Refinement-based verification of implementations 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's Stateflow is a graphical notation widely adopted in industry. Since it is frequently used to model safety-critical systems, correctness of implementations of Stateflow charts is a major concern. In previous work, we have shown how we can generate formal models for refinement of Stateflow charts automatically. Here, we define a refinement strategy that supports the automated verification of implementations with respect to these models. We consider the verification of implementations that follow architectural patterns used in the Stateflow code generator. We present a detailed procedure for application of refinement laws. If the implementation is correct, the procedure succeeds. If a law application fails, the implementation is either incorrect or does not use the expected architectural pattern. The very low proof burden associated with the refinement verification makes a high level of automation possible.

1 Introduction

Simulink [Matc] is a graphical notation widely used in industry for specification of control systems. A Simulink diagram contains blocks that encapsulate some functionality, and wires connecting their inputs and outputs. Stateflow [Matd] is part of Simulink, and supports the specification of state transition systems via a particular kind of block, namely, a Stateflow block. These are defined by Stateflow charts, a variant of Statecharts [Har87] that extends standard state transition systems by introducing new features, such as hierarchy and backtracking. While Simulink diagrams are typically used to specify control laws, Stateflow charts usually model reactive behaviour in response to events.

Simulink and Stateflow are extensively used in the development of safety-critical systems. Additionally, emerging certification standards and guidelines, like DO178B, ISO DIS 26262, and EN50128, now recommend the use of formal methods for the specification, design, development, and verification of software. Formal treatments of Simulink and Stateflow are, therefore, both useful and timely. We are concerned here with a formal approach to the assessment of the correctness of implementations of Stateflow charts.

A frequent treatment of this problem is based on the verification of automatic code generators [CCM$^+$03, TNP$^+$08, LST09]. We propose an orthogonal path based on the verification of implementations with respect to a model of the chart. An overview of our approach is given in Figure 1.

It consists of deriving formal models of the Stateflow chart and its implementation automatically, and applying a refinement calculus to check the correctness of the model of the implementation with respect to

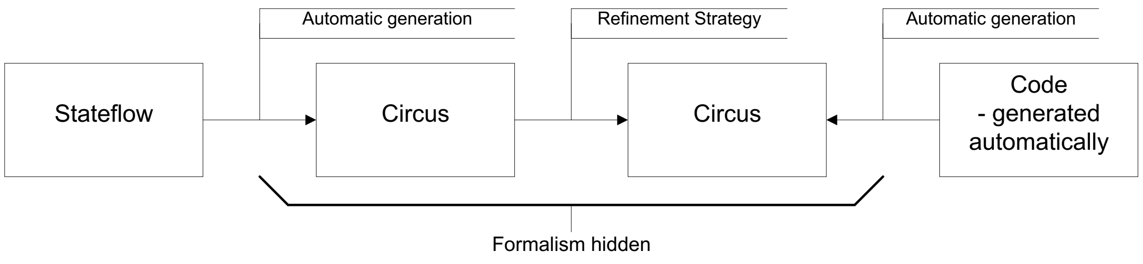


Figure 1: An approach for the verification of implementations of Stateflow charts.

the model of the chart. This is particularly well suited for situations where automatically generated code is not entirely applicable or convenient, because for instance, hardware and performance requirements call for changes in the generated code. Moreover, it addresses the problem raised by Simulink and Stateflow frequent updates, which can have a heavy impact on the cost of the verification of any code generator.

In previous work, we have already addressed the automatic generation of models of charts [MC12] indicated in Figure 1. We have formalised and mechanised the semantics of Stateflow via a function that maps charts to models described in the state-rich process algebra *Circus* [CSW03, Oli06]. This semantic function is described by translation rules; they have been extensively validated, encoded in Z [WD96] using a Z tool, and implemented in a tool that generates *Circus* models of Stateflow diagrams.

In this paper, our main contribution is to show how these *Circus* models can be used to prove the correctness of implementations of the charts as indicated in Figure 1. The verification technique presented here is a refinement strategy that uses the *Circus* laws to prove that the chart model is refined by the implementation model. Soundness of the technique stems from that of the laws established using the *Circus* semantics. In [MC11], we have presented an overview of an initial version of our refinement tactic. Here, we provide a detailed algorithmic description of the tactic.

The notion of refinement is that of *Circus*, which is similar to that of failures-divergences refinement in CSP [Ros11]. First of all, it ensures that safety properties are preserved, in the sense that all traces of interactions of the implementations are allowed by the specification. In addition, it ensures that the implementations do not introduce deadlocks or divergences (livelocks) not allowed by the specifications.

The implementations that we consider are those that employ a specific architectural pattern. This allows us to specialise our refinement strategy to increase automation. The architectural pattern is simple, but general. The structure is that adopted by the MATLAB code generator. There are other code generators [SSC$^+$04, TNP$^+$08, Tar], but they either are not verified or cover a limited subset of the Stateflow notation.

