Available online at www.sciencedirect.com
ScienceDirect

Electronic Notes in Theoretical Computer Science 240 (2009) 23–41

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Stateflow Diagrams in *Circus*

Ana Cavalcanti[1],[2]

Department of Computer Science
University of York
York, UK

Abstract

The Matlab Simulink tool is widely used to construct and analyse control law diagrams. Many have worked on techniques to enhance analysis facilities, and previously, we have considered the complementary problem of proving correctness of implementations of diagrams. We use *Circus*, a refinement language that combines Z and CSP, and can capture both functional and behavioural aspects of diagrams and programs. We defined a *Circus* semantics for an extensive subset of discrete-time diagrams, and now extend it to cover Stateflow blocks, which are themselves defined by diagrams written in (a variant of) the statechart notation. We highlight the challenging features of the semantics of a diagram, describe how *Circus* models can be constructed, and discuss the formalisation of the *Circus* semantics as algebraic translation rules.

Keywords: Simulink, semantics, refinement.

1 Introduction

Control law diagrams are very popular among engineers as a design notation for control systems; in particular, the Simulink tool is widely used in the avionics and automotive sections [11]. It supports construction of diagrams and analysis based on simulation. A diagram is composed of blocks connected by wires. The input and output signals are indicated by special blocks; the wires determine the flow of signals and blocks embed functionality that determines how the outputs are calculated.

Typically, the mathematical model of a physical system is given by differential equations that relate the inputs and outputs. Control law diagrams provide a graphical representation of the calculations specified by the equations. Often, however, the behaviour of the system changes when certain events occur or conditions are met. To specify the different modes of operation, finite state machines are

[1] This work is funded by EPSRC (research grant EP/E025366/1). We are grateful for Jim Woodcock's comments on a draft of this paper.

[2] Email: Ana.Cavalcanti@cs.york.ac.uk

doi:10.1016/j.entcs.2009.05.043

convenient. Simulink allows the seamless combination of blocks that embed standard mathematical calculations, and blocks whose outputs are defined by Stateflow diagrams: a variant of Harel's statecharts [9].

There are many variants of the statechart notation [10]; that used as part of Simulink is called Stateflow. In [2], Stateflow models are translated to the SMV model checker input language, which describes finite-state machines. The models used in [18] are communicating pushdown automata, which are then translated to a language called SAL, which specifies transition systems. Translation to Lustre, a synchronous dataflow language, is discussed informally in [17]. Restrictions on the use of recursion in Lustre impose restrictions on the kind of diagrams that can be handled, but the approach includes checking that features of Stateflow that may lead to undesirable models are avoided. Formal semantics are considered in [7,8]. An operational semantics is provided in [8]; it was used to translate from Stateflow diagrams to SAL. Later, in [7], the subset of Stateflow covered is extended: a denotational style is adopted, and continuations are used to cope with the challenging structure of Stateflow diagrams. The denotational semantics defines diagrams as functions on environments that record the active states and the value of the data.

We are concerned with verification of implementations, rather than analysis, of discrete-time Simulink diagrams. In [4], we define the semantics of these diagrams using *Circus* [5], a language for refinement that combines the Z notation [20] and CSP [16] to specify state-rich reactive systems, and has a refinement theory and calculational technique. Using *Circus*, we have extended an industrial highly automated Z-based technique for verification of Ada procedures that implement block functionality [1]. We verify both the Ada procedures and the scheduler [3]. The technique is based on the controlled application of algebraic laws, and is amenable to high levels of automation using tactics of refinement [14].

The subset of Simulink that we model using *Circus* is extensive, including action and enabled subsystems and merge blocks. Here, we take a first step to extend this work to tackle Stateflow blocks. We give a *Circus* semantics to Stateflow diagrams that can be used as a component of the model of the Simulink diagram that includes it. We aim at using refinement to verify implementations of Stateflow blocks.

A CSP semantics for the statechart notation as used in UML state diagrams is provided in [13]. In addition, *Circus* is used in [15] to give semantics to UML-RT, a UML profile tailored for concurrent applications. That work covers the state diagrams, which are again a variant of statecharts, and also part of a more comprehensive description of the system including class and structure diagrams. In both works, refinement is the basis for reasoning, either using the CSP refinement model checker, or refinement laws for model transformation. The statechart diagrams that we consider here, however, are rather different from those in UML and UML-RT.

The Stateflow notation is rather complex. A state can have outer transitions that lead to state change, and inner transitions that model computations carried out whenever the state is active: a state can itself include flowcharts. Junctions can be used to break an outer or inner transition in segments. Transitions to junctions backtrack if they do not eventually lead to a new state, or to a terminal junction.

Each transition has two associated actions: one is executed every time the transition is tried, and another only if it is not backtracked.

In this paper, we propose a *Circus* semantics that covers all these features. In the next section, we give a brief overview of *Circus*, and Section 3 explains the main features of Stateflow diagrams. The structure of the *Circus* models of Stateflow diagrams that we propose is discussed in Section 4; formalisation of the semantics is considered later on in Section 5. In Section 6 we use small examples to illustrate how we can handle some of the more intricate features of the diagrammatic notation. Finally, in Section 7, we discuss related and future work.

2 *Circus*

Various combinations of state-based formalisms with process algebras have been proposed [6,19]. *Circus* distinguishes itself as a refinement language. Apart from the constructs of the well established notations Z and CSP, *Circus* also includes imperative commands from Dijkstra's language of guarded commands, and a refinement theory and technique in the style of Morgan's calculus [12].

Circus programs are sequences of paragraphs like in Z. It is, however, also possible to declare channels, channel sets, and processes.

A process encapsulates a state and exhibits some behaviour. The state is defined by schemas, like in Z. The behaviour is defined using a (main) action, which possibly combines data operations specified in Z, CSP constructs like prefixing, choice, and parallelism, and imperative commands like assignments and conditionals.

Processes can themselves be combined using CSP operators. The state of the resulting process contains all components of the combined processes. Its behaviour is defined by combining their main actions using the CSP operator.

Several examples of *Circus* processes are presented in Section 4.

3 Stateflow diagrams

Figure 1 shows a Simulink diagram, a variant of which can be found in the Simulink demonstration files [11]. The application is an automatic transmission controller; the diagram models the engine, the transmission, and the wheels.

The transmission uses the power generated by the engine to move the wheels. The block ManeuversGUI defines the inputs Throttle and BrakeTorque; the block PlotResults defines the outputs EngineRPM and VehicleSpeed. The system takes the amount of air entering the engine (that is, the throttle) and the force applied on the brakes, and outputs the speed of the engine and of the vehicle. Its behaviour is cyclic: it repeatedly samples the inputs and produces the outputs, possibly using information calculated in the previous cycle.

