Refinement-oriented models of Stateflow charts

Alvaro Miyazawa[1] and Ana Cavalcanti[2]

[1]alvaro.miyazawa@york.ac.uk
[2]ana.cavalcanti@york.ac.uk
Department of Computer Science, University of York, York, YO10 5GH, UK

July 26, 2015

Abstract

Simulink block diagrams are widely used in industry for specifying control systems, and of particular interest and complexity are Stateflow blocks, which are themselves defined by separate charts. To make formal reasoning about diagrams and charts possible, we need to formalise their semantics; for the formal verification of their implementations, a refinement-based semantics is appropriate. An extensive subset of Simulink has been formalised in a language for refinement, namely, *Circus*, and here, we propose an approach to cover Stateflow charts. Our models are distinctive in their operational nature, which closely reflects the informal description of the Stateflow (simulation) semantics. We describe, formalise, and automate a strategy to generate our *Circus* models. The result is a solid foundation for reasoning based on refinement.

1 Introduction

MATLAB Simulink [1] is a graphical notation that supports the specification of control systems at a level of abstraction convenient for engineers; it is, for example, widely used in the avionics and automotive industries. Stateflow [2] is part of Simulink, and consists of a statechart notation used to define Simulink blocks. Simulink diagrams are typically used to specify aspects of a system that can be modelled by differential equations relating inputs and outputs. On the other hand, Stateflow charts frequently model the control aspects, like, for instance, changes in modes of operations triggered by events and conditions. Both Simulink and Stateflow provide extensive tool support to handle diagrams; there are facilities for simulation and analysis [1, 2], verification, validation and testing [3, 4], code generation [5, 6], and prototyping [7].

Many of the systems specified and designed using Simulink diagrams and Stateflow charts are safety-critical systems, and various certification standards [8, 9] recommend the use of formal methods for the specification, design, development and verification of software. This suggests that formal techniques that support graphical notations like Simulink and Stateflow are extremely useful, if not essential.

We are concerned with the assessment of the correctness of implementations of Stateflow charts: we want to be able to assert that a program correctly implements a chart. This has been frequently dealt with by approaches based on the verification of automatic code generators [10, 11, 12]. The approach that we pursue is orthogonal, and can be used in situations where code generators are not applicable or convenient. For instance, frequent updates to the generator have a heavy impact on the cost of its verification. In addition, customised hardware or performance requirements often impose the need for changes in code generated.

In this paper, we provide a formal semantics of Stateflow charts suitable for reasoning based on refinement. It is written in a way that facilitates validation, and integration of models of Simulink diagrams. With this, we set the foundation not only for analysing complex diagrams, but also for verifying the correctness of systems specified using both standard Simulink blocks and Stateflow charts. Although we do not tackle program verification here, the models that we present can be the starting point, for instance, to extend the refinement-based verification technique for implementations of Simulink diagrams presented in [13].

Our semantics of Stateflow charts is based on the *Circus* notation [14], which is a refinement language that combines Z [15], CSP [16], Dijkstra's language of guarded commands [17], and Morgan's specification statement [18]. *Circus* has a formal semantics given in terms of the Unifying Theories of Programming [19], and a refinement strategy that integrates different theories of refinement (action and process refinement). *Circus* supports the abstract specification of state-rich reactive systems and provides a refinement calculus [20] that

[1]email: alvaro.miyazawa@york.ac.uk

allows the verification of implementations. It is supported by various tools, such as a type-checker [21], a refinement editor [22], a translator from *Circus* to Java [23], and a theorem prover [24].

Cavalcanti et al. [25] define a semantics for Simulink diagrams in *Circus*. It builds upon the Z-based approach described in Arthan et al. [26], and Adams and Clayton [27], and extends it to cover a larger subset of Simulink, but not Stateflow blocks. Therefore, 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. By using *Circus* as a specification language, we provide support to reason formally about the model, to verify code of proposed implementations, and to integrate the model with existing models of Simulink diagrams.

As far as we know, our modelling technique is unique in that it covers a wide range of Stateflow constructs that have not been treated before, such as history junctions, unrestricted transitions, and multi-dimensional data. Additionally, the models are specified in a formal notation that has been extensively used for the verification of programs, namely *Circus*. It is possible to apply the *Circus* refinement calculus to reason about our models and their implementations. Finally, our models can be integrated with *Circus* models of Simulink diagrams. All this caters for a very comprehensive coverage of the Simulink/Stateflow notation.

Because the established semantics of Stateflow is only available through simulation or in an informal description in the *Stateflow User's Guide* [2], we provide a formal model based on the expected behaviour of charts during simulation. There is no way of formally comparing our model to the semantics encoded in the simulator (without access to the simulator's code), and this is a problem inherent to the formalisation of any language that does not have a well established formal model already.

In order to circumvent this issue, we have validated our model through alternative approaches. We have used inspection: systematic comparison of our formal model to the informal descriptions found in the Stateflow documentation. In particular, to facilitate validation, our model is constructed in a way that provides a direct correspondence with these descriptions. We have also used simulation: comparing traces of the simulation tool to traces of our models. Defining and implementing rules to translate Stateflow charts to *Circus*, and applying them to case studies, has also provided further validation. Finally, we are currently applying the *Circus* refinement calculus to verify chart implementations based on our model; this effort validates the modelling technique and its appropriateness for reasoning.

The main contributions of this work are an approach to modelling Stateflow charts using a state-rich process algebra like *Circus*, the formalisation of a strategy to translate Stateflow to *Circus* models, and the *s2c* tool that implements this strategy and generates *Circus* models automatically. These models are partitioned in two components that capture separately the semantics of Stateflow charts and the structure of a particular chart. The formalisation of the translation strategy is encoded in Z and consists of a formal syntax of Stateflow charts, well-formedness conditions, an embedding of *Circus* in Z, and a set of translation rules defined as functions from well-formed elements of the syntax of Stateflow charts to elements of the *Circus* notation. The *s2c* tool takes a textual representation of one or more charts, and automatically produces a *Circus* specification containing the corresponding models.

This article is structured as follows. Section 2 introduces the Stateflow and *Circus* notations. Section 3 presents our approach to construct formal models of Stateflow charts; Section 4 introduces the formalisation of the translation rules; Section 5 discusses the implementation of the translation rules in the *s2c* tool, and describes the validation of the proposed approach. Section 6 reviews our contributions, discusses the limitations of our work, examines related work, and proposes future directions for this work.

2 Preliminaries

In this section, we introduce the notations that form the basis for this work: Stateflow and *Circus*.

2.1 Stateflow

A Simulink diagram consists of blocks and wires connecting the inputs and outputs of the blocks. Its execution is carried out in a cyclic fashion, in which the blocks are executed in an order determined by the wiring of the diagram, and at a time determined by simulation time steps. Stateflow provides a new type of Simulink block, namely a Stateflow chart. This is a graphical notation that supports the specification of state transition systems. It is a variant of Harel's statecharts [28], which extends standard state-transition systems by introducing a number of features, such as hierarchy and parallelism.

Figure 1 shows a Stateflow chart adapted from an example supplied with the Simulink/Stateflow tool; it is part of the model of an automatic transmission controller. This chart specifies a system that monitors the speed of a vehicle, and changes gears appropriately. To decided whether or not to change gears, it calculates upper and lower thresholds based on the current gear and the engine throttle.

A Stateflow chart is the root for the execution of the corresponding Stateflow model; it encapsulates the states, transitions, junctions, functions, data and events that characterise it. The only means of interaction

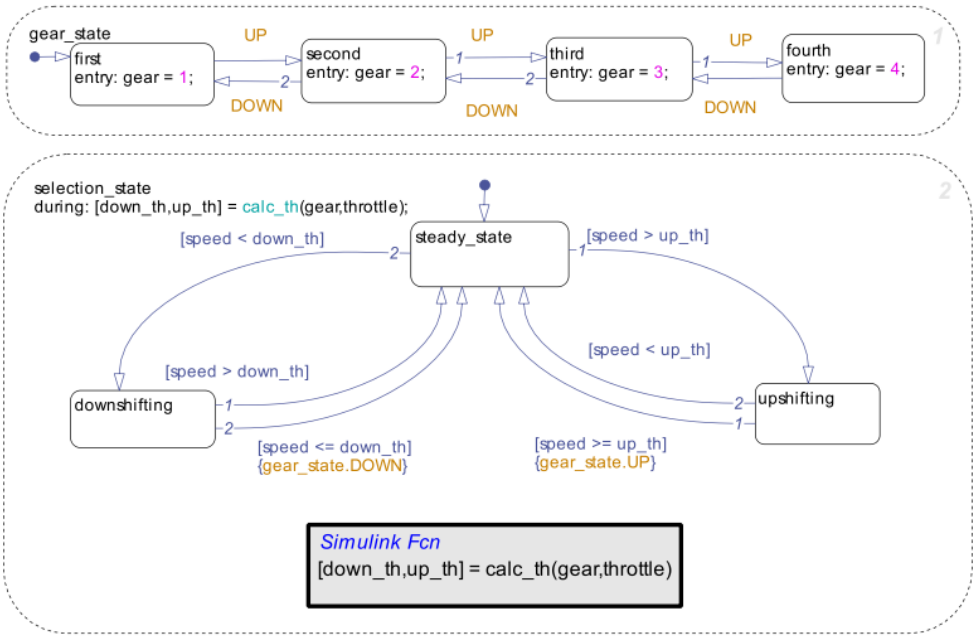

Figure 1: Shift logic example.

with the Simulink diagram are through input and output events and data. The life cycle of a Stateflow chart starts with the chart in an inactive state, that is, none of its states are active. When inactive, the chart can be triggered by an event, or by the Simulink diagram. In this case, the chart is executed, and at this point, the chart is active. Once the execution of the chart has been completed, the chart sleeps, that is, it waits to be triggered again. When an inactive chart is executed, its default transitions are entered possibly leading to the entry of states. When a sleeping chart is executed, its active states are executed.

In Figure 1, the two default transitions are those represented by an arrow with a closed circle. They lead to the states first and steady_state. When this chart is first executed, these states are entered.

A state is either active or inactive, and this status is altered by entering or exiting the state. States can have substates, which can be organised in two ways: parallel or sequential decomposition. In this respect, a chart is a special kind of state: it always has a decomposition. If a state (or chart) has a sequential decomposition, at most one of its immediate substates can be active at any time. If it has a parallel decomposition, all the immediate substates must be either active or inactive. The substates of a state with a parallel decomposition are ordered, and sequentially entered, executed and exited.

The chart in Figure 1 has a parallel decomposition and contains two immediate states, gear_state and selection_state, and each of them has a sequential decomposition. The state gear_state contains four substates, first, second, third and fourth, and selection_state has three, steady_state, upshifting and downshifting.

A state can have a series of actions that are executed at particular points of the life-cycle of a state. There are five types of actions that can be defined in association with a state. The *entry* actions are executed when a state is entered. For example, in Figure 1, the state first has an entry action "gear = 1". The *during* actions are executed when a state is active, and there are no valid transitions out of the state. The state selection_state contains a during action "[down_th,up_th]=calc_th(gear,throttle)" that assigns to the array formed by the variables down_th and up_th the result of the application of the function calc_th to the variables gear and throttle. The *on* actions are executed in the situations that a *during* action would be executed and a particular event occurs. The *exit* actions are executed when a state is exited. The *bind* actions specify which states can change a particular variable or broadcast a specific event.

Junctions are connective nodes that define decision points; they can, for example, model `if-then-else` and `for` statements. A history junction, on the other hand, records the most recent active substate of the state where it occurs. It can stand by itself in a state with sequential decomposition, or be reached by a transition. The example in Figure 2 contains connective and history junctions. It is a system that iteratively calculates the factorial or the absolute value of an input n, and outputs the result through a variable result.

