence of transitions; it takes as parameters the identifier *tid* of the first transition, the sequence *path* of identifiers of the transitions that have been successfully followed so far, the *source* state of the transition path, the current event *ce* and the value-result parameter *success* used to indicate the success or failure of the execution. When executing a transition loop through a call to *ExecuteTransition*, the uncontrolled application of the procedure copy leads to nontermination. To avoid that, we use the parameters $loopT$ and $treatedT$ of parallelism-resolution.

If in a call to *ExecuteTransition*, the first argument t is the identifier of a transition that starts a loop ($t \in loopT$), there are two possible situations: a call to *ExecuteTransition* with t as argument has already been expanded previously ($t \in treatedT$), or not. If this is the first call to *ExecuteTransition* with t as argument to which parallelism-resolution is applied, the call is expanded as usual, using the procedure copy, and parallelism-resolution is applied recursively action with parameters $loopT$ and $treatedT \cup \{t\}$, that is, t is marked as "treated". If t is already in $treatedT$, we leave the parallelism unresolved. It is refined later in the Step 3 of the structuring phase. If the call to *ExecuteTransition* has an argument t that does not start a loop ($t \notin loopT$), or the call is not to *ExecuteTransition*, we expand the call, and recursively apply parallelism-resolution to the parallel action with the remaining parameters unchanged.

Whatever the outcome of this case, we are left with a parallel action. Additionally, if as a result we have a recursive call to `parallelism-resolution`, we now have eliminated the action call.

$$(SchAct\ ;\ (\mu Y \bullet AllActions\ ;\ Y \Box end_cycle \longrightarrow \mathbf{Skip}))\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A$$

Figure 5.38: parallelism-resolution: leading schema operation on the left-hand side.

$$SchAct\ ;\ \boxed{((\mu Y \bullet AllActions\ ;\ Y \Box end_cycle \longrightarrow \mathbf{Skip})\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A)}$$

Figure 5.39: parallelism-resolution: result for leading schema operation on the left-hand side.

$$A\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ (\mu X \bullet B[X])$$

Figure 5.40: parallelism-resolution: explicit recursion on the right-hand side.

$$\boxed{A\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ B[(\mu X \bullet B[X])]}$$

Figure 5.41: parallelism-resolution: result for explicit recursion on the right-hand side.

J. Leading schema operation on the left-hand side In our example, after expanding *ExecuteChart*, several of the previous cases apply. Eventually, we reach a parallelism where on the left-hand side we have a schema operation. The process *Simulator* does not have schema operations, and so this kind of action can only appear on the left. They are used to model chart actions (entry, during, and so on) and activation and deactivation operations. Just like assignments, these schemas are extracted from the parallelism.

Figure 5.38 shows the general pattern covered in this case; as said above, it arises from applications of the previous cases to unfold the recursion on *AllActions*. Figure 5.39 presents the result of this step of the procedure, which is just an application of Law C.73 [76]. The provisos require that the variables modified by the schema are in the name set of the parallelism, and are not used by the other parallel action. These always hold by construction of the model. (The *Simulator* process does not change or use the state of the chart process.) As usual, we recursively apply the procedure to the remaining parallel action as indicated in Figure 5.39. The other parameters of the recursive application are *loopT* and *treatedT* (both unchanged).

K. Explicit recursion on the right-hand side The action *ExecuteChart* involves communication that request the execution of a chart action. These are specified as sequences of *Circus* actions, some of which may correspond to local event broadcasts. In *Step*, at this point, we have, therefore, a call to one of the *Simulator* actions *LocalEventEntry*, *LocalEventDuring*, *LocalEventExit*, *LocalEventCondition* or *LocalEventTransition*, depending on the type of chart action. These *Simulator* actions are all recursions (that offer a choice between treating a local event and recursing, or signalling the end of the chart action). Additionally, at several points in *Step*, we have a call to *transitionActionCheck*, which is also a recursion (that checks the status of the substates of another state as part of the modelling of the check of early return logic condition).

In this case, we unfold the recursions. It applies to actions of the form shown in

1. **Distribute sequence.** Apply Law C.112 [76].
2. **Distribute parallelism.** Apply Law C.87 [76].
3. **Introduce deadlocks.** Apply Law C.92 [76].
4. **Eliminate deadlocks.** Apply Law C.114 [76].
5. **Recurse.** Apply the procedure parallelism-resolution.

Figure 5.42: parallelism-resolution: steps for leading prefixing over channel in the synchronisation set on the left-hand side.

Figure 5.40, applies Law C.128 [76], and produces the action in Figure 5.41, to which parallelism-resolution is applied recursively. This does not lead to nontermination because any chart action involves only a finite number of local event broadcasts, and every state has a finite number of substates.

L. Leading prefixing over channel in the synchronisation set on the left-hand side A *Circus* action that models a local event broadcast uses internal channels (*local_event* and *end_action*, both in the synchronisation set) to control the *Simulator*. In this case, we consider a parallelism where the left-hand action is a communication on such a channel. In the right-hand action, at these points, *Step* always offers a(n external) choice that accepts the communication, followed by some other action. The general pattern that is covered is $c \longrightarrow A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c \longrightarrow B_1 \Box d \longrightarrow B_2) ; C$, where both c and d are in cs.

In this step, refinement resolves the external choice. The particular refinement steps to be carried out are shown in Figure 5.42. We distribute the sequence and the parallelism over the external choice, eliminate the deadlocked parallel actions. The provisos of the laws applied follow from the fact that c and d are both in the synchronisation set, and are different. The result is the action $c \longrightarrow A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket c \longrightarrow (B_1 ; C)$; a recursive application of parallelism-resolution to it solves the synchronisation.

M. Local event broadcast In the *Simulator* actions *LocalEventEntry*, *LocalEventExit*, *LocalEventDuring*, *LocalEventCondition* or *LocalEventTransition*, we have a call to the action *TreatLocalEvent*. It decides whether to reexecute the whole chart or a specific chart state, depending on the kind of broadcast.

This call to *TreatLocalEvent* cannot be expanded indiscriminately (as in the case **call action on either side**, for instance) because the resulting action can involve a recursive call to *ExecuteChart*, or to *ExecuteState*, the *Simulator* action that models the execution of a chart state. The situation is similar to that of calls to *ExecuteTransition*, which are singled out in the case **call action on either side**. Here, however, we have a different pattern because we do not have a tail recursion.

Calls to *TreatLocalEvent* occur in parallel with the *Circus* actions of the chart process that model local event broadcasts. In those, we have recursions, which provide the services

$$\left(\begin{array}{c} (\mu Z \bullet AllActions\ ;\ Z \Box end_local_execution \longrightarrow \mathbf{Skip})\ ;\ A) \\ [\![ns_1 \mid cs \mid ns_2]\!] \\ (TreatLocalEvent(e,s)\ ;\ B \end{array}\right)$$

Figure 5.43: parallelism-resolution: local event broadcast.

1. **Use state component** *sfEvent_C* Apply Law use-state-comp to $TreatLocalEvent(e, s)$ to introduce a local variable *c_previousEvent* of type $\mathbb{N}$, with initial value *sfEvent_C*, whose temporary value becomes e.
2. **Extend and distribute.** Apply Laws var-seq-ext-right and C.138 [76], and apply Law C.84 [76] twice.
3. **Expand action call.** Apply the copy procedure to $TreatLocalEvent(e, s)$.
4. **Simplify the alternation.** Try to apply Law alt-elim to eliminate each of the branches of the alternation that defines *TreatLocalEvent*.
5. **Distribute the parallelism.** Apply Law par-seq-dist.
6. **Recurse.** Apply the procedure parallelism-resolution to the second parallelism.

Figure 5.44: parallelism-resolution: steps for local event broadcast.

of *AllActions* for the *Simulator*, while it reexecutes the chart, or part of the chart, as a consequence of the broadcast. Termination of this recursion is triggered by a synchronisation on the channel *end_local_execution*, whose role is similar to that of *end_cycle* in the main action of the chart process. It is concerned, however, with executions triggered by a local event broadcast.

The pattern for this case is as shown in Figure 5.43. The objectives of the refinement are twofold. First, we resolve the decision embedded in *TreatLocalEvent*, which is based solely on the syntactic structure of the chart: namely, the target of the broadcast. Second, we split the parallelism to isolate the encoding of the execution of the broadcast from that of the continuation of the execution of the chart. Figure 5.44 presents the refinement steps to be carried out, and Figure 5.45 the resulting action, which contains a sequence of parallelisms: the first corresponds to the execution of the local event broadcast and is further refined in the Step 3 of the structuring phase, and the second is the target of a recursive application of parallelism-resolution. In splitting the parallelism, a local variable *c_previousEvent* is used to store the current event in *sfEvent_C*, before it is updated to the broadcast event, so that later, the value of *sfEvent_C* can be restored.

Step 1 applies the novel, but simple, Law use-state-comp presented below.

$$\left(\begin{array}{l}\mathbf{var}\ c_previousEvent : \mathbb{N} \bullet \\ \quad\left(\begin{array}{l} c_previousEvent := sfEvent_C\ ;\ sfEvent_C := e; \\ \left(\begin{array}{l}\mu X \bullet AllActions\ ;\ X \Box end_local_execution \longrightarrow \mathbf{Skip} \\ \quad [\![ns_1 \mid cs \mid ns_2]\!] \\ ExecuteChart(sfEvent_C)\ ;\ end_local_execution \longrightarrow \mathbf{Skip}\end{array}\right); \\ \boxed{(A\ [\![ns_1 \mid cs \mid ns_2]\!]\ sfEvent_C := c_previousEvent\ ;\ B)}\end{array}\right)\end{array}\right)$$

Figure 5.45: parallelism-resolution: result for local event broadcast.

Law[use-state-comp]

$A(e) = (\mathbf{var}\, x : T \bullet x := v;\ v := e;\ A(v);\ v := x)$

where *e, of type T, is a value argument of A.*
provided

- $v \notin FV(A)$;
- *x is fresh.*

This law applies to an action call $A(e)$, where e is a value argument. A more general version explicitly allows for more arguments, as is the case of the call $TreatLocalEvent(e, s)$ in the pattern in Figure 5.43. For simplicity, we only indicate e in the specification of Law use-state-comp. It can be used to declare a fresh local variable x (of type T), which is initialised with the value of a variable v not used in A, which is then updated to hold the value of e temporarily for the execution of A, after which it recovers its original value.

In the refinement carried out in this case, we use Law use-state-comp to substitute $TreatLocalEvent(e, s)$ with the declaration of a local variable $c_previousEvent$ of type $\mathbb{N}$, record the value of the state component $sfEvent_C$ in $c_previousEvent$, store the argument e of the call to $TreatLocalEvent$ in $sfEvent_C$, call $TreatLocalEvent$ with the e substituted with $sfEvent_C$, and restore the value of $sfEvent_C$. In the Step 2, we extend the scope of $c_previousEvent$ over the parallel action, and apply a step-law twice to distribute the assignments over the parallelism. The (partial) result is as follows.

$$\left(\begin{array}{l}\mathbf{var}\ c_previousEvent : \mathbb{N} \bullet c_previousEvent := sfEvent_C\ ;\ sfEvent_C := e; \\ \quad\left(\begin{array}{l}(\mu Z \bullet AllActions\ ;\ Z \Box end_local_execution \longrightarrow \mathbf{Skip})\ ;\ A \\ \quad [\![ns_1 \mid cs \mid ns_2]\!] \\ TreatLocalEvent(sfEvent_C, s)\ ;\ sfEvent_C := c_previousEvent\ ;\ B\end{array}\right)\end{array}\right)$$

In Step 3, we expand $TreatLocalEvent(sfEvent_C, s)$; this results in an alternation whose guards do not refer to state components. It establishes whether the destination of the broadcast is a state or the chart, in order to call the appropriate action. Step 4 simplifies this alternation to one of its branches by attempting to apply Law alt-elim to eliminate the first branch, and then the second, if unsuccessful. One of the applications necessarily succeeds, since the model constants that record the structure of the chart can be used to

determine that one of the guards of the alternation is **True** and the other is **False**. For the sake of example, we assume that the alternation simplifies to the first branch, and we obtain the result below.

$$\left(\begin{array}{l}\mathbf{var}\ c_previousEvent : \mathbb{N} \bullet c_previousEvent := sfEvent_C\ ;\ sfEvent_C := e;\\ \quad\left(\begin{array}{l}(\mu Z \bullet AllActions\ ;\ Z \Box end_local_execution \longrightarrow \mathbf{Skip})\ ;\ A\\ \quad [\![ns_1 \mid cs \mid ns_2]\!]\\ ExecuteChart(sfEvent_C)\ ;\ end_local_execution \longrightarrow \mathbf{Skip}\ ;\ sfEvent_C := c_previousEvent\ ;\ B\end{array}\right)\end{array}\right)$$

Finally, Step 5 applies the novel Law par-seq-dist to separate the parallelism into a sequence of two parallel compositions. It considers a parallelism of sequences (and relates it to a sequence of parallelisms). The action of the left-hand side of the parallelism is the second component of a pair (M, N) of actions defined by mutual recursion. The first component M may offer a communication over a channel l and start the second component N, and N offers a choice between calling M and recursing on N or synchronising on a channel el and terminating. The first action of the sequence on the right-hand side of the parallelism is a simple recursion that communicates on l, conditionally recurses, and synchronises on el afterwards. The second action of the right-hand side starts with a synchronisation on el.

In our application of this law as part of our strategy, N is the recursion offering *AllActions* with el as *end_local_execution*. The action M is *AllActions* itself, which accepts communications on a channel *local_event* and then starts a new recursion identical to the one that called it. So, l is the channel *local_event*. The actions *ExecuteChart* and *ExecuteState* are both recursions: they treat local event broadcasts themselves, and that involves recursive calls to either *ExecuteChart* or *ExecuteState*. For example, *ExecuteChart* can be written as a parametrised explicit recursion as follows.

$$\left(\begin{array}{l}\mu X \bullet ce : EVENT \bullet chart?c \longrightarrow status!(c.identifier)?active \longrightarrow\\ \left(\begin{array}{l}\mathbf{if}\ active = \mathbf{True} \longrightarrow \ldots\\ [\!]\ active = \mathbf{False} \longrightarrow activate!(c.identifier) \longrightarrow\\ \left(\begin{array}{l}\mathbf{if}\ c.default \neq nulltransition.identifier \lor c.decomposition = CLUSTER \longrightarrow \ldots\\ [\!]\ c.default = nulltransition.identifier \land c.decomposition = SET \longrightarrow\\ \left(\begin{array}{l}\mathbf{if}\ c.substates = \langle\rangle \longrightarrow \mathbf{Skip}\\ [\!]\ c.substates \neq \langle\rangle \longrightarrow \ldots;\ executeentryaction!(first.identifier) \longrightarrow\\ \qquad\left(\begin{array}{l}\left(\begin{array}{l}local_event?e?s \longrightarrow\\ \left(\begin{array}{l}\mathbf{if}\ s.type = CHART \longrightarrow X(e)\\ [\!]\ s.type \neq CHART \longrightarrow ExecuteState(s, e)\\ \mathbf{fi}\end{array}\right);\\ end_local_execution \longrightarrow \ldots\end{array}\right)\\ \Box\\ end_action \longrightarrow \mathbf{Skip}\end{array}\right);\ \ldots\\ \mathbf{fi}\end{array}\right)\\ \mathbf{fi}\end{array}\right)\\ \mathbf{fi}\end{array}\right)\end{array}\right)$$

The chart is obtained through channel *chart* and its status is queried through channel *status*, then depending on whether the chart is active or not, the substates are executed or entered. We omit the execution of the substates, and show the entering of the substates. That depends on the type of decomposition and whether or not the chart has default transitions. For example, if there are no default transitions and the chart has a parallel decomposition ($c.decomposition = SET$), if the chart has substates, then the first one is executed. This leads to the execution of the entry action (*executeentryaction*), which caters for local events ($local_event?e?s \longrightarrow \ldots$). The action that carries out the local event execution is a conditional that matches that in Law par-seq-dist. The same pattern occurs for every recursive call X to *ExecuteChart*.

ExecuteState can also be defined in an explicit recursive form. Whenever *ExecuteState* requests the execution of a chart action, it caters for local event broadcasts in the same way as shown above for *ExecuteChart*. We have a communication on *local_event*, conditionally executing *ExecuteChart* or itself, followed by a synchronisation on *end_local_execution*, before continuing with its execution.

Law[par-seq-dist]

$$N \; ; \; B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mu X \bullet A_2[l \longrightarrow \begin{pmatrix} \textbf{if } b \longrightarrow X \\ [] \neg\, b \longrightarrow C \\ \textbf{fi} \end{pmatrix} ; \; el \longrightarrow \textbf{Skip}]) \; ; \; el \longrightarrow B_2$$

$$=$$

$$\begin{pmatrix} (N \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mu X \bullet A_2[l \longrightarrow \begin{pmatrix} \textbf{if } b \longrightarrow X \\ [] \neg\, b \longrightarrow C \\ \textbf{fi} \end{pmatrix} ; \; el \longrightarrow \textbf{Skip}]) \; ; \; el \longrightarrow \textbf{Skip}); \\ (B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2) \end{pmatrix}$$

where $(M, N) \mathrel{\widehat=} \mu X, Y \bullet (F[l \longrightarrow Y], (X \; ; \; Y) \Box el \longrightarrow \textbf{Skip})$

provided

- $\{\!| l, el |\!\} \subseteq cs$;
- $\{\!| l, el |\!\} \cap usedC(F, A_2) = \varnothing$;
- $initials(M) \subseteq cs$;
- $usedV(B_1) \cap ns_2 = usedV(B_2) \cap ns_1 = \varnothing$.

In the parallelism of sequences, l and el are in the synchronisation set and are only used in the recursions as explicitly shown, as stated in the first two provisos. Each communication on l, therefore, triggers a recursive call Y to N on the left-hand side, and, on the right-hand side, a recursive call or a call to C. When either of those calls on the right-hand side terminates, we have a synchronisation on el. Since N is offering to synchronise on el or reexecute X (that is, M, which is waiting for a communication on a channel in the synchronisation set as stated by the third proviso) both sides synchronise on el and the most recent recursive call to N terminates. When the recursion on the right-hand side terminates, a second synchronisation on el prompts the mutual recursion to terminate.

Since there is no possibility of B_2 communicating with the mutual recursion or B_1 communicating with the simple recursion because both recursions terminate synchronously, and B_2 does not use variables written by the mutual recursion and B_1 does not use variables written by the simple recursion (last proviso), we can separate the parallel action in two parallel actions in sequence.

As already mentioned, we apply this Law par-seq-dist to the action obtained from step 4, with the recursion on the left-hand side as N, $ExecuteChart(sfEvent_C)$ as the recursion on the right-hand side, $local_event$ as l and $end_local_execution$ as el. Since $local_event$, $end_local_execution$ and $initials(AllActions)$ are all in the synchronisation set, each parallel action does not use the variables written by the other, and $local_event$ and $end_local_execution$ are, respectively, only used immediately before and after a recursive call in both $AllActions$ and $ExecuteChart$, the application is successful. For our example, we have the following result.

$$\left(\begin{array}{l} \mathbf{var}\ c_previousEvent : \mathbb{N} \bullet c_previousEvent := sfEvent_C\ ;\ sfEvent_C := e; \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} (\mu Z \bullet AllActions\ ;\ Z \Box end_local_execution \longrightarrow \mathbf{Skip}) \\ \qquad [\![ns_1 \mid cs \mid ns_2]\!] \\ ExecuteChart(sfEvent_C)\ ;\ end_local_execution \longrightarrow \mathbf{Skip} \end{array}\right); \\ A\ [\![ns_1 \mid cs \mid ns_2]\!]\ sfEvent_C := c_previousEvent\ ;\ B \end{array}\right) \end{array}\right)$$

Step 6 applies the procedure parallelism-resolution to the second parallel action.

The resulting action shown in Figure 5.45 is for a local event broadcast directed at the whole chart, as is the example shown above. The same steps apply for broadcasts directed at particular states. The resulting action differs solely on the action call that executes the state: instead of $ExecuteChart(sfEvent_C)$, we have $ExecuteState(s, sfEvent_C)$, where s is the target state.

N. Leading local variable declaration on either side Following the execution of a local event broadcast, both the chart and the *Simulator* processes carry out early return logic checks. For that, they use a local boolean variable b that records the result of the check. In our example, the action B above is such a variable block. In this case, therefore, we consider parallelisms $A\ [\![ns_1 \mid cs \mid ns_2]\!]\ (\mathbf{var}\ b : \mathbb{B} \bullet B)$. We apply Laws var-rename and C.138 [76] to give b a fresh name and expand its scope out of the parallelism, before recursing. The result is a block like $\mathbf{var}\ b_i : \mathbb{B} \bullet \underline{(A\ [\![ns_1 \mid cs \mid ns_2 \cup \{b_i\}]\!]\ B[b_i/b])}$, where b_i is a fresh name.

Since parallelism-resolution is recursive, termination is an issue. In the sequel, we present a detailed argument based on the structure of the parallel actions found in our chart models.

Termination of parallelism-resolution Since parallelism-resolution is recursive, we now address the issue of its termination. Our argument is based on the structure of the right-hand parallel action.

There are two base cases: A, and I. The other cases can be divided into three groups: those that do not change the right-hand action, those that reduce its structure, and those that expand it. The cases in the first group can be applied only a finite number of times: they do not lead to an infinite sequence of steps that do not reduce the right-hand action. As we explain next, this is due to the structure of the left-hand parallel action: after applications of such cases, we eventually apply a case in another group.

The cases in this first group are: B, D, E, F, G, H, I, J, N. Case B simplifies the left-hand action to allow the prefix on the right-hand action to be addressed by case C. Cases D, E, F, G and N remove an action on the left to apply the other cases to the remaining action to resolve the parallelism. Any sequence of applications of these cases must be followed by an application of case C, or case B followed by C, which reduce the structure of the right-hand action. Case H simplifies the structure on the left-hand action to allow the other cases to handle the actions in each branch of the alternation. Finally, case I for the left-hand action only expands the definition of *broadcast*, which is not expanded by case B. Each occurrence of this case must necessarily be followed by an application of case L, which reduces the structure of the right-hand action.

For cases in the second group, since each application reduces the structure of the right-hand parallel action to that of one of its components, eventually we reach $end_cycle \longrightarrow \mathbf{Skip}$. Refinement, using cases B and C, leads to the application of the base case A, which eliminates the parallelism and terminates the procedure. This group contains the cases C, D, E, F, H, L, and N.

The cases in the third group, that is, I, K, and M, expand the right parallel action. Case I could potentially expand calls to recursive actions. If such an action allowed a non-terminating recursion, then the procedure parallelism-resolution would not terminate due to successive indiscriminate applications of I. The right parallel actions, however, come from the *Simulator* process, so we know exactly to which actions case I is applied. The only recursive action as above is *ExecuteTransition*, when executing a transition that starts a loop. Case I treats it as a special case, only expanding once the execution of a transition that starts a loop.

Case M is applied to each local event broadcast. Since there is only a finite number of broadcasts in a chart, and recursions introduced by local event broadcasts are not unfolded, this case only expands the right-hand action a fixed number of times. All the actions introduced in this expansion can be treated by the remaining cases.

Case K, similarly to case I, unfolds the recursion, but in the recursive application of parallelism-resolution, case F is applied to extract the assignment to $sfEvent_C$, and then case N is applied to the right-hand side to treat the local variable declaration that always follows a call to *TreatLocalEvent*. In any case, any such recursion can be unfolded up to a finite number of times (number of local event broadcasts in an action). Eventually, an application of case C, resolving a synchronisation over channel *end_action*) will terminate the recursion. The channel *end_action* signals the end of a chart action.

Finally, case K unfolds a recursion on the right hand side, this recursions are associated

1. **Copy-rule.** Apply definitions B.37, B.39 [76], and copy-rule.
2. For each variable block corresponding to a value-result parameter
   (a) **Eliminate the local variable.** Apply law var-value-result.
3. For each variable block corresponding to a value parameter
   (a) **Distribute assignment.** Exhaustively apply laws assign-seq-com and assign-seq-dist-2.
   (b) **Eliminate local variable.** Apply law var-assign-elim.

Figure 5.46: Procedure copy

to the treatment of local event broadcasts and with the checking of properties of lists of substates. In both cases, the recursion can only be unfolded a finite number of times (local event broadcasts are finite, and each state has a finite number of substates), and the action introduced by the unfolding can be treated by the remaining cases.

5.3.3.3 Procedure copy

This procedure simply substitutes a call to an action by the definition of the action with the value and value-result parameters substituted. We illustrate this procedure by applying it to a call to action *transitionActionCheck* with a transition identifier t and variable v.

$$transitionActionCheck(t, v)$$

The first step applies the definitions of action call with value and value-result parameters. The definition copy-rule states that unspecified parameters are treated as value parameters, this differs from [76], where a call to an action with unspecified parameters is defined as the syntactic substitution of the values for the parameters. This step introduces local variable declarations and assignments which are treated by the remaining steps. The result of applying the definition of *transitionActionCheck* to our example is as follows.

$$\begin{array}{l}
\mathbf{var}\, sid : SID;\ b : \mathbb{B} \bullet sid := t\ ;\ b := v;\\
entryActionCheck(sid, b)\ ;\ \mathbf{var}\, ss : \mathrm{seq}\, SID \bullet\\
\left(\begin{array}{l}
state!sid?s \longrightarrow ss := s.substates;\\
\mu X \bullet \left(\begin{array}{l}
\mathbf{if}\ ss = \langle\rangle \longrightarrow \mathbf{Skip}\\
[]\ ss \neq \langle\rangle \longrightarrow\\
\qquad status!(head\ ss)?active \longrightarrow (b := or(b, active)\ ;\ ss := tail\ ss\ ;\ X)\\
\mathbf{fi}
\end{array}\right)
\end{array}\right);\\
v := b
\end{array}$$

The formal parameters sid and b become local variables and are initialised with the actual parameters (t and v), the body of the action is executed, and the local variable corre-

sponding to the value-result parameter is assigned to the actual parameter v.

For each blocks $\mathbf{var}\, x : T \bullet x := y \,;\, A \,;\, y := x$ introduced by definition B.39 [76], Step 2(a) removes the local variable. In our example, this step applies to the parameter b; it is eliminated by an application of the law var-value-result that substitutes the local variable by the actual parameter, provided the actual parameter is not used in the definition of the action, resulting in the following action.

$$
\begin{array}{l}
\mathbf{var}\, sid : SID \bullet sid := t; \\
entryActionCheck(sid, v) \,;\, \mathbf{var}\, ss : \mathrm{seq}\, SID \bullet \\
\left(\begin{array}{l}
state!sid?s \longrightarrow ss := s.substates; \\
\mu X \bullet \left(\begin{array}{l}
\mathbf{if}\ ss = \langle\rangle \longrightarrow \mathbf{Skip} \\
[\!]\ ss \neq \langle\rangle \longrightarrow \\
\qquad status!(head\ ss)?active \longrightarrow (v := or(v, active) \,;\, ss := tail\ ss \,;\, X) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}
$$

In step 3, for each local variable introduced in step 1 by definition B.37 [76], we distribute the assignment over the variable block, and eliminate it. In our example, the result of this step is the action below.

$$
\begin{array}{l}
entryActionCheck(t, v) \,;\, \mathbf{var}\, ss : \mathrm{seq}\, SID \bullet \\
\left(\begin{array}{l}
state!t?s \longrightarrow ss := s.substates; \\
\mu X \bullet \left(\begin{array}{l}
\mathbf{if}\ ss = \langle\rangle \longrightarrow \mathbf{Skip} \\
[\!]\ ss \neq \langle\rangle \longrightarrow \\
\qquad status!(head\ ss)?active \longrightarrow (v := or(v, active) \,;\, ss := tail\ ss \,;\, X) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}
$$

The assignment to *sid* is distributed over the sequential composition towards the end of the local variable block, and the local variable is eliminated.

5.3.3.4 Procedure recursion-introduction

This procedure is used in Step 3 of the structuring phase. It receives the same parameters *loopT* and *treatedT* as parallelism-resolution, which it uses to call that procedure. It acts on the body of the outermost recursion in the main action to transform the remaining parallel actions into recursions.

So far, in resolving the parallelism between the chart and *Simulator* processes and actions, the refinement has basically instantiated the generic operational semantics defined by *Simulator* to the specific chart defined in the chart process. This unravels a sequential structure of conditionals that establish the paths of execution available in the chart. Where, however, we have loops in these paths, we need to introduce recursions. (Some of them become loops in the program.) For that, we calculate the recursive actions, and then show that they refine the parallel actions. Figure 5.47 shows the refinement steps.

While there are remaining parallelism p in the main action

1. **Calculate a possibly recursive action.** Applying the procedure parallelism-resolution to p.
2. **Refine parallel action.**
   (a) If the calculated action is a recursion $\mu X \bullet F(X)$, apply Law unique-fixed-point to p and F.
   (b) Else, substitute the calculated action for the parallelism.

Figure 5.47: Refinement strategy:structuring phase - recursion-introduction.

For each parallel action (left unresolved by the procedure parallelism-resolution in Step 2 of the structuring phase) we first calculate (Step 1 in Figure 5.47), by refinement, a (possibly recursive) sequential action. If the calculated action is recursive, we refine the parallel action to that recursion using a standard fixed-point law (Step 2(a) in Figure 5.47). Otherwise, we simply replace the parallelism with the calculated action (Step 2(b) in Figure 5.47), since the procedure used to calculate it establishes equivalence.

To calculate the new action, we apply the procedure parallelism-resolution to p, which is a parallel action $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B$. The result may be of the form $F[A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B]$ or F, that is, it may or may not contain the same parallelism. If it does, the calculated action is a recursion $\mu X \bullet F[X]$ whose body is the result obtained with parallelism-resolution, where all parallelisms $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B$ are replaced with a recursive call X. If it does not contain the parallelism, the result of this calculation is the action F itself.

A proof obligation generated by the Law unique-fixed-point in Step 2(a) requires $F(p) = p$. This is a consequence of the way in which F is calculated: parallelism-resolution establishes equivalence. The other proof obligation, namely, p is deterministic, is true of our models.

As mentioned previously, we use the procedure recursion-introduction with the assumption that there are no mutually recursive actions to be introduced. This means that the calculated actions do not contain themselves any further parallelisms, which would generate a recursion.

To illustrate this procedure, we consider the parallel action that executes a transition in Figure 5.24. By applying parallelism-resolution to it, we obtain the action below.

$$\left(\begin{array}{l} \textbf{if } ct \longrightarrow \\ \quad \ldots \left(\begin{array}{l} (\mu Y \bullet AllActions \ ; \ Y \Box end_cycle \longrightarrow \textbf{Skip}) \\ \qquad \llbracket ns_1 \mid cs \mid ns_2 \cup \{sfEvent_C\} \rrbracket \\ ExecuteTransition(t, p, s, sfEvent_C) \end{array}\right) \\ [\!] \neg ct \longrightarrow \ldots \\ \textbf{fi} \end{array}\right)$$

This does contain the parallel action that originated it, so we define a recursion by substituting a recursive call for all the occurrences of the parallelism to obtain an action

$\mu X \bullet \mathbf{if}\ ct \longrightarrow \ldots X \ [] \ \neg\ ct \longrightarrow \ldots \mathbf{fi}$.

Finally, we refine the parallelism in Figure 5.24 to the calculated action, by applying Law unique-fixed-point. This generates two provisos: the body of the recursion is deterministic, which follows from the definition of our models, and the parallelism is a fixed point of the recursion. This last proviso is shown below.

$$\left(\begin{array}{l} \mathbf{if}\ ct \longrightarrow \ldots \\ \quad \left(\begin{array}{c} (\mu Y \bullet AllActions \ ; \ Y \ \Box \ end_cycle \longrightarrow \mathbf{Skip}) \\ [\![ns_1 \mid cs \mid ns_2 \cup \{sfEvent_C\}]\!] \\ ExecuteTransition(t, p, s, sfEvent_C) \end{array}\right) \\ [] \ \neg\ ct \longrightarrow \ldots \\ \mathbf{fi} \end{array}\right) = \left(\begin{array}{c} (\mu Y \bullet AllActions \ ; \ Y \ \Box \ end_cycle \longrightarrow \mathbf{Skip}) \\ [\![ns_1 \mid cs \mid ns_2 \cup \{sfEvent_C\}]\!] \\ ExecuteTransition(t, p, s, sfEvent_C) \end{array}\right)$$

This is exactly the result of applying the procedure parallelism-resolution to the right-hand side of the equation.