The blocks of the diagram model the components of the system. A transmission has two impellers: one interacts with the engine and the other with the wheels. Roughly, the Engine block calculates the speed of the engine as a function of the torque of the first impeller and the throttle. The engine speed, the gear, and the

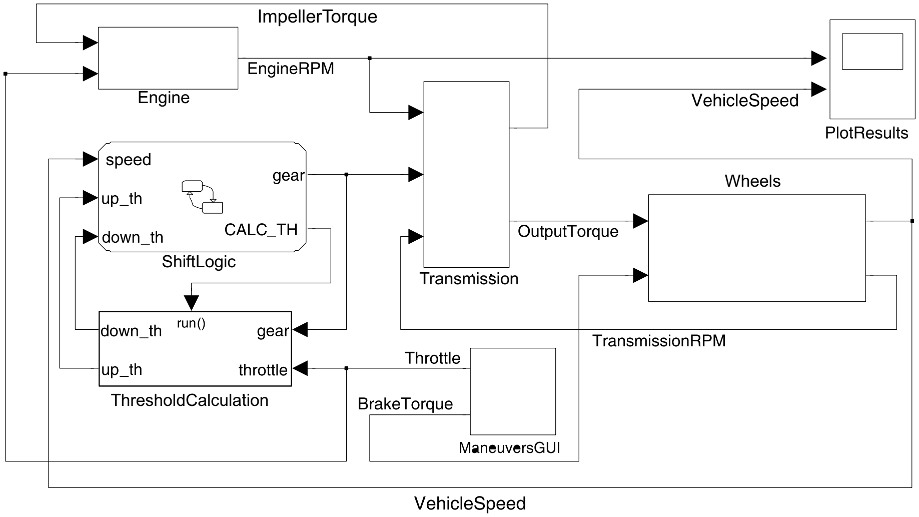


Fig. 1. Simulink diagram: automatic transmission

speed of the wheels are used to calculate the torque of the impellers in Transmission. The block Wheels computes the transmission and vehicle speeds. The speed thresholds for shifting gears up and down are calculated in the block ThresholdCalculation. All these calculations are governed by differential equations that are captured by the functionality of these standard Simulink blocks.

The change of gears, however, is based on conditions involving the current speed and the gear thresholds. To specify that, a Stateflow diagram is used; it defines the behaviour of the block ShiftLogic, and is presented in Figure 2.

In this diagram, the blocks represent states and the lines are transitions. In our example, we have two (parallel) states gear_state and selection_state; they are always active in every cycle. These states have substates, of which just one is active at a time. In the first cycle, the first state of gear_state and the steady_state of selection_state are activated, as indicated by the default transitions: those without a source. In the subsequent cycles, the transitions of each active state are tried, and followed if possible. A cycle finishes when, for every active state, a valid transition, if any, is followed; in addition, if any events local to the Stateflow diagram are generated, they need to be treated in the same cycle.

In the example, if the speed becomes higher or lower than the relevant thresholds, then there is a transition from steady_state to the state upshifting or downshifting. In these states, if the the speed is still too high or too low, an event UP or DOWN local to the Stateflow diagram is generated and handled by the active substate of gear_state; otherwise, the state changes back to steady_state.

In every cycle in which selection_state is active, the output event CALC_TH is generated to trigger the recalculation of the thresholds by the Simulink model. A change in a substate of gear_state updates the value of gear.

Our example shows how a Simulink and a Stateflow diagram can be combined. The top level of the diagram contains two AND (parallel) states; this is indicated by the use of dashed lines to draw their boxes. These states are not really executed

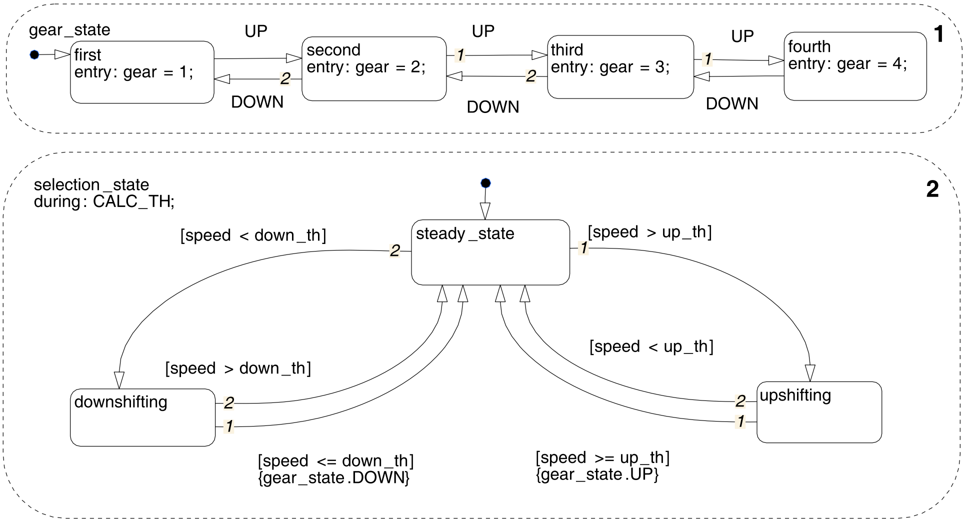


Fig. 2. Stateflow diagram: ShiftLogic block

in parallel, but are active in parallel. Each of them is defined by OR (exclusive) states, which can only be active one at a time. All these substates are basic: they have no further substates. The events UP and DOWN are broadcast internally; they are directed to gear_state and are treated by whichever one of its substates is active.

In the next section, we discuss the model that we propose for this diagram.

4 *Circus* models of Stateflow diagrams

The first concern in defining a model for a diagrammatic notation is giving an account of its abstract syntax in a linear form. We use a slight generalisation of that adopted in [7], which we specify in the sequel using Z.

4.1 Syntax of diagrams

Given sets $SName$, $JName$, Var, and $Event$ contain the valid names of states, junctions, data variables, and events. A full characterisation of a state is a $Path == \mathrm{seq}\, SName$, which lists its superstates in the order determined by the hierarchy of the diagram. In Figure 2, for example, we have states $\langle\, gear\_state, first \,\rangle$ and $\langle\, selection\ \ state, downshifting \,\rangle$. The empty path denotes the stateflow chart.

Just like any state, a chart can have data, events, transitions, and junctions, as well as states, of course. Data variables can be inputs or outputs of the diagram, or local to a state. The same comment applies to events.

$$DataDeclaration == [\, input, output : \mathbb{P}\ Var;\ \ local : Path \pfun \mathbb{P}\ Var \,]$$
$$EventDeclaration == [\, input, output : \mathbb{P}\ Event;\ \ local : Path \pfun \mathbb{P}\ Event \,]$$

A transition is guarded by an event and a condition, but they are both optional.

$$Guard ::= empty \mid e\langle\!\langle Event \rangle\!\rangle$$

A condition is a boolean expression on the data variables. We omit the precise definition of the syntax, and assume that there is a direct correspondence between the elements of a given set *Condition* and *Circus* conditions. If a condition is missing, we define it as *true*. The transitions between the substates of gear_state, for example, are guarded by events, and those in selection_state are guarded by conditions.

As already mentioned, transitions are also associated with actions: condition actions are executed every time the transition is taken, and transition actions are executed when it eventually leads to a new state or a terminal junction. Actions are sequences of simple assignments that change the data variables or event broadcasts. The first transition from downshifting to steady_state, for example, raises the event DOWN directed to gear_state. The action language of Stateflow is rather restricted. We use a given set *Action* to represent the set of all actions, but assume that any of them can be written in a rather direct way in *Circus*.