The chart is controlled by events: ABS and FACT select the function to be calculated, PAUSE temporarily stops the calculations, START restarts them, STOP stops the calculations and resets the system allowing the user to select a new function. Initially, the system is in the state STOP, and a transition leads to a connective junction. The occurrence of FACT or ABS triggers the execution of a transition path that leads to the state

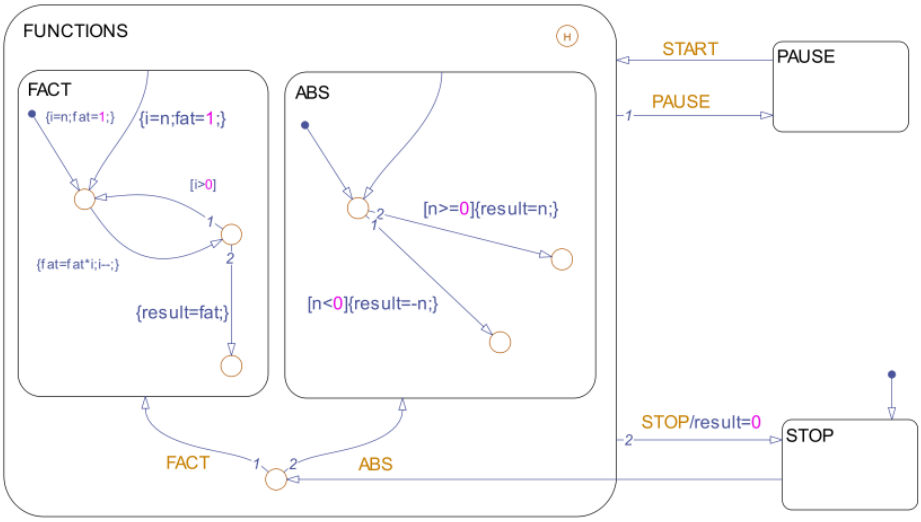

Figure 2: Function selection example.

FACT or to the state ABS; before either of these states is entered, the state FUNCTIONS is entered. If PAUSE occurs, FUNCTIONS and its active substate are exited, and the state PAUSE is entered. If PAUSE is active and START occurs, a transition from PAUSE to FUNCTIONS takes place, and the history junction in FUNCTIONS activates the last active substate, thus restarting the paused calculation.

A transition is usually a connection between two nodes, which can be either states or junctions. 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. For example, in Figure 2 the transition from the state STOP to the bottom connective junction in the state FUNCTIONS is a transition segment. A path is formed by following this transition segment with one of the two segments that lead to either the state FACT or the state ABS. A transition can connect nodes on different levels of the state hierarchy defined by their decomposition; such a transition is called an interlevel transition. For example, the transition segment mentioned above, from the state STOP to the bottom junction in FUNCTIONS, is an interlevel transition.

A default transition has no source node. A transition whose destination node is inside its source node is an inner transition. If the destination node is external to the source node, we have an outer transition.

Transitions can have a trigger, a condition, and condition and transition actions. A trigger is a set of events and temporal expressions. If the event being processed by the chart is in the trigger of a transition, or one of its temporal expressions is true for that event, and its condition is true, the transition is deemed valid (and can be followed). In Figure 1, the transition from the state first to the state second has a trigger formed by the event UP, and the transition from steady_state to upshifting has a condition [speed > up_th].

Condition actions are executed every time a transition (segment) is taken. Transition actions, on the other hand, are only executed once a transition path including the segment with the transition action is followed. The example in Figure 3 models a fumigation control system that handles requests to fumigate a plant (event FUMIGATE). While there are people inside the plant (condition people > 0), it sounds an alarm (condition action alarm()). Once there is no one left, it seals the plant (condition action seal()), and queues the transition action fumigate(). After the plant is sealed, the system does a redundant check of the number of people: if it is still zero (condition people==0), it marks the plant as not clean (exiting the state CLEAN), executes the transition action that had been queued, and marks the plant as being fumigated (entering the state FUMIGATING). If the second count reveals that there are people inside, it means that they entered while the plant was being sealed. In this case, the state FUMIGATING is not reached, the system backtracks to the state CLEAN, and the transition action fumigate() is not executed. The plant is, however, kept sealed because the condition action seal() is not undone.

Events are objects that trigger the execution of a Simulink block, for example, a chart. They can be distinguished according to trigger type and scope. With respect to the trigger type, an event can be edge-triggered or function-call. The scope characterises an event as input, output, local, exported or imported.

An input event is generated in a different block in the Simulink diagram and processed by the chart. An output event is generated by the chart and processed by some other block in the Simulink diagram. Local events are generated and processed inside the chart; they often model communication between parallel substates. The chart in Figure 1 uses two local edge-triggered events UP and DOWN.

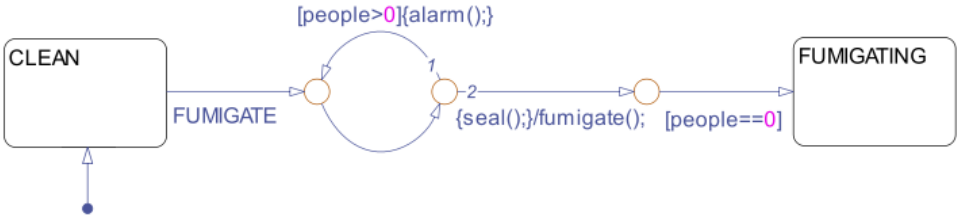

Figure 3: Fumigation control example.

Data objects are variables that record values used by the chart. They can be used to record internal information or to communicate with the Simulink diagram. As events, data can be distinguished according to scope. Local data are defined in a particular state (or chart) and are available to the state and its 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. The example in Figure 1 declares five data variables: speed, gear, up_th, down_th and throttle. The variables speed and throttle are input variables, gear is an output variable, and up_th and down_th are local variables.

A function block is an object that allows for the definition of functions. There are two types of function blocks that can be embedded in a Stateflow chart: graphical functions are defined using junctions, transitions, and the action language, and Simulink functions are defined by Simulink diagrams. The example in Figure 1 defines a Simulink function called calc_th that takes two parameters, gear and throttle, and returns the array [down_th,up_th]. It is worth mentioning that the variables used in the definition of the Simulink functions are not the variables of the same name declared by the chart. The function itself is defined by a Simulink diagram that relates the input variables to the output variables.

The semantics of Stateflow is given by simulation. It is, however, also described in the *User's Guide*, spread across object's descriptions and examples, and, in a more coherent way, in Chapter 3. The description contains inconsistencies that are partially addressed in the appendix called *Semantic Rules Summary*. Although it fixes some of the problems, some are not reviewed and new ones are introduced. These inconsistencies are mostly related to terminology use. We explain in Section 3 how we addressed these issues: where inconsistencies were found, the Stateflow simulator was used to identify the correct behaviour.

The execution of a Stateflow chart is triggered by the Simulink model; there are three ways the execution can be triggered: input events, sampling, or input data. If the chart has one or more input events, the occurrence of any of those events triggers the execution of the chart. If the chart does not have input events, it can be executed in a fixed time step, or its execution depends on the rate of the input data signals.

When executing a sleeping chart, the active states of the chart are executed in a top-down fashion. The execution of an active state consists of trying the outer transitions, and if the state is exited as a result of this, the execution stops. Otherwise, the *during* and valid *on* actions are executed, and the inner transitions are attempted. If no state transition results, the active children are executed (in order, if they are parallel).

When exiting an active state as a result of a state transition, if the state is parallel, all the sibling states after this in the reverse execution order are exited, any active substates are exited (in reverse order, if they are in parallel decomposition), the *exit* action is executed, and the state is marked inactive.

Transitions are considered in groups for execution. All default transitions that have the same parent are executed in sequence. All inner or outer transitions that originate in the same state are also considered as a sequence. The execution of a sequence of transitions consists of checking if the trigger and condition of the first transition is valid. If this is false, the execution restarts at the next transition. Otherwise, the condition action is executed and the execution proceeds to the destination node. If the destination node is a state, the source state and the super-states whose boundaries are crossed by the transition path are exited, the transition actions of the transition path are executed, and the destination state is entered. If the destination is a junction with no outgoing transitions, execution stops and no states are entered or exited. If the junction contains outgoing transitions, these are executed. If no outgoing transition of a junction is feasible, the execution backtracks and the next transition of the previous group is executed. When a transition path is successful, it triggers the entering of an inactive state.

In Section 3, we formalise the operational simulation model of a Stateflow chart that we have just described. First, we present the modelling notation: *Circus*.

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

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

channel $read, write : IMAGE$

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

$\quad$ **state** $S == [image : \operatorname{seq} IMAGE \mid \# image < maxbuff]$

$\quad InitState == [S' \mid image' = \langle\rangle]$

$\quad Read \mathrel{\widehat=} \big(\# image < maxbuff\big) \mathbin{\&} read?x \longrightarrow image := image \frown \langle x \rangle$

$\quad Write \mathrel{\widehat=} \big(\# image > 0\big) \mathbin{\&} write!(\operatorname{head} image) \longrightarrow image := tail\ image$

$\quad \bullet\ InitState\ ;\ \ (\mu X \bullet (Read \Box Write)\ ;\ X)$

end

Figure 4: The *Buffer* process

2.2 *Circus*

In this section, we present some of the *Circus* features using a simple *Circus* process as example (Figure 4); it models a system of image transmitters and receivers. A detailed presentation of *Circus* can be found in [14].

A *Circus* specification is a sequence of paragraphs: Z paragraphs (axiomatic definitions, schemas, and so on), channel and channel set declarations, and process definitions. The first few paragraphs of our example (Figure 4) define the horizontal coordinates as the set *HORIZONTAL* (of numbers from 1 to 800), the vertical coordinates as the set *VERTICAL*, the set of *COLOUR*s, and the set of *IMAGE*s: total functions 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.

A channel declaration introduces channel names and the types of the values that can be communicated through them. A channel with no type does not communicate any values; it is used for synchronisation. 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 4, we define a process *Buffer* whose state is given by the schema S; it 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 for it. Thus the action *Read* has the condition $\# image < maxbuff$ as guard; it requires the size of the image buffer to be smaller than *maxbuff*. If the guard is true, *Buffer* can receive a value through channel *read*, 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 (head *buffer*) through the channel *write*, and remove it from the *buffer* by redefining it as the rest of the sequence (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 its components renamed, have its channels hidden, and so on.

For instance, we can specialise the *Buffer* process as a transmitter that reads images from an *antenna* and sends it 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.

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.

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

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

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*.

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

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.

We present a much larger example of a *Circus* specification, as we describe our Stateflow models.

3 A formal model of Stateflow charts

In this section, we introduce an operational model of Stateflow charts described in *Circus*. We develop a model close enough to the informal description of the behaviour of such charts, so that its validity can be argued with some degree of confidence by comparing the formal and informal specifications.

Our models consist of two *Circus* processes in parallel. The first, *Simulator*, represents the simulator, and is the same for every chart. The second, the chart process, represents a particular chart.

We describe and illustrate our models using the example in Figure 1. The process that models that chart is *c_shift_logic*. Below, we define the process *Shift_logic* that combines *c_shift_logic* and *Simulator* in parallel. They interact via a set *interface* of internal channels plus the channel *end_cycle*.

$$\textbf{process}\ Shift\_logic \mathrel{\widehat=} (c\_shift\_logic \ [\![\ interface \cup \{\!|\, end\_cycle \,|\!\}\ ]\!]\ Simulator) \setminus interface$$

The 23 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. For the sake of conciseness, we omit the declaration of channels and channel sets. We discuss them as they are needed in the sequel. A complete version of the model can be found in [29].

The external channels of our Stateflow models include one channel for each input and output data; in our example in Figure 1, we have *o_gear*, used for output, and *i_speed* and *i_throttle*, used for input. We also have two channels *input_event* and *output_event* to communicate events, which are modelled as elements of a type *EVENT*. A channel *end_cycle* marks the end of each cycle of execution. Finally, a channel *error* flags error situations; for well-formed diagrams, there should be no communications on this channel.

3.1 Process *Simulator*

The process *Simulator* provides the main communication interface with a Simulink model; it accepts communications that allow input events to trigger the execution, and the communication of an output event and of the end of the cycle of execution of the chart. An overview of the structure of the process *Simulator* is shown in Figure 5. It declares 36 actions that are used to build the process' main action.