Figure 5.26 shows the result of applying recursion-introduction to both parallelisms in Figure 5.24.

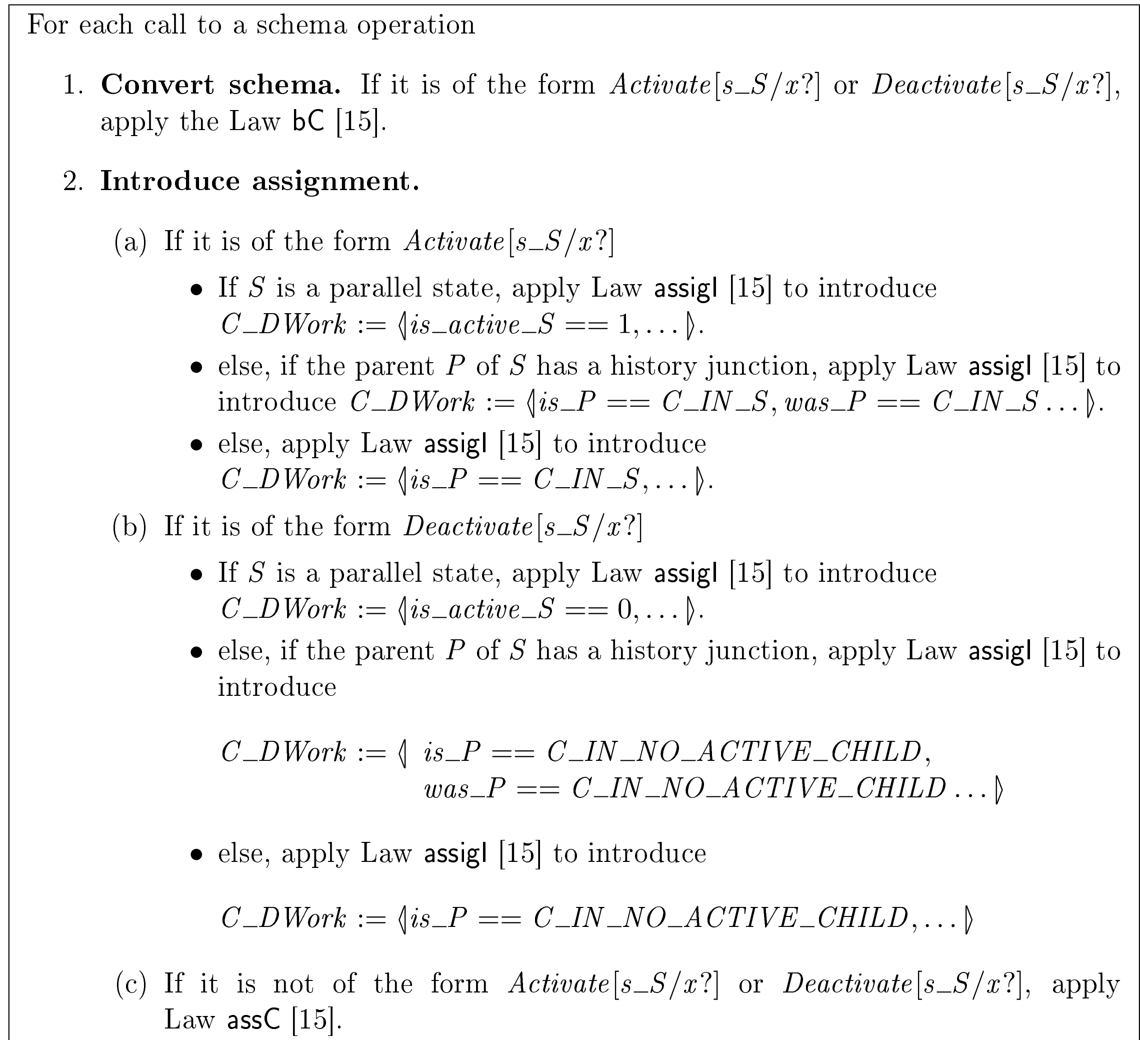
5.3.3.5 Procedure assignment-introduction

Figure 5.48 presents the procedure assignment-introduction. As indicated previously, it refines to assignments all schema operations. We distinguish two types of schema operations: (1) those that activate or deactivate a particular state, and (2) those introduced in the data refinement phase plus the initialisation schema.

The operations in the first group are specified by a renaming of the schema *Activate* or *Deactivate*, which are originally defined in the chart process. They specify data operations that record an input state $x?$ as active or inactive. Renamings $Activate[s_S/x?]$ and $Deactivate[s_S/x?]$ are used in the main action at this stage to define (data refined) operations that activate or deactivate a specific state S.

All assignments introduced are of bindings to the state component C_DWork, which records the status of states and history junctions. The predicates in the schemas of the second group are conjunctions of equalities, and so we convert them into assignments directly (using a Z refinement law). The schemas in the first group do not enjoy such property, so we first convert them to specification statements (also using a Z law).

The actual assignments are determined by the kind of schema operation, by the type of the state being activated or deactivated (parallel or sequential), and by whether the parent P of a sequential state being activated has a history junction or not. These conditions identify the components of C_DWork that are to be modified. The identification is based on the naming conventions described in Section 5.3.1. For instance, activation or deactivation of a parallel state S modifies the component is_active_S of C_DWork. In the steps shown in Figure 5.48, the ellipses in the bindings on the right-hand side of the assignments indicate that all other components of the binding that defines the value of C_DWork are left unchanged.



For each call to a schema operation

1. **Convert schema.** If it is of the form $Activate[s_S/x?]$ or $Deactivate[s_S/x?]$, apply the Law bC [15].

2. **Introduce assignment.**

   (a) If it is of the form $Activate[s_S/x?]$
   - If S is a parallel state, apply Law assigI [15] to introduce $C_DWork := \langle\!| is_active_S == 1, \ldots |\!\rangle$.
   - else, if the parent P of S has a history junction, apply Law assigI [15] to introduce $C_DWork := \langle\!| is_P == C_IN_S, was_P == C_IN_S \ldots |\!\rangle$.
   - else, apply Law assigI [15] to introduce $C_DWork := \langle\!| is_P == C_IN_S, \ldots |\!\rangle$.

   (b) If it is of the form $Deactivate[s_S/x?]$
   - If S is a parallel state, apply Law assigI [15] to introduce $C_DWork := \langle\!| is_active_S == 0, \ldots |\!\rangle$.
   - else, if the parent P of S has a history junction, apply Law assigI [15] to introduce

     $$C_DWork := \langle\!| \; is_P == C_IN_NO_ACTIVE_CHILD, \; was_P == C_IN_NO_ACTIVE_CHILD \ldots |\!\rangle$$

   - else, apply Law assigI [15] to introduce

     $$C_DWork := \langle\!| is_P == C_IN_NO_ACTIVE_CHILD, \ldots |\!\rangle$$

   (c) If it is not of the form $Activate[s_S/x?]$ or $Deactivate[s_S/x?]$, apply Law assC [15].

Figure 5.48: Refinement strategy: structuring phase - assignment-introduction

5.3.3.6 Procedure update-output

This procedure data refines the process to introduce a new state component, namely, C_Y, which, as previously explained, records the values of the output variables and events to be communicated at the end of the step. The procedure update-output also expands the definition of the schema type of C_B, to include boolean components that record whether the output events have occurred or not.

As a parameter, update-output takes the sequence *output_events* of output events in the order in which they are defined in the chart. The starting point of update-output is the second action in the sequence that defines the body of the outmost recursion in the main action: this is the whole body, except *ReadInputs*. It has the general form shown in Figure 5.26, and contains a number of interleavings (omitted in Figure 5.26) repeated at the end of the innermost branches of the nested alternations. These interleavings have the general form shown in Figure 5.49; each is an interleaving of alternations communicating events, and communications of output variables. In Figure 5.49, the interleaved actions shown are an alternation that communicates an output event E, and a prefixing that

$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\textbf{if } C_DWork.counter_E > 0 \longrightarrow \\ \qquad C_DWork := \langle\!| counter_E == (C_DWork.counter_E - 1), \ldots |\!\rangle; \\ \qquad o_E!(\textbf{True}) \longrightarrow \textbf{Skip} \\ \,[\!]\, C_DWork.counter_E = 0 \longrightarrow o_E!(\textbf{False}) \longrightarrow \textbf{Skip} \\ \textbf{fi}\end{array}\right) \\ [\![\{C_DWork.counter_E\} \mid \ldots]\!] \\ \ldots\end{array}\right) \\ [\![\ldots \mid \ldots]\!] \\ (o_v!(C_B.v) \longrightarrow \textbf{Skip} \mid\mid\mid \ldots)\end{array}\right)$$

Figure 5.49: update-output starting point.

communicates the output variable v.

The procedure update-output is shown in Figure 5.50. First, it extracts the interleavings to the end of the action. This is possible because the interleavings, followed by a synchronisation on end_cycle, are the final action in all innermost branches of all alternations. The procedure parallelism-resolution pushes them inside the alternations, and recursion-introduction may introduce tail recursions that terminate with them.

Next, for each event E in the sequence of output events, we identify the alternation that communicates the output event: that with a communication through o_E. We introduce a local variable E of type $\mathbb{B}$, assign to it the value v that is being communicated through o_E, and communicate E instead. Afterwards, we extend the scope of the local variable over the main action, and promote it to a state component. The result of these steps on the alternation in Figure 5.49 is shown below.

$$\left(\begin{array}{l}\textbf{if } C_DWork.counter_E > 0 \longrightarrow \\ \qquad C_DWork := \langle\!| counter_E == (C_DWork.counter_E - 1), \ldots |\!\rangle\, ;\; E := \textbf{True}\, ;\; o_E!(E) \longrightarrow \textbf{Skip} \\ \,[\!]\, C_DWork.counter_E = 0 \longrightarrow E := \textbf{False}\, ;\; o_E!(E) \longrightarrow \textbf{Skip} \\ \textbf{fi}\end{array}\right)$$

We now extract the prefixings $o_E!(E) \longrightarrow \textbf{Skip}$ that communicate the local variables from the alternation, and then extract the alternation from the interleaving obtaining the action below.

$$\left(\begin{array}{l}\textbf{if } C_DWork.counter_E > 0 \longrightarrow \\ \qquad C_DWork := \langle\!| counter_E == (C_DWork.counter_E - 1), \ldots |\!\rangle\, ;\; E := \textbf{True} \\ \,[\!]\, C_DWork.counter_E = 0 \longrightarrow E := \textbf{False} \\ \textbf{fi}\end{array}\right);$$
$$\left(\left(\begin{array}{l} o_E!(E) \longrightarrow \textbf{Skip} \\ [\![\{\} \mid \ldots]\!] \\ \ldots\end{array}\right) [\![\ldots \mid \ldots]\!]\, (o_v!(C_B.v) \longrightarrow \textbf{Skip} \mid\mid\mid \ldots)\right)$$

After all output events have been considered, we data refine the process to include the

1. **Isolate outputs.** Exhaustively apply Laws alt-seq-dist and tail-rec-seq-dist to the whole action.

2. For each output event E in $output_events$

   (a) **Introduce local variable.** Apply Law var-assign-intro to prefixing actions of the form $o_E!v \longrightarrow$ **Skip** to introduce a fresh local variable E of type $\mathbb{B}$ and initialise it with v, and then apply Law assign-seq-dist to the assignment followed by the prefixing.

   (b) **Extend scope.** Apply Laws var-seq-ext-left to the branch of the alternation that initialises E with **True**, var-alt-dist-both to the alternation, var-int-dist to the interleaving, C.137 [76] to the body of the outmost recursion, var-tail-rec-ext to the recursion, and var-seq-ext-left to the outer sequence.

   (c) **Promote to state component.** Apply Law A.5 [14] to the process.

   (d) **Extract communication.** Apply Law alt-seq-dist to the alternation with the communications over o_E.

   (e) **Step-law.** Apply Law int-seq-dist to the interleaving, with A as the alternation.

3. **Data refinement.** Apply Laws C.1-C.25 [76] to the whole process (to include the new state components in C_B).

4. **Introduce local variable.** Apply Law var-assign-intro to the interleaving introduce a local variable C_Y of type $ExternalOutputs_C$ and initialise it with C_B.

5. **Substitute variable for value.** Apply Law assign-seq-dist to the sequence containing the new assignment and the interleaving.

6. **Extend scope.** Apply Law C.137 [76] to the body of the outmost recursion to extend the scope of C_Y, Law var-tail-rec-ext to the recursion and var-seq-ext-left to the outer sequence.

7. **Promote to state component.** Apply Law A.5 [14] to the process.

Figure 5.50: Refinement strategy: structuring phase - update-output

newly added state components in the component C_B. This is achieved by first defining a new concrete state identical to the current state, except that the variables that were just promoted are components of the schema $BlockIO_C$. The retrieve relation equates each promoted variable to the corresponding variable of the component of type $BlockIO_C$ of the new concrete state. The laws of action simulation determine the required data refinement.

After the data refinement, we apply Law var-assign-intro to the remaining interleaving to introduce the variable C_Y and initialise it with C_B. (This is now possible because the bindings of $ExternalOutputs_C$ and $BlockIO$ have the same components). Next, we substitute C_Y for C_B in the interleaved communications, and distribute the local variable over the main action. Finally, we promote it to a state component. The result, as probably expected, is the action shown in Figure 5.21.

5.3.3.7 Procedure simplification

This procedure, which is applied in the last step of the structuring phase, eliminates redundant code, simplifies the control structure, and adds *inputevents* to the record C_U. It is applied to the whole of the main action, and basically, this aligns its specification with the code of the implementation.

Figure 5.51 shows the steps of simplification. Step 1 distributes the hiding of the channel set *interface* over the whole action, thus eliminating the internal communications and the hidings themselves. This is possible because of the sequential and deterministic structure of the starting action.

Before the application of this procedure, the action may contain assignments deactivating a state, followed by the activation of the same state or a sequential sibling. The second step of simplification removes such redundant assignments. It applies to sequences where the first action is a deactivation assignment: an assignment $DWork_C.is_active_S := 0$ or $is_S := C_IN_NO_ACTIVE_CHILD$, for some chart state S. It moves such assignments forward as far a possible, and then eliminates redundancy if it arises. For example, $is_S := C_IN_NO_ACTIVE_CHILD$; A ; $is_S := C_IN_S_p$ may be refined to A ; $is_S := C_IN_S_p$, if A does not use is_S. This is an optimisation frequently used to avoid unnecessary assignments.

The only local variables that should remain in the model of the implementation are those that record the current event when a local event broadcast occurs, namely the local variables *c_previousEvent*. In the third step, we remove all other (spurious) local variables by moving them over sequential compositions (using Laws assign-seq-com and assign-seq-col), extracting them from alternations (using Law alt-seq-dist from right to left), and eliminating them using Laws C.136 [76], from right to left, and var-assign-elim.

The procedure simplification uses three other procedures that we present later on in this section. The procedure early-return-simplification, used in the fourth step, attempts to simplify alternations that verify early return logic conditions. It introduces and distributes assertions about the status of the states, and uses them to eliminate some of these alternations. The fifth step uses the procedure parallel-state-simplification to simplify alternations that model the entering, executing, and exiting of parallel states, whenever possible. It relies on two facts: (1) whenever a state is inactive, all its substates are inactive too; and (2) in most cases, whenever an active state with a parallel decomposition is being executed, all its substates are also active. (The latter is not necessarily true when there is local event broadcast; in this case the alternation cannot be simplified.) The sixth step flattens the alternations that execute sequential states using sequential-state-simplification. It relies on the fact that at any point at most one sequential substate may be active, and that the values of a component $C_DWork.is_S$ are in a one-to-one correspondence with the substates of S.

The execution of transitions may result in nested alternations, the first checking the trigger of the transitions and the second evaluating its condition, and in tail recursions

1. **Distribute the hiding.** Exhaustively apply distribution laws for hiding to the main action: Laws C.120 [76], C.122 [76], C.125 [76], alt-hide-dist, var-rec-hide-dist, rec-hide-dist, var-tail-alt-rec-hide-dist,prefix-hide-dist-1, and prefix-hide-dist-2.
2. **Simplify deactivations.** For each chart state S, and assignment $DWork_C.is_active_S := 0$ or $DWork_C.is_S := S_IN_NO_ACTIVE_CHILD$, try to eliminate it as follows.
   (a) **Distribute assignment.** Apply Law assign-seq-comp exhaustively.
   (b) **Eliminate the assignment.** Apply Law assign-seq-col exhaustively.
3. **Eliminate local variables.** For each local variable different from *c_previousEvent*, exhaustively apply Laws assign-seq-com and assign-seq-col to sequences containing assignments to it, Law alt-seq-dist from right to left to alternations finishing in assignments to it, and Laws C.136 [76], from right to left, and var-assign-elim to its declaration.
4. **Simplify early return logic.** Apply the procedure early-return-simplification to the main action.
5. **Simplify parallel states.** Apply the procedure parallel-state-simplification to the main action.
6. **Flatten sequential states.** Apply the procedure sequential-state-simplification to the main action.
7. **Simplify transition executions.** Exhaustively apply Laws alt-alt-dist, alt-simp, and tail-rec-seq-dist from right to left, if necessary, to nested alternations and recursions that execute transitions.
8. **Fold recursion.** Exhaustively apply Law C.128 [76] from right to left to the main action.
9. **Eliminate assumptions.** Exhaustively apply Law C.35 [76] to the main action.
10. **Eliminate Skip.** Exhaustively apply Laws C.100, from right to left, and C.132 [76] to the main action.
11. **Eliminate actions.** Apply Law action-intr [14] from right to left to the process and each of its actions.
12. **Data refinement.** Data refine the process to make *inputevents* a record component of C_U.

Figure 5.51: Refinement strategy: structuring phase - simplification

whose main body is a alternation that may stop the recursion. In the first case, Step 7 flattens the nested conditionals, and in the second case, it additionally extracts the actions executed when the recursion terminates. The starting point of this step is the whole main action, however the actions to which the law alt-alt-dist applies are those of the form below.

$$\textbf{if } \mathit{trigger} \longrightarrow (\textbf{if } \mathit{condition} \longrightarrow A \,;\; C \,[\!]\, \neg\, \mathit{condition} \longrightarrow B \textbf{ fi}) \,[\!]\, \neg\, \mathit{trigger} \longrightarrow B \textbf{ fi}$$

The resulting actions are, therefore, as shown below.

$$(\textbf{if } \mathit{trigger} \wedge \mathit{condition} \longrightarrow A \,;\; C \,[\!]\, \neg\, \mathit{trigger} \vee \neg\, \mathit{condition} \longrightarrow \textbf{Skip fi}) \,;\; B$$

The guards of the alternation are now determined by the guards of the transitions.

As previously discussed, Step 2 of the structuring phase leaves some parallelisms unresolved, some of which are then refined in the Step 3 of that phase to recursions. This is illustrated by the actions in Figures 5.26 and 5.20. In Figure 5.26, we have an action obtained using the recursion-introduction procedure in the Step 3 of the structuring phase. As indicated, in that action, we use another action shown in Figure 5.20; its definition is similar to that of the action in Figure 5.26 itself. Omitted in Figure 5.26 are internal communications. Once they are eliminated in Step 1 above, we can fold any recursions left unfolded. This is carried out in Step 8.

Step 9 eliminates any assumptions left over from previous steps. Step 10 applies prefix and sequence unit laws to eliminate unnecessary uses of **Skip**. Step 11 eliminates the actions of the original chart and *Simulator* processes that are no longer used: Steps 2 and 3 of the structuring phase expand action calls.

Finally, Step 12 data refines the process to make the state component *inputevents*, added by the Step 1 of the structuring phase, a component of C_U. The retrieve relation equates the component *inputevents* of C_U (in the concrete state) to the state component of the same name in the original state. This results in the state used in the program model, so that the following phases do not involve any further data refinement.

Procedure early-return-simplification This tries to simplify as many alternations that check for early return conditions, as possible. The general form of the actions that early-return-simplification modifies is shown below; A activates and deactivates a number of states, and is followed by an early return check.

$$A \,;\; \textbf{if } \mathit{early_return_condition} \longrightarrow \textbf{Skip} \,[\!]\, \neg\, \mathit{early_return_condition} \longrightarrow B \textbf{ fi}$$

Using early-return-simplification, assumptions are extracted from the assignments in A that change the status of states, distributed towards the early return check, and used to eliminate it, if possible, resulting in $A \,;\; B$.

The steps for this procedure are shown in Figure 5.52. First, in Step 1, we extract assumptions about the status of states (from assignments that activate and deactivate states), and then distribute them throughout the action using a procedure assumption-distribution. This procedure is discussed in the sequel; it systematically distributes an assumption through an action as far as possible, towards the assignments. In Step 2, for each alternation that checks the status of states, early-return-simplification merges any preceding assumptions into one, and tries to simplify the alternation using these assumptions.

1. **Introduce and distribute assumption.** For each assignment to a component *is_* or *is_active_* of *C_DWork* followed in sequence by any action A, apply Law assign-assump-intro to the assignment, and apply the procedure assumption-distribution to the sequence formed by the introduced assumption and A.
2. For each alternation whose guard uses the components *is_* or *is_active_* of *C_DWork*
   (a) **Merge assumptions.** Exhaustively apply Law C.26 [76] to the assumptions that immediately precede the alternation.
   (b) **Simplify.** Try to apply Law assump-alt-elim to the alternation and the preceding assumption.
3. **Remove assumptions.** Exhaustively apply Law C.35 [76].

Figure 5.52: Refinement strategy: structuring phase - early-return-simplification

The final Step 3 removes all the assumptions introduced, since they are no longer needed.

In our example, entering the substates of PowerOn involves an early return check as shown below.

$$Air_DWork.is_active_FAN1 := 1 \,;\; Air_DWork.is_FAN1 := Air_IN_Off \,;$$
$$\left(\begin{array}{l}\mathbf{if}\ \neg\ (Air_DWork.is_active_FAN1 \neq 0) \longrightarrow \mathbf{Skip} \\ [\!]\ Air_DWork.is_active_FAN1 \neq 0 \longrightarrow \text{(Entering state FAN2)} \\ \mathbf{fi}\end{array}\right)$$

First the state FAN1 is activated ($Air_DWork.is_active_FAN1 := 1$), then the default transition whose destination is Off is followed. This leads to the substate Off being entered ($Air_DWork.is_FAN1 := Air_IN_Off$). Since entering a state can generate recursive executions due to a local event broadcast, an early return logic check is executed before the next state is entered. In this case, the state FAN1 must still be active: $Air_DWork.is_active_FAN1 \neq 0$. (This condition actually requires that the state is not inactive.) We omit above the action that models the continuation when there is no need for an early return: namely, entering FAN2.

Step 1 introduces assumptions for the assignments that activate states FAN1 and Off.

$$Air_DWork.is_active_FAN1 := 1 \,;\; \{Air_DWork.is_active_FAN1 = 1\};$$
$$Air_DWork.is_FAN1 := Air_IN_Off \,;\; \{Air_DWork.is_FAN1 = Air_IN_Off\};$$
$$\left(\begin{array}{l}\mathbf{if}\ \neg\ (Air_DWork.is_active_FAN1 \neq 0) \longrightarrow \mathbf{Skip} \\ [\!]\ Air_DWork.is_active_FAN1 \neq 0 \longrightarrow \text{(Entering state FAN2)} \\ \mathbf{fi}\end{array}\right)$$

Next, the first assumption is moved over the second assignment, and both assumptions are

distributed through the alternation. The result is as follows.

$$
\begin{array}{l}
Air_DWork.is_active_FAN1 := 1;\\
Air_DWork.is_FAN1 := Air_IN_Off;\\
\{Air_DWork.is_active_FAN1 = 1\} ; \{Air_DWork.is_FAN1 = Air_IN_Off\};\\
\left(\begin{array}{l}
\mathbf{if} \neg (Air_DWork.is_active_FAN1 \neq 0) \longrightarrow\\
\quad \{Air_DWork.is_active_FAN1 = 1\} ; \{Air_DWork.is_FAN1 = Air_IN_Off\} ; \mathbf{Skip}\\
[\!] \ Air_DWork.is_active_FAN1 \neq 0 \longrightarrow\\
\quad \{Air_DWork.is_active_FAN1 = 1\} ; \{Air_DWork.is_FAN1 = Air_IN_Off\};\\
\quad \text{(Entering state FAN2)}\\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

Next, assumption-distribution proceeds to distribute both assumptions over the omitted action that models entering FAN2. Once the distribution is completed, Step 2(a) joins the assumptions before the alternation above, and Step 2(b) above eliminates it using Law assump-alt-elim and $Air_DWork.is_active_FAN1 = 1$. The assumptions left are eliminated in Step 3. The result is as follows.

$$Air_DWork.is_active_FAN1 := 1 ; Air_DWork.is_FAN1 := Air_IN_Off ; \text{(Entering state FAN2)}$$

We are left with the assignments and the action that models entering FAN2.

Procedure assumption-distribution This procedure systematically distributes an assumption over an action formed solely by assumptions, prefixings, assignments, alternations and interleavings. The steps for this procedure are shown in Figure 5.53.

1. For each A in a sequence of actions, do
   (a) if $A = (v := e)$, apply law assign-assump-dist or assign-assump-dist-nofv;
   (b) if $A = (p \longrightarrow B)$, apply one of the laws C.41 [76], C.43 [76], C.45 [76], C.47 [76] or C.49 [76], and continue with B;
   (c) if $A = (\mathbf{if} [\!] i : I \bullet p_i \longrightarrow B_i \ \mathbf{fi})$, apply law assump-alt-dist, apply this procedure to each B_i, and apply law alt-seq-dist to extract the assumption;
   (d) if $A = |||\ i : I \bullet [\![cs_i]\!] B_i$, apply law C.39 [76], apply this procedure to each B_i, and apply law int-assump-extract;
   (e) if $A = \{p\}$, apply law assump-com.

Figure 5.53: Procedure assumption-distribution.

The procedure assumption-distribution takes an assumption followed by a sequence of action, and, for each action in the sequence, it tries to distribute the assumption over that action. It distinguishes five cases: assignment, prefixing, alternation, interleaving, and assumptions. The cases for assignment and assumption are simple; the procedure applies the appropriate laws to distribute the assumption, and iterates.

1. **Introduce assumption.** Apply Law assign-assump-intro to the initialization of C_DWork in the first action of the outer sequence, and Law C.27 [76] to introduce the assumption described in the text.
2. **Distribute assumption.** Apply Law assump-rec-dist to the recursion, and apply procedure assumption-distribution to the body of the recursion.
3. For each alternation with a guard of the form $is_active_S = e$ or $is_P = C_IN_S$, where S is a state with parallel decomposition
   (a) **Introduce assumption.** Apply Law alt-assump-intro.
   (b) **Distribute assumption.** Apply procedure assumption-distribution to the branches.
   (c) For each alternation (nested in the branch) whose guards contain $is_active_S_i$, where S_i is a substate of S
      i. **Merge assumptions.** Apply Law C.26 [76] to merge the assumptions preceding it.
      ii. **Simpify.** Apply Law assump-alt-elim to simplify the alternation, if possible.

Figure 5.54: Refinement strategy: structuring phase - parallel-state-simplification

For the case of a prefixing, the procedure tries to distribute the assumptions over the prefix, and recursively applies assumption-introduction to the prefixed action. In the case of an alternation, the assumption is distributed to the branches of the alternations, and the procedure is recursively applied to each branch. Once all the recursive calls to assumption-distribution have terminated, if all branches finish in the assumption being distributed, the assumption is extracted from the branches, and the procedure continues. The treatment of interleavings is similar, but uses a simple novel law int-assump-extract to extract the trailing assumption from the interleaved actions.

This is a recursive procedure over the syntactic structure of actions; it terminates because the structure of the actions is finite, and every recursive call is made to a strict subaction of the action being treated.

Procedure parallel-state-simplification When entering, executing or exiting a state with a parallel decomposition, its substates may be entered, executed or exited. In this case, the status of each substate is checked before proceeding. As already mentioned, this procedure tries to simplify the corresponding alternations because some of the checks can be removed. For instance, in a chart without local event broadcasts, it is always the case that at the beginning of the cycle, every parallel substate of an inactive state is also inactive.

The alternations refined by this procedure all contain guards that refer to state components of the form $C_DWork.is_active_S$ or $C_DWork.is_S$. Figure 5.54 presents the steps.

In Step 1, we introduce an assumption from the initialisation of the state component C_DWork (stating that every state is inactive), and introduce the assumption below as a consequence of that.

$$\left\{ \begin{array}{l} \exists\, C_State \bullet \\ \left(\begin{array}{l} RetrieveFunction \land \\ \forall\, s : \operatorname{ran} state \mid s.decomposition = SET \bullet \\ \quad (state_status(s) = \mathbf{True} \Rightarrow (\forall\, ss : \operatorname{ran} s.substates \bullet state_status(ss) = \mathbf{True})) \land \\ \forall\, s : \operatorname{ran} state \bullet \\ \quad (state_status(s) = \mathbf{False} \Rightarrow (\forall\, ss : \operatorname{ran} s.substates \bullet state_status(ss) = \mathbf{False})) \end{array} \right) \end{array} \right\}$$

This assumption states that for every parallel state, if it is active, all its substates are active as well. This holds because the initialisation marks all states as inactive making the assumption trivially true.

Step 2 distributes the assumption over the recursion, and, subsequently, distributes it as far as possible through the body of the recursion towards the assignments. This is achieved using assumption-distribution, already used in the procedure early-return-simplification presented previously, except that we first apply Law assump-rec-dist to move the assumption into the recursion. (This has itself a refinement as a proviso.)

In Step 3, for each alternation with a guard that checks the status of a state S with parallel decomposition (is of the form $is_active_S = e$ or $is_P = C_IN_S$), the procedure introduces the assumptions established by the guards, and distributes them through the branches using assumption-introduction again. Finally, for each alternation inside a branch that checks the status of a substate, Step 3(c) merges the assumptions that precede this inner alternation, and tries to simplify it using the merged assumption.

We illustrate the Step 3 of this procedure by simplifying the model of the execution of the states `FAN1`, `FAN2`, `SpeedValue` in our example. After Step 2, the assumption introduced in Step 1 has been distributed through the whole action, and has been kept before each alternation. Step 3 targets alternations such as the one sketched in Figure 5.55, where $\{\ldots\}$ abbreviates occurrences of the assumption distributed in Step 2, one of which, however, is sketched more fully: that just before the first of the innermost alternation. Step 3(a) introduces the assumptions derived from the guards of the outermost alternation (Figure 5.56).

Step 3(b) distributes these assumptions using the procedure assumption-introduction as before. We show in Figure 5.57 the result for the second branch of the alternation. For the first branch, since `PowerOff` has no substates, the distribution is straightforward, and reaches no further nested alternations.

In this example, Step 3(c) is concerned with the alternations whose guards refer to the components is_active_FAN1, is_active_FAN2 and $is_active_SpeedValue$ of Air_DWork. Starting at the first of them in Figure 5.57, in Step 3(c)-i we merge the assumptions that precede it to obtain the assumption shown in Figure 5.58.