Finally, the destination of a transition can be a state of a junction.

$$DestComponent ::= p\langle\langle Path\rangle\rangle \mid j\langle\langle JName\rangle\rangle$$

To summarise, the components of a transition are described below.

$$\begin{array}{|l}
\multicolumn{1}{l}{Transition} \\ \hline
event : Guard;\ \ cond : Condition;\ \ condAction, transAction : Action \\
dest : DestComponent \\ \hline
\end{array}$$

States can be defined by an OR or an AND composition of other states, but a state can also be basic, in which case it has no composition.

$$Composition ::= Or\langle\langle \mathrm{seq}\ Transition \times \mathbb{P}\ Path\rangle\rangle \mid And\langle\langle \mathrm{seq}\ Path\rangle\rangle \mid none$$

The sequence of transitions in an *Or* composition are the default transitions that establish its initial active state. Their destination may be a junction, for example, so not necessarily one of the states that are composed. If there are no default transitions, it is because the initialisation of the state is determined by other transitions. In addition, the diagram defines the order in which the transitions should be attempted: either explicitly or based on a graphical convention. The abstract syntax records the right order. Similarly, AND states are executed in a particular order that is defined in the syntax of a composition.

States can also have associated actions: entry actions, which are triggered when the state is entered, during actions, which execute in each cycle when it is active, and exit actions executed when the state is exited.

$$\begin{array}{|l}
\multicolumn{1}{l}{StateDefinition} \\ \hline
entry, during, exit : Action \\
comp : Composition \\
outer, inner : \mathrm{seq}\ Transition \\ \hline
\end{array}$$

In each cycle, the outer transitions of a state, that is, those that lead to a different state, are tried before any of its inner transitions.

A junction belongs to a state, and may have transitions. A history junction records the last active substate of the composite state in which it occurs.

$$\begin{array}{|l}
\multicolumn{1}{l}{JunctionDefinition} \\ \hline
path : Path;\ transitions : \mathrm{seq}\ Transition \\
history : BOOL \\ \hline
\end{array}$$

Finally, a Stateflow diagram or chart contains all of data and event declarations, states, and junctions. In summary, a Stateflow can be characterised as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{Stateflow} \\ \hline
data : DataDeclaration \\
event : EventDeclaration \\
state : Path \mathrel{+\!\!\!\!\!\twoheadrightarrow} StateDefinition \\
junction : JName \mathrel{+\!\!\!\!\!\twoheadrightarrow} JunctionDefinition \\ \hline
\end{array}$$

We leave the formalisation of well-formedness restrictions as future work. For example, the data of a diagram must be local to one of its states, and so on.

4.2 Overall structure of diagram models

In the *Circus* model of a Simulink diagram [4], the input and output signals, and all the internal wires are channels. Each block is defined as a process that reads all the inputs of the block, carries out the necessary calculations, communicates the outputs, and then waits for the end of the cycle before starting again. The end of a cycle is signalled by a channel *end_cycle* on which all block processes synchronise.

The diagram is defined by the parallel composition of the block processes, synchronising on their common channels. All the channels corresponding to internal wires are hidden. The *Circus* process that defines the model of the diagram in Figure 1, which is called auto_trans, is as follows. We show only the processes for Engine, ShifLogic, and Transmission, but all block processes are combined in parallel.

$$
\begin{array}{l}
\mathbf{process}\ auto\ \ trans \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
Engine\ \{\!|\ ImpellerTorque, Throttle, EngineRPM, end\ \ cycle\ |\!\}\\
\qquad \|\\
ShiftLogic\ \{\!|\ VehicleSpeed, up\ \ th, down\ \ th, gear, CALC\ \ TH, end\ \ cycle\ |\!\}\\
\qquad \|\\
Transmission\ \{\!|\ EngineRPM, gear, ImpellerTorque, \ldots, end\ \ cycle\ |\!\}\\
\qquad \| \ldots
\end{array}\right)\\
\qquad \setminus\ \{\!|\ ImpellerTorque, up\ \ th, down\ \ th, gear, CALC\ \ TH, \ldots\ |\!\}
\end{array}
$$

For each of them, we give the set of channels that it uses. The parallelism requires that the processes synchronise on their common channels. Internal channels like *ImpellerTorque*, *up_th*, *down_th*, *gear*, *CALC_TH*, and so on are hidden. In this way, *auto_trans* communicates with its environment using the channels *Throttle*, *BrakeTorque*, *EngineRPM*, and *VehicleSpeed*, which model the inputs and outputs of the diagram, and *end_cycle*. The process *Engine* interacts with *Transmission* on *ImpellerTorque*, *EngineRPM*, and *end_cycle*. Similarly, *ShiftLogic* and *Transmission* interact on *gear* and *end_cycle*.

Our models for Stateflow diagrams are appropriate for use as components of the models of Simulink diagrams that include the Stateflow block. The model of the diagram in Figure 2, for instance, is a process *ShiftLogic* that communicates with its environment on the channels *VehicleSpeed*, *up_th*, *down_th*, *gear*, *CALC_TH*, and *end_cycle*. It is partially presented in Figure 3.

The input and output channels are declared in the Simulink model, but we need to declare internal channels that correspond to the local events: in our example, these are *UP* and *DOWN*. In *Circus*, there are no local channels: they are all declared globally, with locality enforced by hiding.

In the diagrams, the hierarchical state structure is used to define the scope of events and data. The transitions, however, do not respect this organised nesting of scope; they can direct flow of execution between states at any level of the hierarchy. For this reason, in our model, we have an action *Cycle*, which reads and stores all data, and makes it available for another action *Chart*, which models the behaviour of the statechart. In each cycle, the inputs are read, and made available to *Chart* through internal channels. Similarly, internal channels are used to read the outputs from *Chart*; they are communicated just before the end of the cycle. There is an internal channel for each input and output channel; in our example, they are *ispeed*, *iup_th*, *idown_th*, *igear*, and *iCALC_TH*. The type of these channels is $\mathbb{U}$; it is not part of standard Z, but a universal type in the dialect of ProofPower-Z, the theorem prover that we use to mechanise our models and verifications.

In the life-cycle of a diagram, all states are initially inactive; in the first cycle, some states are entered and become active, due to (a combination of) default transitions and AND compositions. Afterwards, in each cycle, the active states execute, that is, process its transitions and any events generated, and then sleep. The sleeping status is signalled using a channel *sleeping*, which is used by the action *Cycle* to determine the point in which the outputs should be communicated, before signalling the end of the cycle using *end cycle*.

As already mentioned, AND states execute in a particular order. Channels *start* and *stop* are used to control that order. If a state with an AND composition can be exited, then the execution of the AND states has to be finished. In this case an extra channel *finished* is used; in our example, the parallel states are always active.