3.1.1 Step of execution

The main action of *Simulator* recursively executes the *Circus* action *Step*. It reads the set of events active in the current time step (through *input_event*), stores it in a local variable *aes*, requests the chart process to read the inputs using the channel *read_inputs*, executes the chart with the active events as defined by the action *ExecuteEvents*, requests the chart to write its outputs using the channel *write_outputs*, and flags the end of the execution step through the channel *end_cycle*.

$$Step \mathrel{\widehat=} input\_event?x \longrightarrow \textbf{var}\ aes : \mathbb{P}\, EVENT \bullet \left(\begin{array}{l} aes := x; \\ read\_inputs \longrightarrow ExecuteEvents(events, aes); \\ write\_outputs \longrightarrow end\_cycle \longrightarrow \textbf{Skip} \end{array} \right)$$

The execution of the chart on the active events is specified by the action *ExecuteEvents*; it takes a sequence of events and a set of active events as parameters and executes the chart under each active event. The sequence *events* corresponds to the events declared in the chart ordered by their port numbers.

$$ExecuteEvents \mathrel{\widehat=} es : \mathrm{seq}\, EVENT;\ aes : \mathbb{P}\, EVENT \bullet \left(\begin{array}{l} \textbf{if}\ \#\, events = 0 \longrightarrow \textbf{Skip} \\ [\!]\ \#\, events > 0 \longrightarrow ExecuteEvent(head\ es, aes)\ ;\ ExecuteEvents(tail\ es, aes) \\ \textbf{fi} \end{array} \right)$$

channel *executeentryaction*, *executeduringaction*, *executeexitaction* : *SID*
...
channelset *interface* == {| *executeentryaction*, *executeduringaction*, ... |}
process *Simulator* $\widehat{=}$ **begin**
ExecuteTransition $\widehat{=}$ *tid* : *TID*; *path* : seq *TID*; *source* : *State*; *cevent* : *EVENT* • ...
CheckValidity $\widehat{=}$ *tid* : *TID*; *path* : seq *TID*; *source* : *State*; *cevent* : *EVENT* • ...
Proceed $\widehat{=}$ *tid* : *TID*; *path* : seq *TID*; *source* : *State*; *cevent* : *EVENT* • ...
proceedToState $\widehat{=}$ *src*, *dest* : *State*; *path* : seq *TID*; *cevent* : *EVENT* • ...
executePath $\widehat{=}$ *path* : seq *TID*; *src*, *dest* : *State*; *cevent* : *EVENT* • ...
proceedToJunction $\widehat{=}$ *tid* : *TID*; *path* : seq *TID*; *source* : *State*; *cevent* : *EVENT* • ...
executeJunction $\widehat{=}$ *j* : *Junction*; *path* : seq *TID*; *source* : *State*; *cevent* : *EVENT* • ...
ExecuteDefaultTransition $\widehat{=}$ *s*, *tpp* : *State*; *cevent* : *EVENT* • ...
EnterState $\widehat{=}$ *s*, *tpp* : *State*; *cevent* : *EVENT* • ...
*EnterState*1 $\widehat{=}$ *s*, *tpp* : *State*; *cevent* : *EVENT* • ...
...
*EnterState*7 $\widehat{=}$ *s*, *tpp* : *State*; *cevent* : *EVENT* • ...
EnterStates $\widehat{=}$ *ss* : seq *SID*; *tpp* : *State*; *cevent* : *EVENT* • ...
ExecuteState $\widehat{=}$ *s* : *State*; *cevent* : *EVENT* • ...
AlternativeExecution $\widehat{=}$ *s* : *State*; *cevent* : *EVENT* • ...
ExecuteStates $\widehat{=}$ *ss* : seq *SID*; *cevent* : *EVENT* • ...
ExitState $\widehat{=}$ *s* : *State*; *cevent* : *EVENT* • ...
ExitStates $\widehat{=}$ *ss* : seq *SID*; *cevent* : *EVENT* • ...
ExecuteChart $\widehat{=}$ *cevent* : *EVENT* • ...
ExecuteInactiveChart $\widehat{=}$ *c* : *State*; *cevent* : *EVENT* • ...
ExecuteActiveChart $\widehat{=}$ *c* : *State*; *cevent* : *EVENT* • ...
LocalEvent $\widehat{=}$ ...
TreatLocalEvent $\widehat{=}$ *e* : *EVENT*; *s* : *State* • ...
Step $\widehat{=}$ ...
• μX • *Step* ; *X*
end

Figure 5: Structure of the *Simulator* process.

If the sequence of events is empty, the action terminates. Otherwise, it calls the action *ExecuteEvent* on the first event of the list (*head events*) and the set of active events, and recursively calls the action *ExecuteEvents* on the rest of the list and the set of active events.

$$
\begin{array}{l}
\mathit{ExecuteEvent} \mathrel{\widehat{=}} e : \mathit{EVENT};\ \mathit{aes} : \mathbb{P}\,\mathit{EVENT} \bullet \\
\left(\begin{array}{l}
\textbf{if } e \in \mathit{aes} \longrightarrow \mathit{ExecuteChart}(e) \triangle (\mathit{interrupt\_simulator} \longrightarrow \mathit{interrupt\_chart} \longrightarrow \textbf{Skip}) \\
[\!] \ e \notin \mathit{aes} \longrightarrow \textbf{Skip} \\
\textbf{fi}
\end{array}\right)
\end{array}
$$

The action *ExecuteEvent* takes a single event and the set of active events, and executes the chart if the event is in the set. Otherwise, the action terminates.

The execution of the chart, that is, the action *ExecuteChart*, can, at any moment, be interrupted ($\triangle$) by early return logic. The interrupting action waits for a synchronisation on the channel *interrupt_simulator*, and then synchronises on the channel *interrupt_chart*. Early return logic occurs when a recursive execution (triggered by a local event broadcast) deactivates a state; this is signalled by the chart process using the channel *interrupt_simulator*. If early return logic occurs, immediately after the execution is interrupted, the chart's action is interrupted. This is signalled to the chart process using the channel *interrupt_chart*.

By way of illustration, we consider the chart in Figure 6 adapted from an example in [2]. This rather artificial 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 the condition action is executed. It 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.

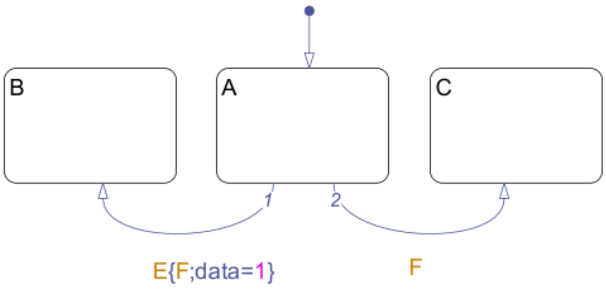

Figure 6: Early return logic example.

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 *NULLEVENT*, and allowing the Simulink model to trigger the execution of the chart by communicating $\{NULLEVENT\}$.

The execution of a chart depends on whether it is inactive. The definition of *ExecuteChart* is omitted. If the chart is inactive, it calls *ExecuteInactiveChart* defined below, otherwise, *ExecuteActiveChart* is called.

The execution of an inactive chart c triggers the activation of the state identified 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, *ExecuteInactiveChart* attempts to execute the default transitions using *ExecuteDefaultTransition*. A parallel action monitors whether they are successfully executed or not. If they are, this is signalled by *ExecuteDefaultTransition* using the channel *stsuccess*. In this case, the whole parallelism terminates. If we have a failure, this is signalled by *stfail*, and the parallel monitoring action checks whether the state has a parallel decomposition ($c.decomposition = SET$) or not. If this is the case, it enters all substates ($c.substates$) in the appropriate order as defined by *EnterStates* explained below. Otherwise, the parallel monitoring action indicates an error to the Simulink diagram using *error*. As previously mentioned, a model of a well-formed Stateflow chart should never lead to the raise of an error.

$$
\begin{array}{l}
ExecuteInactiveChart \mathrel{\widehat=} c : State;\ cevent : EVENT \bullet activate!(c.identifier) \longrightarrow \\
\left(\begin{array}{c} ExecuteDefaultTransition(c, c, cevent) \\ [\![\mid statetransition \mid]\!] \\ \left(\begin{array}{l} stsuccess \longrightarrow \mathbf{Skip} \,\square\, stfail \longrightarrow \left(\begin{array}{l} \mathbf{if}\ c.decomposition = SET \longrightarrow \\ \quad EnterStates(c.substates, c, cevent) \\ [\!]\ c.decomposition \neq SET \longrightarrow \\ \quad error \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array} \right) \end{array} \right) \end{array} \right) \setminus statetransition
\end{array}
$$

The simple definition of *ExecuteActiveChart* is omitted. If the chart is not inactive, *ExecuteActiveChart* executes its active substates ($c.substates$). If there are no substates, it is executed as an inactive chart.

3.1.2 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. It 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, and the current event *cevent*.

$$
\begin{array}{l}
ExecuteTransition \mathrel{\widehat=} tid : TID;\ path : \mathrm{seq}\ TID;\ source : State;\ cevent : EVENT \bullet \\
\left(\begin{array}{l} \mathbf{if}\ tid = nulltransition.identifier \longrightarrow \\ \quad \left(\begin{array}{l} \mathbf{if}\ path = \varnothing \longrightarrow stfail \longrightarrow \mathbf{Skip} \\ [\!]\ path \neq \varnothing \longrightarrow \left(\begin{array}{l} transition!(\mathrm{last}\ path)?lt \longrightarrow \\ ExecuteTransition(lt.next, (\mathrm{front}\ path), source, cevent) \end{array} \right) \\ \mathbf{fi} \end{array} \right) \\ [\!]\ tid \neq nulltransition.identifier \longrightarrow CheckValidity(tid, path, source, cevent) \\ \mathbf{fi} \end{array} \right)
\end{array}
$$

ExecuteTransition compares *tid* to the identifier of the null transitions ($nulltransition.identifier$). 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 signalled using *stfail*. If *path* is not empty, *ExecuteTransition* uses the identifier of the last transition successfully followed (last *path*) to obtain that transition *lt* itself 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, and the current event.

The action *CheckValidity* receives the identifier *tid* of a transition, a sequence *path* of transition identifiers, a state *source*, and an event *cevent*, and requests the chart to evaluate its trigger by communicating *tid* and *cevent* through the channel *checktrigger*. It then takes the boolean response e through the channel *result*. Next, it gets the evaluation c of the condition of the transition through *evaluatecondition*. If the trigger and the condition are true ($e = \mathbf{True} \wedge c = \mathbf{True}$), then *CheckValidity* requests the execution of the condition action through *executeconditionaction*, calls the action *LocalEvent*, and proceeds with the execution of the destination node 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*). *LocalEvent* deals with any local event broadcasts by the condition action. 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.

$$
\begin{array}{l}
CheckValidity \mathrel{\widehat=} tid : TID;\ path : \mathrm{seq}\,TID;\ source : State;\ cevent : EVENT \bullet \\
\quad checktrigger!tid!cevent \longrightarrow result!tid!cevent?e \longrightarrow evaluatecondition!tid?c \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ e = \mathbf{True} \wedge c = \mathbf{True} \longrightarrow \left(\begin{array}{l} executeconditionaction!tid!(source.identifier)!cevent \longrightarrow \\ \quad LocalEvent\ ;\ Proceed(tid, path \frown \langle tid \rangle, source, cevent) \end{array}\right) \\
[\!]\ \neg\,(e = \mathbf{True} \wedge c = \mathbf{True}) \longrightarrow transition!tid?t \longrightarrow ExecuteTransition(t.next, path, source, cevent) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

Whenever the process *Simulator* requests the execution of a chart action, there is a possibility that local events are broadcasted. In this case, *LocalEvent* recursively offers a choice between treating an event and waiting for other local events, or terminating. The first option is modelled by an action that waits for a communication on the channel *local_event*. If it occurs, the action *TreatLocalEvent* is called and *LocalEvent* 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, *LocalEvent* terminates.

$$
LocalEvent \mathrel{\widehat=} (\mu X \bullet local\_event?e?s \longrightarrow (TreatLocalEvent(e, s)\ ;\ X) \,\Box\, end\_action \longrightarrow \mathbf{Skip})
$$

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.

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

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.

Proceed requests the transition identified by *tid* through the channel *transition*. It then identifies 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.