In Step 3(c)-ii we simplify the alternation since this assumption implies the guard of

$$\{\ldots\}\ ;\ \left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_c1_Air = Air_IN_PowerOff \longrightarrow \ldots \\
[]\ Air_DWork.is_c1_Air = Air_IN_PowerOn \longrightarrow \{\ldots\}; \\
\quad \left(\begin{array}{l}
\textbf{if}\ _sfEvent_Air_ = Air_event_SWITCH \longrightarrow \ldots \\
[]\ \neg\ (_sfEvent_Air_ = Air_event_SWITCH) \longrightarrow \\
\quad \left(\begin{array}{l}
\left\{\begin{array}{l}
\ldots \wedge \\
Air_DWork.is_c1_Air = Air_IN_PowerOn \Rightarrow \\
\quad \left(\begin{array}{l}
Air_DWork.is_active_FAN1 = 1 \wedge \\
Air_DWork.is_active_FAN2 = 1 \wedge \\
Air_DWork.is_active_SpeedValue = 1
\end{array}\right)
\end{array}\right\}; \\
\left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_active_FAN1 = 1 \longrightarrow (\mathsf{FAN1}) \\
[]\ \neg\ (Air_DWork.is_active_FAN1 = 1) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right); \\
\{\ldots\}; \\
\left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_active_FAN2 = 1 \longrightarrow (\mathsf{FAN2}) \\
[] \neg\ (Air_DWork.is_active_FAN2 = 1) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right); \\
\{\ldots\}; \\
\left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_active_SpeedValue = 1 \longrightarrow (\mathsf{SpeedValue}) \\
[]\ \neg\ (Air_DWork.is_active_SpeedValue = 1) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right)
\end{array}\right) \\
\textbf{fi}
\end{array}\right) \\
\textbf{fi}
\end{array}\right)$$

Figure 5.55: parallel-state-simplification: example after step 2.

$$\{\ldots\}\ ;\ \left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_c1_Air = Air_IN_PowerOff \longrightarrow \\
\quad \{Air_DWork.is_c1_Air = Air_IN_PowerOff\}\ ;\ \ldots \\
[]\ Air_DWork.is_c1_Air = Air_IN_PowerOn \longrightarrow \\
\quad \{Air_DWork.is_c1_Air = Air_IN_PowerOn\}\ ;\ \{\ldots\}\ ;\ \ldots \\
\textbf{fi}
\end{array}\right)$$

Figure 5.56: parallel-state-simplification: example after step 3(a).

$$
\{\ldots\}\,; \left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_c1_Air = Air_IN_PowerOff \longrightarrow \ldots \\
[]\ Air_DWork.is_c1_Air = Air_IN_PowerOn \longrightarrow \\
\quad \left(\begin{array}{l}
\{\ldots\}\,;\ \{Air_DWork.is_c1_Air = Air_IN_PowerOn\}; \\
\mathbf{if}\ _sfEvent_Air_ = Air_event_SWITCH \longrightarrow \ldots \\
[]\ \neg\,(_sfEvent_Air_ = Air_event_SWITCH) \longrightarrow \\
\left(\begin{array}{l}
\{\ldots\}\,;\ \{Air_DWork.is_c1_Air = Air_IN_PowerOn\}; \\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_active_FAN1 = 1 \longrightarrow (\mathtt{FAN1}) \\
[]\ \neg\,(Air_DWork.is_active_FAN1 = 1) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\{\ldots\}\,;\ \{Air_DWork.is_c1_Air = Air_IN_PowerOn\}; \\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_active_FAN2 = 1 \longrightarrow \ldots \\
[]\ \neg\,(Air_DWork.is_active_FAN2 = 1) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\{\ldots\}\,;\ \{Air_DWork.is_c1_Air = Air_IN_PowerOn\}; \\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_active_SpeedValue = 1 \longrightarrow \ldots \\
[]\ \neg\,(Air_DWork.is_active_SpeedValue = 1) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
$$

Figure 5.57: parallel-state-simplification: example after step 3(b).

$$
\left\{\begin{array}{l}
\ldots \wedge \\
\left(Air_DWork.is_c1_Air = Air_IN_PowerOn \Rightarrow \left(\begin{array}{l}
Air_DWork.is_active_FAN1 = 1 \wedge \\
Air_DWork.is_active_FAN2 = 1 \wedge \\
Air_DWork.is_active_SpeedValue = 1
\end{array}\right)\right) \wedge \\
Air_DWork.is_c1_Air = Air_IN_PowerOn
\end{array}\right\}
$$

Figure 5.58: parallel-state-simplification: example after step 3(c)-i.

$$\{\ldots\}\ ;\ \left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_c1_Air = Air_IN_PowerOff \longrightarrow \ldots\\
[]\ Air_DWork.is_c1_Air = Air_IN_PowerOn \longrightarrow\\
\quad \left(\begin{array}{l}
\{\ldots\}\ ;\ \{Air_DWork.is_c1_Air = Air_IN_PowerOn\};\\
\textbf{if}\ _sfEvent_Air_ = Air_event_SWITCH \longrightarrow \ldots\\
[]\ \neg\ (_sfEvent_Air_ = Air_event_SWITCH) \longrightarrow\\
\quad \left(\begin{array}{l}
\left\{\begin{array}{l}
\ldots \wedge\\
\left(\begin{array}{l}
Air_DWork.is_c1_Air = Air_IN_PowerOn \Rightarrow\\
\quad \left(\begin{array}{l}
Air_DWork.is_active_FAN1 = 1 \wedge\\
Air_DWork.is_active_FAN2 = 1 \wedge\\
Air_DWork.is_active_SpeedValue = 1
\end{array}\right)
\end{array}\right) \wedge\\
Air_DWork.is_c1_Air = Air_IN_PowerOn
\end{array}\right\};\\
\text{(Entering state FAN1)};\\
\{\ldots \wedge Air_DWork.is_c1_Air = Air_IN_PowerOn\};\\
\text{(Entering state FAN2)};\\
\{\ldots \wedge Air_DWork.is_c1_Air = Air_IN_PowerOn\};\\
\text{(Entering state SpeedValue)};
\end{array}\right)\\
\textbf{fi}
\end{array}\right)
\end{array}\right)$$

Figure 5.59: parallel-state-simplification: example after step 3(c)-ii.

1. While there are nested alternations, where one branch of the outermost alternation has a guard containing $C_DWork.is_S$
   (a) **Flatten alternation**. Apply Law alt-alt-dist.
   (b) **Rewrite guard.** For each branch whose guard is a conjunction of one equality and a number of inequalities, apply Law alt-guard-rewrite to rewrite the guard as the single equality.

Figure 5.60: Procedure sequential-state-simplification

its first branch. All other alternations are refined in a similar way. The final result is as in Figure 5.59. All alternations are removed, but the assumptions are left behind (and removed later).

Procedure sequential-state-simplification As explained previously, this is the procedure that flattens the nested alternations that execute sequential states, to obtain a multi-branch alternation. Each branch, except the last, corresponds to the execution of one state of a sequential decomposition, and its guard checks if it is active. The last branch caters for the case where no state is active (and so, its guard is a conjunction of inequalities). The steps of this procedure are in Figure 5.60.

This procedure systematically flattens and rewrites nested alternations whose guards refer to a component $C_DWork.is_S$, and therefore relates to a state S that has a sequential decomposition. Each alternation is transformed by applying the Law alt-alt-dist to flatten it, and by rewriting its guards using Law alt-guard-rew. The guards resulting from applying

law alt-alt-dist may be either conjunctions of inequalities

$C_DWork.is_S \neq C_IN_S_1 \land \ldots \land C_DWork.is_S \neq C_in_S_m$

or an equality conjoined to a conjunction of inequalities

$C_DWork.is_S = C_IN_S_p \land C_DWork.is_S \neq C_IN_S_1 \land \ldots \land$
$C_DWork.is_S \neq C_in_S_n$

The first type of guard is left unchanged; the second type is rewritten to the single equality that it contains.

5.3.4 Parallelism introduction

The previous phase introduced the sequential control structure of the implementation. In this phase, which is necessary only for the verification of parallel implementations, we introduce in the process actions that define servers and their composition. Additionally, we transform the main action into a parallel composition of the client and server actions, according to the architectural pattern described in Section 5.2.

This phase relies on the following information about the implementation and its model: for each server, the block of code that implements its core functionality, the block of code of the client delimited by the synchronisations with that server, the input and output channels it uses to communicate with the client, and the state components modified by it. As previously explained, the input and output channels communicate, respectively, the whole state of the process, and the modified state components. The synchronisation points in the program are modelled in *Circus* as communications between the client and server actions, and the use of shared variables by communication of the relevant values at the synchronisation points: the whole state in the first synchronisation, and the altered components in the second.

Precisely, the parameters for this phase are a sequence *servers* of pairs of blocks of code of the implementation, a sequence *channels* of pairs of channels, and a sequence *changes* of lists of state component names. The first block of code in each pair of *servers* is the implementation of the functionality of the server, and the second is the code of the client with which the server state is run in parallel. The pair of channels in position i of *channels* corresponds to the input and output channels used to model the i-th server code in *servers*. The list in $changes(i)$ gives the state components modified by that same code.

The parallel implementation of our running example has only one server, which is implemented by the function `FAN1` shown in Figure 5.61. It is an infinite loop that, at each step, synchronises with the client, executes the function `Air_FAN1`, also shown in Figure 5.61, and synchronises with the client again signalling the end of its calculation. The function `Air_FAN1` implements the core functionality of this server.

In this example, therefore, the parameter *servers* contains the pair where the first el-

```
void *FAN1(void *arg) {
    while(1) {
        synchronise(); Air_FAN1(); synchronise();
    }
}
static void Air_FAN1(void) {
    switch (Air_DWork.is_FAN1) {
    case Air_IN_Off:
        if (Air_U.temp >= 120.0) {
            Air_DWork.is_FAN1 = Air_IN_On;
        }
        break;
    case Air_IN_On:
        if (Air_U.temp < 120.0) {
            Air_DWork.is_FAN1 = Air_IN_Off;
        }
        break;
    default:
        Air_DWork.is_FAN1 = Air_IN_Off;
        break;
    }
}
```

Figure 5.61: Functions implemented the server for our example

$$\left(\begin{array}{l}
\textbf{if } Air_DWork.is_FAN1 = Air_IN_Off \longrightarrow \\
\quad \left(\begin{array}{l} \textbf{if } Air_U.temp \geq 120 \longrightarrow Air_DWork.is_FAN1 := Air_IN_On \\ [\!] \neg\,(Air_U.temp \geq 120) \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right) \\
[\!] Air_DWork.is_FAN1 = Air_IN_On \longrightarrow \\
\quad \left(\begin{array}{l} \textbf{if } Air_U.temp < 120 \longrightarrow Air_DWork.is_FAN1 := Air_IN_Off \\ [\!] \neg\,(Air_U.temp < 120) \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right) \\
[\!] \left(\begin{array}{l} Air_DWork.is_FAN1 \neq Air_IN_Off \wedge \\ Air_DWork.is_FAN1 \neq Air_IN_On \end{array}\right) \longrightarrow Air_DWork.is_FAN1 := Air_IN_Off \\
\textbf{fi}
\end{array}\right);$$

$$\left(\begin{array}{l}
\textbf{if } Air_DWork.is_FAN2 = Air_IN_Off \longrightarrow \\
\quad \left(\begin{array}{l} \textbf{if } Air_U.temp \geq 150 \longrightarrow Air_DWork.is_FAN2 := Air_IN_On \\ [\!] \neg\,(Air_U.temp \geq 150) \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right) \\
[\!] Air_DWork.is_FAN2 = Air_IN_On \longrightarrow \\
\quad \left(\begin{array}{l} \textbf{if } Air_U.temp < 150 \longrightarrow Air_DWork.is_FAN2 := Air_IN_Off \\ [\!] \neg\,(Air_U.temp < 150) \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right) \\
[\!] \left(\begin{array}{l} Air_DWork.is_FAN2 \neq Air_IN_Off \wedge \\ Air_DWork.is_FAN2 \neq Air_IN_On \end{array}\right) \longrightarrow Air_DWork.is_FAN2 := Air_IN_Off \\
\textbf{fi}
\end{array}\right)$$

Figure 5.62: Parallelism introduction - example of execution of parallel states.

$$
\begin{array}{l}
(in_FAN1!(\theta\ ConcreteState) \longrightarrow \\
\left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_FAN2 = Air_IN_Off \longrightarrow \\
\quad \left(\begin{array}{l}
\textbf{if}\ Air_U.temp \geq 150 \longrightarrow Air_DWork.is_FAN2 := Air_IN_On \\
[\!]\neg\ (Air_U.temp \geq 150) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right) \\
[\!] Air_DWork.is_FAN2 = Air_IN_On \longrightarrow \\
\quad \left(\begin{array}{l}
\textbf{if}\ Air_U.temp < 150 \longrightarrow Air_DWork.is_FAN2 := Air_IN_Off \\
[\!]\neg\ (Air_U.temp < 150) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right) \\
[\!] \left(\begin{array}{l}
Air_DWork.is_FAN2 \neq Air_IN_Off \land \\
Air_DWork.is_FAN2 \neq Air_IN_On
\end{array}\right) \longrightarrow Air_DWork.is_FAN2 := Air_IN_Off \\
\textbf{fi}
\end{array}\right); \\
out_FAN1?x \longrightarrow Air_DWork.is_FAN1 := x
\end{array}
$$

Figure 5.63: Parallelism introduction target - execution of parallel states example.

ement is the body of the function `Air_FAN1`, and the second element is the block of code that executes state `FAN2`. The parameter *channels* contains the pair of input and output channels (in_FAN1, out_FAN1), and *changes* the singleton list containing the state component $Air_DWork.is_FAN1$. The channel in_FAN1 communicates the whole state under which the client is to be executed, and the channel out_FAN1 communicates a value of type $\mathbb{N}$ corresponding to the single element modified by the server $(Air_DWork.is_FAN1)$.

Starting point As previously mentioned, this phase acts upon the process obtained from the last phase, whose main action models a sequential implementation of the chart. Its general structure is shown in Figure 5.19. It initialises the state and recursively reads the inputs, executes the chart, and writes the output. As indicated in Figure 5.19 itself, the execution of a chart is defined by an action like that in Figure 5.20. There, parallel states executions are defined by sequences of alternations. For our example, we have alternations in sequence for `FAN1`, `FAN2`, `SpeedValue`. Those for `FAN1` and `FAN2` are shown in Figure 5.62.

Target As mentioned above, the main action after this phase is a parallelism. The *Client* parallel action is similar to that in Figure 5.19. It initialises the state, and recursively reads the inputs, executes the chart, and writes the outputs. In executing the chart, however, it executes states in parallel. For our example, the execution of the chart involves the execution of `FAN2` and a request for the server to execute `FAN1`; this is depicted in Figure 5.63. The server side of the parallelism executes `FAN1` as shown in Figure 5.64.

Refinement steps Figure 5.65 presents the steps of the parallelism introduction phase.

Step 1(a) calculates the *Circus* actions S_i and C_i that model the blocks of code in each pair of *servers*. In our example, we obtain the *Circus* actions corresponding to the body of the function `Air_FAN1` (see Figure 5.61) and the execution of state `FAN2` in *servers*. The first action is identical to the outermost alternation in Figure 5.64, and the second is

$$
\begin{array}{l}
\mu X \bullet \\
\left(\begin{array}{l}
\mathbf{var}\ Air_B : BlockIO_Air;\ \ldots;\ sfEvent_Air : \mathbb{N} \bullet \\
\left(\begin{array}{l}
in_FAN1?s \longrightarrow (Air_B := s.Air_B\ ;\ \ldots\ ;\ sfEvent_Air := s.sfEvent_Air); \\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_FAN1 = Air_IN_Off \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ Air_U.temp \geq 120 \longrightarrow Air_DWork.is_FAN1 := Air_IN_On \\
[\!]\neg\ (Air_U.temp \geq 120) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!] Air_DWork.is_FAN1 = Air_IN_On \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ Air_U.temp < 120 \longrightarrow Air_DWork.is_FAN1 := Air_IN_Off \\
[\!]\neg\ (Air_U.temp < 120) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!] \left(\begin{array}{l}
Air_DWork.is_FAN1 \neq Air_IN_Off\ \wedge \\
Air_DWork.is_FAN1 \neq Air_IN_On
\end{array}\right) \longrightarrow \\
\quad Air_DWork.is_FAN1 := Air_IN_Off \\
\mathbf{fi}
\end{array}\right); \\
out_FAN1!(Air_DWork.is_FAN1) \longrightarrow \mathbf{Skip}
\end{array}\right);\ X
\end{array}\right)
\end{array}
$$

Figure 5.64: Parallelism introduction target - server example.

1. For $i = 1 \mathrel{..} \#\, servers$
   - (a) **Calculate *Circus* action.** Obtain the *Circus* actions S_i and C_i that, respectively, model the portions of code in $servers(i).1$ and $servers(i).2$
   - (b) **Introduce server.** Apply Law server-intro with parameters $\{d_1, \ldots, d_n\} = usedV(S_i) \cup wrtV(S_i)$, $\{t_1, \ldots, t_m\} = changes(i)$, $in = channels(i).1$, $out = channels(i).2$, and $S = S_i$ to the left parallel action of the innermost parallelism in the main action. (For $i = 1$, this is the main action itself.)
   - (c) **Synchronise server and client.** Exhaustively apply Law sync-client-server to sequences of the form

     $$(channels(i).1!s \longrightarrow channels(i).2?x \longrightarrow t := x)\ ;\ C_i$$

2. **Introduce interleaving in the server.** Apply Law par-int-intro to the main action of the process.

Figure 5.65: Refinement strategy: parallelism introduction phase

exactly the second action in the sequential composition in Figure 5.62.

Step 1(b) refines the main action of the process into a parallelism between the server and the original main action refined to use the server. It applies the novel Law server-intro presented below.

Law[server-intro]

$$A\ ;\ (\mu X \bullet B[S]\ ;\ X)$$
$$=$$
$$\left(\begin{array}{l} A\ ;\ (\mu X \bullet B[in!(d_1,\ldots,d_n) \longrightarrow out?x \longrightarrow t_1,\ldots,t_m := x.1,\ldots,x.m]\ ;\ X) \\ \quad [\![\{d_1,\ldots,d_n\} \mid \{\!|\ in, out\ |\!\} \mid \{\}]\!] \\ \left(\mu X \bullet \left(\begin{array}{l} \mathbf{var}\ d_1 : D_i;\ \ldots;\ d_n : D_n \bullet \\ \quad \left(\begin{array}{l} in?s \longrightarrow d_1 := s.1\ ;\ \ldots;\ d_n := s.n; \\ S; \\ out!(t_1,\ldots,t_m) \longrightarrow \mathbf{Skip} \end{array}\right) \end{array}\right)\ ;\ X\right) \end{array}\right) \setminus \{\!|\ in, out\ |\!\}$$

where

- $\{d_1,\ldots,d_n\} = usedV(S) \cup wrtV(S)$.
- $\{t_1,\ldots,t_m\}$ *are variables and components of record-valued variables that are changed by* S.
- d_i *has type* D_i, *and* t_i *has type* T_i.
- in *has type* $D_1 \times \ldots \times D_n$ *and* out *has type* $T_1 \times \ldots \times T_m$.

provided

- $in \notin usedC(A, B)$;
- $out \notin usedC(A, B)$.

This law applies to a non-terminating tail-recursive action $A\ ;\ (\mu X \bullet B[S]\ ;\ X)$, where a (server) action S occurs in the body of the recursion. It transforms that action into a parallelism where the left-hand parallel action is the original recursive action with all occurrences of S changed. Instead of S, we have, first, a communication over a fresh channel in to send the value of the variables $d_1,\ldots,d_n$ that are used and written by S. A second communication over another fresh channel out reads a tuple of new values x for the variables $t_1,\ldots t_n$ changed by S. An assignment updates these variables according to x. Correspondingly, the right-hand parallel action recursively declares variables corresponding to the used and written variables of S, reads their values from in and assigns the values to the appropriate local variables, executes S and communicates through out the values of the variables written by S.

In the resulting parallel action, every communication on in prompts the execution of a step of the server action S on an exact copy of the state at that moment, when otherwise S would have been executed. Once the server finishes its calculations, it communicates the changed variables through out, and these are read by the client action, which up-

$$
\begin{array}{l}
in_FAN1!(\theta\ ConcreteState) \longrightarrow out_FAN1?x \longrightarrow Air_DWork.is_FAN1 := x;\\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_FAN2 = Air_IN_Off \longrightarrow\\
\quad\left(\begin{array}{l}
\mathbf{if}\ Air_U.temp \geq 150 \longrightarrow Air_DWork.is_FAN2 := Air_IN_On\\
[\!]\neg\ (Air_U.temp \geq 150) \longrightarrow \mathbf{Skip}\\
\mathbf{fi}
\end{array}\right)\\
[\!]Air_DWork.is_FAN2 = Air_IN_On \longrightarrow\\
\quad\left(\begin{array}{l}
\mathbf{if}\ Air_U.temp < 150 \longrightarrow Air_DWork.is_FAN2 := Air_IN_Off\\
[\!]\neg\ (Air_U.temp < 150) \longrightarrow \mathbf{Skip}\\
\mathbf{fi}
\end{array}\right)\\
[\!]\left(\begin{array}{l}
Air_DWork.is_FAN2 \neq Air_IN_Off\ \wedge\\
Air_DWork.is_FAN2 \neq Air_IN_On
\end{array}\right) \longrightarrow Air_DWork.is_FAN2 := Air_IN_Off\\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

Figure 5.66: Parallelism introduction - example: portion of the main action after Step 1.

dates the state accordingly. In this way, the result of executing the substituted action $(in!(\ldots) \longrightarrow out?x \longrightarrow \ldots)$ is exactly the same as if S had been executed at that point.

The result of applying Step 1(b) of this phase, therefore, is a parallel main action calculated by the above law. For our example, we have a parallelism between the server in Figure 5.64 and the original action with the sequence shown in Figure 5.62 replaced with that in Figure 5.66.

At this stage, the main action has the following form, where k is the number of servers considered so far.

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
A\ ;\ \left(\ \mu X \bullet B[in_k!(\ldots) \longrightarrow out_k?t \longrightarrow v_k := e_k\ ;\ C_k]\ ;\ X\ \right)\\
\qquad [\![\alpha ConcreteState \mid \{\!|\ in_k, out_k\ |\!\} \mid \{\}]\!]\\
(\mu X \bullet \left(\begin{array}{l}
\mathbf{var} \ldots \bullet in_k?s \longrightarrow \ldots;\\
Server_k\ ;\ out_k!(\ldots) \longrightarrow \mathbf{Skip}
\end{array}\right)\ ;\ X)
\end{array}\right) \setminus \{\!|\ in_k, out_k\ |\!\}\\
\ldots\\
\qquad [\![\alpha ConcreteState \mid \{\!|\ in_1, out_1\ |\!\} \mid \{\}]\!]\\
(\mu X \bullet \left(\begin{array}{l}
\mathbf{var} \ldots \bullet in_1?s \longrightarrow \ldots;\\
Server_1\ ;\ out_1!(\ldots) \longrightarrow \mathbf{Skip}
\end{array}\right)\ ;\ X)
\end{array}\right) \setminus \{\!|\ in_1, out_1\ |\!\}
$$

In the parallel client action, $in_k!(\ldots) \longrightarrow out_k?t \longrightarrow v_k := e_k\ ;\ C_k$ requests the execution of the server $(in_k!(\ldots))$, and waits for its conclusion $(out_k?t)$, before starting the client action C_k. In Step 1(c), we move the final synchronisation $out_k?t$ with the server to after C_k, so that it can then proceed in parallel with $Server_k$. This is possible because our architectural constraints ensure that C_k and $Server_k$ do not share variables.

Step 1(c) applies the novel Law sync-client-server presented below.

Law[sync-client-server]

$$\left(\begin{array}{l} A\ ;\ (\mu X \bullet B[in!a \longrightarrow out?x \longrightarrow v := e\ ;\ C]\ ;\ X) \\ \qquad [\![ns \mid \{\!|\ in, out\ |\!\} \mid \{\}]\!] \\ (\mu X \bullet (\mathbf{var}\ d : T \bullet in?y \longrightarrow D\ ;\ out!b \longrightarrow \mathbf{Skip})\ ;\ X) \end{array}\right) \setminus \{\!|\ in, out\ |\!\}$$
$$=$$
$$\left(\begin{array}{l} A\ ;\ (\mu X \bullet B[in!a \longrightarrow C\ ;\ out?x \longrightarrow v := e]\ ;\ X) \\ \qquad [\![ns \mid \{\!|\ in, out\ |\!\} \mid \{\}]\!] \\ (\mu X \bullet (\mathbf{var}\ d : T \bullet in?y \longrightarrow D\ ;\ out!b \longrightarrow \mathbf{Skip})\ ;\ X) \end{array}\right) \setminus \{\!|\ in, out\ |\!\}$$

provided

- $in \notin usedC(A, B, C, D)$ and $out \notin usedC(A, B, C, D)$;
- $usedV(out?x \longrightarrow v := e) \cap wrtV(C) = \varnothing$ and $v \notin usedV(C)$.

This law applies to a parallel action where the right action is a server communicating over channels in and out, and the left action is a client that requests the execution of the server through in, reads the server's answer from out, updates the state, and executes an action C. This law modifies the parallelism to allow C to run in parallel with the server. This is possible if C and the preceding prefixing on out do not share variables, and if in and out are not used anywhere else in the action.

In the original parallelism, if a communication over in occurs, the server starts executing D and the client waits for a communication on out. After the communication on out occurs, the server recurses and waits for the next request on in, and the client executes C. In the transformed parallelism, after the initial communication on in, the server executes D and the client immediately executes C, then the client and the server communicate on out and proceed. These are the only possible interactions because channel in and out are not used anywhere else in the action. Since C does not communicate over any channels, it is not possible to distinguish whether C occurs before or after the synchronisation on out. Furthermore, C and $out?x \longrightarrow v := e$ do not share variables, thus we may exchange the two actions.

As mentioned above, Step 1(c) reorganises the communication between the server and the client using the facts that in_k and out_k are not used anywhere else except in the server, and that $v_k := e_k$ and C_k do not share variables. The resulting action is as follows.

$$\left(\begin{array}{l} \left(\begin{array}{l} A\ ;\ \left(\mu X \bullet B[in_k!(\ldots) \longrightarrow C_k\ ;\ out_k?t \longrightarrow v_k := e_k]\ ;\ X \right) \\ \qquad [\![\alpha ConcreteState \mid \{\!|\ in_k, out_k\ |\!\} \mid \{\}]\!] \\ (\mu X \bullet (\mathbf{var} \ldots \bullet in_k?s \longrightarrow \ldots;\ Server_k \longrightarrow out_k!(\ldots) \longrightarrow \mathbf{Skip})\ ;\ X) \end{array}\right) \setminus \{\!|\ in_k, out_k\ |\!\} \\ \ldots \\ \qquad [\![\alpha ConcreteState \mid \{\!|\ in_1, out_1\ |\!\} \mid \{\}]\!] \\ (\mu X \bullet (\mathbf{var} \ldots \bullet in_1?s \longrightarrow \ldots;\ Server_1 \longrightarrow out_1!(\ldots) \longrightarrow \mathbf{Skip})\ ;\ X) \end{array}\right) \setminus \{\!|\ in_1, out_1\ |\!\}$$

In this step, we assume that the server action S_i models the execution of a parallel state

S, C_i models the execution of one or more parallel states CS (that do not share variables with S) and that these states are adjacent to each other with respect to their execution order as defined in the chart. If there are any other states PS between S and CS in that sequence, there are three possible situations: (1) S, PS and CS are independent (do not share variables) and, thus, can all be put in parallel, (2) PS depends on S, or (3) CS depends on PS. We assume that the parallelism of the chart is fully explored at least within an individual group of parallel states, thus, in the first case we require the states PS are also put in parallel. The other two cases present complications because they require reordering of states. In case (2), we must show that CS does not depend on PS and exchange both blocks before applying Step 1(c) to synchronise the client and the server. In case (3), the blocks that execute S and PS must be exchanged. We do not treat these cases as they require further analysis of the chart. We leave this extension of our strategy as future work.

After Step 1 terminates, the main action has the following structure.

$$\left(\begin{array}{l}\left(\begin{array}{l}A\ ;\ \left(\ \mu X \bullet B\ ;\ X\ \right)\\ \qquad [\![\alpha ConcreteState \mid \{\!| in_n, out_n |\!\} \mid \{\}]\!]\\ (\mu X \bullet (\mathbf{var} \ldots \bullet in_n?s \longrightarrow \ldots ;\ Server_n \longrightarrow out_n!(\ldots) \longrightarrow \mathbf{Skip})\ ;\ X)\end{array}\right) \setminus \{\!| in_n, out_n |\!\}\\ \ldots\\ \qquad [\![\alpha ConcreteState \mid \{\!| in_1, out_1 |\!\} \mid \{\}]\!]\\ (\mu X \bullet (\mathbf{var} \ldots \bullet in_1?s \longrightarrow \ldots ;\ Server_1 \longrightarrow out_1!(\ldots) \longrightarrow \mathbf{Skip})\ ;\ X)\end{array}\right) \setminus \{\!| in_1, out_1 |\!\}$$

Since the servers communicate only with the client, it is possible to rearrange the main action as a single parallelism between the client action and all the servers in interleave. This is achieved by Step 2, which exhaustively applies Law par-int-intro below to transform the action into the desired parallelism.

Law[par-int-intro]

$$((A\ [\![\ ns \mid cs_1 \mid \{\}\ ]\!]\ B) \setminus cs_1\ [\![\ ns \mid cs_2 \mid \{\}\ ]\!]\ C) \setminus cs_2$$
$$=$$
$$(A\ [\![\ ns \mid cs_1 \cup cs_2 \mid \{\}\ ]\!]\ (B \mathbin{|||} C)) \setminus cs_1 \cup cs_2$$

provided

- $usedC(B) \cap usedC(C) = \varnothing$;
- $cs_1 = usedC(B)$;
- $cs_2 = usedC(C)$;
- $cs_1 \cup cs_2 \subseteq usedC(A)$.

This law applies to an action formed by two nested parallel compositions. The innermost parallelism is between actions A and B synchronising on cs_1, and only A writes to any variables. The outermost parallel action combines the parallelism just described, and an action C; they synchronise on cs_2, and only the inner parallelism writes to variables (be-

1. **Introduce actions.** Apply Law action-intr [14] to the process and each action in the model of the implementation.
2. **Introduce call actions.** Exhaustively apply the copy rule from right to left to the main action and the newly introduced actions.

Figure 5.67: Refinement strategy: action introduction phase.

cause A does). The hiding over the inner parallelism can be expanded over the whole action, because C does not communicate over the channels on cs_1 (since cs_1 is the set of channels used in B, and the sets of channels used by B and C are disjoint). Since cs_1 and cs_2 are the sets of channels used by, respectively, B and C, and these two actions do not share channels, they can be interleaved. B and C both communicate with A, because the union of cs_1 and cs_2 is in the set of channels used by A, thus, they can be put in parallel synchronising on the union of their communication sets. Finally, in both parallelisms, the only action that can write to variables in the scope of the parallelism is A.

The general form of the action that results from the application of the final Step 2 is shown below.

$$\left(\begin{array}{l} A \; ; \; \left(\; \mu X \bullet B \; ; \; X \; \right) \\ \quad [\![\alpha ConcreteState \mid \{\!| in_1, out_1, \ldots, in_n, out_n |\!\} \mid \{\}]\!] \\ \left(\begin{array}{l} (\mu X \bullet (\mathbf{var} \ldots \bullet in_1?s \longrightarrow \ldots ; \; Server_1 \longrightarrow out_1!(\ldots) \longrightarrow \mathbf{Skip}) \; ; \; X) \\ \quad ||| \cdots ||| \\ (\mu X \bullet (\mathbf{var} \ldots \bullet in_n?s \longrightarrow \ldots ; \; Server_n \longrightarrow out_n!(\ldots) \longrightarrow \mathbf{Skip}) \; ; \; X) \end{array} \right) \end{array} \right) \setminus \{\!| in_1, out_1, \ldots, in_n, out_n |\!\}$$

This structure matches the architectural pattern shown in Figure 5.9. Since our example has only one server, this step has no impact on the final result.

5.3.5 Action introduction

After the parallelism introduction phase, the main action of the refined process should be the same as that of the model of the implementation, except that the main action of the implementation is decomposed into a number of subactions. In this phase, we refine our process to match exactly the process that models the implementation. Figure 5.67 describes the steps for this phase.

In the first step, the actions of the model of the implementations are introduced in the process being refined. Next, in Step 2 we exhaustively apply the copy-rule from right to left to replace occurrences of the definitions of the actions introduced in Step 1 with a call to the appropriate action. The main action of the process resulting from the application of this phase to our example is as follows.

$$ExecuteChart \; [\![\; \{_sfEvent_Air_, Air_U, Air_B, Air_DWork, Air_Y\} \mid \{\!| in_FAN1, out_FAN1 |\!\} \mid \{\} \;]\!] \; FAN1$$

ExecuteChart calls the action *MdlInitialize* to initialise the state, and recursively reads

the inputs using the action *read_inputs*, executes the chart using *Air_output*, writes the outputs using *write_outputs*, and signals the end of the step by synchronising on *end_cycle*. The action $FAN1$ is that in Figure 5.64.

This completes the verification of our example.