The existing *Circus* refinement calculus [CSW03] is enough for the purpose of deriving correct implementations from the models in [MC12]. The expertise required for that, however, is often not available. Moreover, the complexity of Stateflow and the size of real charts potentially renders the manual application of the refinement calculus infeasible. Our approach relies on the structure of the chart models and on the architecture of the implementations to provide guidance and automation where possible.

The elaborate structure of our refinement tactic reflects the particularities of the structure of our models. Our verification technique trades the simplicity and generality of the *Circus* refinement calculus for high levels of automation. It hides the use of formalism and, therefore, facilitates use by engineers. Our technique is closely related to that proposed in [CCO11] for verification of implementations of Simulink diagrams. Our work extends those results to cover a larger class of diagrams and implementations.

Section 2 introduces the background material necessary for the presentation of our strategy: the Stateflow and the *Circus* notations, and our *Circus*-based modelling approach for Stateflow charts. Section 3 discusses the architectural pattern of the implementations of Stateflow charts that we consider, and Section 4 provides general guidelines for deriving *Circus* models of these implementations. Section 5 describes our refinement strategy for the verification of implementations of Stateflow charts. Section 6 assesses our contributions, examines related work, and discusses directions for future developments.

2 Background material

In this section, we introduce the Stateflow and *Circus* notations, and our formal models of Stateflow charts.

2.1 Stateflow charts

The main components of a chart are states, junctions, transitions, data and events. Interaction with the Simulink diagram is via input and output events and data variables.

Our running example is shown in Figure 2; it is a chart (supplied with MATLAB Stateflow) that models a temperature controller for a ventilation system. It has one input variable `temp`, one output variable `airflow`, and is triggered by two events: `SWITCH` and `CLOCK`. The chart does not show `CLOCK`; it is part of the Simulink diagram that includes the Stateflow block defined by this chart.

The execution model of a Stateflow chart is cyclic (like in Simulink) and controlled by events. In every step, the chart is executed once for each of the events that occurred in an order defined by the chart. Events are signalled by and to the Simulink diagram. Events that are not shown in the Stateflow chart, like the `CLOCK` event in our example, have the same status as the events explicitly shown. In our example, this is `SWITCH`. If an event like `CLOCK` occurs, transitions that are not guarded by an event may be executed.

In the first cycle, execution of the chart entails activation by following default transitions, which are represented by arrows without a source. Subsequent executions of a chart execute its active states. In what follows, we explain what the execution of a state entails.

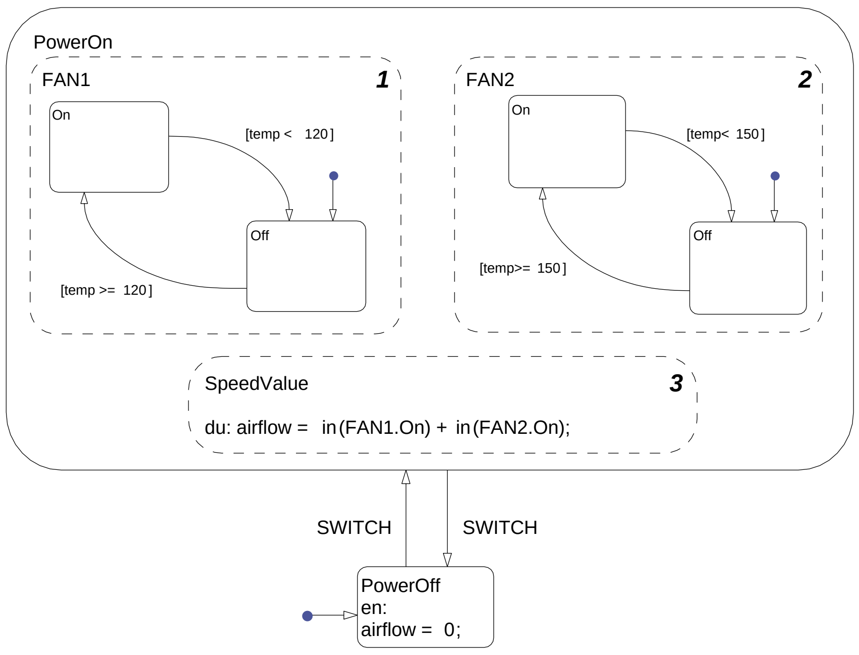


Figure 2: Air controller chart: supplied with Stateflow.

In a chart, states are represented by rectangles with round corners. States (and charts) can have substates grouped in a sequential or parallel decomposition. A state with a sequential decomposition has at most one substate active, and a state with a parallel decomposition has all of its substates active or inactive. There are nine states in our example: `PowerOn` has a parallel decomposition, `FAN1` and `FAN2` have sequential decompositions, and the remaining states (that is, `PowerOff`, `SpeedValue`, and two `On` and two `Off` states) have no substates. The chart itself has a sequential decomposition with two states `PowerOn` and `PowerOff`.

Incidentally, we observe that the graphical representation of a chart does not present all the information that is available about it. For instance, all elements of a chart (including states, transitions, and so on) have identifiers, which are not all shown. In addition, transitions from a single source are ordered; this is implicitly shown in a diagram by organising them clockwise. All this information is included in a textual description of a chart based on a well defined metamodel [Matd]. It is this description that we take as input for our verification technique (and to which we give semantics in [MC12]).