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

The definition of *proceedToState* is based on the closest common parent $lub(src, dest)$ of the source and destination states *src* and *dest*. This 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. The function *lub* determines the least upper bound of an ancestor relation between states [29]. The action *proceedToState* exits all the active substates of the common ancestor ($(lub(src, dest)).substates$), executes *path* as defined by *executePath*, enters *dest* using *EnterState*,

and signals the success of the transition execution through *stsuccess*. *EnterState* takes the closest ancestor of *src* and *dest* as a second parameter; it is further discussed later on.

$$proceedToState \mathrel{\widehat=} src, dest : State;\ path : \text{seq } TID;\ cevent : EVENT \bullet \left(\begin{array}{l} ExitStates((lub(src, dest)).substates, cevent)\ ;\ executePath(path, src, dest, cevent); \\ EnterState(dest, lub(src, dest), cevent)\ ;\ stsuccess \longrightarrow \textbf{Skip} \end{array} \right)$$

The path execution runs each of its transition actions; this is defined by the action *executePath*.

$$executePath \mathrel{\widehat=} path : \text{seq } TID;\ src, dest : State;\ cevent : EVENT \bullet \left(\begin{array}{l} \textbf{if } \#path = 0 \longrightarrow \textbf{Skip} \\ [\!] \ \#path > 0 \longrightarrow \left(\begin{array}{l} executetransitionaction!(\text{head } path)!((lub(src, dest)).identifier)!cevent \longrightarrow \\ LocalEvent\ ;\ executePath(\text{tail } path, src, dest, cevent) \end{array} \right) \\ \textbf{fi} \end{array} \right)$$

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*) and the closest common parent of the source and destination of the path ($lub(src, dest)$) and the current event through *executetransitionaction*. It then calls *LocalEvent* to deal with any local event broadcast by the transition action, and recurses the rest of the path (tail *path*).

It is worth mentioning that the Stateflow User's Guide requires that after the source state (and relevant parent states) are exited, only "the transition action of the final transition segment of the full transition path is executed" [2]. 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 action *proceedToJunction* 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 *cevent*. 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 the channel *history*. If *lsid* is the identifier of the null state ($lsid = nullstate.identifier$), the default transitions of the state are executed using *ExecuteDefaultTransition*. Otherwise, *proceedToJunction* recovers the state *ls* identified by *lsid* through channel *state*, and calls *proceedToState* with *ls* as target state.

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

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 stores *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, it 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.

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 signalled by *stfail*. Otherwise, *executeJunction* calls *ExecuteTransition*.

$$executeJunction \mathrel{\widehat=} j : Junction;\ path : \text{seq } TID;\ source : State;\ cevent : EVENT \bullet \left(\begin{array}{l} \textbf{if } j.transition = nulltransition.identifier \longrightarrow stfail \longrightarrow \textbf{Skip} \\ [\!] \ j.transition \neq nulltransition.identifier \longrightarrow ExecuteTransition(j.transition, path, source, cevent) \\ \textbf{fi} \end{array} \right)$$

The key actions that model the execution of transitions are *ExecuteTransition*, which we presented above, and *ExecuteDefaultTransition*. The latter extends the former 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 definition of *ExecuteDefaultTransition* is in [29].

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 that all siblings with a higher (i.e., earlier) entry order are active. If not, perform entry steps 1-5 for these states 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. 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 7: Steps for entering a state

3.1.3 Entering a state

The process of entering an inactive state is described in Chapter 3 of the Stateflow User's Guide, and in the appendix *Semantic Rules Summary*. These descriptions are inconsistent with each other, and with the behaviour observed in the Stateflow tool. In particular, there are inconsistencies in the use of terminology regarding entry actions and the entry steps. For example, 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. (This type of inconsistency can also be found in the description of the process of exiting a state in Chapter 3 of the User's Guide; however, it was corrected in the appendix *Semantic Rules Summary*.)

Figure 7 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 define the correct steps that are recursively executed. In the case described above, 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 state is already active, as it triggered the entering of this state.

A second source of problems is the range of entry steps that are executed in certain situations. For instance, when entering a parallel state, if the left sibling is not active, the description in Chapter 3 requires the execution of all entry steps. The appendix, on the other hand, requires the execution of the steps from 1 to 5 only. We have observed that the latter is correct, and include it in our description (step 2 in Figure 7).

To keep close to the informal description, we preserve its structure and granularity in the actions that model the simulation semantics. The procedure for entering a state s is defined by the action *EnterState* below. For each step in Figure 7, 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. These actions are composed in sequence to define actions that represent the execution of several steps. For example, the execution of steps 1-4 is modelled by *EnterState*14, which is the sequential composition of *EnterState*1, *EnterState*2, *EnterState*3, and *EnterState*4. Finally, all the steps are combined in sequence to defined *EnterState*.

$$EnterState \mathrel{\widehat=} s, tpp : State;\ cevent : EVENT \bullet \left(\begin{array}{l} EnterState14(s, tpp, cevent); \\ EnterState5(s, tpp, cevent); \\ EnterState67(s, tpp, cevent) \end{array} \right)$$

Here we present the action *EnterState*5a that formalises the step 5a of the procedure in Figure 7. The

complete formalisation of all steps can be found in [29].

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

If s does not have a sequential decomposition ($s.decomposition \neq CLUSTER$), then $EnterState5a$ terminates immediately. Otherwise, if s has a history junction ($s.history = \textbf{True}$), then it 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 ($lsid = nullstate.identifier$), $EnterState5a$ calls $ExecuteDefaultTransition$ on s. Otherwise, it obtains the state ls identified by $lsid$ through $state$, and executes the steps from 1 to 5. If s does not have a history junction, the default transitions are executed.

Checking that s has a sequential decomposition is not explicitly required in the informal description of step 5a, but it is needed to avoid attempting to execute default transitions of a state with parallel decomposition and issuing an error. Although it is possible to define default transitions to parallel states, they are ignored. This fact is confirmed by the simulation tool of Stateflow.

EnterStates takes as parameter a sequence ss of state identifiers to be entered. If ss is empty ($\# ss = 0$), it terminates. Otherwise, it takes the first identifier in the sequence, and determines whether the corresponding state is active or not through $status$. If it is not, *EnterStates* recovers that state through $state$, and calls *EnterState* on it. If it is active, nothing is done. Finally, we have a recursive call on the rest of the sequence.

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

The processes of executing and exiting states are modelled by the actions *ExecuteState* and *ExitState*, and are omitted. In the next section, we describe the process that represents the structure of the chart and complements the *Simulator* process in order to model the execution of the chart.

3.2 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 state records the data declared in the chart, and information particular to its execution. The structure of the chart process *c_shift_logic* corresponding to our example in Figure 1 is depicted in Figure 8.

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

$$
\begin{array}{l}
identifier : SID \\
states : SID \pfun State \\
transitions : TID \pfun Transition \\
junctions : JID \pfun Junction \\
\hline
nullstate \notin \operatorname{ran} states \\
nulltransition \notin \operatorname{ran} transitions \\
nulljunction \notin \operatorname{ran} junctions \\
\#\{s : \operatorname{ran} states \mid s.type = CHART\} = 1 \\
(states(identifier)).type = CHART \\
\forall n : SID \mid n \in \operatorname{dom} states \bullet (states(n)).identifier = n \\
\forall n : JID \mid n \in \operatorname{dom} junctions \bullet (junctions(n)).identifier = n \\
\forall n : TID \mid n \in \operatorname{dom} transitions \bullet (transitions(n)).identifier = n
\end{array}
$$

process $c\_shift\_logic \mathrel{\widehat=}$ **begin**

$StateflowChart$

$\ldots$

$SimulationData == [state\_status : SID \pfun \mathbb{B};\ state\_history : SID \pfun SID \mid \ldots]$
$InitSimulationData == [SimulationData' \mid \ldots]$
$SimulationInstance == [v\_gear, v\_speed, v\_throttle, v\_up\_th, v\_down\_th : \mathbb{R}]$
$InitSimulationInstance == [SimulationInstance' \mid \ldots]$
$InitState == (InitSimulationInstance) \land (InitSimulationData)$
$\ldots$
$Activate == \Xi SimulationInstance \land ((ActivateWithHistory \lor ActivateNoHistory) \lor ActivateFail)$
$\ldots$
$Deactivate == \Xi SimulationInstance \land (DeactivateSuccess \lor DeactivateFail)$
state $(SimulationInstance \land SimulationData)$
$entryaction\_downshifting \mathrel{\widehat=} executeentryaction.(s\_downshifting)?o?ce \longrightarrow$ **Skip**
$\ldots$
$entryactions \mathrel{\widehat=} entryaction\_downshifting \Box \ldots \Box entryaction\_steady\_state$
$duringaction\_downshifting \mathrel{\widehat=} executeduringaction.(s\_downshifting)?o?ce \longrightarrow$ **Skip**
$\ldots$
$duringactions \mathrel{\widehat=} duringaction\_downshifting \Box \ldots \Box duringaction\_steady\_state$
$exitaction\_downshifting \mathrel{\widehat=} executeexitaction.(s\_downshifting)?o?ce \longrightarrow$ **Skip**
$\ldots$
$exitactions \mathrel{\widehat=} exitaction\_downshifting \Box \ldots \Box exitaction\_steady\_state$
$conditionaction\_third\_fourth \mathrel{\widehat=} executeconditionaction.(t\_third\_fourth)?o?ce \longrightarrow$ **Skip**
$\ldots$
$conditionactions \mathrel{\widehat=} conditionaction\_third\_fourth \Box \ldots \Box conditionaction\_default\_steady\_state$
$transitionaction\_third\_fourth \mathrel{\widehat=} executetransitionaction.(t\_third\_fourth)?o?ce \longrightarrow$ **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**
$\ldots$
$triggers \mathrel{\widehat=} trigger\_third\_fourth \Box \ldots \Box trigger\_upshifting\_steady\_state26$
$getstate \mathrel{\widehat=} state?x : (x \in \mathrm{dom}(states))!(states(x)) \longrightarrow$ **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**
$activation \mathrel{\widehat=} activate?x \longrightarrow Activate$
$deactivation \mathrel{\widehat=} deactivate?x \longrightarrow Deactivate$
$ChartActions \mathrel{\widehat=} \ldots$
$InterfaceActions \mathrel{\widehat=} \ldots$
$Inputs \mathrel{\widehat=} \ldots$
$Outputs \mathrel{\widehat=} \ldots$
$AllActions \mathrel{\widehat=} \ldots$
$broadcast \mathrel{\widehat=} e : EVENT;\ sid, dest : SID \bullet \ldots$
$broadcast\_taction \mathrel{\widehat=} e : EVENT;\ sid, dest : SID \bullet \ldots$

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

end

Figure 8: Structure of the *c_shift_logic* process.

The component *identifier* indicates the state that corresponds to the chart. Identifiers for states, transitions, and junctions are drawn from the sets *SID*, *TID* and *JID*, defined as disjoint given sets. The sets *State*, *Transition* and *Junction* are defined by schemas (presented later); they are sets of bindings (records) all including a component *identifier*. The components *states*, *transitions* and *junctions* are, therefore, partial 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.

The schema *State* 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.

State

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

The schema *Junction* records the identifier of a junction, the identifier of the first (if any) transition leaving it, the identifier of its parent state (or chart), and whether or not it is a history junction.

$$Junction == [identifier : JID;\ transition : TID;\ parent : SID;\ history : \mathbb{B}]$$

Transition records the identifier of a transition, the identifiers of its source and destination nodes, the identifier of the next transition (if any) in a sequence, and the parent state (or chart) of that transition.

$$Transition == [identifier : TID;\ source, destination : NID;\ next : TID;\ parent : SID]$$

The set *NID* contains node identifiers, which are taken from the sets *SID* and *JID*.

In the chart process (see Figure 8), an axiomatic definition includes *StateflowChart*, promoting its components to process constants, and identifying their values. For our example, the axiomatic definition defines that $identifier = c\_shift\_logic$ and that *states* includes the pairs $(s\_downshifting, S\_downshifting)$, $(s\_gear\_state, S\_gear\_state)$ and so on, where $s\_downshifting$ and s_gear_state are state identifiers.