5.4 Final considerations

In this chapter, we have identified a simple, but general architectural pattern for the parallel implementation of Stateflow charts, based on the sequential architecture implemented by MATLAB's automatic code generator [98, 100]. Based on this pattern, we have proposed a refinement strategy for the verification of implementations. While parts of the strategy are dependent on the architecture of the implementation (namely, data refinement, structuring (except steps 2 and 3) and parallelism introduction phases), other parts can be reused in strategies that target different architectures.

In particular, the normalisation phase and the procedure parallelism-resolution are central to any strategy that deals with our models as they support the collapsing of the process parallelism. The procedure parallelism-resolution is particularly important as it supports the use of our modelling approach, where the core of the semantics is separated from structural aspects of the notation.

Additionally, the architecture-dependent procedures can also be a starting point for other strategies. While their details may require revision, the fundamental underlying principles are bound to remain the same. For instance, procedures similar to those used in the structuring phase, where appropriate assumptions are introduced and distributed through the action to simplify the structure, are likely to be widely applicable.

Our refinement strategy makes a number of assumptions about the automatically generated models of Stateflow chart: deadlock free, divergence free, and deterministic. All of these properties can be checked with a model checker, but they are also evident from the structure of the model. It is not inconceivable that it is possible to prove that every model of a well formed chart satisfies these properties. The strategy uses the *Circus* refinement calculus and derives its soundness from the soundness of the refinement laws.

Our refinement strategy should produce the process that models the implementation. If this is not true, then either the implementation is incorrect, or it does not follow the architectural patterns verified by this strategy. In the first case, the comparison between the two process may shed some light into what is the actual problem. Although the issue of error traceability is interesting one, we leave it as future work. In the latter case, we can directly apply the refinement calculus to the model, or identify the architectural patterns used in the implementation, and adapt our refinement strategy to explore them.

Additionally, as already mentioned, we do not treat programs that include mutual recursions. Our strategy, however, can be extended to treat mutual recursions by modifying the procedure recursion-introduction to extract information about mutual recursions from the implementation, calculate the appropriate recursive action, and apply a version of the

law unique-fixed-point that refines actions to mutual recursions.

In general, the provisos generated by the refinement laws are simple and can be discharged using a theorem prover, except for the proviso of the law assump-rec-dist, which requires that the assumption holds before and after the body of the recursion. It is used to distribute an assumption through the recursion that characterises the step of execution of the chart. For charts without recursions originating from local event broadcasts, it is possible to describe a refinement strategy that discharges this proviso, and we leave this as future work. For cases where local event broadcast leads to recursive behaviour in the chart, the occurrence of early return logic may leave the chart in a state that violates some of the assumptions. In this case, the proviso does not hold, and the model is not further simplified. It is important to note, however, that the Stateflow code generator does not simplify the implementation in these cases either.

The refinement strategy we propose shows that our choice of a more operational model can be easily treated by a appropriately designed verification strategy. The verification of implementation through refinement has its advantages and disadvantages. It support a higher degree of automation and possibly scales better to larger cases studies[1] as it does not rely so strongly on theorem proving. The main disadvantage of our approach is that the verifiable implementations are very restricted, and, for more general implementations, new refinement strategies need to be developed.

For any similar treatments of the problem of verification of implementations, it is extremely important that the assumptions about the implementation are well understood and delimited because this facilitates the definition of the refinement strategy.

Overall, our strategy is a general approach for the verification of, possibly parallel, implementations of Stateflow charts. Moreover, it is a substantial starting point for the development of other refinement strategies, tailored for other architectural patterns, as it tackles aspects of the refinement process that are fundamental to any verification based on our automated technique for generation of chart models.

[1] This point is speculative, and needs further investigation

Chapter 6

Conclusions

In this chapter, we discuss the main contributions of our work, compare it with the closely related results in the literature, and provide directions for future work.

6.1 Thesis contributions

The use of graphical notations in general, and more specifically Simulink and Stateflow, is widespread in industry for the specification of a variety of systems. Furthermore, some of the systems modelled in such notations are safety critical, and thus require a higher level of assurance of correctness. At present, the use of formal methods in this context is almost exclusively restricted to the verification of properties of the models [20, 7, 101, 87]. While there has been some work on the verification of code generators [102, 86], and verification of implementations of Simulink diagrams [19], we are not aware of any proposed approaches for the verification of implementations of Stateflow charts. Our work stands as a different direction for the use of formal specification and refinement for graphical notations. The main contributions of the work presented in this thesis can be divided in five areas.

A new approach to semantic description. Traditionally, models of graphical notations that are tailored for verification of properties and implementations follow a denotational style, where each component of the notation is associated with some complete and independent element of the model. We adopt a completely different approach, in which we prioritise comprehensive coverage of the notation, and support for validation by inspection of the informal description, over the particular scenarios in which the models are to be used.

Our approach yields a number of advantages. Firstly, it provides a better support for the formalisation of the semantics of inherently non-compositional notations such as Stateflow. Secondly, it allows us to separate the semantics of the notation from the structure of particular charts, reducing the size of the chart-specific model. Finally, our approach does not embed simplifications of the semantics based on the particular chart structures, as such simplifications may introduce errors in the semantics of the charts.

A comprehensive modelling approach for Stateflow charts. Most of the formalisation of Stateflow charts (and other varieties of state diagram notations) identify a well-behaved subset of the notation, and restrict the defined semantics to this subset. In particular, as far as we know, there has not been a formalisation of Stateflow charts that completely treat local event broadcasts and early return logic. Our models provide a thorough treatment of these features, and describes without restrictions most of the core elements of the notation.

We have validated our models through:

- Translation of the models of some examples into CSP;
- Analysis and simulation of the CSP models;
- Comparison of the results of simulations to the expected behaviour;
- Translation of large industrial case-studies.

Formalisation and implementation of a technique for automatic generation of models. We have formalised the translation rules that characterise our models of Stateflow charts and define a technique for the automatic generation of these models. While most of the literature on formal semantics of Stateflow charts either does not give an account of the translation process, or provide rather informal translation rules, we formalise in Z the syntax of Stateflow and *Circus*, and the translation rules for each element of Stateflow that we cover in our models. Furthermore, frequently the translation of smaller aspects of the notation are overlooked (e.g. action language). Our translation rules cover these aspects as well. We provide an implementation of the translation rules in the tool *s2c*. We have thoroughly validated this tool and the models it generates by means of a number of examples, including industrial case-studies. The tool was further validated by parsing and type checking the generated models.

Architecture for implementations. We described an architecturals pattern for (parallel) implementations of Stateflow charts based on the architecture enforced by Stateflow code generator, extended with a client-server pattern for the execution of parallel states. This architecture identifies the main aspects of implementations that are required to support the automation of a refinement strategy. As far as we know, this is the first approach to parallel implementations of Stateflow charts.

Refinement strategy. We propose a refinement strategy that supports the verification of implementations of Stateflow charts with respect to our models. This strategy relies on the identified architecture to support a high degree of automation, and its soundness is derived from the soundness of the refinement laws used to define it. The motivation for a highly specialised verification strategies is that they are necessary to support a high

degree of automation in the verification task, which is a key requirement for the adoption of formal techniques by industry.

The strategy is structured in five phases: data refinement, normalisation, structuring, parallelism introduction and action introduction. The normalisation and action introduction phases are completely independent of the architecture of the implementations.

The structuring phase describes how to systematically collapse the parallelism between the process that models the semantics of Stateflow and the process that models the chart's structure. It also determines how to carry out simplifications in the nested structure of alternations that characterise the execution of the chart. This phase is key to the success of our approach to the description of graphical notations using an operational style, as it produces simpler, correct by construction, models of the chart by eliminating the unnecessary aspects of the semantics, which in other approaches are simplified in the translation strategy. While the portion of this phase that resolves the parallelism of the model is completely independent of the implementation, the portion that simplifies the structure of the model depends on the particular architectural patterns of the implementation.

The data refinement phase depends heavily on the state of the implementation. Simple restructuring of the state of implementations, however, can easily be reflected in this phase, provided the data patterns are equivalent. More sophisticated changes to the state of implementations can be incorporated provided the appropriate retrieve relation is defined.

The parallelism introduction phase relies on the client-server pattern used in our parallel implementations of Stateflow charts. In general, it can be adapted to different patterns of parallelism based on synchronisation. The use of explicit schedulers may require some changes to this phase, but we note that it has been previously treated in *Circus* for Simulink diagrams [14].

Our refinement strategy is extremely general in that it supports a wide range of Stateflow features as well as a number of architectural choices for the implementations. Furthermore, it can be extended to cover new architectural patterns and Stateflow features.

The failure of the refinement strategy indicates either that the implementation is incorrect or that it does not follow the associated architecture. In the latter case, the architectural patterns can be identified, and we can define a new strategy potentially reusing parts of the strategy proposed in this thesis. In the former, knowledge of the exact point of failure may help correct the program, as the phases and procedures of the refinement strategy target specific aspects of the implementation.

While the semantics of Stateflow is rather cumbersome in some points (e.g., local event broadcast), it is undeniably a notation that is actively used in industry. Most approaches to the verification of Stateflow tend to restrict the subset of the notation that is allowed, however, it is our belief that such a restriction should come from the users of the notation. If difficult aspects of the notation such as early return logic are used in industry, the issue of verification of this aspect is relevant and cannot be simply ignored if verification approaches are to have a chance of industrial adoption.

Our approach aims at minimising the need for expert guidance during the verification

process, this limits the robustness of our verification strategy, but increases the chances of industrial adoption. We believe that this is a reasonable trade-off.

6.2 Related Work

Our approach combines a direct semantic formulation of Stateflow charts, and a specialised refinement strategy to verify implementations. It partially follows the approach proposed for Simulink diagrams presented in [19, 17], but diverges mainly on the style of the proposed semantics and the complexity of the required refinement strategy. The technique used to define the semantics of Simulink diagrams in [19] does not translate well for the task of defining a semantics for Stateflow charts because the behaviour of Stateflow charts is not as compositional as that of Simulink diagrams.

Hamon and Rushby [35] propose an operational semantics of Stateflow, but it does not cover history junctions and imposes restrictions on transitions. Hamon [34] describes a denotational semantics heavily based on the notion of continuations. They claim to cover most of the notation, but Simulink functions and events of type function-call are not mentioned, and early return logic is not taken into consideration in the treatment of local event broadcast.

While we do not use continuations in our models, our treatment of transition backtracking and state execution is similar to that of [34]. For transition backtracking, we keep a record of the transitions that have been successfully executed, which upon failure allows us to recover the last executed transition and attempt the next transition. This behaviour is modelled in [34] by passing a continuation that executes the appropriate transition in case of failure. The same continuation scheme is used to execute outer and inner transitions, during actions and substates when executing a state. We use a value-result parameter to assess the success or failure of a transition (in causing a state transition), and decide how to proceed based on this parameter.

Banphawatthanarak et al. [7] restrict input signals to boolean values, and do not calculate output signals. It also imposes restrictions on the number of transitions reaching a junction, and the types of actions supported.

The work in [87] imposes a series of restrictions on the charts it treats. It does not allow multi-segment transition paths that represent loops. Variable assignment on transitions must be made solely on transition actions or the condition action of the last segment. It avoids backtracking of transitions by requiring that conditions of outgoing transitions from junctions form a cover, that is, the disjunction of the conditions is true. It also avoids relying on orderings determined by position of elements in the diagram by requiring that transitions leaving a node have disjunct conditions, for instance. Toyn and Galloway [104] impose even stronger restrictions; they do not cover parallel states, junctions, local variables, and so on.

In contrast to the above, in defining our models of Stateflow charts, we provide an extensive coverage of the notation. Our models cover edge-triggered input and output

events, local events, input and output data, entry, during, exit, condition, transition and on actions, parallel and sequential states, connective and history junctions, and transitions without imposing restrictions other that those already imposed by the Stateflow notation. The features that distinguish these models are the unrestricted treatment of states, junctions and transitions, and the thorough account of local event broadcasts and early return logic. Simulink and graphical functions are partially supported, but function-call events, temporal expressions, fixed-point and enumeration types are not supported.

There are few approaches to the verification of implementations of graphical notations. Arthan et al. [4] and Adams and Clayton [3] describe ClawZ, a tool for translating Simulink diagrams into Z [109] in order to formally verify implementations in Ada. This approach does not cover the Stateflow notation and can only deal with sequential implementations. To overcome the latter limitation, concurrent aspects were specified in CSP [85] and analysed through the model checker `FDR2` [30].

Cavalcanti and Clayton [17] define the semantics of control law diagrams in the *Circus* notation [79]. This semantics reuses ClawZ and the CSP approach to concurrency, and extends these works to cover a larger subset of the Simulink notation, but it still does not cover the Stateflow notation. The *Circus* model of Stateflow charts presented in this thesis is a natural extension of previous work, allowing for the verification of a broader variety of control law diagrams.

Cavalcanti et al. [14] proposes a refinement strategy that supports the verification of parallel implementations of Simulink diagrams. It builds upon the models proposed in [17], therefore it does not consider Stateflow blocks. Our work is a natural extension of [17, 14], but differs in the approach to the definition of the semantics of Stateflow, and, as a consequence of this, the requirements of the refinement strategy.

6.3 Future work

Due to the complexity of the semantics of Stateflow charts and the size of our models, the manual application of the refinement strategy to the smallest of charts is already an extremely time consuming task. We have rigourously applied the refinement calculus to verify a simple example, and selectively applied specific phases to simple examples. The complete verification of most Stateflow charts requires tool support that is currently unavailable, and the complete verification of industrial case studies requires the automation of the refinement strategy. Currently, even the task of simulating our models is not possible. Therefore, it is indispensable that better tool support is developed. In particular, simulation and refinement tools are of extreme importance to the approach proposed in this thesis.

Our models of Stateflow chart do not model the interaction between the chart and other Simulink blocks. Since a Stateflow chart is always defined in the context of a Simulink diagram, a model that integrates both notations is essential, but goes beyond the scope of this thesis. To solve this drawback, we must first refine the treatment of input events to

accept Simulink signals. Next, the treatment of type in the translation of Stateflow chart and Simulink diagrams must be harmonised, and the translation strategy for Simulink diagrams must be extended to include Stateflow blocks.

When considered in the context of a Simulink diagram, the current treatment of events is not sufficiently detailed. Our current treatment of events must be extended to distinguish different types of triggered events and to include function-call events. Different types of edge-triggered events can be supported by extending the definition of events to include type, and by refining the treatment of input events to accept Simulink signals.

Function-call events require some further modifications. The Simulink models must be extended to support enabled subsystems, and the step of execution of Simulink blocks needs to be revised to allow multiple executions of a block in the same step of the diagram. Furthermore, the models of Stateflow charts must also be updated to reflect these changes.

Our models do not treat bind actions, which in general specify scope properties that can be checked statically, but, in the special case of function-call events, may yield a behaviour similar to that of enabled subsystems. A complete treatment of bind actions must start by integrating the models of Simulink diagrams and Stateflow charts, and by extending both models to support function-call events.

Temporal expressions are not treated; they can be supported by refining the definition of events in order to record information about the number of times an event has been broadcast. Simulink and graphical functions can be fully supported by integrating our models to the models of Simulink diagrams: Simulink functions are Simulink diagrams, and graphical functions are Stateflow charts containing only junctions and transitions.

While the operational style adopted proved suitable for most of the semantics of Stateflow charts, the difficulties faced in modelling local event broadcasts and early return logic raise the question of whether a denotational approach would be better suited. As future work, we would like to further develop the denotational model that inspired this work [16] and compare the treatment of local event broadcast and early return logic in the two models.

The current refinement strategy targets parallel implementations that follow a fairly restrictive architecture. In particular, only parallel states that do not share variables can be run in parallel. As future work, we would like to relax some of the restrictions and define new refinement strategies to support the verification of a wider variety of implementations. Furthermore, once an integration of the models of Stateflow charts and Simulink diagrams exists, a natural development is the integration of the available refinement strategies to support the verification of Simulink diagrams that contain Stateflow charts.

Currently, our refinement strategy must be thoroughly validated with large industrial case-studies. As previously discussed, two industrial case studies were used to validate the translation strategy, but they are too large to be manually verified using our refinement strategy. In order to support semi-automatic verification, we must first formalise the strategy in a tactic language for refinement, such as ArcAngel [77], and then use a refinement tool like `CRefine` [80] to verify larger examples. However, in its current state `CRefine`

does not fully support the application of a refinement strategy such as the one proposed in this thesis. Thus, as previously mentioned, a more robust refinement tool needs to be developed to fully support our verification approach.

Finally, since the refinement strategy derives its soundness from the refinement laws used, providing mechanised proofs for all the refinement laws is a requirement for any practical use of the strategy. Since *Circus* has theorem proving support [111], these law can be formalised in *ProofPowerZ* and verified. This task, however, is not simple, and requires deep understanding of the semantics of *Circus* and of the theorem prover.

It is still the case that the use of formal methods in industry is limited, and in order to change this, both communities must bridge the gap between academic and industrial practice. Our work contributes to this goal by providing a formal treatment of the problem of correctness of software in the context of a graphical notation widely used in industry.

Appendix A

Syntax of *Circus*

The syntax presented in this appendix is based on that published in [76].

$\mathit{Program}$	$::=$	$\mathit{CircusPar}*$
$\mathit{CircusPar}$	$::=$	$\mathit{Par} \mid \textbf{channel}\ \mathit{CDecl} \mid \textbf{channelset}\ N == \mathit{CSExp} \mid \mathit{ProcDecl}$
CDecl	$::=$	$\mathit{SimpleCDecl} \mid \mathit{SimpleCDecl};\ \mathit{CDecl}$
$\mathit{SimpleCDecl}$	$::=$	$N^+ \mid N^+ : \mathit{Exp} \mid [N^+]N^+ : \mathit{Exp} \mid \mathit{SchemaExp}$
CSExp	$::=$	$\{\!\lvert\ \rvert\!\} \mid \{\!\lvert N^+ \rvert\!\} \mid N \mid \mathit{CSExp} \cup \mathit{CSExp} \mid \mathit{CSExp} \cap \mathit{CSExp} \mid \mathit{CSExp} \setminus \mathit{CSExp}$
$\mathit{ProcDecl}$	$::=$	$\textbf{process}\ N \mathrel{\widehat=} \mathit{ProcDef} \mid \textbf{process}\ N[N^+] \mathrel{\widehat=} \mathit{ProcDef}$
$\mathit{ProcDef}$	$::=$	$\mathit{Decl} \bullet \mathit{ProcDef} \mid \mathit{Decl} \odot \mathit{ProcDef} \mid \mathit{Proc}$
Proc	$::=$	$\textbf{begin}\ \mathit{PPar} * \textbf{state}\ \mathit{SchemaExpPPar}* \bullet \mathit{Action}\ \textbf{end}$
		$\mid \mathit{Proc}\ ;\ \mathit{Proc} \mid \mathit{Proc} \mathbin{\Box} \mathit{Proc} \mid \mathit{Proc} \sqcap \mathit{Proc} \mid \mathit{Proc} \llbracket \mathit{CSExpr} \rrbracket \mathit{Proc} \mid$
		$\mid \mathit{Proc} \interleave \mathit{Proc} \mid \mathit{Proc} \setminus \mathit{CSExpr} \mid (\mathit{Decl} \bullet \mathit{ProcDef})(\mathit{Exp}^+) \mid N(\mathit{Exp}^+) \mid N$
		$\mid (\mathit{Decl} \odot \mathit{ProcDef})\lfloor \mathit{Exp}^+ \rfloor \mid N\lfloor \mathit{Exp}^+ \rfloor \mid \mathit{Proc}[N^+ := N^+] \mid N[\mathit{Exp}^+]$
		$\mid \mathop{;}\ \mathit{Decl} \bullet \mathit{Proc} \mid \mathop{\Box} \mathit{Decl} \bullet \mathit{Proc} \mid \mathop{\sqcap} \mathit{Decl} \bullet \mathit{Proc} \mid \llbracket \mathit{CSExpr} \rrbracket\ \mathit{Decl} \bullet \mathit{Proc}$
		$\mid \mathop{\interleave} \mathit{Dec} \bullet \mathit{Proc}$
NSExp	$::=$	$\{\} \mid \{N^+\} \mid N \mid \mathit{NSExp} \cup \mathit{NSExp} \mid \mathit{NSExp} \cap \mathit{NSExp} \mid \mathit{NSExp} \setminus \mathit{NSExp}$
PPar	$::=$	$\mathit{Par} \mid N \mathrel{\widehat=} \mathit{ParAction} \mid \textbf{nameset}\ N == \mathit{NSExp}$
$\mathit{ParAction}$	$::=$	$\mathit{Action} \mid \mathit{Decl} \bullet \mathit{ParAction}$
Action	$::=$	$(\mathit{SchemaExp}) \mid \mathit{Command} \mid N \mid \mathit{CSPAction} \mid \mathit{Action}[N^+ := N^+]$

$$
\begin{array}{lll}
\mathit{CSPAction} & ::= & \mathbf{Skip} \mid \mathbf{Stop} \mid \mathbf{Chaos} \mid \mathit{Comm} \longrightarrow \mathit{Action} \mid (\mathit{Pred}) \mathbin{\&} \mathit{Action} \\
 & \mid & \mathit{Action} \mathbin{;} \mathit{Action} \mid \mathit{Action} \mathbin{\Box} \mathit{Action} \mid \mathit{Action} \sqcap \mathit{Action} \\
 & \mid & \mathit{Action} \llbracket \mathit{NSExp} \mid \mathit{CSExp} \mid \mathit{NSExp} \rrbracket \mathit{Action} \\
 & \mid & \mathit{Action} \mathbin{|\!|\![} \mathit{NSExp} \mid \mathit{NSExpr} \mathbin{]\!|\!|} \mathit{Action} \mid \mathit{Action} \setminus \mathit{CSExp} \mid \mathit{ParAction}(\mathit{Exp}^{+}) \\
 & \mid & \mu\, N^{+} \bullet \mathit{Action} \mid \mathop{;} \mathit{Decl} \bullet \mathit{Action} \mid \mathop{\Box} \mathit{Decl} \bullet \mathit{Action} \mid \bigsqcap \mathit{Decl} \bullet \mathit{Action} \\
 & \mid & \llbracket\, \mathit{CSExp} \,\rrbracket\, \mathit{Decl} \bullet \llbracket \mathit{NSExp} \rrbracket\, \mathit{Action} \mid \mathop{|\!|\!|} \mathit{Decl} \bullet |\!|\![\mathit{NSExp}\,]\!|\!|\, \mathit{Action} \\
\mathit{Comm} & ::= & N \mathit{CParameter}* \mid N\,[\mathit{Exp}^{+}]\,\mathit{CParameter}* \\
\mathit{CParameter} & ::= & ?N \mid ?N : (\mathit{Pred}) \mid !\mathit{Exp} \mid .\mathit{Exp} \\
\mathit{Command} & ::= & N^{+} := \mathit{Exp}^{+} \mid \mathbf{if}\ \mathit{GActions}\ \mathbf{fi} \mid \mathbf{var}\ \mathit{Decl} \bullet \mathit{Action} \mid N^{+} : [\mathit{Pred}, \mathit{Pred}] \\
 & \mid & \{\mathit{Pred}\} \mid [\mathit{Pred}] \mid \mathbf{val}\ \mathit{Decl} \bullet \mathit{Action} \mid \mathbf{res}\ \mathit{Decl} \bullet \mathit{Action} \\
 & \mid & \mathbf{vres}\ \mathit{Decl} \bullet \mathit{Action} \\
\mathit{GActions} & ::= & \mathit{Pred} \longrightarrow \mathit{Action} \mid \mathit{Pred} \longrightarrow \mathit{Action} \mathbin{[\!]} \mathit{GActions}
\end{array}
$$

Appendix B

Circus model of Stateflow semantics

B.1 Basic definitions

section *basic_toolkit* **parents** *circus_toolkit*

$[NAME]$

$\mathbb{R} : \mathbb{P}\,\mathbb{A}$

$r : \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{R}$

generic(opt _)

$\operatorname{opt} X == \{s : \operatorname{seq} X \mid \# s \leq 1\}$

$b2r : \mathbb{B} \rightarrow \mathbb{R}$

$\forall v : \mathbb{B} \bullet b2r(v) = (\mathbf{if}\ v = \mathbf{True\ then}\ 1\ \mathbf{else}\ 0)$

$truthvalue : \mathbb{A} \rightarrow \mathbb{B}$

$\forall x : \mathbb{A} \mid x > 0 \bullet truthvalue(x) = \mathbf{True}$

$\forall x : \mathbb{A} \mid x \leq 0 \bullet truthvalue(x) = \mathbf{False}$

B.2 Stateflow semantics

section *stateflow_toolkit* **parents** *circus_toolkit*, *basic_toolkit*

function 40 **leftassoc** $(_ \land_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ \lor_{\mathcal{A}} _)$
function$(\neg_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ \land_{\mathcal{B}} _)$
function 40 **leftassoc** $(_ \lor_{\mathcal{B}} _)$
function 40 **leftassoc** $(_ \oplus_{\mathcal{B}} _)$
function$(\neg_{\mathcal{B}} _)$
function 40 **leftassoc** $(_ << _)$
function 40 **leftassoc** $(_ >> _)$
function 40 **leftassoc** $(_ <_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ >_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ \leq_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ \geq_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ =_{\mathcal{A}} _)$
function 40 **leftassoc** $(_ \neq_{\mathcal{A}} _)$

$$
\begin{array}{l}
_ \land_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ \lor_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
\neg_{\mathcal{A}} _ : \mathbb{A} \to \mathbb{A} \\
_ \land_{\mathcal{B}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ \lor_{\mathcal{B}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ \oplus_{\mathcal{B}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
\neg_{\mathcal{B}} _ : \mathbb{A} \to \mathbb{A} \\
_ << _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ >> _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ <_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ >_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ \leq_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ \geq_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ =_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
_ \neq_{\mathcal{A}} _ : \mathbb{A} \times \mathbb{A} \to \mathbb{A} \\
\hline
\forall x, y : \mathbb{A} \bullet x \land_{\mathcal{A}} y = \neg_{\mathcal{A}}(\neg_{\mathcal{A}} x \lor_{\mathcal{A}} \neg_{\mathcal{A}} y) \\
\forall x, y : \mathbb{A} \bullet x \land_{\mathcal{B}} y = \neg_{\mathcal{B}}(\neg_{\mathcal{B}} x \lor_{\mathcal{B}} \neg_{\mathcal{B}} y) \\
\forall x, y : \mathbb{A} \bullet x \oplus_{\mathcal{B}} y = (x \land_{\mathcal{B}} \neg_{\mathcal{B}} y) \lor_{\mathcal{B}} (\neg_{\mathcal{B}} x \land_{\mathcal{B}} y)
\end{array}
$$

$DECOMPOSITION ::= SET \mid CLUSTER$
$TYPE ::= AND \mid OR \mid CHART$
$DESTINATION ::= STATE \mid JUNCTION$
$[SID, TID, JID, EVENT]$
$NID ::= snode\langle\langle SID\rangle\rangle \mid jnode\langle\langle JID\rangle\rangle$
$SFBOOL == \mathbb{Z}$

State

$identifier : SID$
$default, inner, outer : TID$
$parent, left, right : SID$
$substates : \mathrm{seq}\, SID$
$decomposition : DECOMPOSITION$
$type : TYPE$
$history : \mathbb{B}$

Junction

$identifier : JID$
$transition : TID$
$parent : SID$
$history : \mathbb{B}$

Transition

$identifier : TID$
$source, destination : NID$
$next : TID$
$parent : SID$

$nullstate : State$
$nulljunction : Junction$
$nulltransition : Transition$

$nullstate.history = \mathbf{False}$
$nulljunction.history = \mathbf{False}$

StateflowChart

$identifier : SID$
$states : SID \pinj State$
$transitions : TID \pinj Transition$
$junctions : JID \pinj Junction$

$nullstate \notin \operatorname{ran} states$
$nulltransition \notin \operatorname{ran} transitions$
$nulljunction \notin \operatorname{ran} junctions$
$\#\{s : \operatorname{ran} states \mid s.type = CHART\} = 1$
$(states(identifier)).type = CHART$
$\forall n : SID \mid n \in \operatorname{dom} states \bullet (states(n)).identifier = n$
$\forall n : JID \mid n \in \operatorname{dom} junctions \bullet (junctions(n)).identifier = n$
$\forall n : TID \mid n \in \operatorname{dom} transitions \bullet (transitions(n)).identifier = n$

$parent : State \leftrightarrow State$

$\forall s_1, s_2 : State \bullet parent(s_1) = s_2 \Leftrightarrow s_1.parent = s_2.identifier$

$ancestors : State \rightarrow \mathbb{P}\, State$

$\forall s : State \bullet ancestors(s) = (parent^{+}) \llparenthesis \{s\} \rrparenthesis \setminus \{nullstate\}$

$la : (State \times State) \rightarrow State$

$\forall s_1, s_2 : State \bullet la(s_1, s_2) = \mu\, x : (ancestors(s_1) \cap ancestors(s_2)) \mid$
$\quad (\forall y : (ancestors(s_1) \cap ancestors(s_2)) \bullet x = y \lor y \in ancestors(x)) \bullet x$

channel $read_inputs, write_outputs$
channel $input_event : \operatorname{seq} \mathbb{B}$
channel $events : \operatorname{seq} EVENT$
channel $local_event : EVENT \times State$
channel $executeentryaction, executeexitaction : SID$
channel $executeduringaction : SID \times EVENT$
channel $executeconditionaction, executetransitionaction : TID$
channel $evaluatecondition : TID \times \mathbb{B}$
channel $checktrigger : TID \times EVENT$
channel $result : TID \times EVENT \times \mathbb{B}$

$\textbf{channel}\ activate, deactivate : SID$
$\textbf{channel}\ junction : JID \times Junction$
$\textbf{channel}\ transition : TID \times Transition$
$\textbf{channel}\ state : SID \times State$
$\textbf{channel}\ chart : State$
$\textbf{channel}\ status : SID \times \mathbb{B}$
$\textbf{channel}\ history : SID \times SID$
$\textbf{channel}\ end_local_execution, end_action$
$\textbf{channel}\ interrupt : \mathbb{B}$
$\textbf{channel}\ end_cycle$
$\textbf{channelset}\ interface == \{\!| executeentryaction, executeduringaction, executeexitaction, executeconditionaction, executetransitionaction, evaluatecondition, checktrigger, result, junction, transition, state, chart, status, history, activate, deactivate, read_inputs, write_outputs, end_local_execution, end_action, local_event, interrupt, events |\!\}$

$$and, or : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$
$$not : \mathbb{B} \to \mathbb{B}$$

$$or(\textbf{False}, \textbf{False}) = and(\textbf{False}, \textbf{False}) = and(\textbf{True}, \textbf{False}) = and(\textbf{False}, \textbf{True}) = \textbf{False}$$
$$and(\textbf{True}, \textbf{True}) = or(\textbf{True}, \textbf{True}) = or(\textbf{True}, \textbf{False}) = or(\textbf{False}, \textbf{True}) = \textbf{True}$$
$$not(\textbf{True}) = \textbf{False}$$
$$not(\textbf{False}) = \textbf{True}$$

$\textbf{process}\ Simulator \mathrel{\widehat=} \textbf{begin}$

$entryActionCheck \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{vres}\ b : \mathbb{B} \bullet status!sid?active \longrightarrow b := not(active)$
$duringActionCheck \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{vres}\ b : \mathbb{B} \bullet entryActionCheck(sid, b)$
$exitActionCheck \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{vres}\ b : \mathbb{B} \bullet entryActionCheck(sid, b)$
$conditionActionCheck \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{vres}\ b : \mathbb{B} \bullet entryActionCheck(sid, b)$
$transitionActionCheck \mathrel{\widehat=} \textbf{val}\ sid : SID;\ \textbf{vres}\ b : \mathbb{B} \bullet$

$$entryActionCheck(sid, b)\ ;\ \textbf{var}\ ss : \mathrm{seq}\, SID \bullet \left(\begin{array}{l} state!sid?s \longrightarrow ss := s.substates; \\ \mu X \bullet \left(\begin{array}{l} \textbf{if}\ ss = \langle\rangle \longrightarrow \textbf{Skip} \\ [\!]\ ss \neq \langle\rangle \longrightarrow \left(\begin{array}{l} status!(head\ ss)?active \longrightarrow b := or(b, active); \\ ss := tail\ ss\ ;\ X \end{array}\right) \\ \textbf{fi} \end{array}\right) \end{array}\right)$$