The main action of a Stateflow block process is a parallel composition of the *Chart* and *Cycle* actions, synchronising on the internal channels, which are hidden.

$$\mathbf{process}\ \mathit{ShiftLogic} \mathrel{\widehat{=}} \mathbf{begin}$$

$$\begin{array}{l}
\mathbf{state}\ \mathit{SShiftLogic} == [\,\mathit{vspeed}, \mathit{vup\_th}, \mathit{vdown\_th}, \mathit{vgear}, \mathit{vCALC\_TH} : \mathbb{U}\,] \\
\mathbf{nameset}\ \mathit{vVar} == \{\, \mathit{vspeed}, \mathit{vup\_th}, \mathit{vdown\_th}, \mathit{vgear}, \mathit{vCALC\_TH} \,\} \\
\mathit{Input} \mathrel{\widehat{=}} \mathit{speed}?x \rightarrow \mathit{vspeed} := x \interleave \mathit{up\_th}?x \rightarrow \mathit{vup\_th} := x \interleave \mathit{gear}?x \rightarrow \mathit{vgear} := x \\
\mathit{Output} \mathrel{\widehat{=}} \mathit{gear}!\mathit{vgear} \rightarrow \mathit{Skip} \interleave \mathit{CALC\_TH}!\mathit{vCALC\_TH} \rightarrow \mathit{Skip}
\end{array}$$

$$\mathit{Cycle} \mathrel{\widehat{=}} \left(\begin{array}{l}
\mathit{vCALC\_TH} := 0 \mathbin{;} \mathit{Input} \mathbin{;} \\
\left(\mu\, Y \bullet \left(\begin{array}{l}
\mathit{ispeed}!\mathit{vspeed} \rightarrow Y \Box \mathit{iup\_th}!\mathit{vup\_th} \rightarrow Y \Box \mathit{idown\_th}!\mathit{vdown\_th} \rightarrow Y \\
\Box\ \mathit{igear}?x \rightarrow \mathit{vgear} := x \mathbin{;} Y \Box \mathit{iCALC\_TH} \rightarrow \mathit{vCALC\_TH} := 1 \mathbin{;} Y \\
\Box\ \mathit{sleeping} \rightarrow \mathit{Output} \mathbin{;} \mathit{end\_cycle} \rightarrow \mathit{Cycle}
\end{array} \right) \right)
\end{array} \right)$$

$$\mathit{Chart} \mathrel{\widehat{=}} \mathit{Top\_Enter}$$

$$\mathit{Top\_Enter} \mathrel{\widehat{=}} \left(\begin{array}{l}
\left(\left(\begin{array}{l}
\mathit{Top\_gear\_state\_Enter} \\
\quad \llbracket \{\!|\, \mathit{sleeping}, \mathit{UP}, \mathit{DOWN} \,|\!\} \rrbracket \\
\mathit{Top\_selection\_state\_Enter}
\end{array} \right) \setminus \{\!|\, \mathit{UP}, \mathit{DOWN} \,|\!\} \right) \\
\quad \llbracket \{\!|\, \mathit{sleeping}, \mathit{start}, \mathit{stop} \,|\!\} \rrbracket \\
\mathit{Top}
\end{array} \right) \setminus \{\!|\, \mathit{start}, \mathit{stop} \,|\!\}$$

$$\begin{array}{l}
\mathit{Top\_gear\_state\_Enter} \mathrel{\widehat{=}} \mathit{start}.1 \rightarrow \mathit{Top\_gear\_state\_first\_Enter} \\
\mathit{Top\_gear\_state\_first\_Enter} \mathrel{\widehat{=}} \\
\quad \left(\begin{array}{l}
((\mathit{gear}!1 \rightarrow \mathit{stop}.1 \rightarrow \mathit{sleeping} \rightarrow \mathit{Skip}) \lhd (\mathit{UP} \rightarrow \mathit{Top\_gear\_state\_second\_Enter})) \mathbin{;} \\
(\mathit{end\_cycle} \rightarrow \mathit{Top\_gear\_state\_first})
\end{array} \right) \\
\mathit{Top\_gear\_state\_first} \mathrel{\widehat{=}} \\
\quad \left(\begin{array}{l}
\mathit{start}.1 \rightarrow ((\mathit{stop}.1 \rightarrow \mathit{sleeping} \rightarrow \mathit{Skip}) \lhd (\mathit{UP} \rightarrow \mathit{Top\_gear\_state\_second\_Enter})) \mathbin{;} \\
(\mathit{end\_cycle} \rightarrow \mathit{Top\_gear\_state\_first})
\end{array} \right) \\
\ldots \\
\mathit{Top\_selection\_state\_Enter} \mathrel{\widehat{=}} \mathit{start}.2 \rightarrow \mathit{Top\_selection\_state\_steady\_state\_Enter} \\
\mathit{Top\_selection\_state\_steady\_state\_Enter} \mathrel{\widehat{=}} \\
\quad \mathit{stop}.2 \rightarrow \mathit{sleeping} \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Top\_selection\_state\_steady\_state} \\
\mathit{Top\_selection\_state\_steady\_state} \mathrel{\widehat{=}} \mathit{start}.2 \rightarrow \mathit{Top\_selection\_state} \mathbin{;} \\
\quad \left(\begin{array}{l}
\mathbf{var}\ \mathit{vspeed}, \mathit{vup\_th}, \mathit{vdown\_th} : \mathbb{U} \bullet \\
\quad \left(\begin{array}{l}
(\mathit{ispeed}?x \rightarrow \mathit{vspeed} := x \interleave \mathit{iup\_th}?x \rightarrow \mathit{vup\_th} := x \interleave \ldots) \mathbin{;} \\
\mathbf{if}\ \mathit{vspeed} > \mathit{vup\_th} \rightarrow \mathit{Top\_selection\_state\_upshifting\_Enter} \\
[\!]\ \mathit{vspeed} < \mathit{vdown\_th} \rightarrow \mathit{Top\_selection\_state\_downshifting\_Enter} \\
[\!]\ \mathit{else} \rightarrow \mathit{stop}.2 \rightarrow \mathit{sleeping} \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Top\_selection\_state\_steady\_state} \\
\mathbf{fi}
\end{array} \right)
\end{array} \right) \\
\mathit{Top\_selection\_state} \mathrel{\widehat{=}} \mathit{iCALC\_TH} \rightarrow \mathit{Skip} \\
\mathit{Top\_selection\_state\_upshifting\_Enter} \mathrel{\widehat{=}} \\
\quad \mathit{stop}.2 \rightarrow \mathit{sleeping} \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Top\_selection\_state\_upshifting} \\
\mathit{Top\_selection\_state\_upshifting} \mathrel{\widehat{=}} \mathit{start}.2 \rightarrow \mathit{Top\_selection\_state} \mathbin{;} \\
\quad \left(\begin{array}{l}
\mathbf{var}\ \mathit{vspeed}, \mathit{vup\_th} : \mathbb{U} \bullet \\
\quad \left(\begin{array}{l}
(\mathit{ispeed}?x \rightarrow \mathit{vspeed} := x \interleave \mathit{iup\_th}?x \rightarrow \mathit{vup\_th} := x) \mathbin{;} \\
\mathbf{if}\ \mathit{vspeed} \geq \mathit{vup\_th} \rightarrow \mathit{UP} \rightarrow \mathit{Top\_selection\_state\_steady\_state\_Enter} \\
[\!]\ \mathit{vspeed} < \mathit{vup\_th} \rightarrow \mathit{Top\_selection\_state\_steady\_state\_Enter} \\
\mathbf{fi}
\end{array} \right)
\end{array} \right) \\
\ldots \\
\mathit{Top} \mathrel{\widehat{=}} \mathit{start}.1 \rightarrow \mathit{stop}.1 \rightarrow \mathit{start}.2 \rightarrow \mathit{stop}.2 \rightarrow \mathit{sleeping} \rightarrow \mathit{sleeping} \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Top}
\end{array}$$

$$\bullet\ (\mathit{Chart} \llbracket \{\,\} \mid \mathit{IChannels} \mid \mathit{vVar} \rrbracket \mathit{Cycle}) \setminus \mathit{IChannels}$$

$$\mathbf{end}$$

Fig. 3. *Circus* model of the ShiftLogic block

These channels are grouped in the set *IChannels*, defined as follows for our example.

$$\mathbf{channelset}\ IChannels \mathrel{\widehat{=}} \{\!|\ ispeed, iup\_th, idown\_th, igear, iCALC\_TH, sleeping\ |\!\}$$

The components of the state of a Stateflow process hold the values of the inputs and outputs, and of any local data variables. The action *Cycle* updates them.