The definition of *SimulationData* is independent of a particular chart. It declares the state components of a 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

$$\begin{array}{l} state\_status : SID \pfun \mathbb{B} \\ state\_history : SID \pfun SID \\ \hline \operatorname{dom} state\_status = \operatorname{dom} states \\ \operatorname{dom} state\_history = \{j : \operatorname{ran} junctions \mid j.history = \mathbf{True} \bullet j.parent\} \end{array}$$

The predicate of *SimulationData* states that the domain of *state_status* contains the identifiers of all states, and that the domain of *state_history* contains the identifiers of all parent states of history junctions. 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 8) declares the data defined in the chart. For our example, these 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 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*, which can be interrupted by the *Simulator* through the channel *interrupt_chart*. As shown below, *AllActions* is defined as the external choice of actions called *conditions*, *triggers*, *Inputs*, *Outputs*, *ChartActions*, and *InterfaceActions*.

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

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* signaling 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 (in Figure 1) is defined as follows.

$$condition\_steady\_state\_downshifting \mathrel{\widehat=} \left(\begin{array}{l} \textbf{if } (v\_speed <_{\mathcal{A}} v\_down\_th) \neq 0 \longrightarrow evaluatecondition.t\_steady\_state\_downshifting!\textbf{True} \longrightarrow \textbf{Skip} \\ [\!] \neg\ ((v\_speed <_{\mathcal{A}} v\_down\_th) \neq 0) \longrightarrow evaluatecondition.t\_steady\_state\_downshifting!\textbf{False} \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array} \right)$$

The predicate $(v\_speed <_{\mathcal{A}} v\_down\_th) \neq 0$ is evaluated, and if it is true, this 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, e, and the boolean result of the evaluation through *result*. For the trigger of the transition between the states first and second, we have the action below.

$$trigger\_first\_second \mathrel{\widehat=} checktrigger.t\_first\_second?e \longrightarrow \left(\begin{array}{l} \textbf{if } e = e\_UP \longrightarrow result.t\_first\_second.e!\textbf{True} \longrightarrow \textbf{Skip} \\ [\!] \neg\ (e = e\_UP) \longrightarrow result.t\_first\_second.e!\textbf{False} \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array} \right)$$

It returns **True** if the event received through the channel *checktrigger* is *e_UP*, otherwise, it returns **False**.

The *Circus* action *Inputs* waits for a synchronisation on *read_inputs* and reads the values of the input variables in interleaving; there is a channel for each input. 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 declaration of the state components that are altered by each parallel action. This action only terminates when all the input variables are read.

The *Circus* action *Outputs* is defined in a similar fashion. For our example, it is as follows.

$$Outputs \mathrel{\widehat=} write\_outputs \longrightarrow o\_gear!(v\_gear) \longrightarrow \textbf{Skip}$$

The only output variable in the example is *v_gear*, so there is no need for an interleaving.

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.

$$entryaction\_first \mathrel{\widehat=} executeentryaction.(s\_first)?o?ce \longrightarrow v\_gear := 1$$

This action waits for the communication of the state's identifier on *executeentryaction*, and the input of a state identifier o and an event ce, and assigns the value 1 to the state component *v_gear*. The input o records the state whose execution requested the execution of the entry action; this is used when broadcasting a local event. The input ce is necessary because one type of during action, namely an *on action*, is only executed for particular events. We keep ce as an input to all types of actions for uniformity, and to allow the extension of the model to treat temporal actions, which depend on the current event as well.

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.

$$conditionaction\_upshifting\_steady\_state26 \mathrel{\widehat=} executeconditionaction.(t\_upshifting\_steady\_state26)?o?ce \longrightarrow broadcast(e\_UP, o, s\_gear\_state)$$

We postfix this action's name with a number that makes the transition unique, given that there are two transitions with the same source and destination states. The action waits for the communication of the

transition identifier through *executeconditionaction*, and broadcasts the event *e_UP* to the state *s_gear_state*. In this example, the broadcast is used to trigger a change of state in the parallel state gear_state.

The *Circus* action *broadcast* that specifies the broadcasting mechanism takes three parameters: the local event e, the identifier *dest* of the destination state, and the state identifier *origin* for either the state, or the source or parent of the transition that issued the broadcast.

$$broadcast \mathrel{\widehat=} e : EVENT;\ origin, dest : SID \bullet local\_event!(e, states(dest)) \longrightarrow \left(\mu X \bullet \left(\begin{array}{l} AllActions\ ;\ X \\ \Box \\ end\_local\_execution \longrightarrow \textbf{Skip} \end{array} \right) \right) ; \left(\begin{array}{l} \textbf{if}\ state\_status(origin) = \textbf{True} \longrightarrow \textbf{Skip} \\ [\!]\ state\_status(origin) = \textbf{False} \longrightarrow \\ \quad interrupt\_simulator \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array} \right)$$

The action *broadcast* communicates e and destination state *states*(*dest*) through *local_event*, and recursively offers a choice between *AllActions* followed by a recursive call to X, and a terminating action that waits for a synchronisation on *end_local_execution*, and finishes the recursion. Once the recursion ends, *broadcast* checks the status of the state *origin* according to the function *state_status*, and if it is no longer active (*state_status*(*origin*) = **False**), interrupts the simulator by synchronising on *interrupt_simulator*.

The early return logic for broadcasts from transition actions contains an extra condition. If any of the substates of *origin* is active, the execution is interrupted. This is modelled by the following action.

$$broadcast\_taction \mathrel{\widehat=} e : EVENT;\ origin, dest : SID \bullet \textbf{var}\ \_s : State;\ \_b : \mathbb{B} \bullet broadcast(e, origin, dest); \_s := states(dest)\ ;\ \left(HasActiveSubstates\right)\ ;\ \left(\begin{array}{l} \textbf{if}\ \_b = \textbf{True} \longrightarrow interrupt\_simulator \longrightarrow \textbf{Skip} \\ [\!]\ \_b = \textbf{False} \longrightarrow \textbf{Skip} \\ \textbf{fi} \end{array} \right)$$

This action calls *broadcast*, and, if it completes, *broadcast_taction* recovers the state identified by *dest*, stores it in the local variable *_s*, and calls the action *HasActiveSubstates* that assigns **True** to *_b* if the state *_s* has active substates, and **False** otherwise. If *_s* has active substates, *broadcast_taction* signals on *interrupt_simulator* that *ExecuteChart* must be interrupted. Otherwise, *broadcast_taction* terminates. Notice that the *origin* of a transition action is the parent of the transition path that contains the action.

For our example, we have the *Circus* action below, which models the execution of the condition action gear_state.DOWN of the transition of number 2 from downshifting to steady_state.

$$conditionaction\_downshifting\_steady\_state25 \mathrel{\widehat=} executeconditionaction.t\_downshifting\_steady\_state25?o?ce \longrightarrow broadcast(e\_DOWN, o, s\_gear\_state)$$

The arguments used in the call to the *Circus* action *broadcast* are the event *e_DOWN*, which models the chart event DOWN, the identifier o received through the channel *executeconditionaction*, and which, in this case, is *s_downshifting*, and the identifier *s_gear_state* of the target of the broadcast.

The only during action in the chart in Figure 1 involves the use of the Simulink function *calc_th*. This is specified as the application of a Z function of the same name whose definition must be provided in a separate Z library. The *Circus* action that models the during action waits for the communication of the state identifier, *s_selection_state*, through *executeduringaction* and executes the assignment.

$$duringaction\_selection\_state \mathrel{\widehat=} executeduringaction.(s\_selection\_state) \longrightarrow \textbf{var}\ \_aux : \mathbb{R} \times \mathbb{R} \bullet (\_aux := calc\_th(v\_gear, v\_throttle)\ ;\ v\_down\_th := \_aux.1\ ;\ v\_up\_th := \_aux.2)$$

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. The action *getstate*, for example, receives a state identifier and outputs the corresponding state.

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

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 specification of these actions is omitted and can be found in [29].

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

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

4 Translation from Stateflow charts to *Circus*

As explained in the previous section, our models of Stateflow charts are formed by two *Circus* processes composed in parallel: one is the model of the simulator, and the other is the model of a particular chart. The separation between the simulator and chart processes allows us to reduce the amount of specification that must be generated for each chart. In this section, we explain and formalise an algebraic strategy to translate a Stateflow chart into a *Circus* model. It takes the form of a set of translation rules; Section 4.1 introduces the preliminary encodings necessary to formalise the rules, and Section 4.2 discusses the formalisation.

4.1 Preliminary encodings

In this section, we lay the ground for the formalisation of the translation rules. Each rule takes an element of the Stateflow notation and translates it into an element of the *Circus* notation. To allow their formalisation, it is necessary to have a formal model of both notations, so we specify their syntaxes in Z.

The syntax of Stateflow charts is captured by seven main groups of objects: charts, states, junctions, transitions, functions, events and data. Each of these objects is defined by a set of parameters that can include identifiers, names, expressions and actions. We use Z schemas to formalise these objects; for example, a state is formalised by the schema *State* below. This is a concrete representation of a state as it is encoded in the textual accounts of Stateflow charts defined in the tool. It differs from the representation provided by the schema *State* in the previous section in that it contains more information than is necessary in the *Circus* models of Stateflow charts. For instance, the schema *State* in this section includes the syntactical representation of the actions of the state (entry, during and exit actions). In the *Circus* model, this information is encoded as *Circus* actions, not components of the schema *State*.

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

The schema *State* above records the *identifier* and *name* of the state, the identifier of the *parent* of the state, which is either a state identifier (*SID*) or a chart identifier (*CID*), the decomposition (*decomp*) of the state and its *type*, the sequences of *entry*, *exit* and *during* actions, the sequence of event and data names that are bound to the state (*binding*), information about whether the state has a history junction or not (*history*), the identifiers of the first *default* transition (if one exists), the first *inner* transition (if one exists), and the first *outer* transition (if one exists), the identifier of *left* sibling (if one exists), the identifier of the *right* sibling (if one exists), and the sequence of the identifiers of the *substates*.

The set *SID* contains the state identifiers, *SNAME* is the set of names of states, and *CID* is the set of chart identifiers. We define disjoint sets of identifiers for all seven types of objects, and disjoint sets of names for charts, states, functions, events and data. The sets *DECOMP* and *STYPE* identify the decomposition and type of a state. The sets *ACTION* and *DACTION* define the possible actions and during actions. *ENAME* is the set of event names, and *DNAME* is the set of data names. The set of transition identifiers is *TID*. The opt operator is used to declare a component whose value is optional; for example, the component *left* can contain a state identifier, or be empty, indicating that the state has no left sibling.

A during action is either an action from the set *ACTION* defined below, or an *on* action, which is defined as an association between an event name and a sequence of actions.

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

The constructors of *ACTION*, that is, *bcast*, *expr*, *assign*, and *massign*, reflect the various types of actions available. An action can be the broadcasting of an event (represented by the constructor *bcast*), an expression (constructor *expr*), an assignment (constructor *assign*), or a multiple assignment (constructor *massign*). In general, Stateflow actions alter the values of the chart variables and do not have an associated value. The set *EXPR* denotes the set of expressions; these can be logical and arithmetical expressions, name expressions, value expressions, function applications, temporal expressions, and so on.

We formalise well-formedness conditions for the syntax of Stateflow charts. These conditions are not surprising, but are necessary to formalise the translation rules as total functions from well-formed elements of the Stateflow notation. Their definitions can be found in [30].

The *Circus* notation is embedded in Z by defining a series of free types whose constructors are used to build elements of the language, such as actions and processes, as well as elements of the Z notation, such as axiomatic definitions and schemas. The well-formedness conditions and the embedding of *Circus* in Z can be found in [30]. The formalisation of the translation rules is described next.

4.2 Formal translation rules

The model obtained by applying these rules to a particular chart consists of a set of preliminary definitions (such as, identifier, names, bindings, and so on), and a chart process. As already said, the process *c_shift_logic* presented in Section 3.2 is the chart process for the example in Figure 1.