A state has actions associated with it. Entry and exit actions are executed when the state is entered and exited, respectively. During and on actions are executed when the state is executed, but on actions also require the occurrence of a particular event. In our example, the state `SpeedValue` has a during action identified by the label `du`, and the state `PowerOff` has an entry action, identified by the label `en`. Both actions are assignments. The expression `in(FAN1.On)` in the during action evaluates to 1 if the substate `On` of `FAN1` is active, and to 0 otherwise. Similar comments apply to `in(FAN2.On)`.

States can be connected by one or more transitions represented by arrows. These can have a condition, a trigger, and two types of actions, namely condition and transition actions. A condition is an expression over the variables of the chart, and a trigger is an expression over the events of the chart. In our example, four of the transitions have a condition, `temp<120`, `temp>=120`, `temp<150`, or `temp>=150`, and two are guarded by `SWITCH`. The condition and the trigger define a guard for the transition: the guard is true if the condition holds, and one of the events in the trigger happens. In this case, the condition actions are executed.

Transitions may be in a path: a series of transitions connected by junctions, which are represented by circles. So taking a transition may not lead to exiting a state. Junctions are decision points, so that a transition path is conditional. It is completed when a state is reached, or all the transitions are attempted (possibly via backtracking) without reaching a state. When a state is reached, the source state of the path is exited, the transition actions of the path are executed, and the target state is entered. A transition does not lead to exiting a state if it is part of a path that does not successfully reach another state.

Additionally, there is a special type of junction, called history junction, which records the most recently activated substate of the state that contains it. There are no junctions or history junctions in our example.

Transitions are classified according to the presence of a source and the relative position between its source and target. Default transitions have no source and indicate the default path to be taken when a state is entered or the chart is activated. Transitions that connect a state to one of its substates are inner transitions,

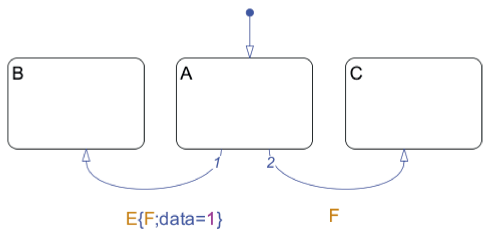


Figure 3: Local event broadcast and early return logic example.

and all others are called outer transitions. In our example, there are three default transitions (with `PowerOff` and the `Off` states as targets); the rest are outer transitions. The first time the chart is executed, by following its default transition, `PowerOff` is entered, and the assignment `airflow = 0` is executed.

Entering a state makes it active, and leads to the execution of its entry actions and to entering the substates: either by following a default transition, in the case of a state with a sequential decomposition, or by entering all substates in an order defined in the diagram, in the case of a state with a parallel decomposition. As already said, the execution of a state is related to the cyclic behaviour of charts. In every cycle, the active states are executed, and this entails trying the outer transitions, and stopping if the states are exited as a result. If not, the during and on actions are executed, and the inner transitions are attempted. If no transition takes place, the active substate(s) are then executed, in order for parallel states.

In our example, when `PowerOn` is entered, all its substates are entered in the order determined by the numbers on their top right corner. First, `FAN1` is entered, and its substate `Off` is entered as a result of the default transition. The same occurs for `FAN2`, and, finally, `SpeedValue` is entered. After `PowerOn` is entered, it can be either executed (if `CLOCK` occurred), or exited (if `SWITCH` occurred). When `PowerOn` is executed, all these substates are executed in order. When `FAN1` and `FAN2` are executed, a transition between their substates can take place according to the value of `temp`. When `SpeedValue` is executed, the sum of the statuses (1, if active, and 0, if inactive) of the substates `On` of `FAN1` and `FAN2` is assigned to `airflow`. If `PowerOn` is exited, `PowerOff` is entered, all the active substates of `PowerOn` are exited, and `airflow` is set to zero.

A Stateflow action can raise a local event broadcast. This is illustrated by the chart in Figure 3, which contains states `A`, `B`, and `C`, and only one input event `E`. A local event broadcast leads to the reexecution of the chart or of an identified target state in the current step of the diagram. In our example, there are two transitions from `A`. After the first step of execution, the state `A` is active. In the next step, if the chart is executed for the event `E`, in the execution of the active state `A` the transition from `A` to `B` is attempted. This is possible since the trigger of that transition is `E`, and leads to the broadcast of the local event `F`, which prompts the reexecution of the whole chart. At this point, `A` is still active, and the transition from `A` to `B` is not possible because the current event is now `F`. The transition from `A` to `C` is attempted and successfully executed; this leads to the state `A` being exited, the state `C` being entered, and the reexecution terminates.

In some situations, reexecution could lead to an inconsistency if the original execution were to proceed. In our example, after the reexecution, the initial execution recovers control; if it continued, the condition action that generated the local event broadcast would proceed. In this case, however, `A` would be exited (although it is already inactive), and `B` would be entered, leading to a configuration where both `B` and `C` would be active at the same time, which is inconsistent since the chart has a sequential decomposition.