In a parallelism of actions, we need to associate with each parallel action the subset of the variables in scope that it can update; these subsets must be disjoint to avoid racing conditions. In our case, *Cycle* has control over all state components; they are grouped in a set called $vVar$. The action *Chart* modifies no variables.

The action *Input* reads all the inputs of the diagram; it is used by *Cycle* at the beginning of each cycle. It is an interleaving that reads each input and records it in the corresponding variable. Strictly speaking, in an interleaving, we also need to define the name set associated with each action; we omit these in our example, as they are obvious. Similarly, the *Output* action communicates the values of the outputs, and is used by *Cycle* after the chart sleeps.

In each cycle, the action *Cycle* initialises the value of the variables that correspond to output events to 0; in our example, *Cycle* initialises $vCALC\_TH$. Event signals can take the value 0 or 1. If a chart does not raise an event in a cycle, then by default its value is 0. For output data we do not have such concern, since the chart should always define its value explicitly.

Each state S of the diagram is modelled by three actions called S_Enter, S, and S_Exit, as needed. The S_Enter action models the behaviour of S when it is entered, S models its behaviour when it is active, and S_Exit when it becomes inactive. The definitions of these state actions depend on whether S has an OR composition, an AND composition, or is a basic state.

For a state with an AND composition, the Circus *Enter* action executes the entry action of the state, and then executes all *Enter* actions of the substates in parallel. In addition, the S action is run in parallel to control their order of execution, using the channels *start* and *stop*. The synchronisation sets are defined by the pattern of communication between states, which can be determined by the raising and accepting states of the local events . This is further discussed in Section 5.

In our example, the top state has an AND composition, so the action Top_Enter executes $Top\_gear\_state\_Enter$ and $Top\_selection\_state\_Enter$ in parallel. Since UP and DOWN are raised in selection state and handled in gear state, they synchronise on these channels, as well as on *sleeping*.

In the case of an OR composition, the *Enter* action executes the entry action, and then tries the default transitions in order; the target of the first transition that can be successfully followed is entered. In our example, both OR compositions have a single default transition with no guard or condition. For example, the action *Top gear state Enter* just executes *Top gear state first Enter*, since the state first is the target of the default transition of gear state.

For basic states, the *Enter* action executes the entry action and sleeps, that is, waits for synchronisation on *sleeping*, since there are no substates to be activated.

Afterwards, the action waits for the end of the cycle and calls the S action in the next cycle. In Figure 3, *selection_steady_state_Enter* is a simple example.

The S action of a state with an AND composition controls the order of execution of the substates. In our example, in each cycle *Top* uses *start*.1 to start the execution of *Top_gear_state*, waits for it to become ready to sleep, which is signalled by *stop*.1, and then does the same for *Top_selection_state* using *start*.2 and *stop*.2. The *Enter* actions of *Top_gear_state* and *Top_selection_state* also use *start* and *stop* to determine when they can start and signal when they are ready to sleep.

In general, there may be during actions, and inner and outer transitions, and all these need to be executed or tried before the parallel states are executed. If S has an OR composition, all that needs to be done is handling these components.

In our example, the state gear_state does not have any of these components, so that *Top_gear_state* is simply *Skip*, and we omit it. On the other hand, selection_state has a during action, which raises CALC_TH. This is modelled in *Top_selection_state* by synchronisation on the internal channel *iCALC_TH*.

Similarly, an action of a basic state handles the during action, the transitions, and then sleeps if no outer transition is taken. An example is the action *Top_selection_state_upshifting*. It calls the action for its superstate, then it reads the variables that it needs to make decisions on the transitions, and then checks which transition conditions are valid. In this case, the conditions are mutually exclusive and the conditional is quite simple. In the general case, we need to guarantee the order of testing; this is further discussed in Section 5.

Since parallel states execute in order, their substates have to wait for a synchronisation on *start* before they can execute. As already mentioned, the *Enter* actions of the parallel states wait to synchronise on *start*. The *Enter* action of a basic substate signals that it is ready to *sleep* using *stop*; the action *Top_selection_state_upshifting_Enter* is an example. Since there is no entry action in this case, it immediately signals *stop*. The action *Top_selection_state_upshifting*, as a basic substate of a parallel composition, also uses *start* and *stop* to control its points of execution. In fact, in our example, since the chart is composed of parallel states, every basic state uses the *start* and *stop* channels.

The action *Top_selection_state_upshifting* raises the event *UP*. The actions that model the substates of gear_state need to handle it; this is achieved using the interrupt operator. For example, *Top_gear_state_first_Enter* first executes the entry action of first; it is an assignment of 1 to gear, which we model by outputting the value 1 through *igear*. The *Cycle* action takes this value and stores it in the variable that holds the value of gear. Afterwards, *Top gear state first Enter* signals that it is ready to sleep, and then sleeps. In this period, however, at any moment it may be interrupted by the event *UP*, in which case the state second is entered. After synchronisation on *sleeping*, however, the state is not active anymore, and so an interruption by *UP* is no longer possible. Similarly, *Top gear state first* once it starts, it accepts interruptions up to when it sleeps.

This example shows how we handle AND compositions, OR compositions, basic states, and event broadcasting. In Section 5, we consider a general strategy for building models; it formalises the semantics of Stateflow diagrams.

5 Translating Stateflow to *Circus*

Our semantics of a Stateflow diagram is a function $[\![\, st \,]\!] n$ that takes a diagram st, that is, an element of the type *Stateflow* defined in Section 4, and the name n of the Simulink block defined by this diagram, and produces a *Circus* specification.

As discussed and exemplified in Section 4, the *Circus* model starts with a declaration of local events and internal channels.

$$\begin{array}{l}
\mathbf{channel}\ \ \mathrm{ran}\ st.\mathsf{event.local};\ \ \mathsf{iReadWrite}(\bigcup \mathrm{ran}\ st.\mathsf{data.local}) : \mathbb{U} \\
\mathbf{channel}\ \ \mathsf{i}(st.\mathsf{data.input}), \mathsf{i}(st.\mathsf{data.output}), \mathsf{i}(st.\mathsf{event.input}), \mathsf{i}(st.\mathsf{event.output}) : \mathbb{U} \\
\mathbf{channel}\ \ sleeping;\ \ start, stop, finish : \mathbb{Z}
\end{array}$$

The local events are synchronisation channels, so they do not have a type; we use the set $\mathrm{ran}\ st.\mathsf{event.local}$ of local event names to denote a list of these names. Similarly, we consider the set $\bigcup \mathrm{ran}\ st.\mathsf{data.local}$ of names of local variables, and use it where a list of the names is expected. The syntactic function $\mathsf{iReadWrite}$ prefixes all the names with an *iread* to form the names of the internal channels used by the action *Chart* to read the values of the local variables, and with an *iwrite* to name channels used to update their values. For the input and output data and events we need only one channel, because they can only be either read or written; the i-prefixed names are created by the function i. We also need to declare the extra control channels. The channels *start*, *stop*, and *finish* are not always necessary, and may be eliminated if there are no AND compositions. In particular, *finish* is only needed if there are outer transitions from a parallel state.