A translation rule is an equation that contributes to the definition of a function mapping a construct of the Stateflow notation, to a *Circus* construct expressed in the embedding discussed in the previous section. The main translation rule defines the total function *translate*; it takes a well-formed chart, and produces a *Circus* program that is composed of Z paragraphs, channel declarations, and process declarations.

$$
\begin{array}{|l}
translate : WF\_CHART \to Program \\
\hline
\forall\, c : WF\_CHART \bullet translate(c) = \textbf{let} \\
identifiers == \langle zparagraphdeclaration(StateIdentifierDeclaration(c)), \\
\quad zparagraphdeclaration(JunctionIdentifierDeclaration(c)), \\
\quad zparagraphdeclaration(TransitionIdentifierDeclaration(c))\rangle; \\
states == set2seq(\{s : \operatorname{ran} c.states \bullet zparagraphdeclaration(StateDeclaration(c, s))\}); \\
junctions == set2seq(\{j : \operatorname{ran} c.junctions \bullet zparagraphdeclaration(JunctionDeclaration(c, j))\}); \\
transitions == set2seq(\{t : \operatorname{ran} c.transitions \bullet zparagraphdeclaration(TransitionDeclaration(c, t))\}); \\
events == \langle zparagraphdeclaration(EventsDeclaration(c))\rangle; \\
channels == \langle ChannelsDeclaration(c)\rangle; \\
proc == \langle processdeclaration(ProcessDeclaration(c))\rangle \bullet \\
program(identifiers \frown states \frown junctions \frown transitions \frown events \frown channels \frown proc)
\end{array}
$$

The function *translate* is defined for all well-formed charts: elements of the set WF_CHART. The result for a chart c is the application of the *Circus* constructor *program* to a sequence of paragraphs formed by the concatenation of seven sequences: *identifiers*, *states*, *junctions*, *transitions*, *events*, *channels* and *proc*.

The sequence *identifiers* contains three paragraphs defining the identifiers of states, junctions and transitions. In our example, it contains, for instance, a paragraph that defines the state identifier *s_gear_state* as a constant of type *SID*. The sequence *states* holds all Z paragraphs that declare state bindings; in our example, this sequence contains an axiomatic definition that declares the name *S_gear_state* of type *State* and equates it to the binding that corresponds to the state gear_state. The function *set2seq* takes a set and returns a sequence whose range is equal to the set. The sequences *junctions* and *transitions* are similar to *states*; they contain the paragraphs that declare junction and transition bindings. Our example does not contain junctions, therefore *junctions* is empty. The sequence *transitions* contains an axiomatic definition that declares, for instance, the name *T_first_second* of type *Transition* and equates it to the binding that specifies the transition from first to second. The paragraph that declares the events is contained in the sequence *events*, and in our example consists of an axiomatic definition that declares the names *e_UP* and *e_DOWN* of type *EVENT*. The channel declarations are in the sequence *channels*. In our example, *channels* contains, for instance, the declaration of *i_speed* and *i_throttle*. The sequence *proc* contains just the definition of the *Circus* process that characterises the chart (for our example, *c_shift_logic*).

To define *translate*, we use the functions *StateIdentifierDeclaration*, *JunctionIdentifierDeclaration*, and *TransitionIdentifierDeclaration*, which produce axiomatic definitions that declare the identifiers of the states, junctions and transitions. For example, *StateIdentifierDeclaration* is defined as follows.

$$
\begin{array}{|l}
StateIdentifierDeclaration : WF\_CHART \to Paragraph \\
\hline
\forall\, c : WF\_CHART \bullet StateIdentifierDeclaration(c) = \textbf{let} \\
\quad snames == set2seq(\{s : \operatorname{ran} c.states \bullet decl(stateid(c, s))\}); \\
\quad cname == \langle decl(chartid(c))\rangle; \\
\quad type == reference(ref(STATEID)) \bullet \textbf{let} \\
\qquad decl == declpart(\langle variable(snames \frown cname, type)\rangle) \bullet \\
\qquad\quad axiomaticdescription(schematext(\langle decl\rangle, \langle\rangle))
\end{array}
$$

For all well-formed charts c, the application of *StateIdentifierDeclaration* to c yields an axiomatic definition built using the constructor *axiomaticdescription* applied to a *SchemaText*. It is obtained by the application of the constructor *schematext* to a sequence of declarations and a sequence of predicates. In the case above, the sequence of predicates is empty, and the sequence of declarations contains the local name *decl* whose value is the application of the constructor *declpart* to a sequence of variables. The sequence of variables is built

by the constructor *variable* applied to a sequence of names and a type expression. The names are those in *snames* and *cname*, that is, the identifiers of the states and of the chart. The type expression is formed from the name *STATEID* that identifies the type *SID* of identifiers of states.

As illustrated above, the formal definition of the translation rules, while useful for reasoning purposes, is hard to read because of the amount of constructor applications used to build the expressions. In the sequel, we present the rules in a semi-formal fashion. The complete formalisation of the rules can be found in [30]. As an example, the rule for *StateIdentifierDeclaration* presented above can be restated as follows.

$$
\begin{array}{l}
StateIdentifierDeclaration(c) = \\
\quad \mid \quad stateid(c.states(1)), \ldots, stateid(c.states(\#\, c.states)), chartid(c) : STATEID
\end{array}
$$

This rule takes a well-formed chart c, and yields an axiomatic definition that declares the state and the chart identifiers to be of the type *STATEID*. The identifiers of the states $c.states(1), \ldots, c.states(\#\, c.states)$ in c are constructed by the function *stateid*. The identifier of the chart is constructed by *chartid*.

The application of this rule to the chart in Figure 1 results in the axiomatic definition sketched below.

$$
\mid \quad s\_downshifting, s\_gear\_state, s\_fourth, s\_second, s\_third, s\_first, \ldots, c\_shift\_logic : SID
$$

An identifier of type *SID* is defined for each state and for the chart.

The functions *StateDeclaration*, *JunctionDeclaration* and *TransitionDeclaration* take a chart and, respectively, a state, a junction, or a transition as parameters, and produce an axiomatic definition that declares a variable of the appropriate type and equates it to a binding that represents the state, junction or transition received as parameter. For example, the function *JunctionDeclaration* is defined as follows.

$$
\begin{array}{l}
JunctionDeclaration(j) = \\
\quad \mid \quad junctionname(j) : JUNCTION \\
\quad \mid \quad \hline \\
\quad \mid \quad junctionname(j) = \langle\!| junction\_identifier(j), junction\_transition(j), junction\_parent(j), junction\_history(j) |\!\rangle
\end{array}
$$

The name of the junction j is determined by the function *junctionname* that guarantees its uniqueness; this name is declared to be a constant of type *JUNCTION*. The predicate of the axiomatic definition defines that the value of the constant is the binding formed by the results of the applications of the functions *junction_identifier*, *junction_transition*, *junction_parent*, and *junction_history* to j. Each of these functions produces an element of a binding that equates a component of the schema *Junction* (see Section 3.2), and a value of that component. For example, *junction_identifier* is defined as follows.

$$
junction\_identifier(j) = IDENTIFIER == junctionid(c, j)
$$

The result of this function is an association between the component whose name is given by *IDENTIFIER* and the identifier of the junction obtained by the function *junctionid*. The functions *StateDeclaration* and *TransitionDeclaration* are similarly defined, as are *EventsDeclaration* and *ChannelsDeclaration*.

The rule *junction_identifier* when applied to the bottom junction in the example in Figure 2 yields the following expression $identifier == j\_23$. This is used in the application of the rule *JunctionDeclaration* to the same junction resulting in the following axiomatic definition.

$$
\begin{array}{l}
\quad \mid \quad J\_23 : JID \\
\quad \mid \quad \hline \\
\quad \mid \quad J\_23 = \langle\!| identifier == j\_23, transition == t\_23\_FACT, parent == s\_FUNCTIONS, history == \mathbf{False} |\!\rangle
\end{array}
$$

The name J_23 and the identifier j_23 are unique in the model, as is t_23_FACT.

The process declaration rule takes a well-formed chart and produces a *Circus* process. We show a partial definition of this rule below. The result of its application to the chart shown in 1 is the process sketched in Figure 8. For a chart c, this gives a process whose name is the result of applying *processname* to c; this yields the name of the chart. The body of the process consists of a series of schema and action declarations

obtained by applying the appropriate translation functions.

$$
ProcessDeclaration(c) = \left(\begin{array}{l}
\textbf{process}\ processname(c) \mathrel{\widehat=} \textbf{begin} \\
\quad ChartDeclaration(c) \\
\quad SimulationInstanceDeclaration(c) \\
\quad InitSimulationInstanceDeclaration(c) \\
\quad SimulationDataDeclaration \\
\quad InitSimulationDataDeclaration \\
\quad InitState == InitSimulationInstance \land InitSimulationData \\
\quad HasActiveSubstateDeclaration \\
\quad ActivateDeclarations \\
\quad DeactivateDeclarations \\
\quad \textbf{state}\ (SimulationInstance \land SimulationData) \\
\quad EntryActionDeclaration(s_1) \\
\quad \ldots \\
\quad EntryActionDeclaration(s_n) \\
\quad EntryActionsDeclaration(c) \\
\quad \ldots \\
\quad ConditionActionsDeclaration(c) \\
\quad \ldots \\
\quad ConditionsDeclaration(c) \\
\quad \ldots \\
\quad TriggersDeclaration(c) \\
\quad GetStateDeclaration \\
\quad \ldots \\
\quad \bullet\ MainActionDeclaration \\
\textbf{end}
\end{array}\right)
$$

For instance, the axiomatic definition that declares the chart structure (states, junctions, and transitions) is specified by the application of the translation rule *ChartDeclaration*. Other functions define the *SimulationInstance* schema and its initialisation operation. The state is always the same: the conjunction of *SimulationInstance* and *SimulationData*. The schemas that define the activation and deactivation operations are declared by the rules *ActivateDeclarations* and *DeactivateDeclarations*. All the groups of actions are also constructed by specific functions. For example, the entry actions are translated by *EntryActionDeclaration*. Finally, the function *MainActionDeclaration* determines the main action of the chart process.

For a state s, the function *EntryActionDeclaration* gives the encoding of a *Circus* action modelling the entry action of s.

$$
\begin{array}{l}
EntryActionDeclaration(s) = (EntryActionName(s) \mathrel{\widehat=} \\
\quad executeentryaction.stateid(s)?o?ce \longrightarrow translate_{Acts}(s.entry))
\end{array}
$$

The name of the resulting *Circus* action is obtained by the function *EntryActionName* based on the name of s. The action itself is a prefixing with a communication through *executeentryaction* of the identifier of the state and of inputs o and ce, followed by the translation of the sequence of the entry actions $s.entry$ of s as defined by $translate_{Acts}$. The application of *EntryActionDeclaration* to the state first in Figure 1 is shown in Section 3.2, as well as the result of the application of the rule $translate_{Acts}$ to the assignment in the entry action of that state. All the rules for state and transition actions are specified in a similar fashion.

The function $translate_{Acts}$ translates sequences of actions from the set $ACTION$. The rule that specifies it constructs a *Circus* action by composing in sequence the results of applying the function $translate_{Act}$ to each action in the sequence. The function $translate_{Act}$ is defined inductively for each type of Stateflow action; for example, the definition of this function for an assignment $name = e$ is shown below.

$$
translate_{Act}(name = e) = dataname(getdatabyname(name)) := translate_{Expr}(e)
$$

We recover the data to which *name* refers, using the function *getdatabyname*, determine its name using the function *dataname*, and use it and the translation of the expression e to build an assignment.

Multi-dimensional vectors are specified as finite functions, and the assignment $name[e_1] \ldots [e_n] = e$ of a value e to a position identified by the indices $[e_1] \ldots [e_n]$ needs to alter the function *name* only in that position. The corresponding translation rule is presented below. We obtain the name of the data, and build an assignment from that and from an expression built by applying the override operator ($\oplus$) 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.

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

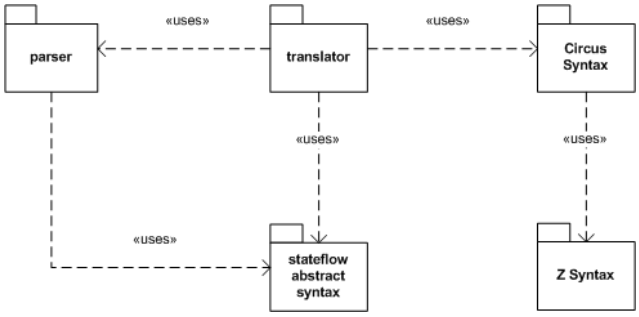

Figure 9: Architecture of *s2c*: main packages.

The complete formalisation of the translation strategy that supports the construction of *Circus* models of Stateflow charts consists of over 80 formal translation rules. It has been mechanically checked, and can be found in [30]. In the next section, we discuss the implementation of the translation rules in the tool *s2c*.