To avoid such situations, early return logic conditions identify, immediately after a reexecution prompted by a local event broadcast, the configurations that can lead to an inconsistent state and the remaining actions that must, therefore, be interrupted. In our example, these are all the remaining actions of the execution step. If, on the other hand, the three states in our example were inside a parallel state, only the execution of `A` would be interrupted. Any other parallel states would still be executed in the same step of execution.

The definitive semantics of Stateflow is given by simulation, and informally in a user's guide. Our modelling technique provides a formal semantics. It is distinctive in its coverage of intricate features of Stateflow, like connective and history junctions, including the possibility of backtracking in transition paths, all forms of transitions, including inter-level transitions, local event broadcasts, and early return logic.

Before presenting our modelling approach, we give, in the next section, an overview of *Circus*.

2.2 *Circus*

We present the main *Circus* features using the example in Figure 4. It models a parallel sorter that reads a sequence of natural numbers through the channel *in*, and writes on the channel *out* an ordered version of the

```
**channel** $in, in1, in2, out : \mathrm{seq}\,\mathbb{N}$

**process** $Merger \mathrel{\widehat=}$ **begin**
    **state** $S == [y : \mathrm{seq}\,\mathbb{N}]$
    $InitS == [S' \mid y' = \langle\rangle]$
    $Merge \mathrel{\widehat=} x1, x2 : \mathrm{seq}\,\mathbb{N} \bullet$
        **if** $\# x1 = 0 \longrightarrow y := y \frown x2$
        $[\!]$ $\# x2 = 0 \longrightarrow y := y \frown x1$
        $[\!]$ $\# x1 \neq 0 \land \# x2 \neq 0 \longrightarrow$
            **if** $head\,x1 \leq head\,x2 \longrightarrow y := y \frown \langle head\,x1 \rangle \ ; \ Merge(tail\,x1, x2)$
            $[\!]$ $head\,x1 > head\,x2 \longrightarrow y := y \frown \langle head\,x2 \rangle \ ; \ Merge(x1, tail\,x2)$
            **fi**
        **fi**
    $\bullet\ InitS \ ; \ in1?x1 \longrightarrow in2?x2 \longrightarrow Merge(x1, x2) \ ; \ out!y \longrightarrow$ **Skip**
**end**

**process** $SplitSorter \mathrel{\widehat=} \dots$

**process** $ParallelSorter \mathrel{\widehat=} (SplitSorter \llbracket \{\!| in1, in2 |\!\} \rrbracket Merger) \setminus \{\!| in1, in2 |\!\}$
```

Figure 4: The *ParallelSorter* specification.

input sequence. A detailed presentation of *Circus* can be found in [Oli06].

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 main concept is, like in CSP, that of a process; both systems and their components are modelled by processes. A process encapsulates a state (specified like in Z) and exhibits a behaviour in the form of a pattern of interactions specified by a (main) action. Processes communicate with each other and their environment via channels (like in CSP). An action is written using a mixture of Z data operations, CSP constructs, and Dijkstra's guarded commands. CSP process operators can also be used in *Circus* to combine processes.

The first paragraph of Figure 4 defines channels in, $in1$, $in2$, and out, which communicate sequences of natural numbers: elements of the type $\mathrm{seq}\,\mathbb{N}$. The following paragraphs define processes. The second and third paragraphs define basic processes *Merger* and *SplitSorter*, specified using a combination of Z and CSP notation. The last paragraph defines a process *ParallelSorter* in terms of *Merger* and *SplitSorter*.

The definition of *Merger* declares its state using a Z schema named S. Afterwards, a few local actions are declared before the specification the main action. The schema S has only one component y of type $\mathrm{seq}\,\mathbb{N}$. The action *InitS* is a state initialisation operation defined by a Z operation schema; it sets y to the empty sequence ($\langle\rangle$). Like in Z, we use y' to refer to the value of y after the operation.

The action *Merge* with parameters $x1$ and $x2$ (of type $\mathrm{seq}\,\mathbb{N}$) is defined using a conditional (and assignments). We present in Table 1 the notation used to specify *Circus* actions that are relevant in this paper. More details of the notation are provided as needed. *Merge* appends $x1$ and $x2$ to y, so that if both input sequences are ordered, the final sequence in y is also ordered. If one of the sequences is empty ($\# x1 = 0$ or $\# x2 = 0$), the non-empty sequence is appended. When both sequences are not empty, *Merge* compares the first element (that is, the head) of each sequence ($head\,x1 \leq head\,x2$). It appends the smallest of them to y (using the sequence concatenation operator $\frown$), and recursively calls *Merge* on the rest of the sequence that had the smallest element ($tail\,x1$ or $tail\,x2$), and the whole of the other sequence.