The set of internal channels is given a name.

$$\begin{array}{l}
\mathbf{channelset}\ \ IChannels \mathrel{\widehat=} \\
\qquad\qquad\qquad\qquad \{\!|\ \mathsf{i}(st.\mathsf{data.input}), \mathsf{i}(st.\mathsf{data.output}), \mathsf{i}(st.\mathsf{event.input}), \\
\qquad\qquad\qquad\qquad\ \ \mathsf{i}(st.\mathsf{event.output}), \mathsf{iReadWrite}(\bigcup \mathrm{ran}\ st.\mathsf{data.local}), sleeping\ |\!\}
\end{array}$$

The name of the block is used to declare the process that models the diagram.

$$\mathbf{process}\ \ n \mathrel{\widehat=} \mathbf{begin}$$

The state components record the values of the input and output data and events, and of all local data; the syntactic function v prefixes all their names with a v to form the names of the corresponding state components.

$$\begin{array}{|l}
\mathbf{state}\ S \\
\hline
\mathsf{v}(st.\mathsf{data.input}), \mathsf{v}(st.\mathsf{data.output}), \mathsf{v}(st.\mathsf{event.input}), \mathsf{v}(st.\mathsf{event.output}) : \mathbb{U} \\
\mathsf{v}(\bigcup \mathrm{ran}\ st.\mathsf{data.local}) : \mathbb{U} \\
\hline
\end{array}$$

The chart is characterised by the *Enter* action of the fictional *Top* state that corresponds to the topmost level of the hierarchy of states.

$$Chart \mathrel{\widehat{=}} Top\_Enter$$

The function $[\![st]\!]^{SJ}$ defines the actions that model the states and junctions of st, and follow the definition of *Chart*. It is discussed later on in this section.

To define *Cycle*, we introduce four sets of pairs of names. The first, inputs, contains the pairs formed by a name of an input data or event, associated with the corresponding v-prefixed name; similarly, $\mathsf{outputs}$ contains the pairs corresponding to names of outputs. Finally, $\mathsf{iinputs}$ and $\mathsf{ioutputs}$ contains pairs of matching i-prefixed and v-prefixed names; *iinputs* contains the names coming from inputs and local variables, and *ioutputs* those from outputs and local variables. In *iinputs* the local variables are prefixed with *iread*, and in *ioutputs*, with *iwrite*. Using these sets, we define the actions below used in the definition of *Cycle*

$$\begin{array}{l}
Input \mathrel{\widehat{=}} |||(\mathsf{inp}, \mathsf{vinp}) : \mathsf{inputs} \bullet \mathsf{inp}?x \rightarrow \mathsf{vinp} := x \\
Output \mathrel{\widehat{=}} |||(\mathsf{out}, \mathsf{vout}) : \mathsf{outputs} \bullet \mathsf{out}!\mathsf{vout} \rightarrow Skip \\
IChartOut \mathrel{\widehat{=}} \Box(\mathsf{iinp}, \mathsf{vinp}) : \mathsf{iinputs} \bullet \mathsf{iinp}!\mathsf{vinp} \rightarrow Skip \\
IChartInp \mathrel{\widehat{=}} \Box(\mathsf{iout}, \mathsf{vout}) : \mathsf{ioutputs} \bullet \mathsf{iout}?x \rightarrow \mathsf{vout} := x
\end{array}$$

The first of these actions reads the inputs of the chart and stores them in the corresponding state component; the second outputs the value of the state components. The actions *IChartOut* and *IChartInp* define the interface between *Cycle* and *Chart*. The *Cycle* is always willing to output the value of any of the input data, or read the value of any of the outputs; local variables can be both read and written by the chart. The name sets associated with the interleaved actions are the singletons that contain the assigned variable, if any; they are omitted above.

$$Cycle \mathrel{\widehat{=}} \left(\begin{array}{l}
Input \mathbin{;} \mathsf{v}st.\mathsf{event.output} := 0 \mathbin{;} \\
\left(\mu\, Y \bullet \left(\begin{array}{l}
IChartOut \mathbin{;} Y \\
\Box\ IChartInp \mathbin{;} Y \\
\Box\ sleeping \rightarrow Output \mathbin{;} end\_cycle \rightarrow Cycle
\end{array}\right)\right)
\end{array}\right)$$

In our example in Figure 3, we did not introduce *IChartOut* and *IChartInp*, but included them directly in *Cycle*. The main action is the same in all models.

$\bullet\ (Chart \mathbin{[\![} \{\,\} \mid IChannels \mid \alpha S \mathbin{]\!]} Cycle) \setminus IChannels$
end

The set αS contains all the names of the components of the state S of the process; the action *Cycle* has control to update all of them.

The result of $[\![st]\!]^{SJ}$ is a sequence of paragraphs; for each state or junction p in the domain of $st.\mathsf{state}$ or $st.\mathsf{junction}$, we have a few actions specified by the function

$[\![\, p \,]\!]^{P}_{st}$. Its definition is by cases, considering whether p is a state or a junction, and if it is a state, whether it has an OR composition, an AND composition, or is a basic state, and whether it is a substate of an OR or an AND composition.

We sketch here the definition of $[\![\, p \,]\!]^{P}_{st}$ for a state p that has an OR composition, that is, $p \in \operatorname{dom} \mathit{st}.\mathit{state}$ and $\mathit{st}.\mathit{state}(p).\mathit{composition} \in \operatorname{ran} \mathit{Or}$, and that is not a substate of any parallel composition, precisely characterised as follows.

$$\begin{array}{l} \forall\, s_1 : \operatorname{dom} \mathit{st}.\mathit{state} \mid \mathit{st}.\mathit{state}(s_1).\mathit{composition} \in \operatorname{ran} \mathit{And} \bullet \\ \quad \forall\, s_2 : \operatorname{ran} \mathit{And}^{\sim}\, \mathit{st}.\mathit{state}(s_1).\mathit{composition} \bullet \neg\, s_2 \ \mathsf{prefix}\ p \end{array}$$

For a path p, the syntactic function $\mathcal{N}_p$ defines the action name corresponding to p. As already explained, the fictitious top state is called Top.

$$\mathcal{N}_{\langle\rangle} = \mathit{Top} \qquad\qquad \mathcal{N}_{p ^\frown \langle n \rangle} = \mathcal{N}_{p\_n}$$

The function $[\![\, p \,]\!]^{P}_{st}$ introduces three actions: $\mathcal{N}_{p\_}\mathit{Enter}$, $\mathcal{N}_p$, and $\mathcal{N}_{p\_}\mathit{Exit}$.