5 The *s2c* translation tool

The process of manual translation is extremely error prone. Moreover, the translation of larger diagrams is prohibitively costly. One of our experiments on an industrial case study yielded a 180-page specification.

A mechanisation of the translation process can reduce costs, thus increasing the viability of any task that depends on a *Circus* model. In addition, it 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 of translation errors.

In this section, we describe the tool *s2c*, which implements the translation rules discussed in the previous section. This tool takes a *.mdl* file, that is, the file provided by Simulink that contains a textual representation of charts, and produces a *Circus* process for each of the charts. The *.mdl* file is assumed to define a well-formed Stateflow chart, in the sense that it is properly compiled by the simulator. Since Simulink and graphical functions are not translated, but are widely used in charts along with *MATLAB* and *C* functions, we deal with them by requiring the user to supply a Z definition for them (in Z libraries).

The implementation of *s2c* has approximately 5000 lines of Java code, excluding automatically generated parsing classes. It can be found in `www.cs.york.ac.uk/circus/s2c`. There we also present various examples of *.mdl* files describing diagrams whose *Circus* models we have generated using *s2c*.

Section 5.1 describes the architecture of *s2c*, and Section 5.2 elaborates on the implementation of rules.

5.1 Architecture

The implementation of *s2c* is organised in five main packages: `parser`, `stateflow abstract syntax`, *Circus* `syntax`, `Z syntax` and `translator`. Figure 9 depicts them and their relationships. The `translator` package is the main one, and uses all the other packages to perform the translation. The packages `stateflow abstract syntax`, *Circus* `syntax`, and `Z syntax` mechanise the formalisation of the Stateflow syntax and *Circus* syntax, and the package `parser` contains the automatically generated classes that take a textual representation of Stateflow charts and builds the corresponding syntactic representation. The package `translator` holds the main part of the tool, that is, the implementation of the translation rules. Figure 10 shows the architecture of the `translator` package. In the sequel, we discuss each package in more detail.

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 has an implementation of the *Circus* syntax and of the `NameGenerator` interfaces. They determine the methods that must be implemented to generate a specification in the appropriate mark-up language. Our implementations generate text that uses the *Circus* LaTeXmark-up; this is the notation currently accepted by the parser.

The name generator takes Stateflow objects and constructs their names in an appropriate format. For example, we disambiguate state names (when needed) by prefixing the name of the state's parent. 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". The `Translator` class uses a static class `ObjectGetter` that contains methods for recovering objects from the chart using their names or identifiers.

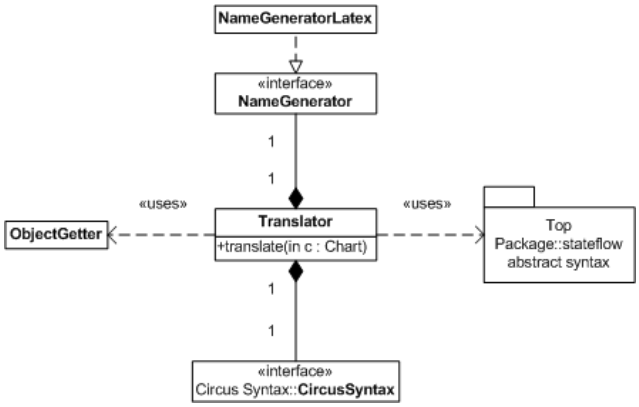

Figure 10: Architecture of *s2c*: the package `translator`.

5.2 Implementation of translation rules

The translation of Stateflow diagrams into *Circus* models is defined in a manner that makes it suitable for mechanisation: using formal, yet very concrete, translation rules as presented in the previous section. The design 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 is that it is easy to identify how components of the code map to the formal rules, thus facilitating the validation of the implementation and of the rules.

We illustrate the strategy used for implementing the translation rules through an example. The formal specification of the translation function for name expressions is shown below.

$$
\begin{array}{|l}
translate_{Expr} : WF\_EXPR \rightarrow Expression \\
\hline
\forall\, n : DNAME;\ es : \mathrm{seq}\, EXPR;\ c : WF\_CHART \mid (c, name(n, es)) \in WF\_EXPR \bullet \\
\quad translate_{Expr}(c, name(n, es)) = \\
\qquad \left(\begin{array}{l} \textbf{if } \#\, es = 0 \\ \textbf{then } reference(ref(dataname(c, getdatabyname(c, n)))) \\ \textbf{else } functionapplication(app(dataname(c, getdatabyname(c, n)), translate_{Exprs}(c, es))) \end{array} \right)
\end{array}
$$

This function takes a well-formed expression (an element of the set WF_EXPR) and produces a Z (*Circus*) expression. In the case of the rule above, the resulting Z expression depends on whether the name expression $name(n, es)$ represents a simple variable or a position in an vector. If it is a simple variable ($\#\, es = 0$), the function returns a reference to the appropriate data by applying the constructors *reference* and *ref* to the unique name of the data. If the name expression is a position in an array, $translate_{Expr}$ returns the application of the name of the appropriate data ($dataname(c, getdatabyname(c, n))$) to the translated expressions contained in the index sequence ($translate_{Exprs}(c, es)$) by using the constructors *functionapplication* and *app*; multi-dimensional vectors are implemented as functions from a set of indexes to $\mathbb{R}$.

Figure 11 shows the implementation of the rule $translate_{Expr}$ for name expressions: 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` of the package `translator`), 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 translation rule for name expressions is straightforward once we provide an embedding of the syntax of Stateflow diagrams, of *Circus*, and of Z operations used in the formal specification.

For a complete discussion about the implementation of the translation rules, see [30]. In the next section, we discuss the validation of *s2c*, covering also the model of Stateflow charts and the translation rules.