The main action of *Merge* (at the end after a $\bullet$) is nameless; it defines, using the previously defined local actions, the overall behaviour of the process. It initialises the state using *InitS*, reads a value $x1$ through the channel $in1$, reads a value $x2$ through $in2$, calls *Merge* with $x1$ and $x2$ as parameters, and outputs y through out. A prefixing (synchronisation $c \longrightarrow A$, input $c\,?\,x \longrightarrow A(x)$, or output $c\,!\,E \longrightarrow A$) defines an action that is ready to engage in a communication and then behave like A. In our example, for instance, we have communications like the input $in1?x1$, which reads a value through the channel $in1$ and assigns it to the variable $x1$ local to the prefixing, and the output $out!y$. The action **Skip** terminates immediately.

The fourth paragraph in Figure 4 defines *ParallelSorter* as the parallel composition (operator $\llbracket \dots \rrbracket$) of *SplitSorter* and *Merge*, communicating over $in1$ and $in2$. The process *SplitSorter* in the third paragraph is omitted; it inputs a sequence of natural numbers through in, splits it in two, sorts each sequence in parallel, and outputs them through $in1$ and $in2$. In the definition of *ParallelSorter*, the channels $in1$ and $in2$ are in the synchronisation set of the parallelism. This means that communications over these channels require synchronisation between the parallel processes *SplitSorter* and *Merge*, but communications over other

Table 1: Selection of *Circus* action constructs.

Action	**Syntax**	**Description**
Skip	**Skip**	Immediately terminates without changing the state.
Synchronisation	$c \longrightarrow A$	Simplest form of an interaction: no value is communicated.
Input Prefix	$c?x \longrightarrow A(x)$	Binds the variable x to the value read through the channel c.
Output Prefix	$c!E \longrightarrow A$	Outputs the value of the expression E on the channel c.
External Choice	$A_1 \Box A_2$	A choice that is resolved by the environment.
Sequence	$A_1 \ ; \ A_2$	Executes the two actions A_1 and A_2 in sequence.
Parallelism	$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$	Parallelism where actions must synchronise on the channels in cs, A_1 only modifies state components in ns_1, and A_2 only modifies state components in ns_2.
Interleaving	$A_1 \lvert\lvert\lvert ns_1 \mid ns_2 \rvert\rvert\rvert A_2$	Parallelism where A_1 and A_2 do not synchronise, A_1 only modifies state components in ns_1, and A_2 only modifies state components in ns_2.
Iterated sequence	$; \ x : S \bullet A(x)$	Sequence of all actions $A(x)$ with $x \in \operatorname{ran} S$, in the order defined by the sequence S.
Interrupt	$A_1 \triangle c \longrightarrow A_2$	A synchronisation on c interrupts the execution of A_1 and subsequently transfers control to A_2.
Hiding	$A \setminus cs$	Interactions via channels in cs are hidden and take place immediately when they are enabled.
Recursion	$\mu X \bullet F(X)$	Occurrences of X in F constitute recursive calls.
Assignment	$x := E$	Changes the value of a state component or local variable.
Local Variable	$\mathbf{var}\, x : T \bullet A(x)$	Declaration of a local variable x.
Conditional	$\mathbf{if}\ g_1 \longrightarrow A_1 \ [\!] \ldots [\!] \ g_n \longrightarrow A_n\ \mathbf{fi}$	Executes an action whose guard g_i is true.
Value Parameter	$\mathbf{val}\, p : T \bullet A(p)$	Action with a value-parameter p of type T.

channels can be carried out independently by them. The channels $in1$ and $in2$ are hidden (operator $\setminus$) in *ParallelSorter*, thus yielding a process whose interface contains only *in* and *out*.

Circus has a refinement calculus [CSW03] based on Morgan's and the Z calculi [Mor94, CW99], but extended to deal with the CSP constructs. In Table 2, we present a description of the laws that we use here. They include laws presented elsewhere [Oli06, CCO11], as well as novel laws found in Appendix B. As already explained, the *Circus* notion of refinement is akin to that of failures-divergences refinement. It is a transitive relation, so that the repeated application of *Circus* refinement laws, as usual in a refinement calculus and as prescribed in the strategy that we present here, preserves refinement. The semantics of the *Circus* constructs also guarantees that refinement of different components of a model can be considered independently.

This notion of refinement is based on the related notion of observation and captures a notion of correctness based on reduction of observation. More precisely, a *Circus* process P is refined by another *Circus* process Q if every observation of Q is a possible observation of P. In this respect, if Q is (the model of) a proposed implementation of a given specification P, for example, then refinement guarantees that a user that agreed on the specification P has to be satisfied by Q because every observation of the behaviour of Q is in accordance with the behaviours prescribed by P.

Embedded in this view is reduction of nondeterminism. An abstract specification P typically embeds some nondeterminism to express freedom of design and implementation. Refinement reduces this nondeterminism as it moves towards more specific architectural designs and patterns of implementation.

The notion of observation is therefore central to our understanding of refinement. In Z, it corresponds to the observation of the behaviour of data operations in terms of acceptable inputs and properties of the produced outputs. In CSP, it can be about (1) traces of interactions; (2) traces and refusals, also known as failures; or (3) failures and divergences. If we can only observe traces of interactions, we can only reason about safety properties. With failures, we can also reason about liveness: absence of deadlock. Finally, with a failure-divergences observation model, we can in addition reason about the possibility of an operation aborting or entering in an infinite loop of internal actions.