The *Enter* action first of all reads all the inputs and the data in its scope, that is, the variables that are local to p or to any of its superstates.

$$\mathit{local}(p) = \bigcup \{\, s : \operatorname{dom} \mathit{st}.\mathit{data}.\mathit{local} \mid s\ \mathsf{prefix}\ p \bullet \mathit{st}.\mathit{data}.\mathit{local}(s) \,\}$$

Local v-prefixed variables in $\mathcal{N}_{p\_}\mathit{Enter}$ hold the value of the inputs and of these variables. The set $\mathsf{iinputsL}$ used below pairs the i-prefixed names corresponding to inputs and the iread-prefixed names corresponding the local variables with the corresponding v-prefixed names.

$$\mathcal{N}_{p\_}\mathit{Enter} \mathrel{\widehat{=}} \left(\begin{array}{l} \mathbf{var}\ \mathsf{v}(\mathit{st}.\mathsf{data.input}), \mathsf{v}(\mathit{st}.\mathsf{event.input}), \mathsf{v}(\mathsf{local}(p)) \bullet \\ \quad \left(\begin{array}{l} \interleave (\mathsf{iinp}, \mathsf{vinp}) : \mathsf{iinputsL} \bullet \mathsf{iinp?x} \longrightarrow \mathsf{vinp} := \mathsf{x}) \,; \\ [\![\, \mathit{st}.\mathit{state}(p).\mathit{entry} \,]\!]^{A}_{st} \,; \\ [\![\, (\mathit{Or}^{\sim}(\mathit{st}.\mathit{state}(p).\mathit{composition})).1 \,]\!]^{DTS} \end{array} \right) \end{array} \right)$$

After reading the necessary data, $\mathcal{N}_{p\_}\mathit{Enter}$ executes the entry action of the state p. Translation of actions is rather simple, and we omit the definition of the semantic function $[\![\, a \,]\!]^{A}_{st}$ which takes a Stateflow action a and produces a corresponding (unnamed) *Circus* action. In the end $\mathcal{N}_p$ *Enter* executes the default transitions. These are translated as defined by the semantic function $[\![\, \mathit{ts} \,]\!]^{DTS}$. We omit its definition, but discuss the translation of transitions below.

The action $\mathcal{N}_p$ is similar to $\mathcal{N}_p$ *Enter*, but instead of executing the entry action, it executes the during action $\mathit{st}.\mathit{state}(s).\mathit{during}$, and instead of executing the default transitions, it executes the outer and inner transitions. In fact, the transitions can only be handled here if the substates do not have exit actions. Otherwise, each substate has to handle the transition of this superstate. The function $O[\![\, \mathit{st}.\mathit{state}(s).\mathit{outer} \,]\!]^{TS}$ translates the outer transitions; it takes as an extra parameter the action to be taken if none of the transitions are available: we give the argument $I[\![\, st.state(s).inner \,]\!]^{TS} Skip$, which executes the inner transitions, and skips if none of these can be executed either. We present the definition of $O[\![\, ts \,]\!]^{TS}$ to illustrate our formalisation of the treatment of transitions.

$$O[\![\, \langle\,\rangle \,]\!]^{TS}_{p} A = A \qquad O[\![\, \langle t \rangle \frown ts \,]\!]^{TS}_{p} = \textbf{if } O[\![\, \langle t \rangle \frown ts \,]\!]^{T}_{p} \textbf{ fi}$$

If there are no transitions, then the argument action A is the result; otherwise we have a conditional whose guarded commands are given by $O[\![\, \langle t \rangle \frown ts \,]\!]^{T}$ defined as follows. If there are no transitions left, the result is A, which should be a guarded command. In our case, it is the result of the translation of the inner transitions. For a list $\langle t \rangle \frown ts$, we need to consider whether the target $t.dest$ of t is a state or a junction; we present below the translation when the target is a state. For transitions targeted at junctions, we need to consider the possibility of backtracking.

$$O[\![\, \langle t \rangle \frown ts \,]\!]^{T}_{p} \mathrel{\widehat{=}} \left(\begin{array}{l} \mathsf{v(t.event)} = 1 \wedge [\![\, \mathsf{t.cond} \,]\!]^{C} \rightarrow \\ \qquad \left(\begin{array}{l} [\![\, \mathsf{t.condAction} \,]\!]^{A}_{st} \,;\, [\![\, \mathsf{t.transAction} \,]\!]^{A}_{st} \,; \\ (\,;\, ss : rev\ nesting(p \vee t.dest, p) \bullet \mathcal{N}_{ss}\_Exit)\,; \\ (\,;\, ss : nesting(p \vee t.dest, t.dest) \bullet \mathcal{N}_{ss}\_Enter) \end{array} \right) \\ [\!]\ \ O[\![\, ts \,]\!]^{T}_{p} \end{array} \right)$$

The translation of conditions is direct, so we omit the definition of $[\![\, c \,]\!]^{C}$. In the absence of backtracking, both the condition and transition actions are executed. Finally, if a transition is taken, we need to exit the current state and enter the target state $t.dest$. For that, we need to exit all superstates of p that do not include $t.dest$ as a substate; this is the state $p \vee t.dest$, where $\vee$ is the least upper bound operator for the prefix relation. For paths p_1 and p_2 such that p_1 is a prefix (superstate) of p_2, $nesting(p_1, p_2)$ gives the chain of states from p_1 to p_2, including p_1 and p_2. We reverse the list $nesting(p \vee t.dest, p)$ to determine the sequence of states from p to $p \vee t.dest$ that need to be exited, and use $nesting(p \vee t.dest, t.dest)$ to determine the list of states from $p \vee t.dest$ to p that need to be entered.

The $\mathcal{N}_{p}\_Exit$ action only executes $st.state(p).exit$. Since these actions are called by the action for p or for one of its substates, there is no need to read any data.

The complete formalisation of the translation rules is quite extensive. The definitions that we have presented illustrate the approach. The most interesting feature that is omitted is the treatment of local events. We need en environment that defines the states that contain the sources and targets of each event. For the diagram in Figure 3, we have a local event UP, for example. The source of UP is upshifting, and the targets are first, second, and third. This information is used to define the structure and synchronisation sets for the parallelisms that model the AND compositions. For each event, we need to find the parallel superstates that contain all its sources and targets. In the case of UP, they are selection_state and gear_state; this explains the synchronisation set in Figure 3.

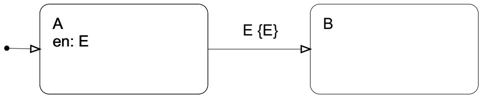
6 Additional features

Many Stateflow features can lead to intricate designs, with negative impact on understandability and analysis. The work in [17], for example, identifies a subset of Stateflow that is regarded as safe; in our approach, as in theirs, we aim at giving semantics and reasoning about arbitrary diagrams.

A first concern of many works is the reliance on positioning of parallel states and transitions to determine the order in which they are executed. In our work, this is handled by the parser of Stateflow diagrams; our abstract notation is explicit about the order. So, this issue is handled in the semantics in a straightforward way, as already explained and exemplified in the previous sections.

Nontermination

Another issue is related to the possibility of the treatment of a local event raising that same event again, which leads to nonterminating behaviour. The following diagram is considered in [17]. It is an OR composition, whose default transition enters a state A. The entry action of A raises the event E, and it is treated by A.