$exitStatesCheck \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet transitionActionCheck(sid, b)$

$enterState1Check \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet$
$\quad state!sid?s \longrightarrow entryActionCheck(s.parent, b)$

$enterState2Check \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet state!sid?s \longrightarrow$

$$\left(\begin{array}{l} \mathbf{if}\ s.left = nullstate.identifier \longrightarrow b := \mathbf{False} \\ [\!]\ s.left \neq nullstate.identifier \longrightarrow entryActionCheck(s.left, b) \\ \mathbf{fi} \end{array}\right)$$

$executeStateCheck \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet$
$\quad state!sid?s \longrightarrow entryActionCheck(s.parent, b)$

$exitStateCheck \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet state!sid?s \longrightarrow entryActionCheck(s.parent, b)$

$enterState15Check \mathrel{\widehat=} \mathbf{val}\, sid : SID;\ \mathbf{vres}\, b : \mathbb{B} \bullet entryActionCheck(sid, b)$

$ExecuteTransition \mathrel{\widehat=}$
$\mathbf{val}\, tid : TID;\ \mathbf{val}\, path : \mathrm{seq}\, TID;\ \mathbf{val}\, source : State;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet$

$$\left(\begin{array}{l} \mathbf{if}\ tid = nulltransition.identifier \longrightarrow \\ \left(\begin{array}{l} \mathbf{if}\ path = \varnothing \longrightarrow success := \mathbf{False} \\ [\!]\ path \neq \varnothing \longrightarrow \left(\begin{array}{l} transition!(last\ path)?lt \longrightarrow \\ ExecuteTransition(lt.next, (front\ path), source, ce, success) \end{array}\right) \\ \mathbf{fi} \end{array}\right) \\ [\!]\ tid \neq nulltransition.identifier \longrightarrow \\ CheckValidity(tid, path, source, ce, success) \\ \mathbf{fi} \end{array}\right)$$

$CheckValidity \mathrel{\widehat=}$
$\mathbf{val}\, tid : TID;\ \mathbf{val}\, path : \mathrm{seq}\, TID;\ \mathbf{val}\, source : State;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet$
$checktrigger!tid!ce \longrightarrow result!tid!ce?e \longrightarrow evaluatecondition!tid?c \longrightarrow$

$$\left(\begin{array}{l} \mathbf{if}\ e = \mathbf{True} \land c = \mathbf{True} \longrightarrow \\ \qquad \left(\begin{array}{l} executeconditionaction!tid \longrightarrow LocalEventCondition(source.identifier); \\ \mathbf{var}\ b : \mathbb{B} \bullet \\ \left(\begin{array}{l} conditionActionCheck(source.identifier, b); \\ \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow success := \mathbf{True} \\ [\!]\ b = \mathbf{False} \longrightarrow Proceed(tid, path \frown \langle tid \rangle, source, ce, success) \\ \mathbf{fi} \end{array}\right) \end{array}\right) \end{array}\right) \\ [\!]\ \neg\,(e = \mathbf{True} \land c = \mathbf{True}) \longrightarrow \left(\begin{array}{l} transition!tid?t \longrightarrow \\ ExecuteTransition(t.next, path, source, ce, success) \end{array}\right) \\ \mathbf{fi} \end{array}\right)$$

$$
\begin{array}{l}
Proceed \mathrel{\widehat=} \\
\mathbf{val}\, tid : TID;\ \mathbf{val}\, path : \operatorname{seq} TID;\ \mathbf{val}\, source : State;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet \\
transition!tid?t \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ t.destination \in \operatorname{ran} snode \longrightarrow \left(\begin{array}{l} state!((snode^{\sim})\, t.destination)?dest \longrightarrow \\ proceedToState(source, dest, path, ce, success) \end{array}\right) \\
[]\ t.destination \in \operatorname{ran} jnode \longrightarrow proceedToJunction(tid, path, source, ce, success) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
proceedToState \mathrel{\widehat=} \\
\mathbf{val}\, src, dest : State;\ \mathbf{val}\, path : \operatorname{seq} TID;\ \mathbf{val}\, ce : EVENT;\ \mathbf{vres}\, success : \mathbb{B} \bullet \\
\left(\begin{array}{l}
ExitStates((la(src, dest)).substates, ce); \\
\mathbf{var}\, b : \mathbb{B} \bullet \\
\left(\begin{array}{l}
exitStatesCheck((la(src, dest)).identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[]\ b = \mathbf{False} \longrightarrow \\
\qquad \left(\begin{array}{l}
executePath(path, src, dest, ce); \\
transitionActionCheck((la(src, dest)).identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[]\ b = \mathbf{False} \longrightarrow EnterState(dest, la(src, dest), ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right); \\
success := \mathbf{True}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
executePath \mathrel{\widehat=} path : \operatorname{seq} TID;\ src, dest : State;\ ce : EVENT \bullet \\
\left(\begin{array}{l}
\mathbf{if}\ \#\, path = 0 \longrightarrow \mathbf{Skip} \\
[]\ \#\, path > 0 \longrightarrow \left(\begin{array}{l}
executetransitionaction!(head\ path) \longrightarrow \\
LocalEventTransition((la(src, dest)).identifier); \\
\mathbf{var}\, b : \mathbb{B} \bullet \\
\left(\begin{array}{l}
transitionActionCheck((la(src, dest)).identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[]\ b = \mathbf{False} \longrightarrow executePath(tail\ path, src, dest, ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
proceedToJunction \mathrel{\widehat=} \\
\mathbf{val}\ tid : TID;\ \mathbf{val}\ path : \operatorname{seq} TID;\ \mathbf{val}\ source : State;\ \mathbf{val}\ ce : EVENT;\ \mathbf{vres}\ success : \mathbb{B} \bullet \\
\quad transition!tid?t \longrightarrow junction!((jnode^{\sim})\ t.destination)?dj \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ dj.history = \mathbf{False} \longrightarrow executeJunction(dj, path, source, ce, success) \\
[\!]\ dj.history = \mathbf{True} \longrightarrow history!(dj.parent)?lsid \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ lsid = nullstate.identifier \longrightarrow \\
\quad \left(\begin{array}{l}
state!(dj.parent)?s \longrightarrow \\
ExecuteDefaultTransition(s, s, ce)\ ;\ success := \mathbf{True}
\end{array}\right) \\
[\!]\ lsid \neq nullstate.identifier \longrightarrow \\
\quad \left(\begin{array}{l}
state!lsid?ls \longrightarrow \\
proceedToState(source, ls, path, ce, success)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
executeJunction \mathrel{\widehat=} \mathbf{val}\ j : Junction;\ \mathbf{val}\ path : \operatorname{seq} TID; \\
\mathbf{val}\ source : State;\ \mathbf{val}\ ce : EVENT;\ \mathbf{vres}\ success : \mathbb{B} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{if}\ j.transition = nulltransition.identifier \longrightarrow \\
\quad success := \mathbf{False} \\
[\!]\ j.transition \neq nulltransition.identifier \longrightarrow \\
\quad ExecuteTransition(j.transition, path, source, ce, success) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecuteDefaultTransition \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet \\
\quad \left(\begin{array}{l}
\mathbf{if}\ s.default \neq nulltransition.identifier \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{var}\ success : \mathbb{B} \bullet ExecuteTransition(s.default, \langle\rangle, s, ce, success); \\
(\mathbf{if}\ success = \mathbf{True} \longrightarrow \mathbf{Skip}\ [\!]\ success = \mathbf{False} \longrightarrow \mathbf{Stop}\ \mathbf{fi})
\end{array}\right) \\
[\!]\ s.default = nulltransition.identifier \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ \#\, s.substates = 0 \longrightarrow \mathbf{Skip} \\
[\!]\ \#\, s.substates = 1 \longrightarrow \left(\begin{array}{l}
state!(head\ s.substates)?saux \longrightarrow \\
EnterState(saux, tpp, ce)
\end{array}\right) \\
[\!]\ \#\, s.substates > 1 \longrightarrow \mathbf{Stop} \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$EnterState \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState16(s, tpp, ce)$$

$$\begin{array}{l} EnterState24 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState2(s, tpp, ce); \\ \quad \mathbf{var}\ b : \mathbb{B} \bullet enterState2Check(s.identifier, b); \\ \quad \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ b = \mathbf{False} \longrightarrow EnterState34(s, tpp, ce) \\ \mathbf{fi} \end{array}\right) \end{array}$$

$$\begin{array}{l} EnterState14 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState1(s, tpp, ce); \\ \quad \mathbf{var}\ b : \mathbb{B} \bullet enterState1Check(s.identifier, b); \\ \quad \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ b = \mathbf{False} \longrightarrow EnterState24(s, tpp, ce) \\ \mathbf{fi} \end{array}\right) \end{array}$$

$$\begin{array}{l} EnterState35 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState34(s, tpp, ce); \\ \quad status!(s.identifier)?active \longrightarrow \left(\begin{array}{l} \mathbf{if}\ active = \mathbf{True} \longrightarrow EnterState5(s, tpp, ce) \\ [\!]\ active = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \end{array}$$

$$\begin{array}{l} EnterState25 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState2(s, tpp, ce); \\ \quad \mathbf{var}\ b : \mathbb{B} \bullet enterState2Check(s.identifier, b); \\ \quad \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ b = \mathbf{False} \longrightarrow EnterState35(s, tpp, ce) \\ \mathbf{fi} \end{array}\right) \end{array}$$

$$\begin{array}{l} EnterState15 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState1(s, tpp, ce); \\ \quad \mathbf{var}\ b : \mathbb{B} \bullet enterState1Check(s.identifier, b); \\ \quad \left(\begin{array}{l} \mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ b = \mathbf{False} \longrightarrow EnterState25(s, tpp, ce) \\ \mathbf{fi} \end{array}\right) \end{array}$$

$$\begin{array}{l} EnterState56 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState5(s, tpp, ce); \\ \quad status!(s.identifier)?active \longrightarrow \left(\begin{array}{l} \mathbf{if}\ active = \mathbf{True} \longrightarrow EnterState6(s, tpp, ce) \\ [\!]\ active = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \end{array}$$

$$EnterState36 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState34(s, tpp, ce);$$
$$status!(s.identifier)?active \longrightarrow \left(\begin{array}{l}\mathbf{if}\ active = \mathbf{True} \longrightarrow EnterState56(s, tpp, ce)\\ [\!]\ active = \mathbf{False} \longrightarrow \mathbf{Skip}\\ \mathbf{fi}\end{array}\right)$$

$$EnterState26 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState2(s, tpp, ce);$$
$$\mathbf{var}\ b : \mathbb{B} \bullet enterState2Check(s.identifier, b);$$
$$\left(\begin{array}{l}\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip}\\ [\!]\ b = \mathbf{False} \longrightarrow EnterState36(s, tpp, ce)\\ \mathbf{fi}\end{array}\right)$$

$$EnterState16 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet EnterState1(s, tpp, ce);$$
$$\mathbf{var}\ b : \mathbb{B} \bullet enterState1Check(s.identifier, b);$$
$$\left(\begin{array}{l}\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip}\\ [\!]\ b = \mathbf{False} \longrightarrow EnterState26(s, tpp, ce)\\ \mathbf{fi}\end{array}\right)$$

$$EnterState1 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet status!(s.parent)?active \longrightarrow$$
$$\left(\begin{array}{l}\mathbf{if}\ active = \mathbf{False} \longrightarrow\\ \qquad state!(s.parent)?p \longrightarrow EnterState14(p, tpp, ce)\\ [\!]\ active = \mathbf{True} \longrightarrow \mathbf{Skip}\\ \mathbf{fi}\end{array}\right)$$

$$EnterState2 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l}\mathbf{if}\ s.type = AND \longrightarrow\\ \left(\begin{array}{l}\mathbf{if}\ s.left \neq nullstate.identifier \longrightarrow status!(s.left)?active \longrightarrow\\ \left(\begin{array}{l}\mathbf{if}\ active = \mathbf{True} \longrightarrow \mathbf{Skip}\\ [\!]\ active \neq \mathbf{True} \longrightarrow state!(s.left)?ls \longrightarrow EnterState15(ls, tpp, ce)\\ \mathbf{fi}\end{array}\right)\\ [\!]\ s.left = nullstate.identifier \longrightarrow \mathbf{Skip}\\ \mathbf{fi}\end{array}\right)\\ [\!]\ s.type \neq AND \longrightarrow \mathbf{Skip}\\ \mathbf{fi}\end{array}\right)$$

$$EnterState3 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet activate!(s.identifier) \longrightarrow \mathbf{Skip}$$

$$EnterState4 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\quad executeentryaction!(s.identifier) \longrightarrow LocalEventEntry(s.identifier)$$

$$EnterState34 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet status.(s.identifier)?active \longrightarrow$$
$$\left(\begin{array}{l} \textbf{if } active = \textbf{True} \longrightarrow \textbf{Skip} \\ [\!]\ active = \textbf{False} \longrightarrow EnterState3(s, tpp, ce)\ ;\ EnterState4(s, tpp, ce) \\ \textbf{fi} \end{array}\right)$$

$$EnterState5 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\quad EnterState5a(s, tpp, ce)\ ;\ EnterState5b(s, tpp, ce)$$

$$EnterState5a \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l} \textbf{if } s.history = \textbf{True} \longrightarrow history!(s.identifier)?lsid \longrightarrow \\ \qquad \left(\begin{array}{l} \textbf{if } lsid \neq nullstate.identifier \longrightarrow state!lsid?ls \longrightarrow EnterState15(ls, tpp, ce) \\ [\!]\ lsid = nullstate.identifier \longrightarrow ExecuteDefaultTransition(s, tpp, ce) \\ \textbf{fi} \end{array}\right) \\ [\!]\ s.history = \textbf{False} \longrightarrow \\ \qquad \left(\begin{array}{l} \textbf{if } s.default \neq nulltransition.identifier \longrightarrow ExecuteDefaultTransition(s, tpp, ce) \\ [\!] s.default = nulltransition.identifier \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right) \\ \textbf{fi} \end{array}\right)$$

$$EnterState5b \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l} \textbf{if } s.decomposition = SET \land s.default = nulltransition.identifier \longrightarrow \\ \qquad EnterStates15(s.substates, tpp, ce) \\ [\!]\ s.decomposition \neq SET \lor s.default \neq nulltransition.identifier \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array}\right)$$

$$EnterStates15 \mathrel{\widehat=} ss : \operatorname{seq} SID;\ tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l}
\mathbf{if}\ \#\, ss = 0 \longrightarrow \mathbf{Skip} \\
[\!]\ \ \#\, ss > 0 \longrightarrow status!(head\ ss)?active \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{False} \longrightarrow state!(head\ ss)?first \longrightarrow EnterState15(first, tpp, ce) \\
[\!]\ active = \mathbf{True} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l}
enterState15Check(head\ ss, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow EnterStates15(tail\ ss, tpp, ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)$$

$$EnterState6 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l}
\mathbf{if}\ s.type = AND \land s.right \neq nullstate.identifier \longrightarrow \\
\qquad state!(s.right)?rs \longrightarrow EnterState(rs, tpp, ce) \\
[\!]\ s.type \neq AND \lor s.right = nullstate.identifier \longrightarrow EnterState7(s, tpp, ce) \\
\mathbf{fi}
\end{array}\right)$$

$$EnterState7 \mathrel{\widehat=} s, tpp : State;\ ce : EVENT \bullet$$
$$\left(\begin{array}{l}
\mathbf{if}\ s.type \neq CHART \longrightarrow \\
\qquad state!(s.parent)?p \longrightarrow \left(\begin{array}{l}
\mathbf{if}\ tpp \neq p \longrightarrow EnterState6(p, tpp, ce) \\
[\!]\ tpp = p \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ s.type = CHART \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)$$

$$
\begin{array}{l}
ExecuteState \mathrel{\widehat=} s : State;\ ce : EVENT \bullet status!(s.identifier)?active \longrightarrow \\
\left(\begin{array}{l}
\textbf{if } active = \textbf{True} \longrightarrow \\
\left(\begin{array}{l}
\textbf{var } success : \mathbb{B} \bullet \\
ExecuteTransition(s.outer, \langle\rangle, s, ce, success); \\
\left(\begin{array}{l}
\textbf{if } success = \textbf{True} \longrightarrow \textbf{Skip} \\
[]\ success = \textbf{False} \longrightarrow \\
\qquad \left(\begin{array}{l}
executeduringaction!(s.identifier)!ce \longrightarrow \\
LocalEventDuring(s.identifier); \\
\textbf{var } b : \mathbb{B} \bullet \\
\left(\begin{array}{l}
duringActionCheck(s.identifier, b); \\
\left(\begin{array}{l}
\textbf{if } b = \textbf{True} \longrightarrow \textbf{Skip} \\
[]\ b = \textbf{False} \longrightarrow AlternativeExecution(s, ce) \\
\textbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\textbf{fi}
\end{array}\right)
\end{array}\right) \\
[]\, active = \textbf{False} \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
AlternativeExecution \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \\
\quad \left(\begin{array}{l}
\textbf{var } success : \mathbb{B} \bullet ExecuteTransition(s.inner, \langle\rangle, s, ce, success); \\
\left(\begin{array}{l}
\textbf{if } success = \textbf{True} \longrightarrow \textbf{Skip} \\
[]\, success = \textbf{False} \longrightarrow ExecuteSubstates(s, ce) \\
\textbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecuteSubstates \mathrel{\widehat=} s : State;\ ce : EVENT \bullet \\
\quad \left(\begin{array}{l}
\textbf{if } s.decomposition = SET \longrightarrow ExecuteParallelStates(s.substates, ce) \\
[]\ s.decomposition = CLUSTER \longrightarrow ExecuteSequentialStates(s.substates, ce) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecuteParallelStates \mathrel{\widehat=} ss : \operatorname{seq} SID;\ ce : EVENT \bullet \\
\quad \left(\begin{array}{l}
\textbf{if } \# ss = 0 \longrightarrow \textbf{Skip} \\
[]\ \# ss > 0 \longrightarrow state!(head\ ss)?first \longrightarrow ExecuteState(first, ce); \\
\textbf{var } b : \mathbb{B} \bullet \left(\begin{array}{l}
executeStateCheck(head\ ss, b); \\
\left(\begin{array}{l}
\textbf{if } b = \textbf{True} \longrightarrow \textbf{Skip} \\
[]\ b = \textbf{False} \longrightarrow ExecuteParallelStates(tail\ ss, ce) \\
\textbf{fi}
\end{array}\right)
\end{array}\right) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExecuteSequentialStates} \mathrel{\widehat=} ss : \operatorname{seq} \mathit{SID};\ ce : \mathit{EVENT} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{if}\ \#\, ss = 0 \longrightarrow \mathbf{Skip} \\
[\!]\ \#\, ss > 0 \longrightarrow status!(head\ ss)?active \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{True} \longrightarrow state!(head\ ss)?first \longrightarrow \mathit{ExecuteState}(first, ce) \\
[\!]\ active = \mathbf{False} \longrightarrow \mathit{ExecuteSequentialStates}(tail\ ss, ce) \\
\mathbf{fi}
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExitState} \mathrel{\widehat=} s : \mathit{State};\ ce : \mathit{EVENT} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{if}\ s.right \neq nullstate.identifier \longrightarrow status!(s.right)?active \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{True} \longrightarrow state!(s.right)?rs \longrightarrow \mathit{ExitState}(rs, ce) \\
[\!]\ active = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\ s.right = nullstate.identifier \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\mathit{ExitStates}(s.substates, ce); \\
executeexitaction!(s.identifier) \longrightarrow \mathit{LocalEventExit}(s.identifier); \\
\mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l}
exitActionCheck(s.identifier, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow deactivate!(s.identifier) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExitStates} \mathrel{\widehat=} ss : \operatorname{seq} \mathit{SID};\ ce : \mathit{EVENT} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{if}\ \#\, ss = 0 \longrightarrow \mathbf{Skip} \\
[\!]\ \#\, ss > 0 \longrightarrow status!(last\ ss)?active \longrightarrow \\
\qquad \left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{True} \longrightarrow state!(last\ ss)?l \longrightarrow \mathit{ExitState}(l, ce) \\
[\!]\ active = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\mathbf{var}\ b : \mathbb{B} \bullet \left(\begin{array}{l}
exitStateCheck(head\ ss, b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow \mathit{ExitStates}(front\ ss, ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
\mathit{ExecuteChart} \mathrel{\widehat=} ce : \mathit{EVENT} \bullet chart?c \longrightarrow status!(c.identifier)?active \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ active = \mathbf{True} \longrightarrow \mathit{ExecuteActiveChart}(c, ce) \\
[\!]\ active = \mathbf{False} \longrightarrow \mathit{ExecuteInactiveChart}(c, ce) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecuteInactiveChart \mathrel{\widehat=} c : State;\ ce : EVENT \bullet activate!(c.identifier) \longrightarrow \\
\quad \left(\begin{array}{l}
\textbf{if}\ c.default \neq nulltransition.identifier \lor c.decomposition = CLUSTER \longrightarrow \\
\qquad ExecuteDefaultTransition(c, c, ce) \\
[\!]\ c.default = nulltransition.identifier \land c.decomposition = SET \longrightarrow \\
\qquad \left(\begin{array}{l}
\textbf{if}\ c.substates = \langle\rangle \longrightarrow \textbf{Skip} \\
[\!]\ c.substates \neq \langle\rangle \longrightarrow \\
\qquad state!(head(c.substates))?first \longrightarrow EnterState(first, c, ce) \\
\textbf{fi}
\end{array}\right) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
ExecuteActiveChart \mathrel{\widehat=} c : State;\ ce : EVENT \bullet \\
\quad \left(\begin{array}{l}
\textbf{if}\ c.substates = \varnothing \longrightarrow ExecuteInactiveChart(c, ce) \\
[\!]\ c.substates \neq \varnothing \longrightarrow ExecuteSubstates(c, ce) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
LocalEventEntry \mathrel{\widehat=} sid : SID \bullet \mu X \bullet \\
\left(\begin{array}{l}
local_event?e?s \longrightarrow \left(\begin{array}{l}
TreatLocalEvent(e, s); \\
\textbf{var}\ b : \mathbb{B} \bullet \\
\left(\begin{array}{l}
entryActionCheck(sid, b); \\
\left(\begin{array}{l}
\textbf{if}\ b = \textbf{True} \longrightarrow \\
\qquad interrupt.\textbf{True} \longrightarrow end_action \longrightarrow \textbf{Skip} \\
[\!]\ b = \textbf{False} \longrightarrow interrupt.\textbf{False} \longrightarrow X \\
\textbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
end_action \longrightarrow \textbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
LocalEventDuring \mathrel{\widehat=} sid : SID \bullet \mu X \bullet \\
\left(\begin{array}{l}
local_event?e?s \longrightarrow \left(\begin{array}{l}
TreatLocalEvent(e, s); \\
\textbf{var}\ b : \mathbb{B} \\
\bullet \left(\begin{array}{l}
duringActionCheck(sid, b); \\
\left(\begin{array}{l}
\textbf{if}\ b = \textbf{True} \longrightarrow \\
\qquad interrupt.\textbf{True} \longrightarrow end_action \longrightarrow \textbf{Skip} \\
[\!]\ b = \textbf{False} \longrightarrow interrupt.\textbf{False} \longrightarrow X \\
\textbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
end_action \longrightarrow \textbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
LocalEventExit \mathrel{\widehat=} sid : SID \bullet \mu X \bullet \\
\left(\begin{array}{l}
local_event?e?s \longrightarrow \left(\begin{array}{l}
TreatLocalEvent(e,s); \\
\mathbf{var}\ b : \mathbb{B} \\
\bullet \left(\begin{array}{l}
exitActionCheck(sid,b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \\
\qquad interrupt.\mathbf{True} \longrightarrow end_action \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow interrupt.\mathbf{False} \longrightarrow X \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
end_action \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
LocalEventCondition \mathrel{\widehat=} sid : SID \bullet \mu X \bullet \\
\left(\begin{array}{l}
local_event?e?s \longrightarrow \left(\begin{array}{l}
TreatLocalEvent(e,s); \\
\mathbf{var}\ b : \mathbb{B} \\
\bullet \left(\begin{array}{l}
conditionActionCheck(sid,b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \\
\qquad interrupt.\mathbf{True} \longrightarrow end_action \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow interrupt.\mathbf{False} \longrightarrow X \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
end_action \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
LocalEventTransition \mathrel{\widehat=} sid : SID \bullet \mu X \bullet \\
\left(\begin{array}{l}
local_event?e?s \longrightarrow \left(\begin{array}{l}
TreatLocalEvent(e,s); \\
\mathbf{var}\ b : \mathbb{B} \\
\bullet \left(\begin{array}{l}
transitionActionCheck(sid,b); \\
\left(\begin{array}{l}
\mathbf{if}\ b = \mathbf{True} \longrightarrow \\
\qquad interrupt.\mathbf{True} \longrightarrow end_action \longrightarrow \mathbf{Skip} \\
[\!]\ b = \mathbf{False} \longrightarrow interrupt.\mathbf{False} \longrightarrow X \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\Box \\
end_action \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
TreatLocalEvent \mathrel{\widehat=} e : EVENT;\ s : State \bullet \\
\qquad \left(\begin{array}{l}
\mathbf{if}\ s.type = CHART \longrightarrow ExecuteChart(e) \\
[\!]\ s.type \neq CHART \longrightarrow ExecuteState(s,e) \\
\mathbf{fi}
\end{array}\right);\ end_local_execution \longrightarrow \mathbf{Skip}
\end{array}
$$

$$ExecuteEvent \mathrel{\widehat=} e : EVENT;\ v : \mathbb{B} \bullet \left(\begin{array}{l} \mathbf{if}\ v = \mathbf{True} \longrightarrow ExecuteChart(e) \\ [\!]\ \ v = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$ExecuteEvents \mathrel{\widehat=} es : \operatorname{seq} EVENT;\ vs : \operatorname{seq} \mathbb{B} \bullet (\mathbin{;}\ \ i : id(1 \mathbin{..} \# es) \bullet ExecuteEvent(es(i), vs(i)))$$

$$\bullet\ (\mu X \bullet Step\ ;\ \ end_cycle \longrightarrow X)$$

end

Appendix C

Circus models of Stateflow charts

C.1 *Circus* model of Shift Logic Chart

section $calc_th$ **parents** $basic_toolkit$

$calc_th : (\mathbb{R} \times \mathbb{R}) \rightarrow (\mathbb{R} \times \mathbb{R})$

section $sf_car_shift_logic$ **parents** $stateflow_toolkit, calc_th$

$s_downshifting, s_gear_state, s_fourth, s_second, s_third, s_first,$
$s_selection_state, s_upshifting, s_steady_state, c_shift_logic : SID$

$t_third_fourth, t_second_third, t_first_second, t_fourth_third, t_default_first,$
$t_second_first, t_third_second, t_steady_state_upshifting, t_default_steady_state,$
$t_upshifting_steady_state23, t_steady_state_downshifting,$
$t_downshifting_steady_state25, t_downshifting_steady_state24,$
$t_upshifting_steady_state26 : TID$

$C_shift_logic : State$

$$C_shift_logic = \left(\begin{array}{l} \langle\!| identifier == c_shift_logic, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == nullstate.identifier, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle s_gear_state, s_selection_state \rangle, \\ decomposition == SET, type == CHART, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_downshifting : State$

$$S_downshifting = \left(\begin{array}{l} \langle\!| identifier == s_downshifting, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_downshifting_steady_state24, \\ parent == s_selection_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_gear_state : State$

$$S_gear_state = \left(\begin{array}{l} \langle\!| identifier == s_gear_state, default == t_default_first, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == c_shift_logic, left == nullstate.identifier, \\ right == s_selection_state, substates == \langle s_fourth, s_third, s_first, s_second \rangle, \\ decomposition == CLUSTER, type == AND, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_fourth : State$

$$S_fourth = \left(\begin{array}{l} \langle\!| identifier == s_fourth, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_fourth_third, \\ parent == s_gear_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_second : State$

$$S_second = \left(\begin{array}{l} \langle\!| identifier == s_second, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_second_third, \\ parent == s_gear_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_third : State$

$$S_third = \left(\begin{array}{l} \langle\!| identifier == s_third, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_third_fourth, \\ parent == s_gear_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_first : State$

$$S_first = \\ \left(\begin{array}{l} \langle\!| identifier == s_first, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_first_second, \\ parent == s_gear_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_selection_state : State$

$$S_selection_state = \\ \left(\begin{array}{l} \langle\!| identifier == s_selection_state, default == t_default_steady_state, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == c_shift_logic, left == s_gear_state, right == nullstate.identifier, \\ substates == \langle s_upshifting, s_downshifting, s_steady_state\rangle, \\ decomposition == CLUSTER, type == AND, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_upshifting : State$

$$S_upshifting = \left(\begin{array}{l} \lblot identifier == s_upshifting, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_upshifting_steady_state26, \\ parent == s_selection_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} \rblot \end{array} \right)$$

$S_steady_state : State$

$$S_steady_state = \left(\begin{array}{l} \lblot identifier == s_steady_state, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_steady_state_upshifting, \\ parent == s_selection_state, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} \rblot \end{array} \right)$$

$T_third_fourth : Transition$

$$T_third_fourth = \left(\begin{array}{l} \lblot identifier == t_third_fourth, source == snode(s_third), \\ destination == snode(s_fourth), next == t_third_second, \\ parent == s_gear_state \rblot \end{array} \right)$$

$T_second_third : Transition$

$$T_second_third = \left(\begin{array}{l} \lblot identifier == t_second_third, source == snode(s_second), \\ destination == snode(s_third), next == t_second_first, \\ parent == s_gear_state \rblot \end{array} \right)$$

$T_first_second : Transition$

$$T_first_second = \left(\begin{array}{l} \lblot identifier == t_first_second, source == snode(s_first), \\ destination == snode(s_second), next == nulltransition.identifier, \\ parent == s_gear_state \rblot \end{array} \right)$$

$T_fourth_third : Transition$

$$T_fourth_third = \left(\begin{array}{l} \langle\!| identifier == t_fourth_third, source == snode(s_fourth), \\ destination == snode(s_third), next == nulltransition.identifier, \\ parent == s_gear_state |\!\rangle \end{array} \right)$$

$T_default_first : Transition$

$$T_default_first = \left(\begin{array}{l} \langle\!| identifier == t_default_first, source == snode(nullstate.identifier), \\ destination == snode(s_first), next == nulltransition.identifier, \\ parent == s_gear_state |\!\rangle \end{array} \right)$$

$T_second_first : Transition$

$$T_second_first = \left(\begin{array}{l} \langle\!| identifier == t_second_first, source == snode(s_second), \\ destination == snode(s_first), next == nulltransition.identifier, \\ parent == s_gear_state |\!\rangle \end{array} \right)$$

$T_third_second : Transition$

$$T_third_second = \left(\begin{array}{l} \langle\!| identifier == t_third_second, source == snode(s_third), \\ destination == snode(s_second), next == nulltransition.identifier, \\ parent == s_gear_state |\!\rangle \end{array} \right)$$

$T_steady_state_upshifting : Transition$

$$T_steady_state_upshifting = \left(\begin{array}{l} \langle\!| identifier == t_steady_state_upshifting, source == snode(s_steady_state), \\ destination == snode(s_upshifting), next == t_steady_state_downshifting, \\ parent == s_selection_state |\!\rangle \end{array} \right)$$

$T_default_steady_state : Transition$

$$T_default_steady_state = \left(\begin{array}{l} \langle\!| identifier == t_default_steady_state, source == snode(nullstate.identifier), \\ destination == snode(s_steady_state), next == nulltransition.identifier, \\ parent == s_selection_state |\!\rangle \end{array} \right)$$

$T_upshifting_steady_state23 : Transition$

$$T_upshifting_steady_state23 = \left(\begin{array}{l} \langle\!| identifier == t_upshifting_steady_state23, source == snode(s_upshifting), \\ destination == snode(s_steady_state), next == nulltransition.identifier, \\ parent == s_selection_state |\!\rangle \end{array} \right)$$

$T_steady_state_downshifting : Transition$

$$T_steady_state_downshifting = \left(\begin{array}{l} \langle\!| identifier == t_steady_state_downshifting, source == snode(s_steady_state), \\ destination == snode(s_downshifting), next == nulltransition.identifier, \\ parent == s_selection_state |\!\rangle \end{array} \right)$$