The refinement notion of *Circus* is based on its semantics given using Hoare and He's Unifying Theories of Programming [HH98]. This is a notion based on failures and divergences, but also on the behaviour of data operations (which are internal to processes). In this respect, traditional refinement techniques based on data refinement, for instance, are valid, as are techniques for model checking traditionally adopted in CSP, after

Table 2: Selection of *Circus* refinement laws.

Law	Description	Source
assign-schema-conv	Convert assignment with assumption to a schema	Appendix B
cond-elim	Eliminate conditional with guard true	Appendix B
hid-distr-seq	Distribute hiding over sequence	[Oli06]
hid-ident	Remove hiding of channels that are not used	[Oli06]
par-distr-cond	Distribute parallelism over conditional	Appendix B
par-unit	**Skip** in parallel with **Skip** is **Skip**	[Oli06]
par-prefix-step	Step law for parallelism: independent communication on the left	Appendix B
par-seq-dist	Exchange law for parallelism and sequence	Section 5.3.1
par-seq-step	Step law for parallelism: sequence on the left	[Oli06]
rec-par-merge	Transform parallelism of recursions into recursion of parallelisms	Section 5.2
sch-par-distr	Distribute a schema over a parallelism in sequence	[Oli06]
seq-assign-conv	Refine a schema to a sequence of assignments	Appendix B
seq-distr-cond	Distribute sequence over conditional - sequence on the right	Appendix B
tail-rec-seq-dist	Insert sequence in a recursion	Appendix B
use-loc-var	Transform calls to use a local variable	Section 5.3.1
var-par-ext-right	Extend scope of variable block over parallelism	[Oli06]
var-seq-ext-right	Extend scope of variable block over sequence	Appendix B

extension to deal with the rich data types of Z (also adopted in *Circus*).

In the next section (Figure 6), we have another example of a process: the model of the chart in Figure 2.

2.3 A formal model of Stateflow charts

In this section, we describe the *Circus* models of Stateflow charts that we can generate automatically. A more detailed description, including an account of our efforts to validate the models and of our tool can be found in [MC12]. For the purposes of the work that we present in this paper, it is enough to understand the structure of the models. It is described diagrammatically in Figure 5 and illustrated in Figure 6.

In our models, the execution of one step of the chart is initiated by reading inputs via channels that represent input events and variables of the chart, and is concluded by writing outputs via channels for output events and variables, and then synchronising on a channel called *end_cycle*. As shown in Figure 5, the models consist of two processes combined in parallel. The process *Simulator* captures the operational semantics embedded in the simulator, and is the same for every chart. The second, the chart process called *P_Chart* in Figure 5, represents a particular chart. These processes communicate over the channels in a set *interface* plus the channel *end_cycle*, with the channels in *interface* hidden. The result is a process named after the chart, which in Figure 5 we call *Chart*, and which communicates with the environment only through *end_cycle* and channels that represent input and output events and variables of the chart.

The chart process uses a data model that defines the state, transition, and junction identifiers, as well as the states, transitions, and junctions themselves as Z bindings (records). These are constants that capture information about the structure of the chart (top rectangle in Figure 5). The chart process *P_Air* for our example is sketched in Figure 6. Its first paragraph introduces some constants. These include, for instance, *s_Off*3 and *c_Air*, the identifiers of the state `Off` (in `FAN1`) and of the chart. Charts, just like states, have identifiers defined by Simulink. We also have *S_Off*3, a binding that records information about `Off` (its identifier, the fact that it has no history junctions, and so on).

The second paragraph in Figure 6 defines two channels *o_airflow* and *i_temp*, corresponding to the output variable `airflow` and to the input variable `temp`. The types of the channels are those of the variables.

The constants that record the structure of the chart, like *s_Off*3, *S_Off*3, and *c_Air*, for example, are collected in four other constants defined within the chart process: *identifier*, *states*, *transitions*, and *junctions*. The constant *identifier* records the identifier of the chart, and *states*, *transitions*, and *junctions* are partial functions that map identifiers to the corresponding bindings. In Figure 6, the first definition sketched in the scope of *P_Air* defines these values for our example.

Next, the chart process defines a series of schemas that specify its state and initialisation operation. Chart variables are recorded in the schema *SimulationInstance*. For each output event, we also have a component in *SimulationInstance* that records how many occurrences of that event have happened so far, and have not yet been communicated to the Simulink diagram. All these counters are initially 0. In our example, the components of *SimulationInstance* are *v_airflow* and *v_temp* corresponding to variables.