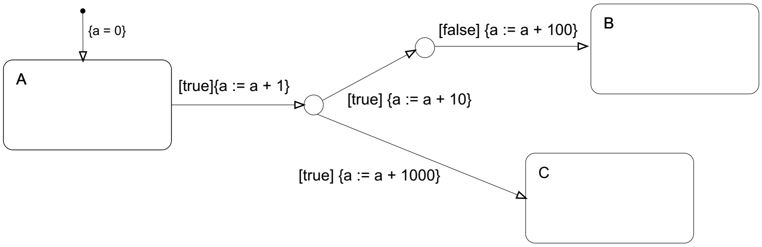
In the environment that records the source and target of E, we can immediately identify that E is both raised and received by A. In this case, the use of the interrupt operator to treat E is not appropriate. Instead we conflate the generation and treatment of E into a single event. Since the channel E is hidden in the model of the diagram, the event happens as soon as it becomes available.

In the example diagram, if the E event were not generated by action condition action, the model of *Top_A_Enter* would be just $E \rightarrow Top\_B\_Enter$, where E is raised and treated, with the corresponding outer transition to B taken.

In the above diagram, however, during the outer transition, E is once again raised. This leads to recursion since the raised event is treated by the execution of the transitions of A itself. So, *Top A Enter* is defined as $\mu X \bullet E \rightarrow X$. This is indeed an infinite loop, but it is also the accurate semantics of the diagram.

Backtracking

Transitions to a new state are final, but transitions to a junction may be backtracked if they do not eventually lead to a final state or to a terminal junction, because they lead to a junction from which no transition can be taken. The following diagram, which is presented in [17], gives an example.



The default transition sets a to 0, and the outer transition from A sets it to 1 and leads to a junction, represented by a circle. Afterwards the upper transition from the junction is tried and increments the value of a by 10, but then the next transition available fails, since its condition is *false*. Consequently, the execution backtracks and the lower transition from the first junction is tried. It succeeds, but the assignments to a are not lost. The final value of a is 1011.

To model backtracking, transitions need to be executed in parallel, with channels *ok* and *backtrack* used to signal whether a transition succeeded, and the others should be abandoned, or whether it failed and the second should take over. For each pair of transitions, a fresh pair of *ok* and *backtrack* channels is needed. Since the data is managed separately by *Cycle*, all transitions have access to it.

For the diagram above, the conditions are the constants *true* and *false*, so using laws of *Circus*, we can simplify the resulting action to another that takes the viable route directly, and updates the value of a to 1011.

History junctions

If a history junction is included in a state with an OR composition, every time that state is re-entered, the last active substate becomes active again.

To model such behaviour, a *History* action is run in parallel with the immediate superstate of the composite state. It keeps track of the active substates, and is used by the *Enter* action to determine the substate to be activated. In addition, the *History* action uses the default transitions to determine the state that becomes active the first time the composite state is entered.

If a history junction is the target of an inner transition, if it is followed, the current substate is exited and re-entered. So, the *Enter* action is not affected in this case. It is possible to have a *History* action that determines the next state to be entered when the inner transaction is taken, or a model that executes the exit and enter actions directly. The first approach is more general.

We leave the formalisation of the translation strategy that considers all these special features as future work.

7 Conclusions

We have proposed a semantics for Stateflow diagrams that is appropriate as a basis for reasoning techniques based on refinement. We have discussed how to construct models, and how the semantics can be formalised algebraically. Our models can be used as components of an existing *Circus* semantics for Simulink.

The work presented in [2] models Stateflow diagrams as finite state machines described in SMV. The translation from the Stateflow notation to SMV is automated, but not formalised. The subset of Stateflow considered does not allow nested event generation and does not include non-Boolean input signals, junctions with more than one input transition, transition actions, or output events.

An automaton model is used in [18]; this work suggests a reasoning approach for hybrid systems based on traditional model checking techniques. The encoding of Stateflow diagrams as communicating pushdown automata is informally described, as is its further translation to SAL. The subset of the Stateflow notation covered does not allow inner transitions or junctions, although the work can be extended.

Like in our approach, the CSP and the *Circus* semantics of UML (or UML-RT) state diagrams in [13,15] are also defined by algebraic rules that specify a function that maps a diagram to a CSP or to a *Circus* model. In addition, the reasoning approaches advocated are based on refinement, and, in particular, the work on UML-RT considers state diagrams as part of a richer design notation.

The diagrams considered in these works, however, are very different from Stateflow diagrams. They exhibit some nondeterminism, but there is no notion of cycle, no backtrack, and no event broadcast; in the CSP work, data is also not covered.

Our model does present some limitations; we still have to study the impact of transitions to and from substates of a parallel composition that cross the parallel state, for example. In addition, further validation of the models is also necessary, since so far we have considered only small examples, and still have to complete the definition of the semantic functions. Our next step is the automation of the model construction, so that large case studies can be conducted. Our long-term goal is to extend the refinement-based verification technique for implementations of Simulink diagrams to cover implementations of Stateflow blocks.

References

[1] M. M. Adams and P. B. Clayton. Cost-Effective Formal Verification for Control Systems. In K. Lau and R. Banach, editors, *ICFEM 2005: Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 465 – 479. Springer-Verlag, 2005.

[2] C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic Verification of Executable Control Specifications. In *IEEE International Symposium on Computer Aided Control System Design*, pages 581 – 586. IEEE Press, 1999.

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

[4] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in *Circus*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 253 – 268. Springer-Verlag, 2005.

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

[6] C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*. Springer-Verlag, 1998.

[7] G. Hamon. A Denotational Semantics of Stateflow. In *International Conference on Embedded Software*. ACM Press, 2005.

[8] G. Hamon and J. Rushby. An operational semantics for Stateflow. *International Journal on Sofware, Tools, and Technology Transfer*, pages 447 – 456, 2007.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 19:87 – 152, 1992.

[10] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293 – 333, 1996.

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

[12] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[13] M. Y. Ng and M. Butler. Towards Formalizing UML State Diagrams in CSP. In *International Conference on Software Engineering and Formal Methods*, pages 138 – 148. IEEE Computer Society, 2003.

[14] M. V. M. Oliveira and A. L. C. Cavalcanti. ArcAngelC: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, **214C**:203 – 229, 2008.

[15] R. Ramos, A. C. A. Sampaio, and A. C. Mota. A Semantics for UML-RT Active Classea via Mapping into *Circus*. In *Formal Methods for Open Object-based Distributed Systems*, volume 3535 of *Lecture Notes in Computer Science*, pages 99 – 114, 2005.

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

[17] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and Translating a “Safe” Subset of Simulink/Stafeflow into Lustre. In *International Conference on Embedded Software*, pages 259 – 268. ACM Press, 2004.

[18] A. Tiwari. Formal Semantics and Analysis Methods for Simulink Stateflow Models. Technical report, SRI International, 2002. `http://www.csl.sri.com/∼tiwari/stateflow.html`.

[19] H. Treharne and S. Schneider. Using a process algebra to control B OPERATIONS. In *1st International Conference on Integrated Formal Methods – IFM’99*, pages 437 – 457. Springer-Verlag, 1999.

[20] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.