$T_downshifting_steady_state25 : Transition$

$$T_downshifting_steady_state25 = \left(\begin{array}{l} \langle\!| identifier == t_downshifting_steady_state25, source == snode(s_downshifting), \\ destination == snode(s_steady_state), next == nulltransition.identifier, \\ parent == s_selection_state |\!\rangle \end{array} \right)$$

$T_downshifting_steady_state24 : Transition$

$$T_downshifting_steady_state24 = \left(\begin{array}{l} \langle\!| identifier == t_downshifting_steady_state24, source == snode(s_downshifting), \\ destination == snode(s_steady_state), next == t_downshifting_steady_state25, \\ parent == s_selection_state |\!\rangle \end{array} \right)$$

$T_upshifting_steady_state26 : Transition$

$$T_upshifting_steady_state26 = \left(\begin{array}{l}\langle\!| identifier == t_upshifting_steady_state26, source == snode(s_upshifting), \\ destination == snode(s_steady_state), next == t_upshifting_steady_state23, \\ parent == s_selection_state |\!\rangle\end{array}\right)$$

$e_UP, e_DOWN, ENULL : EVENT$

channel $o_gear : \mathbb{R};\ i_speed : \mathbb{R};\ i_throttle : \mathbb{R}$

process $P_shift_logic \mathrel{\widehat=}$ **begin**

$StateflowChart$

$$\begin{array}{l}identifier = c_shift_logic \\ states = \{(c_shift_logic, C_shift_logic), (s_downshifting, S_downshifting), \\ \quad (s_gear_state, S_gear_state), (s_fourth, S_fourth), (s_second, S_second), \\ \quad (s_third, S_third), (s_first, S_first), (s_selection_state, S_selection_state), \\ \quad (s_upshifting, S_upshifting), (s_steady_state, S_steady_state)\} \\ transitions = \{(t_third_fourth, T_third_fourth), (t_second_third, T_second_third), \\ \quad (t_first_second, T_first_second), (t_fourth_third, T_fourth_third), \\ \quad (t_default_first, T_default_first), (t_second_first, T_second_first), \\ \quad (t_third_second, T_third_second), \\ \quad (t_steady_state_upshifting, T_steady_state_upshifting), \\ \quad (t_default_steady_state, T_default_steady_state), \\ \quad (t_upshifting_steady_state23, T_upshifting_steady_state23), \\ \quad (t_steady_state_downshifting, T_steady_state_downshifting), \\ \quad (t_downshifting_steady_state25, T_downshifting_steady_state25), \\ \quad (t_downshifting_steady_state24, T_downshifting_steady_state24), \\ \quad (t_upshifting_steady_state26, T_upshifting_steady_state26)\} \\ junctions = \{\}\end{array}$$

$SimulationInstance$

$v_gear, v_up_th, v_speed, v_down_th, v_throttle : \mathbb{R}$

InitSimulationInstance

$SimulationInstance'$

$v_gear' = 0$
$v_up_th' = 0$
$v_speed' = 0$
$v_down_th' = 0$
$v_throttle' = 0$

SimulationData

$state_status : SID \nrightarrow \mathbb{B}$
$state_history : SID \nrightarrow SID$

$\mathrm{dom}\, state_status = \mathrm{dom}\, states$
$\mathrm{dom}\, state_history = \{j : \mathrm{ran}\, junctions \mid j.history = \mathbf{True} \bullet j.parent\}$
$\forall s : \mathrm{ran}\, states \mid s.decomposition = CLUSTER \bullet$
$\quad \#\{ss : \mathrm{ran}\, s.substates \mid state_status(ss) = \mathbf{True}\} \leq 1$

InitSimulationData

$SimulationData'$

$state_status' = \{n : \mathrm{dom}\, states \bullet n \mapsto \mathbf{False}\}$
$state_history' = \{n : \mathrm{dom}\, state_history' \bullet n \mapsto nullstate.identifier\}$

ActivateNoHistory

$\Delta SimulationData$
$x? : SID$

$x? \in \mathrm{dom}\, state_status$
$(parent\,(states\, x?)).history = \mathbf{False}$
$state_history' = state_history$
$state_status' = state_status \oplus \{x? \mapsto \mathbf{True}\}$

ActivateWithHistory

$\Delta SimulationData$
$x? : SID$

$x? \in \mathrm{dom}\ state_status$
$(parent\ (states\ x?)).history = \mathbf{True}$
$state_history' = state_history \oplus \{((states\ x?).parent) \mapsto x?\}$
$state_status' = state_status \oplus \{x? \mapsto \mathbf{True}\}$

$$Activate == (ActivateWithHistory \lor ActivateNoHistory) \land \Xi SimulationInstance$$

Deactivate

$\Delta SimulationData$
$\Xi SimulationInstance$
$x? : SID$

$x? \in \mathrm{dom}\ state_status$
$state_history' = state_history$
$state_status' = state_status \oplus \{x? \mapsto \mathbf{False}\}$

$$InitState == (InitSimulationInstance) \land (InitSimulationData)$$

$$\mathbf{state}\ shift_logic_state == (SimulationInstance\ ) \land (SimulationData\ )$$

$$entryaction_downshifting \mathrel{\widehat=} (executeentryaction.(s_downshifting) \longrightarrow \mathbf{Skip})$$
$$entryaction_gear_state \mathrel{\widehat=} (executeentryaction.(s_gear_state) \longrightarrow \mathbf{Skip})$$
$$entryaction_fourth \mathrel{\widehat=} (executeentryaction.(s_fourth) \longrightarrow (v_gear := 4\ ;\ \mathbf{Skip}))$$
$$entryaction_second \mathrel{\widehat=} (executeentryaction.(s_second) \longrightarrow (v_gear := 2\ ;\ \mathbf{Skip}))$$
$$entryaction_third \mathrel{\widehat=} (executeentryaction.(s_third) \longrightarrow (v_gear := 3\ ;\ \mathbf{Skip}))$$
$$entryaction_first \mathrel{\widehat=} (executeentryaction.(s_first) \longrightarrow (v_gear := 1\ ;\ \mathbf{Skip}))$$
$$entryaction_selection_state \mathrel{\widehat=} (executeentryaction.(s_selection_state) \longrightarrow \mathbf{Skip})$$
$$entryaction_upshifting \mathrel{\widehat=} (executeentryaction.(s_upshifting) \longrightarrow \mathbf{Skip})$$
$$entryaction_steady_state \mathrel{\widehat=} (executeentryaction.(s_steady_state) \longrightarrow \mathbf{Skip})$$
$$entryactions \mathrel{\widehat=} \left(\begin{array}{l} entryaction_downshifting \Box entryaction_gear_state \Box \\ entryaction_fourth \Box entryaction_second \Box entryaction_third \Box \\ entryaction_first \Box entryaction_selection_state \Box \\ entryaction_upshifting \Box entryaction_steady_state \end{array}\right)$$

$duringaction_downshifting \mathrel{\widehat=} (executeduringaction.(s_downshifting)?ce \longrightarrow \mathbf{Skip})$
$duringaction_gear_state \mathrel{\widehat=} (executeduringaction.(s_gear_state)?ce \longrightarrow \mathbf{Skip})$
$duringaction_fourth \mathrel{\widehat=} (executeduringaction.(s_fourth)?ce \longrightarrow \mathbf{Skip})$
$duringaction_second \mathrel{\widehat=} (executeduringaction.(s_second)?ce \longrightarrow \mathbf{Skip})$
$duringaction_third \mathrel{\widehat=} (executeduringaction.(s_third)?ce \longrightarrow \mathbf{Skip})$
$duringaction_first \mathrel{\widehat=} (executeduringaction.(s_first)?ce \longrightarrow \mathbf{Skip})$
$duringaction_selection_state \mathrel{\widehat=} executeduringaction.(s_selection_state)?ce \longrightarrow$
$$\left(\begin{array}{l}\mathbf{var}\ _aux : \mathbb{R} \times \mathbb{R} \bullet _aux := calc_th(v_gear, v_throttle);\\ v_down_th := _aux.1\ ;\ v_up_th := _aux.2\end{array}\right)$$
$duringaction_upshifting \mathrel{\widehat=} (executeduringaction.(s_upshifting)?ce \longrightarrow \mathbf{Skip})$
$duringaction_steady_state \mathrel{\widehat=} (executeduringaction.(s_steady_state)?ce \longrightarrow \mathbf{Skip})$
$$duringactions \mathrel{\widehat=} \left(\begin{array}{l}duringaction_downshifting \mathrel{\Box} duringaction_gear_state \mathrel{\Box}\\ duringaction_fourth \mathrel{\Box} duringaction_second \mathrel{\Box}\\ duringaction_third \mathrel{\Box} duringaction_first \mathrel{\Box}\\ duringaction_selection_state \mathrel{\Box} duringaction_upshifting \mathrel{\Box}\\ duringaction_steady_state\end{array}\right)$$

$exitaction_downshifting \mathrel{\widehat=} (executeexitaction.(s_downshifting) \longrightarrow \mathbf{Skip})$
$exitaction_gear_state \mathrel{\widehat=} (executeexitaction.(s_gear_state) \longrightarrow \mathbf{Skip})$
$exitaction_fourth \mathrel{\widehat=} (executeexitaction.(s_fourth) \longrightarrow \mathbf{Skip})$
$exitaction_second \mathrel{\widehat=} (executeexitaction.(s_second) \longrightarrow \mathbf{Skip})$
$exitaction_third \mathrel{\widehat=} (executeexitaction.(s_third) \longrightarrow \mathbf{Skip})$
$exitaction_first \mathrel{\widehat=} (executeexitaction.(s_first) \longrightarrow \mathbf{Skip})$
$exitaction_selection_state \mathrel{\widehat=} (executeexitaction.(s_selection_state) \longrightarrow \mathbf{Skip})$
$exitaction_upshifting \mathrel{\widehat=} (executeexitaction.(s_upshifting) \longrightarrow \mathbf{Skip})$
$exitaction_steady_state \mathrel{\widehat=} (executeexitaction.(s_steady_state) \longrightarrow \mathbf{Skip})$
$$exitactions \mathrel{\widehat=} \left(\begin{array}{l}exitaction_downshifting \mathrel{\Box} exitaction_gear_state \mathrel{\Box}\\ exitaction_fourth \mathrel{\Box} exitaction_second \mathrel{\Box}\\ exitaction_third \mathrel{\Box} exitaction_first \mathrel{\Box}\\ exitaction_selection_state \mathrel{\Box} exitaction_upshifting \mathrel{\Box}\\ exitaction_steady_state\end{array}\right)$$

$conditionaction_third_fourth \mathrel{\widehat=} (executeconditionaction.(t_third_fourth) \longrightarrow \mathbf{Skip})$
$conditionaction_second_third \mathrel{\widehat=} (executeconditionaction.(t_second_third) \longrightarrow \mathbf{Skip})$
$conditionaction_first_second \mathrel{\widehat=} (executeconditionaction.(t_first_second) \longrightarrow \mathbf{Skip})$
$conditionaction_fourth_third \mathrel{\widehat=} (executeconditionaction.(t_fourth_third) \longrightarrow \mathbf{Skip})$
$conditionaction_default_first \mathrel{\widehat=} (executeconditionaction.(t_default_first) \longrightarrow \mathbf{Skip})$
$conditionaction_second_first \mathrel{\widehat=} (executeconditionaction.(t_second_first) \longrightarrow \mathbf{Skip})$
$conditionaction_third_second \mathrel{\widehat=} (executeconditionaction.(t_third_second) \longrightarrow \mathbf{Skip})$

$$conditionaction_steady_state_upshifting \mathrel{\widehat=} \\ \quad (executeconditionaction.(t_steady_state_upshifting) \longrightarrow \mathbf{Skip})$$
$$conditionaction_default_steady_state \mathrel{\widehat=} \\ \quad (executeconditionaction.(t_default_steady_state) \longrightarrow \mathbf{Skip})$$
$$conditionaction_upshifting_steady_state23 \mathrel{\widehat=} \\ \quad (executeconditionaction.(t_upshifting_steady_state23) \longrightarrow \mathbf{Skip})$$
$$conditionaction_steady_state_downshifting \mathrel{\widehat=} \\ \quad (executeconditionaction.(t_steady_state_downshifting) \longrightarrow \mathbf{Skip})$$

$$conditionaction_downshifting_steady_state25 \mathrel{\widehat=} \\ \quad executeconditionaction.(t_downshifting_steady_state25) \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{var}\ _b : \mathbb{B} \bullet broadcast(e_DOWN, s_gear_state)\ ;\ \ check(_b); \\ \left(\begin{array}{l} \mathbf{if}\ _b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ \neg\ (_b = \mathbf{True}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \end{array}\right)$$
$$conditionaction_downshifting_steady_state24 \mathrel{\widehat=} \\ \quad (executeconditionaction.(t_downshifting_steady_state24) \longrightarrow \mathbf{Skip})$$
$$conditionaction_upshifting_steady_state26 \mathrel{\widehat=} \\ \quad executeconditionaction.(t_upshifting_steady_state26) \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{var}\ _b : \mathbb{B} \bullet broadcast(e_UP, s_gear_state)\ ;\ \ check(_b); \\ \left(\begin{array}{l} \mathbf{if}\ _b = \mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ \neg\ (_b = \mathbf{True}) \longrightarrow (\mathbf{Skip}) \\ \mathbf{fi} \end{array}\right) \end{array}\right)$$

$$conditionactions \mathrel{\widehat=} \left(\begin{array}{l} conditionaction_third_fourth \Box conditionaction_second_third \Box \\ conditionaction_first_second \Box conditionaction_fourth_third \Box \\ conditionaction_default_first \Box conditionaction_second_first \Box \\ conditionaction_third_second \Box \\ conditionaction_steady_state_upshifting \Box \\ conditionaction_default_steady_state \Box \\ conditionaction_upshifting_steady_state23 \Box \\ conditionaction_steady_state_downshifting \Box \\ conditionaction_downshifting_steady_state25 \Box \\ conditionaction_downshifting_steady_state24 \Box \\ conditionaction_upshifting_steady_state26 \end{array}\right)$$

$transitionaction_third_fourth \widehat{=} (executetransitionaction.(t_third_fourth) \longrightarrow \mathbf{Skip})$
$transitionaction_second_third \widehat{=} (executetransitionaction.(t_second_third) \longrightarrow \mathbf{Skip})$
$transitionaction_first_second \widehat{=} (executetransitionaction.(t_first_second) \longrightarrow \mathbf{Skip})$
$transitionaction_fourth_third \widehat{=} (executetransitionaction.(t_fourth_third) \longrightarrow \mathbf{Skip})$
$transitionaction_default_first \widehat{=} (executetransitionaction.(t_default_first) \longrightarrow \mathbf{Skip})$
$transitionaction_second_first \widehat{=} (executetransitionaction.(t_second_first) \longrightarrow \mathbf{Skip})$
$transitionaction_third_second \widehat{=} (executetransitionaction.(t_third_second) \longrightarrow \mathbf{Skip})$

$transitionaction_steady_state_upshifting \widehat{=}$
$\quad (executetransitionaction.(t_steady_state_upshifting) \longrightarrow \mathbf{Skip})$
$transitionaction_default_steady_state \widehat{=}$
$\quad (executetransitionaction.(t_default_steady_state) \longrightarrow \mathbf{Skip})$
$transitionaction_upshifting_steady_state23 \widehat{=}$
$\quad (executetransitionaction.(t_upshifting_steady_state23) \longrightarrow \mathbf{Skip})$
$transitionaction_steady_state_downshifting \widehat{=}$
$\quad (executetransitionaction.(t_steady_state_downshifting) \longrightarrow \mathbf{Skip})$
$transitionaction_downshifting_steady_state25 \widehat{=}$
$\quad (executetransitionaction.(t_downshifting_steady_state25) \longrightarrow \mathbf{Skip})$
$transitionaction_downshifting_steady_state24 \widehat{=}$
$\quad (executetransitionaction.(t_downshifting_steady_state24) \longrightarrow \mathbf{Skip})$
$transitionaction_upshifting_steady_state26 \widehat{=}$
$\quad (executetransitionaction.(t_upshifting_steady_state26) \longrightarrow \mathbf{Skip})$

$$transitionactions \widehat{=} \left(\begin{array}{l} transitionaction_third_fourth \Box transitionaction_second_third \Box \\ transitionaction_first_second \Box transitionaction_fourth_third \Box \\ transitionaction_default_first \Box transitionaction_second_first \Box \\ transitionaction_third_second \Box \\ transitionaction_steady_state_upshifting \Box \\ transitionaction_default_steady_state \Box \\ transitionaction_upshifting_steady_state23 \Box \\ transitionaction_steady_state_downshifting \Box \\ transitionaction_downshifting_steady_state25 \Box \\ transitionaction_downshifting_steady_state24 \Box \\ transitionaction_upshifting_steady_state26 \end{array}\right)$$

$condition_third_fourth \mathrel{\widehat=} (evaluatecondition.(t_third_fourth)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_second_third \mathrel{\widehat=} (evaluatecondition.(t_second_third)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_first_second \mathrel{\widehat=} (evaluatecondition.(t_first_second)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_fourth_third \mathrel{\widehat=} (evaluatecondition.(t_fourth_third)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_default_first \mathrel{\widehat=} (evaluatecondition.(t_default_first)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_second_first \mathrel{\widehat=} (evaluatecondition.(t_second_first)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_third_second \mathrel{\widehat=} (evaluatecondition.(t_third_second)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_steady_state_upshifting \mathrel{\widehat=}$

$$\left(\begin{array}{l} \mathbf{if}((v_speed >_{\mathcal{A}} v_up_th) \neq 0) \longrightarrow \\ \quad evaluatecondition.(t_steady_state_upshifting)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!] \neg\ (((v_speed >_{\mathcal{A}} v_up_th) \neq 0)) \longrightarrow \\ \quad evaluatecondition.(t_steady_state_upshifting)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$condition_default_steady_state \mathrel{\widehat=}$

$\quad (evaluatecondition.(t_default_steady_state)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$condition_upshifting_steady_state23 \mathrel{\widehat=}$

$$\left(\begin{array}{l} \mathbf{if}((v_speed <_{\mathcal{A}} v_up_th) \neq 0) \longrightarrow \\ \quad evaluatecondition.(t_upshifting_steady_state23)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!] \neg\ (((v_speed <_{\mathcal{A}} v_up_th) \neq 0)) \longrightarrow \\ \quad evaluatecondition.(t_upshifting_steady_state23)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$condition_steady_state_downshifting \mathrel{\widehat=}$

$$\left(\begin{array}{l} \mathbf{if}((v_speed <_{\mathcal{A}} v_down_th) \neq 0) \longrightarrow \\ \quad evaluatecondition.(t_steady_state_downshifting)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!] \neg\ (((v_speed <_{\mathcal{A}} v_down_th) \neq 0)) \longrightarrow \\ \quad evaluatecondition.(t_steady_state_downshifting)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$condition_downshifting_steady_state25 \mathrel{\widehat=}$

$$\left(\begin{array}{l} \mathbf{if}((v_speed \leq_{\mathcal{A}} v_down_th) \neq 0) \longrightarrow \\ \quad evaluatecondition.(t_downshifting_steady_state25)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!] \neg\ (((v_speed \leq_{\mathcal{A}} v_down_th) \neq 0)) \longrightarrow \\ \quad evaluatecondition.(t_downshifting_steady_state25)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$condition_downshifting_steady_state24 \mathrel{\widehat=}$

$$\left(\begin{array}{l} \mathbf{if}((v_speed >_{\mathcal{A}} v_down_th) \neq 0) \longrightarrow \\ \quad evaluatecondition.(t_downshifting_steady_state24)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!] \neg\ (((v_speed >_{\mathcal{A}} v_down_th) \neq 0)) \longrightarrow \\ \quad evaluatecondition.(t_downshifting_steady_state24)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$condition_upshifting_steady_state26 \mathrel{\widehat=} \left(\begin{array}{l} \mathbf{if}((v_speed \geq_{\mathcal{A}} v_up_th) \neq 0) \longrightarrow \\ \quad evaluatecondition.(t_upshifting_steady_state26)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ (((v_speed \geq_{\mathcal{A}} v_up_th) \neq 0)) \longrightarrow \\ \quad evaluatecondition.(t_upshifting_steady_state26)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$conditions \mathrel{\widehat=} \left(\begin{array}{l} condition_third_fourth \Box condition_second_third \Box \\ condition_first_second \Box condition_fourth_third \Box \\ condition_default_first \Box condition_second_first \Box \\ condition_third_second \Box \\ condition_steady_state_upshifting \Box \\ condition_default_steady_state \Box \\ condition_upshifting_steady_state23 \Box \\ condition_steady_state_downshifting \Box \\ condition_downshifting_steady_state25 \Box \\ condition_downshifting_steady_state24 \Box \\ condition_upshifting_steady_state26 \end{array}\right)$$

$$trigger_third_fourth \mathrel{\widehat=} checktrigger.(t_third_fourth)?e \longrightarrow \left(\begin{array}{l} \mathbf{if}\ e = e_UP \longrightarrow result.(t_third_fourth).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ (e = e_UP) \longrightarrow result.(t_third_fourth).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$trigger_second_third \mathrel{\widehat=} checktrigger.(t_second_third)?e \longrightarrow \left(\begin{array}{l} \mathbf{if}\ e = e_UP \longrightarrow result.(t_second_third).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ (e = e_UP) \longrightarrow result.(t_second_third).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$trigger_first_second \mathrel{\widehat=} checktrigger.(t_first_second)?e \longrightarrow \left(\begin{array}{l} \mathbf{if}\ e = e_UP \longrightarrow result.(t_first_second).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ (e = e_UP) \longrightarrow result.(t_first_second).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$trigger_fourth_third \mathrel{\widehat=} checktrigger.(t_fourth_third)?e \longrightarrow \left(\begin{array}{l} \mathbf{if}\ e = e_DOWN \longrightarrow result.(t_fourth_third).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ (e = e_DOWN) \longrightarrow result.(t_fourth_third).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$

$$\begin{array}{l}
trigger_default_first \mathrel{\widehat=} checktrigger.(t_default_first)?e \longrightarrow \\
\quad result.(t_default_first).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_second_first \mathrel{\widehat=} checktrigger.(t_second_first)?e \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ e = e_DOWN \longrightarrow result.(t_second_first).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
[\!]\neg\ (e = e_DOWN) \longrightarrow result.(t_second_first).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
trigger_third_second \mathrel{\widehat=} checktrigger.(t_third_second)?e \longrightarrow \\
\quad \left(\begin{array}{l}
\mathbf{if}\ e = e_DOWN \longrightarrow result.(t_third_second).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
[\!]\neg\ (e = e_DOWN) \longrightarrow result.(t_third_second).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
trigger_steady_state_upshifting \mathrel{\widehat=} checktrigger.(t_steady_state_upshifting)?e \longrightarrow \\
\quad result.(t_steady_state_upshifting).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_default_steady_state \mathrel{\widehat=} checktrigger.(t_default_steady_state)?e \longrightarrow \\
\quad result.(t_default_steady_state).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_upshifting_steady_state23 \mathrel{\widehat=} checktrigger.(t_upshifting_steady_state23)?e \longrightarrow \\
\quad result.(t_upshifting_steady_state23).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_steady_state_downshifting \mathrel{\widehat=} checktrigger.(t_steady_state_downshifting)?e \longrightarrow \\
\quad result.(t_steady_state_downshifting).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_downshifting_steady_state25 \mathrel{\widehat=} checktrigger.(t_downshifting_steady_state25)?e \longrightarrow \\
\quad result.(t_downshifting_steady_state25).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_downshifting_steady_state24 \mathrel{\widehat=} checktrigger.(t_downshifting_steady_state24)?e \longrightarrow \\
\quad result.(t_downshifting_steady_state24).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\
trigger_upshifting_steady_state26 \mathrel{\widehat=} checktrigger.(t_upshifting_steady_state26)?e \longrightarrow \\
\quad result.(t_upshifting_steady_state26).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip}
\end{array}$$

$$triggers \mathrel{\widehat=} \left(\begin{array}{l}
trigger_third_fourth \mathbin{\Box} trigger_second_third \mathbin{\Box} \\
trigger_first_second \mathbin{\Box} trigger_fourth_third \mathbin{\Box} \\
trigger_default_first \mathbin{\Box} trigger_second_first \mathbin{\Box} \\
trigger_third_second \mathbin{\Box} trigger_steady_state_upshifting \mathbin{\Box} \\
trigger_default_steady_state \mathbin{\Box} trigger_upshifting_steady_state23 \mathbin{\Box} \\
trigger_steady_state_downshifting \mathbin{\Box} trigger_downshifting_steady_state25 \mathbin{\Box} \\
trigger_downshifting_steady_state24 \mathbin{\Box} trigger_upshifting_steady_state26
\end{array}\right)$$

$$\begin{array}{l}
getevents \mathrel{\widehat=} (events!(\langle ENULL \rangle) \longrightarrow \mathbf{Skip}) \\
getstate \mathrel{\widehat=} (state?x : (x \in \mathrm{dom}(states))!(states(x)) \longrightarrow \mathbf{Skip}) \\
getjunction \mathrel{\widehat=} (junction?x : (x \in \mathrm{dom}(junctions))!(junctions(x)) \longrightarrow \mathbf{Skip}) \\
gettransition \mathrel{\widehat=} (transition?x : (x \in \mathrm{dom}(transitions))!(transitions(x)) \longrightarrow \mathbf{Skip}) \\
getchart \mathrel{\widehat=} (chart!(states(identifier)) \longrightarrow \mathbf{Skip})
\end{array}$$

$broadcast \mathrel{\widehat=} e : EVENT;\ dest : SID \bullet local_event!(e, states(dest)) \longrightarrow$
$\quad \mu X \bullet (AllActions\ ;\ X \Box end_local_execution \longrightarrow \mathbf{Skip})$
$check \mathrel{\widehat=} \mathbf{res}\ erl : \mathbb{B} \bullet \mu X \bullet (InterfaceActions\ ;\ X \Box interrupt?x \longrightarrow erl := x)$
$status \mathrel{\widehat=} (status?x : (x \in \mathrm{dom}(state_status))!(state_status(x)) \longrightarrow \mathbf{Skip})$
$history \mathrel{\widehat=} (history?x : (x \in \mathrm{dom}(state_history))!(state_history(x)) \longrightarrow \mathbf{Skip})$
$activation \mathrel{\widehat=} (activate?x \longrightarrow \big(Activate\big))$
$deactivation \mathrel{\widehat=} (deactivate?x \longrightarrow \big(Deactivate\big))$

$$ChartActions \mathrel{\widehat=} \left(\begin{array}{l} entryactions \Box duringactions \Box \\ exitactions \Box conditionactions \Box \\ transitionactions \end{array}\right);\ end_action \longrightarrow \mathbf{Skip}$$

$$InterfaceActions \mathrel{\widehat=} \left(\begin{array}{l} getevents \Box getchart \Box getstate \Box getjunction \Box gettransition \Box \\ status \Box history \Box activation \Box deactivation \end{array}\right)$$

$$Inputs \mathrel{\widehat=} read_inputs \longrightarrow \left(\begin{array}{c} i_speed?x \longrightarrow v_speed := x \\ \lbrack\!\lbrack \{v_speed\} \mid \{v_throttle\} \rbrack\!\rbrack \\ i_throttle?x \longrightarrow v_throttle := x \end{array}\right)$$

$Outputs \mathrel{\widehat=} write_outputs \longrightarrow o_gear!(v_gear) \longrightarrow \mathbf{Skip}$

$$AllActions \mathrel{\widehat=} \left(\begin{array}{l} conditionactions \Box triggers \Box Inputs \Box Outputs \Box \\ ChartActions \Box InterfaceActions \end{array}\right)$$

$$\bullet \big(InitState\big);\ \left(\mu X \bullet \left(\mu Y \bullet \left(\begin{array}{l} AllActions\ ;\ \ Y \\ \Box \\ end_cycle \longrightarrow \mathbf{Skip} \end{array}\right)\right);\ X\right)$$

end

$\mathbf{process}\ shift_logic \mathrel{\widehat=} (P_shift_logic \lbrack\!\lbrack interface \cup \{\!| end_cycle |\!\} \rbrack\!\rbrack Simulator) \setminus interface$

C.2 *Circus* model of Air Controller Chart

section *air_Controller* **parents** *stateflow_toolkit*

$$s_Off3, s_On4, s_FAN2, s_SpeedValue, s_Off7, s_On8, s_FAN1, s_PowerOn, s_PowerOff, c_Controller : SID$$

$$t_On8_Off7, t_Off7_On8, t_PowerOff_PowerOn, t_default_PowerOff, t_PowerOn_PowerOff, t_On4_Off3, t_Off3_On4, t_default_Off3, t_default_Off7 : TID$$

$C_Controller : State$

$$C_Controller = \left(\begin{array}{l} \langle\!| identifier == c_Controller, default == t_default_PowerOff, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == nullstate.identifier, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle s_PowerOn, s_PowerOff \rangle, \\ decomposition == CLUSTER, type == CHART, history == \mathbf{False} |\!\rangle \end{array}\right)$$

$S_Off3 : State$

$$S_Off3 = \left(\begin{array}{l} \langle\!| identifier == s_Off3, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_Off3_On4, \\ parent == s_FAN2, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array}\right)$$

$S_On4 : State$

$$S_On4 = \left(\begin{array}{l} \langle\!| identifier == s_On4, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_On4_Off3, \\ parent == s_FAN2, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array}\right)$$

$S_FAN2 : State$

$$S_FAN2 = \left(\begin{array}{l} \langle\!| identifier == s_FAN2, default == t_default_Off3, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == s_PowerOn, left == s_FAN1, \\ right == s_SpeedValue, substates == \langle s_Off3, s_On4 \rangle, \\ decomposition == CLUSTER, type == AND, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_SpeedValue : State$

$$S_SpeedValue = \left(\begin{array}{l} \langle\!| identifier == s_SpeedValue, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == s_PowerOn, left == s_FAN2, \\ right == nullstate.identifier, substates == \langle \rangle, \\ decomposition == CLUSTER, type == AND, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_Off7 : State$

$$S_Off7 = \left(\begin{array}{l} \langle\!| identifier == s_Off7, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_Off7_On8, \\ parent == s_FAN1, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle \rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_On8 : State$

$$S_On8 = \left(\begin{array}{l} \langle\!| identifier == s_On8, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_On8_Off7, \\ parent == s_FAN1, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle \rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False} |\!\rangle \end{array} \right)$$

$S_FAN1 : State$

$$S_FAN1 = \left(\begin{array}{l} \langle\!|identifier == s_FAN1, default == t_default_Off7, \\ inner == nulltransition.identifier, outer == nulltransition.identifier, \\ parent == s_PowerOn, left == nullstate.identifier, \\ right == s_FAN2, substates == \langle s_Off7, s_On8\rangle, \\ decomposition == CLUSTER, type == AND, history == \mathbf{False}|\!\rangle \end{array}\right)$$

$S_PowerOn : State$

$$S_PowerOn = \left(\begin{array}{l} \langle\!|identifier == s_PowerOn, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_PowerOn_PowerOff, \\ parent == c_Controller, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle s_FAN1, s_FAN2, s_SpeedValue\rangle, \\ decomposition == SET, type == OR, history == \mathbf{False}|\!\rangle \end{array}\right)$$

$S_PowerOff : State$

$$S_PowerOff = \left(\begin{array}{l} \langle\!|identifier == s_PowerOff, default == nulltransition.identifier, \\ inner == nulltransition.identifier, outer == t_PowerOff_PowerOn, \\ parent == c_Controller, left == nullstate.identifier, \\ right == nullstate.identifier, substates == \langle\rangle, \\ decomposition == CLUSTER, type == OR, history == \mathbf{False}|\!\rangle \end{array}\right)$$

$T_On8_Off7 : Transition$

$$T_On8_Off7 = \left(\begin{array}{l} \langle\!|identifier == t_On8_Off7, source == snode(s_On8), \\ destination == snode(s_Off7), next == nulltransition.identifier, \\ parent == s_FAN1|\!\rangle \end{array}\right)$$