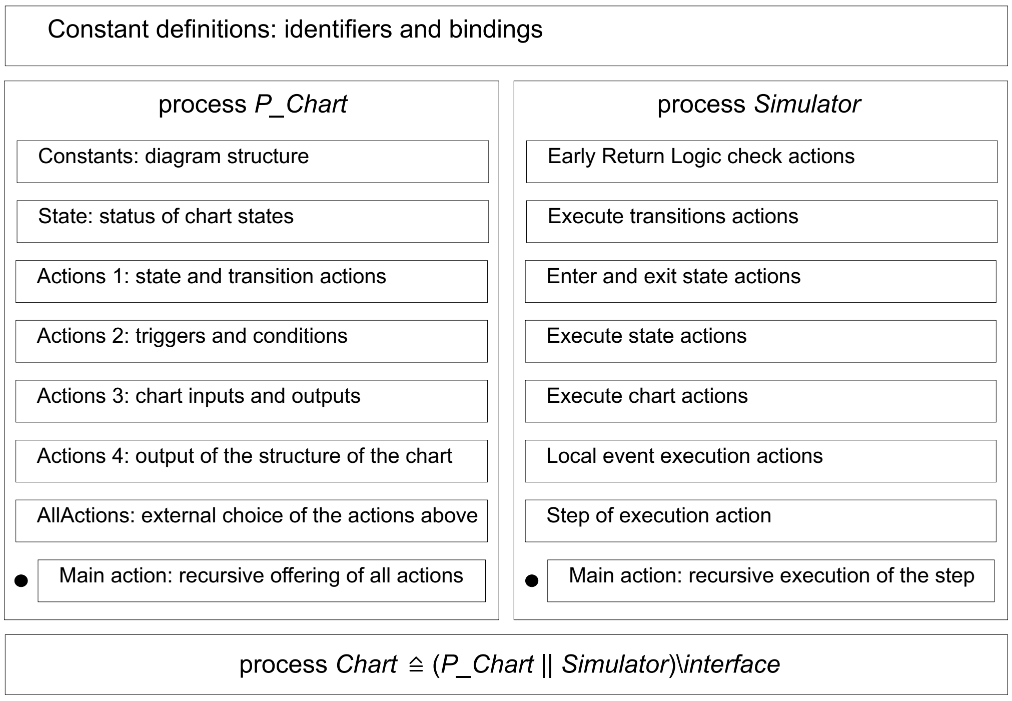

Constant definitions: identifiers and bindings
process P_Chart
Constants: diagram structure
State: status of chart states
Actions 1: state and transition actions
Actions 2: triggers and conditions
Actions 3: chart inputs and outputs
Actions 4: output of the structure of the chart
AllActions: external choice of the actions above
● Main action: recursive offering of all actions
process Simulator
Early Return Logic check actions
Execute transitions actions
Enter and exit state actions
Execute state actions
Execute chart actions
Local event execution actions
Step of execution action
● Main action: recursive execution of the step
process Chart ≙ (P_Chart || Simulator)\interface

Figure 5: Structure of the model of charts.

Information about which chart states are active and about chart states monitored by history junctions is recorded in the schema *SimulationData*. Its definition is the same for all charts. Its first component is a function *state_status* that associates a state (identifier) to a boolean that indicates whether it is active or not. The second component *state_history* associates a state with a history junction to the identifier of its last active substate. The schemas *SimulationInstance* and *SimulationData* are conjoined to define the schema that specifies the state of the chart process, in our example, *Air_State*.

As shown in Figure 5, the chart process defines a series of *Circus* actions that can be divided into four groups corresponding to: state and transition Stateflow actions, calculation of triggers and conditions, reading inputs and writing outputs, and outputting information about the structure of the chart, such as the parent of a state. These actions define services that are offered to the *Simulator* process via specific channels.

The first group contains actions such as *entryaction_PowerOff* in Figure 6; they are prefixed by a communication (over, for example, *executeentryaction* for entry actions) of the identifier of the state (or transition) to which the action is associated, followed by a *Circus* action that models the Stateflow action. In our example, the entry action associated to `PowerOff` is encoded as the assignment $v\_airflow := 0$.

The second group is formed by actions such as *trigger_PowerOff_PowerOn*; it encodes the trigger of the transition between *s_PowerOff* and *s_PowerOn* as a prefixing whose communication gives the transition identifier and the current event (e). The associated *Circus* action consists of a conditional that compares e to the event that guards the transition (in this case e_SWITCH). A channel *result* communicates the result of the guard evaluation as a boolean. (We define $\mathbb{B}$ to be the set of boolean values.)

The third group contains two actions *Inputs* and *Outputs*, which are a prefixing on the events *read_inputs* and *write_outputs*, respectively. *Inputs* reads the value of the input variables (in interleaving, if there are several) through the channels that represent them, and assigns the values read to the corresponding state components. In our example, as already mentioned, we have just one input variable `temp`, and the corresponding channel *i_temp* and state component *v_temp*. It is worth mentioning that input events are not handled by *Inputs*, or even by the chart process. In our model, a step of execution of the Stateflow block is triggered by the input events, and therefore they are handled by the *Simulator* process.

Outputs communicates (in interleaving, if needed) the values of the state components through the channels that represent the corresponding output variables. In addition, for each output event of the chart, if there are occurrences not yet signalled, as indicated by a positive value of its counter in the state, *Outputs* signals one occurrence of the event in interleaving, and decrements the counter. In our example, for the single output variable `airflow`, we have channel *o_airflow* and component *v_airflow*.