5.3 Validation

As a first step in the validation of our approach to formal modelling of Stateflow charts, we have compared our models to the informal description of the semantics of Stateflow charts. This is made possible by the way the model of the simulator is built; the strong relation between the structure of the formalisation and of the informal description allows us to compare them and assess their correspondence.

```
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 11: Implementations of the translation rule for variable expressions.

We have also animated the models and compared the results to those of the Stateflow simulator. For that, we have translated the *Circus* model to CSP, and used ProBE [31] (a CSP animator) and FDR2 [32] (a CSP model checker) to simulate the CSP specification. We have used ProBE to manually investigate traces of the model, and FDR2 to automatically test if a trace obtained from the simulator is a trace of the model.

The formalisation of the translation strategy has yielded a third form of validation of the models. The validation of the translation rules has reinforced our confidence on the model of Stateflow charts by forcing a systematic review of the model, and by supporting the formal derivation of models of Stateflow charts. The formalisation of the translation rules are correctly parsed and type-checked by the CZT toolkit [33].

Finally, the implementation and validation of *s2c* provides validation also for the model and translation rules. An automatic translation tool like *s2c* supports the derivation of models of large and complex Stateflow charts. This allows us to evaluate the robustness of the tool and of the translation rules. In this way, we have a chain of validation that culminates in the evaluation of *s2c*.

Our evaluation strategy has consisted of testing *s2c* on three sets of examples: basic charts, complete examples, and industrial case studies. These examples, excluding the industrial case studies and those that accompany Stateflow, are in `www.cs.york.ac.uk/circus/s2c`. The basic charts exercise specific features of Stateflow; for example, for each type of action, we have a chart with a single state with an entry action of that type. These examples revealed mainly programming errors and limitations of the translation rules.

The complete examples are charts (or collection of charts) that model a simplified system (usually toy-examples), but whose complexity is larger; the chart in Figure 1 belongs to this group. The tests using charts in this set pointed out to the same types of error reveled by the tests with the basic charts, and some errors in the translation of events, in particular, in the calculation of the source and destination states.

The third group of examples includes two models supplied by industrial collaborators. The tests with the industrial case studies pointed out 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.

We have used the basic-charts test set in different stages of development, and errors found have been corrected. The same has been done for the other examples, but less frequently.

In testing with the basic charts, we can track any errors 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 Section 3. 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 specifications generated by industrial case-studies are very large, and, therefore, not tractable manually. While it is possible to track translation errors to pieces of the model, it is a demanding task. Testing industrial case studies, nevertheless, has allowed us to evaluate the robustness and scalability of the tool.

The treatment of functions is not a simple issue, because they are not restricted to the types of functions that can be defined within a chart: MATLAB and even C functions can be used. As previously discussed, our approach to this assumes the existence of a library of Z definitions of the functions 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, that is, graphical functions that change the state of the chart. The symbol t represents the "absolute time inherited from a Simulink signal" [2]. Its proper treatment is, therefore,

delegated to the Simulink diagram and updated through a channel *time*.

The industrial case studies have been provided by Embraer and Airbus. The example from Embraer has been translated without difficulties. It consists of a single chart, with a very simple structure. It uses the *abs* function; this reinforced the need to consider the treatment of functions, and motivated the use of Z libraries. The case study from Airbus is larger; it contains over 50 charts, organised in a five-level hierarchy. In general, the individual charts have a simple structure. History junctions are not used, but parallel and sequential states, connective junctions, and functions are extensively used. The Stateflow model seems to avoid the use of events, and transitions are controlled by boolean variables. The translation of four charts resulted in a 180-page specification. It also requires the treatment of functions and the symbol t. The charts contained in both case studies extensively used functions (MATLAB functions and graphical functions), and that the approach to the treatment of general functions by use of Z libraries has proved successful.

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 [33].

6 Conclusions

In this paper, we have proposed a semantics of Stateflow charts based on *Circus*, a formal notation that supports analysis of models and verification of implementations based on refinement. The operational nature and the structure of the model provide an informal, but intuitive, relationship with the description of Stateflow in the manual; this has allowed validation by inspection. The separation between the structure of a Stateflow chart and its interpretation (simulation semantics) reduces the amount of specification that must be derived for each chart, and also limits the scope of any changes that might be necessary.

We have defined and formalised (using Z) a strategy for the translation of Stateflow charts to a *Circus* process. The formalisation depends on Z embeddings of the Stateflow and *Circus* notations, and is structured inductively over the syntax of Stateflow charts. The translation process is carried out in a top-down manner, and produces a specification that contains the chart process, all auxiliary definitions (identifier, bindings, channels, and so on), and references to external libraries of function definitions.

We have also implemented the translation strategy in the tool *s2c*. The design of *s2c* implements the translation rules in a manner that emphasises the correspondence between the code and the formalisation of the rules. We have extensively tested *s2c* with various examples (from toy examples to industrial case studies), and the results of the translation are correctly parsed and type-checked.

Our approach to modelling Stateflow charts can cater for edge-triggered events, data, actions, parallel and exclusive states, connective and history junctions, and all forms of transitions. The features that distinguish this model are the unrestricted treatment of states, junctions and transitions, the partial support of functions, the treatment of local event broadcasts and multi-dimensional data types, and the possibility of verifying implementations of Stateflow charts in the context of Simulink diagrams.

The handling of transitions to history junctions has raised some issues. This kind of scenario is poorly documented in the *User's Guide*, and only inner transitions to history junctions are mentioned. In our model, inner and outer transitions to history junctions work as observed in the Stateflow simulator, but default transitions lead to nontermination, instead of issuing an error.

There are several approaches to the formal analysis of Stateflow diagrams. Operational and denotational semantics are proposed by Hamon and Rushby [34], and Hamon [35]; these support static analysis, interpretation, and compilation of Stateflow charts. Concerns with coverage of the simulation technique provided by Simulink are addressed in [36, 37] using symbolic analysis of simulation traces. Translations of Stateflow into notations that support model checking are presented in Banphawatthanarak and Krogh [38], Tiwari [39], and Scaife et al. [40]. Verification in these approaches is based on temporal logics and bisimulation, rather than refinement, thus verification of implementations is not the objective. Toyn [41] uses Z to verify that the chart satisfies a set of requirements, but places strong restrictions on the Stateflow notation.

Most of the work done on formalising the Stateflow notation tries to identify a well-behaved subset of the language, and then defines the semantics of the notation restricted to that subset. Hamon and Rushby [34] provide an operational semantics of a subset of the Stateflow notation. Their semantics has been validated against the Stateflow simulator, and supports execution and efficient compilation into other notations (notably imperative languages and input languages of model checkers). This work, however, does not cover history junctions and imposes restrictions on transitions.

Hamon [35] proposes a denotational semantics based on an interesting use of continuations to capture the execution of transitions. The main goal is to formalise the compilation process of Stateflow charts into other notations. Hamon claims that the denotational semantics covers most of the notation, but history junctions, Simulink functions, and events of type function-call are not mentioned. Moreover, the possibility of early return logic in the context of local event broadcasts seems not to be taken into account.

Banphawatthanarak et al. [42] propose a translation from Stateflow charts to the notation of the SMV symbolic model checker. This allows the verification of chart properties using a very successful tool. They applied their technique to a chart that specifies a two-mode estimation algorithm, and proved three properties. They, however, impose restrictions on the types of input signals, number of transitions reaching a junction, outputs signals, and transition actions used in a chart. A customised tool is available.

Scaife et al. [40] propose and implement a strategy to translate Stateflow to the synchronous language Lustre [43], which also provides support for model checking. They applied their technique to an alarm controller, and their results lead to a review of the original model. This work, however, also imposes a series of restrictions on the charts. It does not allow multi-segment transition paths that represent loops. Variable assignment on transitions must be made on transition actions or the condition action of the last segment. It forbids backtracking by requiring that conditions of outgoing transitions from junctions form a cover, that is, the conjunction of the conditions is true. It also avoids relying on ordering determined by position of elements in the chart by requiring that the transitions leaving a node have disjunct conditions, for instance. On the other hand, they can also construct Lustre models for Simulink diagrams.

Toyn and Galloway [41] formalise Stateflow charts in Z. The models are combined with requirements on the states of the chart to produce healthiness conditions. Their verification validates the model with respect to the requirements of the system. This work imposes even stronger restrictions on charts; it does not cover parallel states, junctions, local variables, and so on. A tool automates the construction of healthiness conditions, and the Z theorem prover CADiZ is used to prove them. CADiZ tactics are used to automate these proofs. Their case study is a simplified model of a jet-engine starting process.

We treat graphical and Simulink function only partially. In principle, however, given the current work and the existing treatment of Simulink diagrams [25] in *Circus*, we are able to use the models of Stateflow charts and Simulink diagrams in order to represent such functions.

We do not treat function-call events. This type of event can be supported in the current model by refining the definition of events to account for type, modifying the procedure for reading input variables and writing output variables, and modifying the model of Simulink diagrams in order to execute the function-call block and the chart in interleaving. 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 broadcasted. Elements such as sub-charts and super-transitions are also not covered by our approach; we believe that the treatment of these elements will be simple, because they are basically the encapsulation of already supported elements. As far as we know, none of the works discussed above cover any of these features either.

Our treatment of input and output events and data is not entirely consistent with that of the Simulink model presented in [25]. To fully integrate them, the Simulink model must be extended to conform to the input and output protocols of the Stateflow block. The standard protocol is for the chart to read the input variables in the beginning of a cycle, and pass the output variables to the Simulink diagram in the end of the cycle. This does not occur when we have a broadcast of an output event of type function call. In this case, the execution of the function-call subsystem is parallel to that of the broadcasting chart, and the outputs of the subsystem are immediately available to the chart during the cycle.

As a more substantial piece of future work, we will devise a refinement strategy that allows us to verify implementations of Stateflow charts. It will be compatible with the strategy formalised for verification of Simulink diagrams, so that implementations of a wider range of diagrams can be verified. The *Circus* model of a Simulink diagram is a parallel composition of processes that model the blocks. The refinement strategy presented in [44] consists of four steps that systematically collapse the massive parallelism of the specification in order to match the processes of the implementation model, prove that each of the procedures in the implementation refine the action that specifies it in the corresponding process, and finally, prove that the main programs refine the process that specifies the system. The main actions of component processes are always put in a normal form where they are defined as the iterative execution of a step. The steps consists of reading the inputs in interleaving, calculating the outputs and updating the state, writing the outputs in interleaving, and synchronising on the channel *end_cycle*.

The step of execution of the *Simulator* process is similar, but it has an extra initial step that waits for an event; moreover, the step can be interrupted by the chart. We need to define a strategy to collapse the parallelism between the chart process and the *Simulator* process, and put the main action of this new process in the normal form. It is possible that the requirements of the refinement strategy of the implementation of a Stateflow chart are different from those of a Simulink diagram; this needs to be assessed and the refinement strategy must cater for this scenario. The strategy will then be applied to the case studies.

References

[1] The MathWorks,Inc., Simulink, `http://www.mathworks.com/products/simulink`.

[2] The MathWorks,Inc., Stateflow and Stateflow Coder 7 User's Guide, http://www.mathworks.com/products/stateflow.

[3] The MathWorks,Inc., Simulink Design Verifier, http://www.mathworks.co.uk/products/sldesignverifier.

[4] The MathWorks,Inc., Simulink Verification and Validation, http://www.mathworks.com/products/simverification.

[5] The MathWorks,Inc., Real-Time Workshop, http://www.mathworks.com/products/rtw.

[6] The MathWorks,Inc., Stateflow Coder, http://www.mathworks.com/products/sfcoder.

[7] The MathWorks,Inc., xPC Target, http://www.mathworks.com/products/xpctarget.

[8] BS EN 61508-3:2002, Functional safety of electrical/electronic/programmable electronic safety related systems – Part 1: General requirements (2002).

[9] DO-178b, Software considerations in airborne systems and equipment certification (1992).

[10] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, P. Niebert, From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications, ACM SIGPLAN Notices 38 (7) (2003) 153–162. doi:http://doi.acm.org/10.1145/780731.780754.

[11] A. Toom, T. Naks, M. Panter, M. Grandriau, 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.

[12] R. Lublinerman, C. Szegedy, S. Tripakis, Modular code generation from synchronous block diagrams: modularity vs. code size, in: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '09, ACM, New York, NY, USA, 2009, pp. 78–89. doi:http://doi.acm.org/10.1145/1480881.1480893.
URL http://doi.acm.org/10.1145/1480881.1480893

[13] A. Cavalcanti, P. Clayton, C. O'Halloran, From control law diagrams to Ada via *Circus*, Formal Aspects of Computing (2011) 1–4810.1007/s00165-010-0170-3.
URL http://dx.doi.org/10.1007/s00165-010-0170-3

[14] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, Unifying Theories in ProofPowerZ, Formal Aspects of Computing, online firstDOI 10.1007/s00165-007-0044-5.

[15] J. Woodcock, J. Davies, Using Z: specification, refinement, and proof, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[16] A. W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall Series in Computer Science, Prentice-Hall, New York, 1998, Oxford.

[17] E. W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (8) (1975) 453–457. doi:http://doi.acm.org/10.1145/360933.360975.

[18] C. C. Morgan, Programming from Specifications, 2nd Edition, Prentice Hall International Series in Computer Science, 1994.

[19] C. A. R. Hoare, H. Jifeng, Unifying Theories of Programming, Prentice Hall, 1998.

[20] A. L. C. Cavalcanti, A. C. A. Sampaio, J. C. P. Woodcock, A Refinement Strategy for *Circus*, Formal Aspects of Computing 15 (2 - 3) (2003) 146 — 181.

[21] M. A. Xavier, A. L. C. Cavalcanti, A. C. A. Sampaio, Type Checking *Circus* Specifications, in: A. M. Moreira, L. Ribeiro (Eds.), SBMF 2006: Brazilian Symposium on Formal Methods, 2006, pp. 105 – 120.

[22] M. V. M. Oliveira, A. L. C. Cavalcanti, ArcAngelC: a Refinement Tactic Language for *Circus*, Electronic Notes in Theoretical Computer Science **214C** (2008) 203 – 229, Elsevier B. V.
URL http://dx.doi.org/10.1016/j.entcs.2008.06.010

[23] A. F. Freitas, A. L. C. Cavalcanti, Automatic Translation from *Circus* to Java, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), FM 2006: Formal Methods, Vol. 4085 of Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 115 – 130.

[24] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, Unifying theories in ProofPower-Z, in: S. Dunne, B. Stoddart (Eds.), UTP 2006: First International Symposium on Unifying Theories of Programming, Vol. **4010** of LNCS, Springer-Verlag, 2006, pp. 123–140.

[25] A. L. C. Cavalcanti, P. Clayton, C. O'Halloran, Control Law Diagrams in *Circus*, in: J. Fitzgerald, I. J. Hayes, A. Tarlecki (Eds.), FM 2005: Formal Methods, Vol. 3582 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 253 – 268.

[26] R. Arthan, P. Caseley, C. O'Halloran, A. Smith, ClawZ: control laws in Z, in: ICFEM, 2000, pp. 169.

[27] M. M. Adams, P. B. Clayton, ClawZ: Cost-Effective Formal Verification for Control Systems, in: K.-K. Lau, R. Banach (Eds.), ICFEM, Vol. 3785 of Lecture Notes in Computer Science, Springer, 2005, pp. 465–479.
URL `http://dx.doi.org/10.1007/11576280_32`

[28] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274.

[29] A. Miyazawa, A. Cavalcanti, Towards the formal verification of implementations of Stateflow Diagrams, Tech. Rep. YCS-2010-449, University of York (2010).

[30] A. Miyazawa, A. Cavalcanti, A formal semantics of Stateflow charts, Tech. Rep. YCS-2011-461, University of York (2011).

[31] Formal Systems (Europe) Ltd., Process Behaviour Explorer, `www.fsel.com`.

[32] Formal Systems (Europe) Ltd., Failures-Divergence Refinement, `www.fsel.com`.

[33] P. Malik, M. Utting, CZT: A Framework for Z Tools, in: ZB. Lecture, Springer, 2005, pp. 65–84.

[34] G. Hamon, J. Rushby, An operational semantics for Stateflow, in: M. Wermelinger, T. Margaria-Steffen (Eds.), Fundamental Approaches to Software Engineering:7th International Conference (FASE), Vol. 2984 of Lecture Notes in Computer Science, Springer-Verlag, Barcelona, Spain, 2004, pp. 229–243.

[35] G. Hamon, A denotational semantics for stateflow, in: W. Wolf (Ed.), EMSOFT, ACM, 2005, pp. 164–172.
URL `http://doi.acm.org/10.1145/1086228.1086260`

[36] R. Alur, A. Kanade, S. Ramesh, K. C. Shashidhar, Symbolic analysis for improving simulation coverage of simulink/stateflow models, in: Proceedings of the 8th ACM international conference on Embedded software, EMSOFT '08, ACM, New York, NY, USA, 2008, pp. 89–98. doi:http://doi.acm.org/10.1145/1450058.1450071.
URL `http://doi.acm.org/10.1145/1450058.1450071`

[37] A. Kanade, R. Alur, F. Ivančić, S. Ramesh, S. Sankaranarayanan, K. C. Shashidhar, Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models, in: Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 430–445. doi:http://dx.doi.org/10.1007/978-3-642-02658-4_33.
URL `http://dx.doi.org/10.1007/978-3-642-02658-4\_33`

[38] C. Banphawatthanarak, B. H. Krogh, Verification of stateflow diagrams using smv: sf2smv 2.0, Tech. Rep. CMU-ECE-2000-020, Carnegie Mellon University (2000).

[39] A. Tiwari, Formal semantics and analysis methods for Simulink Stateflow models, Tech. rep., SRI International, `http://www.csl.sri.com/~tiwari/stateflow.html` (2002).

[40] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, F. Maraninchi, Defining and translating a "safe" subset of simulink/stateflow into lustre, in: G. C. Buttazzo (Ed.), EMSOFT, ACM, 2004, pp. 259–268.
URL `http://doi.acm.org/10.1145/1017753.1017795`

[41] I. Toyn, A. Galloway, Proving properties of Stateflow models using ISO Standard Z and CADiZ, in: In ZB-2005, vol. 3455 of LNCS, 2005, pp. 104–123.

[42] C. Banphawatthanarak, B. Krogh, K. Butts, Symbolic verification of executable control specifications, in: Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design, 1999, pp. 581–586.

[43] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data-flow programming language LUSTRE, in: Proceedings of the IEEE, Vol. 79, 1991, pp. 1305 – 1320.

[44] A. L. C. Cavalcanti, P. Clayton, Verification of Control Systems using *Circus*, in: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, 2006, pp. 269 – 278.