$T_Off7_On8 : Transition$

$$T_Off7_On8 = \left(\begin{array}{l} \langle\!|identifier == t_Off7_On8, source == snode(s_Off7), \\ destination == snode(s_On8), next == nulltransition.identifier, \\ parent == s_FAN1|\!\rangle \end{array}\right)$$

$$T_PowerOff_PowerOn : Transition$$

$$T_PowerOff_PowerOn = \left(\begin{array}{l} \langle\!| identifier == t_PowerOff_PowerOn, source == snode(s_PowerOff), \\ destination == snode(s_PowerOn), next == nulltransition.identifier, \\ parent == c_Controller |\!\rangle \end{array} \right)$$

$$T_default_PowerOff : Transition$$

$$T_default_PowerOff = \left(\begin{array}{l} \langle\!| identifier == t_default_PowerOff, source == snode(nullstate.identifier), \\ destination == snode(s_PowerOff), next == nulltransition.identifier, \\ parent == c_Controller |\!\rangle \end{array} \right)$$

$$T_PowerOn_PowerOff : Transition$$

$$T_PowerOn_PowerOff = \left(\begin{array}{l} \langle\!| identifier == t_PowerOn_PowerOff, source == snode(s_PowerOn), \\ destination == snode(s_PowerOff), next == nulltransition.identifier, \\ parent == c_Controller |\!\rangle \end{array} \right)$$

$$T_On4_Off3 : Transition$$

$$T_On4_Off3 = \left(\begin{array}{l} \langle\!| identifier == t_On4_Off3, source == snode(s_On4), \\ destination == snode(s_Off3), next == nulltransition.identifier, \\ parent == s_FAN2 |\!\rangle \end{array} \right)$$

$$T_Off3_On4 : Transition$$

$$T_Off3_On4 = \left(\begin{array}{l} \langle\!| identifier == t_Off3_On4, source == snode(s_Off3), \\ destination == snode(s_On4), next == nulltransition.identifier, \\ parent == s_FAN2 |\!\rangle \end{array} \right)$$

$T_default_Off3 : Transition$

$T_default_Off3 =$
$$\left(\begin{array}{l}\langle\!|identifier == t_default_Off3, source == snode(nullstate.identifier),\\ destination == snode(s_Off3), next == nulltransition.identifier,\\ parent == s_FAN2|\!\rangle\end{array}\right)$$

$T_default_Off7 : Transition$

$T_default_Off7 =$
$$\left(\begin{array}{l}\langle\!|identifier == t_default_Off7, source == snode(nullstate.identifier),\\ destination == snode(s_Off7), next == nulltransition.identifier,\\ parent == s_FAN1|\!\rangle\end{array}\right)$$

$e_SWITCH, e_CLOCK : EVENT$

channel $o_airflow : \mathbb{N};\ i_temp : \mathbb{R}$

process $P_Controller \mathrel{\widehat=}$ **begin**

$StateflowChart$

$identifier = c_Controller$
$states = \{(c_Controller, C_Controller), (s_Off3, S_Off3), (s_On4, S_On4),$
$(s_FAN2, S_FAN2), (s_SpeedValue, S_SpeedValue), (s_Off7, S_Off7),$
$(s_On8, S_On8), (s_FAN1, S_FAN1), (s_PowerOn, S_PowerOn),$
$(s_PowerOff, S_PowerOff)\}$
$transitions = \{(t_On8_Off7, T_On8_Off7), (t_Off7_On8, T_Off7_On8),$
$(t_PowerOff_PowerOn, T_PowerOff_PowerOn),$
$(t_default_PowerOff, T_default_PowerOff),$
$(t_PowerOn_PowerOff, T_PowerOn_PowerOff), (t_On4_Off3, T_On4_Off3),$
$(t_Off3_On4, T_Off3_On4), (t_default_Off3, T_default_Off3),$
$(t_default_Off7, T_default_Off7)\}$
$junctions = \{\}$

SimulationInstance

$v_airflow : \mathbb{N}$

$v_temp : \mathbb{R}$

InitSimulationInstance

$SimulationInstance'$

$v_airflow' = 0$

$v_temp' = 0$

SimulationData

$state_status : SID \rightarrow \mathbb{B}$

$state_history : SID \rightarrow SID$

$\mathrm{dom}\ state_status = \mathrm{dom}\ states$

$\mathrm{dom}\ state_history = \{j : \mathrm{ran}\ junctions \mid j.history = \mathbf{True} \bullet j.parent\}$

$\forall s : \mathrm{ran}\ states \mid s.decomposition = CLUSTER \bullet$

$\quad \#\{ss : \mathrm{ran}\ s.substates \mid state_status(ss) = \mathbf{True}\} \leq 1$

InitSimulationData

$SimulationData'$

$state_status' = \{n : \mathrm{dom}\ states \bullet n \mapsto \mathbf{False}\}$

$state_history' = \{n : \mathrm{dom}\ state_history' \bullet n \mapsto nullstate.identifier\}$

ActivateNoHistory

$\Delta SimulationData$

$x? : SID$

$x? \in \mathrm{dom}\ state_status$

$(parent\ (states\ x?)).history = \mathbf{False}$

$state_history' = state_history$

$state_status' = state_status \oplus \{x? \mapsto \mathbf{True}\}$

ActivateWithHistory

$\Delta SimulationData$
$x? : SID$

$x? \in \mathrm{dom}\ state_status$
$(parent\ (states\ x?)).history = \mathbf{True}$
$state_history' = state_history \oplus \{((states\ x?).parent) \mapsto x?\}$
$state_status' = state_status \oplus \{x? \mapsto \mathbf{True}\}$

$$Activate == (ActivateWithHistory \lor ActivateNoHistory) \land \Xi SimulationInstance$$

Deactivate

$\Delta SimulationData$
$\Xi SimulationInstance$
$x? : SID$

$x? \in \mathrm{dom}\ state_status$
$state_history' = state_history$
$state_status' = state_status \oplus \{x? \mapsto \mathbf{False}\}$

$$InitState == (InitSimulationInstance) \land (InitSimulationData)$$

$$\mathbf{state}\ Controller_state == (SimulationInstance) \land (SimulationData)$$

$$entryaction_Off3 \mathrel{\widehat=} (executeentryaction.(s_Off3) \longrightarrow \mathbf{Skip})$$
$$entryaction_On4 \mathrel{\widehat=} (executeentryaction.(s_On4) \longrightarrow \mathbf{Skip})$$
$$entryaction_FAN2 \mathrel{\widehat=} (executeentryaction.(s_FAN2) \longrightarrow \mathbf{Skip})$$
$$entryaction_SpeedValue \mathrel{\widehat=} (executeentryaction.(s_SpeedValue) \longrightarrow \mathbf{Skip})$$
$$entryaction_Off7 \mathrel{\widehat=} (executeentryaction.(s_Off7) \longrightarrow \mathbf{Skip})$$
$$entryaction_On8 \mathrel{\widehat=} (executeentryaction.(s_On8) \longrightarrow \mathbf{Skip})$$
$$entryaction_FAN1 \mathrel{\widehat=} (executeentryaction.(s_FAN1) \longrightarrow \mathbf{Skip})$$
$$entryaction_PowerOn \mathrel{\widehat=} (executeentryaction.(s_PowerOn) \longrightarrow \mathbf{Skip})$$
$$entryaction_PowerOff \mathrel{\widehat=} (executeentryaction.(s_PowerOff) \longrightarrow v_airflow := 0)$$
$$entryactions \mathrel{\widehat=} \left(\begin{array}{l} entryaction_Off3 \Box entryaction_On4 \Box entryaction_FAN2 \Box \\ entryaction_SpeedValue \Box entryaction_Off7 \Box entryaction_On8 \Box \\ entryaction_FAN1 \Box entryaction_PowerOn \Box entryaction_PowerOff \end{array}\right)$$

$duringaction_Off3 \widehat{=} (executeduringaction.(s_Off3)?ce \longrightarrow \mathbf{Skip})$
$duringaction_On4 \widehat{=} (executeduringaction.(s_On4)?ce \longrightarrow \mathbf{Skip})$
$duringaction_FAN2 \widehat{=} (executeduringaction.(s_FAN2)?ce \longrightarrow \mathbf{Skip})$
$duringaction_SpeedValue \widehat{=} (executeduringaction.(s_SpeedValue)?ce \longrightarrow$
$v_airflow := (b2r(state_status(s_On8)) + b2r(state_status(s_On4))))$
$duringaction_Off7 \widehat{=} (executeduringaction.(s_Off7)?ce \longrightarrow \mathbf{Skip})$
$duringaction_On8 \widehat{=} (executeduringaction.(s_On8)?ce \longrightarrow \mathbf{Skip})$
$duringaction_FAN1 \widehat{=} (executeduringaction.(s_FAN1)?ce \longrightarrow \mathbf{Skip})$
$duringaction_PowerOn \widehat{=} (executeduringaction.(s_PowerOn)?ce \longrightarrow \mathbf{Skip})$
$duringaction_PowerOff \widehat{=} (executeduringaction.(s_PowerOff)?ce \longrightarrow \mathbf{Skip})$

$$duringactions \widehat{=} \left(\begin{array}{l} duringaction_Off3 \Box duringaction_On4 \Box duringaction_FAN2 \Box \\ duringaction_SpeedValue \Box duringaction_Off7 \Box \\ duringaction_On8 \Box duringaction_FAN1 \Box \\ duringaction_PowerOn \Box duringaction_PowerOff \end{array}\right)$$

$exitaction_Off3 \widehat{=} (executeexitaction.(s_Off3) \longrightarrow \mathbf{Skip})$
$exitaction_On4 \widehat{=} (executeexitaction.(s_On4) \longrightarrow \mathbf{Skip})$
$exitaction_FAN2 \widehat{=} (executeexitaction.(s_FAN2) \longrightarrow \mathbf{Skip})$
$exitaction_SpeedValue \widehat{=} (executeexitaction.(s_SpeedValue) \longrightarrow \mathbf{Skip})$
$exitaction_Off7 \widehat{=} (executeexitaction.(s_Off7) \longrightarrow \mathbf{Skip})$
$exitaction_On8 \widehat{=} (executeexitaction.(s_On8) \longrightarrow \mathbf{Skip})$
$exitaction_FAN1 \widehat{=} (executeexitaction.(s_FAN1) \longrightarrow \mathbf{Skip})$
$exitaction_PowerOn \widehat{=} (executeexitaction.(s_PowerOn) \longrightarrow \mathbf{Skip})$
$exitaction_PowerOff \widehat{=} (executeexitaction.(s_PowerOff) \longrightarrow \mathbf{Skip})$

$$exitactions \widehat{=} \left(\begin{array}{l} exitaction_Off3 \Box exitaction_On4 \Box exitaction_FAN2 \Box \\ exitaction_SpeedValue \Box exitaction_Off7 \Box \\ exitaction_On8 \Box exitaction_FAN1 \Box \\ exitaction_PowerOn \Box exitaction_PowerOff \end{array}\right)$$

$conditionaction_On8_Off7 \widehat{=} (executeconditionaction.(t_On8_Off7) \longrightarrow \mathbf{Skip})$
$conditionaction_Off7_On8 \widehat{=} (executeconditionaction.(t_Off7_On8) \longrightarrow \mathbf{Skip})$
$conditionaction_PowerOff_PowerOn \widehat{=}$
$(executeconditionaction.(t_PowerOff_PowerOn) \longrightarrow \mathbf{Skip})$
$conditionaction_default_PowerOff \widehat{=}$
$(executeconditionaction.(t_default_PowerOff) \longrightarrow \mathbf{Skip})$
$conditionaction_PowerOn_PowerOff \widehat{=}$
$(executeconditionaction.(t_PowerOn_PowerOff) \longrightarrow \mathbf{Skip})$

$conditionaction_On4_Off3 \mathrel{\widehat=} (executeconditionaction.(t_On4_Off3) \longrightarrow \mathbf{Skip})$
$conditionaction_Off3_On4 \mathrel{\widehat=} (executeconditionaction.(t_Off3_On4) \longrightarrow \mathbf{Skip})$
$conditionaction_default_Off3 \mathrel{\widehat=} (executeconditionaction.(t_default_Off3) \longrightarrow \mathbf{Skip})$
$conditionaction_default_Off7 \mathrel{\widehat=} (executeconditionaction.(t_default_Off7) \longrightarrow \mathbf{Skip})$

$$conditionactions \mathrel{\widehat=} \left(\begin{array}{l} conditionaction_On8_Off7 \Box \\ conditionaction_Off7_On8 \Box \\ conditionaction_PowerOff_PowerOn \Box \\ conditionaction_default_PowerOff \Box \\ conditionaction_PowerOn_PowerOff \Box \\ conditionaction_On4_Off3 \Box \\ conditionaction_Off3_On4 \Box \\ conditionaction_default_Off3 \Box \\ conditionaction_default_Off7 \end{array}\right)$$

$transitionaction_On8_Off7 \mathrel{\widehat=} (executetransitionaction.(t_On8_Off7) \longrightarrow \mathbf{Skip})$
$transitionaction_Off7_On8 \mathrel{\widehat=} (executetransitionaction.(t_Off7_On8) \longrightarrow \mathbf{Skip})$
$transitionaction_PowerOff_PowerOn \mathrel{\widehat=}$
$\quad (executetransitionaction.(t_PowerOff_PowerOn) \longrightarrow \mathbf{Skip})$
$transitionaction_default_PowerOff \mathrel{\widehat=}$
$\quad (executetransitionaction.(t_default_PowerOff) \longrightarrow \mathbf{Skip})$
$transitionaction_PowerOn_PowerOff \mathrel{\widehat=}$
$\quad (executetransitionaction.(t_PowerOn_PowerOff) \longrightarrow \mathbf{Skip})$
$transitionaction_On4_Off3 \mathrel{\widehat=} (executetransitionaction.(t_On4_Off3) \longrightarrow \mathbf{Skip})$
$transitionaction_Off3_On4 \mathrel{\widehat=} (executetransitionaction.(t_Off3_On4) \longrightarrow \mathbf{Skip})$
$transitionaction_default_Off3 \mathrel{\widehat=} (executetransitionaction.(t_default_Off3) \longrightarrow \mathbf{Skip})$
$transitionaction_default_Off7 \mathrel{\widehat=} (executetransitionaction.(t_default_Off7) \longrightarrow \mathbf{Skip})$

$$transitionactions \mathrel{\widehat=} \left(\begin{array}{l} transitionaction_On8_Off7 \Box \\ transitionaction_Off7_On8 \Box \\ transitionaction_PowerOff_PowerOn \Box \\ transitionaction_default_PowerOff \Box \\ transitionaction_PowerOn_PowerOff \Box \\ transitionaction_On4_Off3 \Box \\ transitionaction_Off3_On4 \Box \\ transitionaction_default_Off3 \Box \\ transitionaction_default_Off7 \end{array}\right)$$

$$condition_On8_Off7 \widehat{=}$$
$$\left(\begin{array}{l}\mathbf{if}(v_temp <_{\mathcal{A}} 120) \neq 0 \longrightarrow evaluatecondition.(t_On8_Off7)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ ((v_temp <_{\mathcal{A}} 120) \neq 0) \longrightarrow evaluatecondition.(t_On8_Off7)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi}\end{array}\right)$$
$$condition_Off7_On8 \widehat{=}$$
$$\left(\begin{array}{l}\mathbf{if}(v_temp \geq_{\mathcal{A}} 120) \neq 0 \longrightarrow evaluatecondition.(t_Off7_On8)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ ((v_temp \geq_{\mathcal{A}} 120) \neq 0) \longrightarrow evaluatecondition.(t_Off7_On8)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi}\end{array}\right)$$

$$condition_PowerOff_PowerOn \widehat{=}$$
$$(evaluatecondition.(t_PowerOff_PowerOn)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$$
$$condition_default_PowerOff \widehat{=}$$
$$(evaluatecondition.(t_default_PowerOff)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$$
$$condition_PowerOn_PowerOff \widehat{=}$$
$$(evaluatecondition.(t_PowerOn_PowerOff)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$$

$$condition_On4_Off3 \widehat{=}$$
$$\left(\begin{array}{l}\mathbf{if}((v_temp <_{\mathcal{A}} 150) \neq 0) \longrightarrow evaluatecondition.(t_On4_Off3)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ ((v_temp <_{\mathcal{A}} 150) \neq 0) \longrightarrow evaluatecondition.(t_On4_Off3)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi}\end{array}\right)$$
$$condition_Off3_On4 \widehat{=}$$
$$\left(\begin{array}{l}\mathbf{if}((v_temp \geq_{\mathcal{A}} 150) \neq 0) \longrightarrow evaluatecondition.(t_Off3_On4)!(\mathbf{True}) \longrightarrow \mathbf{Skip} \\ [\!]\neg\ ((v_temp \geq_{\mathcal{A}} 150) \neq 0) \longrightarrow evaluatecondition.(t_Off3_On4)!(\mathbf{False}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi}\end{array}\right)$$
$$condition_default_Off3 \widehat{=} (evaluatecondition.(t_default_Off3)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$$
$$condition_default_Off7 \widehat{=} (evaluatecondition.(t_default_Off7)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$$
$$conditions \widehat{=} \left(\begin{array}{l}condition_On8_Off7 \Box condition_Off7_On8 \Box \\ condition_PowerOff_PowerOn \Box condition_default_PowerOff \Box \\ condition_PowerOn_PowerOff \Box condition_On4_Off3 \Box \\ condition_Off3_On4 \Box condition_default_Off3 \Box \\ condition_default_Off7\end{array}\right)$$

$trigger_On8_Off7 \mathrel{\widehat=}$
$\quad (checktrigger.(t_On8_Off7)?e \longrightarrow (result.(t_On8_Off7).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip}))$
$trigger_Off7_On8 \mathrel{\widehat=}$
$\quad (checktrigger.(t_Off7_On8)?e \longrightarrow (result.(t_Off7_On8).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip}))$
$trigger_PowerOff_PowerOn \mathrel{\widehat=} checktrigger.(t_PowerOff_PowerOn)?e \longrightarrow$

$$\left(\begin{array}{l} \mathbf{if}\ e = e_SWITCH \longrightarrow (result.(t_PowerOff_PowerOn).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip}) \\ [\!]\neg\ (e = e_SWITCH) \longrightarrow ((result.(t_PowerOff_PowerOn).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip})) \\ \mathbf{fi} \end{array}\right)$$

$trigger_default_PowerOff \mathrel{\widehat=} checktrigger.(t_default_PowerOff)?e \longrightarrow$
$\quad (result.(t_default_PowerOff).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$
$trigger_PowerOn_PowerOff \mathrel{\widehat=} checktrigger.(t_PowerOn_PowerOff)?e \longrightarrow$

$$\left(\begin{array}{l} \mathbf{if}\ e = e_SWITCH \longrightarrow (result.(t_PowerOn_PowerOff).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip}) \\ [\!]\neg\ (e = e_SWITCH) \longrightarrow ((result.(t_PowerOn_PowerOff).(e)!(\mathbf{False}) \longrightarrow \mathbf{Skip})) \\ \mathbf{fi} \end{array}\right)$$

$trigger_On4_Off3 \mathrel{\widehat=}$
$\quad checktrigger.(t_On4_Off3)?e \longrightarrow (result.(t_On4_Off3).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$
$trigger_Off3_On4 \mathrel{\widehat=}$
$\quad checktrigger.(t_Off3_On4)?e \longrightarrow (result.(t_Off3_On4).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$
$trigger_default_Off3 \mathrel{\widehat=}$
$\quad checktrigger.(t_default_Off3)?e \longrightarrow (result.(t_default_Off3).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$
$trigger_default_Off7 \mathrel{\widehat=}$
$\quad checktrigger.(t_default_Off7)?e \longrightarrow (result.(t_default_Off7).(e)!(\mathbf{True}) \longrightarrow \mathbf{Skip})$

$$triggers \mathrel{\widehat=} \left(\begin{array}{l} trigger_On8_Off7 \sqcap trigger_Off7_On8 \sqcap \\ trigger_PowerOff_PowerOn \sqcap trigger_default_PowerOff \sqcap \\ trigger_PowerOn_PowerOff \sqcap trigger_On4_Off3 \sqcap \\ trigger_Off3_On4 \sqcap trigger_default_Off3 \sqcap \\ trigger_default_Off7 \end{array}\right)$$

$getevents \mathrel{\widehat=} (events!(\langle e_SWITCH, e_CLOCK\rangle) \longrightarrow \mathbf{Skip})$
$getstate \mathrel{\widehat=} (state?x : (x \in \mathrm{dom}(states))!(states(x)) \longrightarrow \mathbf{Skip})$
$getjunction \mathrel{\widehat=} (junction?x : (x \in \mathrm{dom}(junctions))!(junctions(x)) \longrightarrow \mathbf{Skip})$
$gettransition \mathrel{\widehat=} (transition?x : (x \in \mathrm{dom}(transitions))!(transitions(x)) \longrightarrow \mathbf{Skip})$
$getchart \mathrel{\widehat=} (chart!(states(identifier)) \longrightarrow \mathbf{Skip})$

$$broadcast \mathrel{\widehat=} e : EVENT;\ dest : SID \bullet local_event!(e, states(dest)) \longrightarrow \mu X \bullet \left(\begin{array}{l} (AllActions\ ;\ X) \\ \Box \\ (end_local_execution \longrightarrow \mathbf{Skip}) \end{array} \right)$$

$$check \mathrel{\widehat=} \mathbf{res}\ erl : \mathbb{B} \bullet \mu X \bullet (InterfaceActions\ ;\ X \Box interrupt?x \longrightarrow erl := x)$$

$$status \mathrel{\widehat=} (status?x : (x \in \mathrm{dom}(state_status))!(state_status(x)) \longrightarrow \mathbf{Skip})$$

$$history \mathrel{\widehat=} (history?x : (x \in \mathrm{dom}(state_history))!(state_history(x)) \longrightarrow \mathbf{Skip})$$

$$activation \mathrel{\widehat=} (activate?x \longrightarrow \big(Activate\big))$$

$$deactivation \mathrel{\widehat=} (deactivate?x \longrightarrow \big(Deactivate\big))$$

$$ChartActions \mathrel{\widehat=} \left(\begin{array}{l} entryactions \Box duringactions \Box \\ exitactions \Box conditionactions \Box \\ transitionactions \end{array} \right);\ (end_action \longrightarrow \mathbf{Skip})$$

$$InterfaceActions \mathrel{\widehat=} \left(\begin{array}{l} getevents \Box getchart \Box getstate \Box \\ getjunction \Box gettransition \Box status \Box \\ history \Box activation \Box deactivation \end{array} \right)$$

$$Inputs \mathrel{\widehat=} (read_inputs \longrightarrow (i_temp?x \longrightarrow v_temp := x))$$

$$Outputs \mathrel{\widehat=} (write_outputs \longrightarrow (o_airflow!(v_airflow) \longrightarrow \mathbf{Skip}))$$

$$AllActions \mathrel{\widehat=} \left(\begin{array}{l} conditionactions \Box triggers \Box Inputs \Box \\ Outputs \Box ChartActions \Box InterfaceActions \end{array} \right)$$

$$\bullet \big(InitState\big);\ \mu X \bullet \left(\left(\mu Y \bullet \left(\begin{array}{l} (AllActions\ ;\ Y) \\ \Box \\ end_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right);\ X \right)$$

end

$$\mathbf{process}\ Controller \mathrel{\widehat=} (P_Controller \llbracket interface \cup \{\!| end_cycle |\!\} \rrbracket Simulator) \setminus interface$$

Appendix D

Circus model of the implementation of Air Controller Chart

section Air_impl **parents** $circus_toolkit, basic_toolkit$

$Air_IN_NO_ACTIVE_CHILD == 0$
$Air_IN_Off == 1$
$Air_IN_On == 2$
$Air_IN_PowerOff == 1$
$Air_IN_PowerOn == 2$
$Air_event_CLOCK == 1$
$Air_event_SWITCH == 0$
$CALL_EVENT == -1$

$BlockIO_Air == [airflow : \mathbb{N}]$

D_Work_Air

$$
\begin{array}{l}
is_active_c1_Air, is_c1_Air : \mathbb{N} \\
is_active_FAN2, is_FAN2 : \mathbb{N} \\
is_active_SpeedValue : \mathbb{N} \\
is_active_FAN1, is_FAN1 : \mathbb{N}
\end{array}
$$

$ExternalInputs_Air == [temp : \mathbb{R};\ inputevents : \operatorname{seq} \mathbb{B}]$
$ExternalOutputs_Air == [airflow : \mathbb{N}]$

$is_active_c1_Air : D_Work_Air \times \mathbb{N} \rightarrow D_Work_Air$

$$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_active_c1_Air(b, v) = \\ \langle\!| is_active_c1_Air == v, is_c1_Air == b.is_c1_Air, \\ is_active_FAN2 == b.is_active_FAN2, is_FAN2 == b.is_FAN2, \\ is_active_SpeedValue == b.is_active_SpeedValue, \\ is_active_FAN1 == b.is_active_FAN1, is_FAN1 == b.is_FAN1 |\!\rangle$$

$is_c1_Air : D_Work_Air \times \mathbb{N} \rightarrow D_Work_Air$

$$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_c1_Air(b, v) = \\ \langle\!| is_active_c1_Air == b.is_active_c1_Air, \\ is_c1_Air == v, is_active_FAN2 == b.is_active_FAN2, \\ is_FAN2 == b.is_FAN2, is_active_SpeedValue == b.is_active_SpeedValue, \\ is_active_FAN1 == b.is_active_FAN1, is_FAN1 == b.is_FAN1 |\!\rangle$$

$is_active_FAN2 : D_Work_Air \times \mathbb{N} \rightarrow D_Work_Air$

$$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_active_FAN2(b, v) = \\ \langle\!| is_active_c1_Air == b.is_active_c1_Air, \\ is_c1_Air == b.is_c1_Air, is_active_FAN2 == v, \\ is_FAN2 == b.is_FAN2, is_active_SpeedValue == b.is_active_SpeedValue, \\ is_active_FAN1 == b.is_active_FAN1, is_FAN1 == b.is_FAN1 |\!\rangle$$

$is_FAN2 : D_Work_Air \times \mathbb{N} \rightarrow D_Work_Air$

$$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_FAN2(b, v) = \\ \langle\!| is_active_c1_Air == b.is_active_c1_Air, \\ is_c1_Air == b.is_c1_Air, is_active_FAN2 == b.is_active_FAN2, \\ is_FAN2 == v, is_active_SpeedValue == b.is_active_SpeedValue, \\ is_active_FAN1 == b.is_active_FAN1, is_FAN1 == b.is_FAN1 |\!\rangle$$

$is_active_SpeedValue : D_Work_Air \times \mathbb{N} \rightarrow D_Work_Air$

$$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_active_SpeedValue(b, v) = \\ \langle\!| is_active_c1_Air == b.is_active_c1_Air, \\ is_c1_Air == b.is_c1_Air, is_active_FAN2 == b.is_active_FAN2, \\ is_FAN2 == b.is_FAN2, is_active_SpeedValue == v, \\ is_active_FAN1 == b.is_active_FAN1, is_FAN1 == b.is_FAN1 |\!\rangle$$

$is_active_FAN1 : D_Work_Air \times \mathbb{N} \to D_Work_Air$

$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_active_FAN1(b, v) =$
$\quad \langle\!| is_active_c1_Air == b.is_active_c1_Air,$
$\quad is_c1_Air == b.is_c1_Air, is_active_FAN2 == b.is_active_FAN2,$
$\quad is_FAN2 == b.is_FAN2, is_active_SpeedValue == b.is_active_SpeedValue,$
$\quad is_active_FAN1 == v, is_FAN1 == b.is_FAN1 |\!\rangle$

$is_FAN1 : D_Work_Air \times \mathbb{N} \to D_Work_Air$

$\forall b : D_Work_Air;\ v : \mathbb{N} \bullet is_FAN1(b, v) =$
$\quad \langle\!| is_active_c1_Air == b.is_active_c1_Air,$
$\quad is_c1_Air == b.is_c1_Air, is_active_FAN2 == b.is_active_FAN2,$
$\quad is_FAN2 == b.is_FAN2, is_active_SpeedValue == b.is_active_SpeedValue,$
$\quad is_active_FAN1 == b.is_active_FAN1, is_FAN1 == v |\!\rangle$

$airflow : BlockIO_Air \times \mathbb{N} \to BlockIO_Air$

$\forall b : BlockIO_Air;\ v : \mathbb{N} \bullet airflow(b, v) =$
$\quad \langle\!| airflow == v |\!\rangle$

$temp : ExternalInputs_Air \times \mathbb{R} \to ExternalInputs_Air$

$\forall b : ExternalInputs_Air;\ v : \mathbb{R} \bullet temp(b, v) =$
$\quad \langle\!| temp == v, inputevents == b.inputevents |\!\rangle$

$inputevents : ExternalInputs_Air \times (\mathrm{seq}\,\mathbb{B}) \to ExternalInputs_Air$

$\forall b : ExternalInputs_Air;\ v : \mathrm{seq}\,\mathbb{B} \bullet inputevents(b, v) =$
$\quad \langle\!| temp == b.temp, inputevents == v |\!\rangle$

Air_state

$_sfEvent_Air_ : \mathbb{Z}$
$Air_B : BlockIO_Air$
$Air_DWork : D_Work_Air$
$Air_U : ExternalInputs_Air$
$Air_Y : ExternalOutputs_Air$

$\mathbf{channel}\, in_FAN1 : Air_state$

$\mathbf{channel}\, out_FAN1 : \mathbb{N}$

$\mathbf{channel}\, input_event : (\mathrm{seq}\, \mathbb{B})$

$\mathbf{channel}\, i_temp : \mathbb{R}$

$\mathbf{channel}\, o_airflow : \mathbb{N}$

$\mathbf{process}\, Air \mathrel{\widehat=} \mathbf{begin}$

$\mathbf{state}\, Air_state$

$$
\begin{array}{l}
Air_FAN1 \mathrel{\widehat=} \\
\left(\begin{array}{l}
\mathbf{var}\, Air_B : BlockIO_Air;\ Air_DWork : D_Work_Air;\ Air_U : ExternalInputs_Air; \\
Air_Y : ExternalOutputs_Air;\ _sfEvent_Air_ : \mathbb{Z} \bullet \\
in_FAN1?s \longrightarrow Air_B, Air_DWork, Air_U, Air_Y, _sfEvent_Air_ := \\
\quad s.Air_B, s.Air_DWork, s.Air_U, s.Air_Y, s._sfEvent_Air_; \\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_FAN1 = Air_IN_Off \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ Air_U.temp \geq 120 \longrightarrow Air_DWork := is_FAN1(Air_DWork, Air_IN_On) \\
[\!]\neg\,(Air_U.temp \geq 120) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\, Air_DWork.is_FAN1 = Air_IN_On \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ Air_U.temp < 120 \longrightarrow Air_DWork := is_FAN1(Air_DWork, Air_IN_Off) \\
[\!]\neg\,(Air_U.temp < 120) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\neg\,(Air_DWork.is_FAN1 = Air_IN_Off \lor Air_DWork.is_FAN1 = Air_IN_On) \longrightarrow \\
\quad Air_DWork := is_FAN1(Air_DWork, Air_IN_Off) \\
\mathbf{fi}
\end{array}\right); \\
out_FAN1!(Air_DWork.is_FAN1) \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
FAN2 \mathrel{\widehat=} \\
\left(\begin{array}{l}
\mathbf{if}\ Air_DWork.is_FAN2 = Air_IN_Off \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ Air_U.temp \geq 150 \longrightarrow Air_DWork := is_FAN2(Air_DWork, Air_IN_On) \\
[\!]\neg\,(Air_U.temp \geq 150) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\, Air_DWork.is_FAN2 = Air_IN_On \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ Air_U.temp < 150 \longrightarrow Air_DWork := is_FAN2(Air_DWork, Air_IN_Off) \\
[\!]\neg\,(Air_U.temp < 150) \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right) \\
[\!]\neg\,(Air_DWork.is_FAN2 = Air_IN_Off \lor Air_DWork.is_FAN2 = Air_IN_On) \longrightarrow \\
\quad Air_DWork := is_FAN2(Air_DWork, Air_IN_Off) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
Air_chartstep_c1_Air \mathrel{\widehat=} \\
\left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_active_c1_Air = 0 \longrightarrow \\
\left(\begin{array}{l}
Air_DWork := is_active_c1_Air(Air_DWork, 1); \\
Air_DWork := is_c1_Air(Air_DWork, Air_IN_PowerOff); \\
Air_B := airflow(Air_B, 0)
\end{array}\right) \\
[\!]\, \neg\ (Air_DWork.is_active_c1_Air = 0) \longrightarrow \\
\left(\begin{array}{l}
\textbf{if}\ Air_DWork.is_c1_Air = Air_IN_PowerOff \longrightarrow \\
\left(\begin{array}{l}
\textbf{if}\ _sfEvent_Air_ = Air_event_SWITCH \longrightarrow \\
\left(\begin{array}{l}
Air_DWork := is_c1_Air(Air_DWork, Air_IN_PowerOn); \\
Air_DWork := is_active_FAN1(Air_DWork, 1); \\
Air_DWork := is_FAN1(Air_DWork, Air_IN_Off); \\
Air_DWork := is_active_FAN2(Air_DWork, 1); \\
Air_DWork := is_FAN2(Air_DWork, Air_IN_Off); \\
Air_DWork := is_active_SpeedValue(Air_DWork, 1)
\end{array}\right) \\
[\!]\, \neg\ (_sfEvent_Air_ = Air_event_SWITCH) \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right) \\
[\!]\, Air_DWork.is_c1_Air = Air_IN_PowerOn \longrightarrow \\
\left(\begin{array}{l}
\textbf{if}\ _sfEvent_Air_ = Air_event_SWITCH \longrightarrow \\
\left(\begin{array}{l}
Air_DWork := is_active_SpeedValue(Air_DWork, 0); \\
Air_DWork := is_FAN2(Air_DWork, Air_IN_NO_ACTIVE_CHILD); \\
Air_DWork := is_active_FAN2(Air_DWork, 0); \\
Air_DWork := is_FAN1(Air_DWork, Air_IN_NO_ACTIVE_CHILD); \\
Air_DWork := is_active_FAN1(Air_DWork, 0); \\
Air_DWork := is_c1_Air(Air_DWork, Air_IN_PowerOff); \\
Air_B := airflow(Air_B, 0)
\end{array}\right) \\
[\!]\, \neg\ (_sfEvent_Air_ = Air_event_SWITCH) \longrightarrow \\
\left(\begin{array}{l}
in_FAN1!(\theta\, Air_state) \longrightarrow \textbf{Skip}; \\
FAN2; \\
out_FAN1?f1 \longrightarrow Air_DWork := is_FAN1(Air_DWork, f1); \\
Air_B := airflow(Air_B, \\
\qquad (\textbf{if}\ Air_DWork.is_FAN1 = Air_IN_On\ \textbf{then}\ 1\ \textbf{else}\ 0) + \\
\qquad (\textbf{if}\ Air_DWork.is_FAN2 = Air_IN_On\ \textbf{then}\ 1\ \textbf{else}\ 0))
\end{array}\right) \\
\textbf{fi}
\end{array}\right) \\
[\!]\, \neg\ (Air_DWork.is_c1_Air = Air_IN_PowerOff \lor \\
\qquad Air_DWork.is_c1_Air = Air_IN_PowerOn) \longrightarrow \\
\qquad \left(\begin{array}{l}
Air_DWork := is_c1_Air(Air_DWork, Air_IN_PowerOff); \\
Air_B := airflow(Air_B, 0)
\end{array}\right) \\
\textbf{fi}
\end{array}\right) \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

$$Air_output \mathrel{\widehat=} \left(\begin{array}{l} \left(\begin{array}{l} \mathbf{if}\ Air_U.inputevents(1) = \mathbf{True} \longrightarrow \\ \qquad _sfEvent_Air_ := Air_event_SWITCH\ ;\ Air_chartstep_c1_Air \\ [\!]\neg\ (Air_U.inputevents(1) = \mathbf{True}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right); \\ \left(\begin{array}{l} \mathbf{if}\ Air_U.inputevents(2) = \mathbf{True} \longrightarrow \\ \qquad _sfEvent_Air_ := Air_event_CLOCK\ ;\ Air_chartstep_c1_Air \\ [\!]\neg\ (Air_U.inputevents(2) = \mathbf{True}) \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right); \\ Air_Y := Air_B \end{array} \right)$$

$$MdlInitialize \mathrel{\widehat=} \left(\begin{array}{l} _sfEvent_Air_ := CALL_EVENT; \\ Air_DWork := is_active_FAN1(Air_DWork, 0); \\ Air_DWork := is_FAN1(Air_DWork, 0); \\ Air_DWork := is_active_FAN2(Air_DWork, 0); \\ Air_DWork := is_FAN2(Air_DWork, 0); \\ Air_DWork := is_active_SpeedValue(Air_DWork, 0); \\ Air_DWork := is_active_c1_Air(Air_DWork, 0); \\ Air_DWork := is_c1_Air(Air_DWork, 0); \\ Air_B := airflow(Air_B, 0) \end{array} \right)$$

$$FAN1 \mathrel{\widehat=} \mu X \bullet Air_FAN1$$

$$read_inputs \mathrel{\widehat=} \left(\begin{array}{l} input_event?es \longrightarrow Air_U := inputevents(Air_U, es); \\ i_temp?v \longrightarrow Air_U := temp(Air_U, v) \end{array} \right)$$

$$write_outputs \mathrel{\widehat=} o_airflow!(Air_Y.airflow) \longrightarrow \mathbf{Skip}$$

$$ExecuteChart \mathrel{\widehat=} MdlInitialize\ ;\ \left(\begin{array}{l} \mu X \bullet read_inputs\ ;\ Air_output; \\ write_outputs\ ;\ end_cycle \longrightarrow X \end{array} \right)$$

$$\bullet \left(\begin{array}{c} ExecuteChart \\ [\![\{_sfEvent_Air_, Air_U, Air_B, Air_DWork, Air_Y\} \mid \{\!| in_FAN1, out_FAN1 |\!\} \mid \{\}]\!] \\ FAN1 \end{array} \right)$$

end

Appendix E

Novel refinement laws

We present here, in alphabetical order, the novel refinement laws of *Circus* that we need.

Law[alt-alt-dist]

$$\left(\begin{array}{l} \mathbf{if}\ g_1 \longrightarrow A_1 \\ \square\ g_2 \longrightarrow \left(\begin{array}{l} \mathbf{if}\ g_3 \longrightarrow A_2 \\ \square\ g_4 \longrightarrow A_3 \\ \mathbf{fi} \end{array}\right) \\ \mathbf{fi} \end{array}\right) = \left(\begin{array}{l} \mathbf{if}\ g_1 \longrightarrow A_1 \\ \square\ g_2 \land g_3 \longrightarrow A_2 \\ \square\ g_2 \land g_4 \longrightarrow A_3 \\ \mathbf{fi} \end{array}\right)$$

provided

- $g_1 \lor g_2$;
- $g_1 \Rightarrow \neg\, g_2$;
- $g_3 \lor g_4$;
- $g_3 \Rightarrow \neg\, g_4$.

Law[alt-assump-intro]

$$(\mathbf{if}\ b_1 \longrightarrow A_1 \ \square\ b_2 \longrightarrow A_2\ \mathbf{fi}) \quad = \quad (\mathbf{if}\ b_1 \longrightarrow \{b_1\}\ ;\ A_1 \ \square\ b_2 \longrightarrow \{b_2\}\ ;\ A_2\ \mathbf{fi})$$

provided

- $b_1 \lor b_2$;
- $b_1 \Rightarrow \neg\, b_2$.

Law[alt-elim]

$(\textbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \textbf{fi}) \quad = \quad A_1$

provided

- $b_1 \vee b_2$;
- $b_1 \Rightarrow \neg\, b_2$;
- $b_1 \Leftrightarrow \textbf{True}$.

Law[alt-guard-rew]

$(\textbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \textbf{fi}) \quad = \quad (\textbf{if}\ b_3 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \textbf{fi})$

provided

- $b_1 \vee b_2$;
- $b_1 \Rightarrow \neg\, b_2$;
- $b_1 \Leftrightarrow b_3$.

Law[alt-hide-dist]

$$(\textbf{if}\ p \longrightarrow A \ [\!]\ \neg\, p \longrightarrow B\ \textbf{fi}) \setminus cs \quad = \quad (\textbf{if}\ p \longrightarrow A \setminus cs \ [\!]\ \neg\, p \longrightarrow B \setminus cs\ \textbf{fi})$$

Law[alt-par-dist]

$(\textbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \textbf{fi}) \ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ B$
$=$
$(\textbf{if}\ b_1 \longrightarrow A_1 \ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ B \ [\!]\ b_2 \longrightarrow A_2 \ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ B\ \textbf{fi})$

provided

- $initials(B) \subseteq cs$
- B *is deterministic.*

Law[alt-seq-dist]

$(\textbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \textbf{fi})\ ;\ B \quad = \quad (\textbf{if}\ b_1 \longrightarrow A_1\ ;\ B \ [\!]\ b_2 \longrightarrow A_2\ ;\ B\ \textbf{fi})$

provided

- $b_1 \vee b_2$
- $b_1 \Rightarrow \neg\, b_2$

Law[alt-var-dist-both]

$(\mathbf{if}\ b_1 \longrightarrow (\mathbf{var}\ x : T \bullet A) [] b_2 \longrightarrow (\mathbf{var}\ x : T \bullet B)\ \mathbf{fi})$
$=$
$\mathbf{var}\ x : T \bullet (\mathbf{if}\ b_1 \longrightarrow A [] b_2 \longrightarrow B\ \mathbf{fi})$

provided

- $b_1 \lor b_2$;
- $b_1 \Rightarrow \neg\, b_2$.

Law[assign-assump-intro]

$v := e = v := e\ ;\ \{v = e\}$

provided $v \notin FV(e)$

Law[assign-par-dist]

$$(v := e\ ;\ A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B \quad = \quad v := e\ ;\ (A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B)$$

provided

- $v \in ns_1$
- $v \cap usedV(A_2) = \varnothing$

Law[assign-schema-conv]

$$\{inv\}\ ;\ v := e \quad = \quad [\Delta S \mid vs' = vs \land v' = e]$$

where

- $S == [dv, dvs \mid inv]$
- *dv is the declaration of v*
- *dvs is the declaration of the remaining variables*
- *inv is the state invariant*

provided $inv \Rightarrow inv \land (\exists\, d' \bullet inv' \land vs' = vs \land v' = e)$

Law[assign-seq-col]

$(x := e_1\ ;\ x := e_2) = x := e_2[e_1/x]$

Law[assign-seq-com]

$(x := e\ ;\ A) = (A[e/x]\ ;\ x := e)$

provided $x \notin wrtV(A)$

Law[assign-seq-dist]

$(x := e\ ;\ A) \quad = \quad (x := e\ ;\ A[e/x])$

provided $x \notin wrtV(A)$

Law[assump-alt-dist]

$\{g\}\ ;\ (\mathbf{if}\ b_1 \longrightarrow A_1 \mathbin{[\!]} b_2 \longrightarrow A_2\ \mathbf{fi}) \quad = \quad \{g\}\ ;\ (\mathbf{if}\ b_1 \longrightarrow \{g\}\ ;\ A_1 \mathbin{[\!]} b_2 \longrightarrow \{g\}\ ;\ A_2\ \mathbf{fi})$

provided

- $b_1 \lor b_2$;
- $b_1 \Rightarrow \neg\, b_2$.

Law[assump-alt-dist-move]

$\{g\}\ ;\ (\mathbf{if}\ b_1 \longrightarrow A_1 \mathbin{[\!]} b_2 \longrightarrow A_2\ \mathbf{fi}) \quad = \quad (\mathbf{if}\ b_1 \longrightarrow \{g\}\ ;\ A_1 \mathbin{[\!]} b_2 \longrightarrow \{g\}\ ;\ A_2\ \mathbf{fi})$

provided

- $b_1 \lor b_2$;
- $b_1 \Rightarrow \neg\, b_2$.

Law[assump-assign-dist]

$\{g\}\ ;\ v := e \quad = \quad v := e\ ;\ \{g\}$

provided $g \land (v' = e) \Rightarrow g'$

Law[assump-rec-dist]

$\{g\}\ ;\ (\mu X \bullet A\ ;\ X) \ \sqsubseteq\ \{g\}\ ;\ (\mu X \bullet \{g\}\ ;\ A\ ;\ X)$

provided $\{g\}\ ;\ A \sqsubseteq A\ ;\ \{g\}$

Law[int-seq-dist]

$$(A\;;\;B) \mathbin{|\!|\!|[} ns_1 \mid ns_2 \mathbin{]\!|\!|\!|} C \quad = \quad A\;;\;(B \mathbin{|\!|\!|[} ns_1 \mid ns_2 \mathbin{]\!|\!|\!|} C)$$

provided

- $usedC(A) = \varnothing$;
- $wrtV(A) \subseteq ns_1$;
- $wrtV(A) \cap usedV(C) = \varnothing$.

Law[prefix-hide-dist-1]

$$(c \longrightarrow A) \setminus cs = c \longrightarrow (A \setminus cs)$$

provided $c \notin cs$

Law[prefix-hide-dist-2]

$$(c \longrightarrow A) \setminus cs = A \setminus cs$$

provided $c \in cs$

Law[prefix-par-dist]

$$\begin{array}{lcl}((c \longrightarrow A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B) & = & c \longrightarrow (A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B)\\ ((c?x \longrightarrow A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B) & = & c?x \longrightarrow (A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B)\\ ((c.e \longrightarrow A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B) & = & c.e \longrightarrow (A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B)\end{array}$$

provided

- $c \notin cs$
- $x \notin usedV(B)$
- $initials(B) \subseteq cs$
- B *is deterministic*

Law[seq-assign-conv]

$$[\Delta S \mid c_1' = e_1 \wedge \ldots \wedge c_m' = e_m \wedge c_{m+1}' = c_{m+1} \wedge \ldots \wedge c_n' = c_n] \quad \sqsubseteq \quad c_1 := e_1\;;\;\ldots\;;\;c_m := e_m$$

where

- $S = [d \mid inv]$
- $c_1, \ldots, c_n$ *are state components (elements of* αd*)*

syntactic restriction αd *and* $\alpha d'$ *are not free in* $e_1, \ldots, e_n$.
provided $inv[e_1, \ldots, e_m / c_1, \ldots, c_m]$

Law[tail-rec-hide-dist]

$$(\mu X \bullet A \; ; \; X) \setminus cs \quad = \quad \mu X \bullet A \setminus cs \; ; \; X$$

provided $A \setminus cs$ *is deterministic*

Law[tail-rec-seq-dist]

$$(\mu X \bullet \textbf{if}\; p \longrightarrow A \; ; \; X \; [] \; q \longrightarrow \textbf{Skip}\; \textbf{fi}) \; ; \; B \quad = \quad (\mu X \bullet \textbf{if}\; p \longrightarrow A \; ; \; X \; [] \; q \longrightarrow B \; \textbf{fi})$$

provided

- $p \vee q$;
- $p \Rightarrow \neg\, q$.

Law[unique-fixed-point]

$A = \mu X \bullet F(X)$

provided

- F *is deterministic;*
- $F(A) = A$.

Law[var-alt-dist]

$(\textbf{if}\; p \longrightarrow (\textbf{var}\; x : T \bullet A) \; [] \; q \longrightarrow B \; \textbf{fi})$
$=$
$\textbf{var}\; x : T \bullet (\textbf{if}\; p \longrightarrow A \; [] \; q \longrightarrow B \; \textbf{fi})$

provided

- $p \vee q$
- $p \Rightarrow \neg\, q$
- $x \notin FV(B, p, q)$

Law[var-assign-elim]

$(\textbf{var}\; x : T \bullet A \; ; \; x := e) = A$

provided $x \notin fv(A)$

Law[var-assign-intro]

$$A \quad = \quad (\mathbf{var}\, y : T \bullet y := e ;\ A)$$

provided $y \notin FV(A)$

Law[var-int-dist]

$$(\mathbf{var}\, x : T \bullet A) \,[\![\, ns_1 \mid ns_2 \,]\!]\, B \quad = \quad (\mathbf{var}\, x : T \bullet (A \,[\![\, ns_1 \cup \{x\} \mid ns_2 \,]\!]\, B))$$

provided $x \notin FV(B) \cup ns_1 \cup ns_2$

Law[var-iter-seq-ext]

$$(\textstyle; \ i : T_i \bullet (\mathbf{var}\, v : T_v \bullet A)) \quad = \quad (\mathbf{var}\, v : T_v \bullet (; \ i : T_i \bullet A))$$

Law[var-prefix-ext]

$$c \longrightarrow (\mathbf{var}\, x : T \bullet A) \quad = \quad (\mathbf{var}\, x : T \bullet c \longrightarrow A)$$

provided $x \notin FV(c)$

Law[var-rec-hide-dist]

$$(\mu X \bullet A ;\ X ;\ B) \setminus cs \quad = \quad (\mu X \bullet A \setminus cs ;\ X ;\ B \setminus cs)$$

provided $A \setminus cs ;\ X ;\ B \setminus cs$ *is deterministic*

Law[var-rename]

$$(\mathbf{var}\, x : T \bullet A) = (\mathbf{var}\, y : T \bullet A[y/x])$$

provided $y \notin FV(A)$

Law[var-seq-ext-left]

$$A ;\ (\mathbf{var}\, x : T \bullet B) \quad = \quad (\mathbf{var}\, x : T \bullet A ;\ B)$$

provided $x \notin FV(A)$

Law[var-seq-ext-right]

$$(\mathbf{var}\, x : T \bullet A) ;\ B = (\mathbf{var}\, x : T \bullet A ;\ B)$$

provided $x \notin FV(B)$

Law[var-tail-alt-rec-hide-dist]

$(\mu X \bullet (\textbf{if}\ p \longrightarrow A\ ;\ X \ [\!]\ \neg\ p \longrightarrow B\ \textbf{fi})) \setminus cs$

$=$

$(\mu X \bullet \textbf{if}\ p \longrightarrow A \setminus cs\ ;\ X \ [\!]\ \neg\ p \longrightarrow B \setminus cs\ \textbf{fi})$

provided $(\textbf{if}\ p \longrightarrow A \setminus cs\ ;\ X \ [\!]\ \neg\ p \longrightarrow B \setminus cs\ \textbf{fi})$ *is deterministic.*

Law[var-tail-rec-ext]$(\mu X \bullet \textbf{var}\ v : T \bullet A\ ;\ X) \sqsubseteq \textbf{var}\ v : T \bullet (\mu X \bullet A\ ;\ X)$

The above is a refinement law because in the action where the variable block is local to the recursion, a new arbitrary value is given to the variable at each iteration. On the other hand, in the action where the recursion is local to the variable block, the value of the variable at each iteration is always that at the end of the previous iteration (and arbitrary just in the first iteration).

Bibliography

[1] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

[2] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010. ISBN 978-0-521-89556-9.

[3] M. M. Adams and P. B. Clayton. ClawZ: Cost-Effective Formal Verification for Control Systems. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *LNCS*. Springer-Verlag, 2005. ISBN 3-540-29797-9. doi: 10.1007/11576280_32.

[4] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: control laws in Z. In *ICFEM*, 2000. pp. 169.

[5] R.-J. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Åbo Akademi, Department of Computer Science, Helsinki, Finland, 1978. Report A–1978–4.

[6] C. Banphawatthanarak and B. H. Krogh. Verification of stateflow diagrams using smv: sf2smv 2.0. Technical Report CMU-ECE-2000-020, Carnegie Mellon University, 2000.

[7] C. Banphawatthanarak, B. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 581–586, August 1999.

[8] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321136160.

[9] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[10] P. Boström and L. Morel. Mode-Automata in Simulink/Stateflow. Technical Report 772, Turku Centre for Computer Science, 2006.

[11] M. Butler and M. Leuschel. Combining CSP and B for Specification and Property Verification. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods*

2005, number LNCS 3582, pages 221–236. Springer-Verlag, January 2005. URL http://eprints.ecs.soton.ac.uk/10388/.

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In R. Alur and I. Lee, editors, *EMSOFT'03 LNCS*. Springer-Verlag, 2003.

[13] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications. *ACM SIGPLAN Notices*, 38(7):153–162, 2003. ISSN 0362-1340. doi: 10.1145/780731.780754.

[14] A. Cavalcanti, P. Clayton, and C. O'Halloran. From control law diagrams to Ada via *Circus*. *Formal Aspects of Computing*, pages 1–48, 2011. ISSN 0934-5043. doi: 10.1007/s00165-010-0170-3.

[15] A. L. C. Cavalcanti. *A refinement calculus for Z*. PhD thesis, Oxford University Computing Laboratory, 1997.

[16] A. L. C. Cavalcanti. Stateflow Diagrams in *Circus*. *Electronic Notes in Theoretical Computer Science*, 240:23–41, 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.05.043.

[17] A. L. C. Cavalcanti and P. Clayton. Verification of Control Systems using *Circus*. In *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 269 – 278. IEEE Computer Society, 2006.

[18] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.

[19] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in *Circus*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *LNCS*, pages 253 – 268. Springer-Verlag, 2005.

[20] C. Chen. Formal Analysis for Stateflow Diagrams. In *Proceedings of the Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, pages 102 –109, june 2010. doi: 10.1109/SSIRI-C.2010.29.

[21] M. L. Crane and J. Dingel. UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal. *Software and Systems Modeling*, 6, 2007. doi: 10.1007/s10270-006-0042-8.

[22] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. ISSN 0001-0782. doi: 10.1145/360933.360975.

[23] P. V. Eijk and M. Diaz, editors. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989. ISBN 0444872671.

[24] Esterel Technologies, Inc. Scade suite$^{\text{TM}}$. `http://www.esterel-technologies.com/`.

[25] European Committee for Electrotechnical Standardization (CENELEC). Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, March 2001.

[26] A. Evans. An Improved Recipe for Specifying Reactive Systems in Z. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notationes*, pages 275–294, London, UK, 1997. Springer-Verlag. ISBN 3-540-62717-0.

[27] H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *ICFEM*, pages 52–65, 2005.

[28] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In *FMOODS '97: Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 423–438, London, UK, UK, 1997. Chapman & Hall, Ltd. ISBN 0-412-82040-4.

[29] Formal Systems (Europe) Ltd. Process Behaviour Explorer, 2003. `www.fsel.com`.

[30] Formal Systems (Europe) Ltd. Failures-Divergence Refinement, 2010. `www.fsel.com`.

[31] A. F. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 115 – 130. Springer-Verlag, 2006.

[32] L. Freitas. *Model-checking Circus*. PhD thesis, Department of Computer Science, The University of York, UK, 2005. YCST-2005/11.

[33] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. In *Proceedings of the IEEE*, volume 79, pages 1305 – 1320, 1991.

[34] G. Hamon. A denotational semantics for Stateflow. In W. Wolf, editor, *EMSOFT*, pages 164–172. ACM, 2005. ISBN 1-59593-091-4. doi: 10.1145/1086228.1086260.

[35] G. Hamon and J. Rushby. An operational semantics for Stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering:7th International Conference (FASE)*, volume 2984 of *LNCS*, pages 229–243, Barcelona, Spain, 2004. Springer-Verlag.

[36] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[37] D. Harel and H. Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) - Preliminary Version. In *SoftSpez Final Report*, pages 325–354, 2004.

[38] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996. ISSN 1049-331X.

[39] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, Inc., New York, NY, USA, 1998. ISBN 0070262055.

[40] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, New York, 1987. IEEE Press.

[41] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. ISBN 0-13-153271-5.

[42] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[43] J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes, data and time. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 245–266, London, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7.

[44] G. J. Holtzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[45] G. J. Holtzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, volume 23, May 1997.

[46] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[47] International Electrotechnical Commission (IEC). Functional safety of electrical/-electronic/programmable electronic safety related systems – Part 1: General requirements (IEC 61508-1), 1997.

[48] ISO. Road vehicles – Funcional safety (ISO 26262), 2011.

[49] N. Izerrouken, X. Thirioux, M. Pantel, and M. Strecker. Certifying an Automated Code Generator Using Formal Tools. In *ERTS'08, 4th European symposium on Real Time Systems*, 2008.

[50] JFlex. JFlex - The Fast Scanner Generator for Java, 2009. `http://jflex.de/` [Last accessed 2009].

[51] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[52] M. P. Jones. jacc: just another compiler compiler for Java, 2004. `http://web.cecs.pdx.edu/~mpj/jacc/` [Last accessed 2009].

[53] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of uml statechart diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V. ISBN 0-7923-8429-6.

[54] M. Leuschel and M. Butler. ProB: A Model Checker for B. In A. Keijiro, S. Gnesi, and M. Dino, editors, *Formal Methods Europe 2003*, volume 2805, pages 855–874. Springer-Verlag, LNCS, 2003. URL `http://eprints.ecs.soton.ac.uk/8341/`.

[55] J. Lilius and I. P. Paltor. The Semantics Of UML State Machines. Technical Report 273, Turku Centre and Computer Science, May 1999.

[56] B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 95–104, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8368-6.

[57] P. Malik and M. Utting. CZT: A Framework for Z Tools. In *ZB. Lecture*, pages 65–84. Springer-Verlag, 2005.

[58] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46: 219–254, March 2003. ISSN 0167-6423. doi: 10.1016/S0167-6423(02)00093-X.

[59] J. McDermid, A. Galloway, S. Burton, J. Clark, I. Toyn, N. Tracey, and S. Valentine. Towards industrially applicable formal methods: Three small steps, and one giant leap. In *Proceedings of the International Conference on Formal Engineering Methods*, pages 76–88. Press, 1998.

[60] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On formal semantics of statecharts as supported by statemate. In *In Second BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, 1997.

[61] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.

[62] Ministry of Defence. Requirements for safety related software in defence equipment (Def Stan 00-55), 1997.

[63] A. Miyazawa. Source files for a parallel implementation of the Stateflow chart in *air.mdl*, 2012. Available from `http://www-users.cs.york.ac.uk/~alvarohm/thesis/`.

[64] A. Miyazawa. Formal syntax of *Circus*, 2012. Available from `http://www-users.cs.york.ac.uk/~alvarohm/thesis/`.

[65] A. Miyazawa. Formal syntax of Stateflow charts, 2012. Available from `http://www-users.cs.york.ac.uk/~alvarohm/thesis/`.

[66] A. Miyazawa. Formal syntax of Z, 2012. Available from `http://www-users.cs.york.ac.uk/~alvarohm/thesis/`.

[67] A. Miyazawa. Formal translation rules for Stateflow charts, 2012. Available from `http://www-users.cs.york.ac.uk/~alvarohm/thesis/`.

[68] A. Miyazawa and A. L. C. Cavalcanti. Towards the formal verification of implementations of Stateflow Diagrams. Technical Report YCS-2010-449, University of York, 2010.

[69] A. Miyazawa and A. L. C. Cavalcanti. A formal semantics of Stateflow charts. Technical Report YCS-2011-461, University of York, 2011.

[70] A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 2011. doi: 10.1016/j.scico.2011.07.007.

[71] A. Miyazawa and A. L. C. Cavalcanti. Refinement-based verification of sequential implementations of Stateflow charts. In J. Derrick, E. Boiten, and S. Reeves, editors, *Proceedings 15th International Refinement Workshop*, volume 55, pages 65–83. EPTCS, 2011. doi: 10.4204/EPTCS.55.5.

[72] A. Miyazawa and A. L. C. Cavalcanti. Refinement-based verifications of implementations of Stateflow charts. 2012. (to be submitted).

[73] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[74] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[75] M. Y. Ng and M. J. Butler. Towards formalizing UML state diagrams in CSP. In *SEFM*, page 138. IEEE Computer Society, 2003. ISBN 0-7695-1949-0.

[76] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.

[77] M. V. M. Oliveira and A. L. C. Cavalcanti. ArcAngelC: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, **214C**:203 – 229, 2008. doi: 10.1016/j.entcs.2008.06.010. © Elsevier B. V.

[78] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Formal development of industrial-scale systems. *Innovations in Systems and Software Engineering*, 1(2): 125 – 146, 2005.

[79] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, **21**(1):3 – 32, 2007. doi: 10.1007/s00165-007-0052-5.

[80] M. V. M. Oliveira, A. C. Gurgel, and C. G. de Castro. CRefine: Support for the *Circus* Refinement Calculus. In A. Cerone and S. Gruner, editors, *6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 281–290. IEEE Computer Society Press, 2008. ISBN 978-0-7695-3437-4.

[81] OMG. UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG), August 2005.

[82] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Theoretical Aspects of Computer Software*. Springer-Verlag, 1991.

[83] Radio Technical Commission for Aeronautics (RTCA). Software Considerations in Airborne Systems and Equipment Certification (DO-178B), 1992.

[84] R. Ramos, A. Sampaio, and A. Mota. A semantics for UML-RT active classes via mapping into *Circus*. In M. Steffen and G. Zavattaro, editors, *FMOODS*, volume 3535 of *LNCS*, pages 99–114. Springer-Verlag, 2005. ISBN 3-540-26181-8. doi: 10.1007/11494881_7.

[85] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, New York, 1998. ISBN 0-13-674409-5. Oxford.

[86] A. E. Rugina, D. Thomas, X. Olive, and G. Veran. Gene-Auto: Automatic software code generation for real-time embedded systems. In *DASIA 2008, the International Space System Engineering Conference*, 2008.

[87] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In G. C. Buttazzo, editor, *EMSOFT*, pages 259–268. ACM, 2004. ISBN 1-58113-860-1. doi: 10.1145/1017753.1017795.

[88] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999. ISBN 0471623733.

[89] S. Schneider and J. Davies. *A brief history of Timed CSP*. Theoretical Computer Science. 1995.

[90] E. Sekerinski and R. Zurob. Translating Statecharts to B. In Springer-Verlag, editor, *IFM 2002*, volume 2335 of *LNCS*, pages 128–144, 2002.

[91] G. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, pages 62–81, London, UK, 1997. Springer-Verlag. ISBN 3-540-63533-5.

[92] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8684-1.

[93] G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems—An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/A:1011269103179.

[94] C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, January 2006. URL `http://eprints.ecs.soton.ac.uk/10169/`.

[95] C. Snook, V. Savicks, and M. Butler. Verification of UML models by translation to UML-B. *LNCS*, 6957:251, 2011. URL `http://eprints.ecs.soton.ac.uk/22921/`.

[96] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *CAV*, pages 709–714, 2009.

[97] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, page 283, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8002-4.

[98] The MathWorks,Inc. Stateflow and Stateflow Coder 7 User's Guide, October 2008. Available at `http://www.mathworks.com/products/stateflow` [Last accessed October 2008].

[99] The MathWorks,Inc. Simulink, October 2009. Available at `http://www.mathworks.com/products/simulink` [Last accessed October 2008].

[100] The MathWorks,Inc. Real-Time Workshop 7 User's Guide, March 2010. Available at `http://www.mathworks.com/products/rtw` [Last accessed March 2010].

[101] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002. URL `http://www.csl.sri.com/users/tiwari/html/stateflow.html`.

[102] A. Toom, T. Naks, M. Panter, M. Grandriau, and I. Wati. Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos. In *ERTS'08, 4th European symposium on Real Time Systems*, 2008.

[103] I. Toyn. Simulink/Stateflow Analyser user's manual. Technical report, Department of Computer Science, University of York, 2005.

[104] I. Toyn and A. Galloway. Proving properties of Stateflow models using ISO Standard Z and CADiZ. In *In ZB-2005, vol. 3455 of LNCS*, pages 104–123, 2005.

[105] H. Treharne and S. Schneider. Using a process algebra to control B OPERATIONS. In *IFM'99 1st International Conference on Integrated Formal Methods*, pages 437–457, York, June 1999. Springer-Verlag. URL `http://www.cs.rhbnc.ac.uk/~helent`.

[106] TTA-Group. TTA – A Dependable Real-Time Communication Platform for Safety-Relevant Applications, 2007. URL `http://www.ttagroup.org/technology/tta.htm`.

[107] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, V1:130–141, 2002.

[108] M. W. Whalen. A Parametric Structural Operational Semantics for Stateflow, UML Statecharts, and Rhapsody. Technical Report 2010-1, University of Minnesota Software Engineering Center, 200 Union St., Minneapolis, MN 55455, August 2010.

[109] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-948472-8.

[110] J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 184—203. Springer-Verlag, 2002.

[111] F. Zeyda and A. Cavalcanti. Encoding *Circus* Programs in ProofPowerZ. LNCS. Springer-Verlag, 2009. Awaiting publication.

[112] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. In *Information Processing Letters*, volume 40, pages 269–276, 1991.