Finally, the fourth group contains actions such as *getstate*, which takes a state identifier and communicates its binding. Actions in each of the groups are combined in external choices (operator $\Box$) to define actions like *conditionactions*, *InterfaceActions*, and so on. In turn, their external choice defines *AllActions*. The choice

$$\begin{array}{|l}
s\_Off3, \ldots, c\_Air : SID;\ \ldots\ S\_Off3 : State;\ \ldots \\
\hline
S\_Off3 = \langle\!| identifier == s\_Off3, \ldots history == \mathbf{False} |\!\rangle \land \ldots
\end{array}$$

channel $o\_airflow : \mathbb{N};\ i\_temp : \mathbb{R}$

process $P\_Air \mathrel{\widehat{=}}$ **begin**

$$\begin{array}{|l}
\ldots \\
\hline
identifier = c\_Air \land states = \{(s\_Off3, S\_Off3), \ldots\} \\
transitions = \{(t\_On8\_Off7, T\_On8\_Off7), \ldots\} \land junctions = \{\}
\end{array}$$

$SimulationInstance == [v\_airflow : \mathbb{N};\ v\_temp : \mathbb{R}]$
$\ldots$
$SimulationData == [state\_status : SID \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{B};\ state\_history : SID \mathrel{+\!\!\!\!\twoheadrightarrow} SID \mid \ldots]$
$\ldots$
state $Air\_State == SimulationInstance \land SimulationData$

$InitState == \ldots$
$\ldots$

$entryaction\_PowerOff \mathrel{\widehat{=}} executeentryaction.s\_PowerOff \longrightarrow v\_airflow := 0$
$\ldots$

$$trigger\_PowerOff\_PowerOn \mathrel{\widehat{=}} checktrigger.t\_PowerOff\_PowerOn?e \longrightarrow \\ \left(\begin{array}{l} \mathbf{if}\ e = e\_SWITCH \longrightarrow result.t\_PowerOff\_PowerOn.e!\mathbf{True} \longrightarrow \mathbf{Skip} \\ [\!]\ \neg\,(e = e\_SWITCH) \longrightarrow result.t\_PowerOff\_PowerOn.e!\mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right)$$
$\ldots$

$Inputs \mathrel{\widehat{=}} (read\_inputs \longrightarrow (i\_temp?x \longrightarrow v\_temp := x))$
$Outputs \mathrel{\widehat{=}} (write\_outputs \longrightarrow (o\_airflow!(v\_airflow) \longrightarrow \mathbf{Skip}))$
$\ldots$

$getstate \mathrel{\widehat{=}} state?x : (x \in \mathrm{dom}(states))!(states(x)) \longrightarrow \mathbf{Skip}$
$\ldots$
$AllActions \mathrel{\widehat{=}} conditionactions \Box \ldots \Box InterfaceActions$

$\bullet\ InitState\ ;\ (\mu X \bullet (\mu Y \bullet (AllActions\ ;\ Y \Box end\_cycle \longrightarrow \mathbf{Skip})\ ;\ X)$

end

process $Air\_Controller \mathrel{\widehat{=}} (P\_Air\_Controller \llbracket \ldots \rrbracket Simulator) \setminus interface$

Figure 6: Sketch of the model of the chart process for the example in Figure 2.

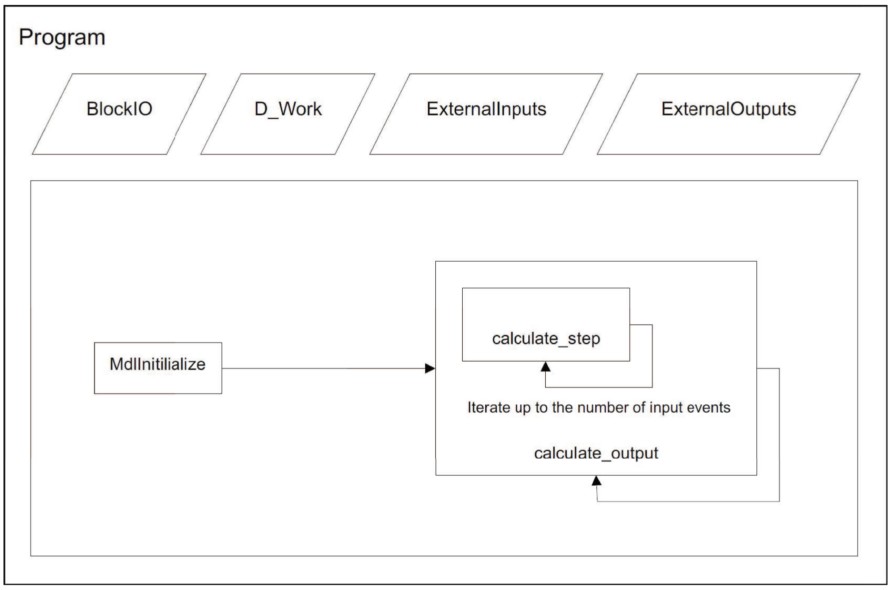


Figure 7: Architecture of implementations of Stateflow charts.

is exercised by the *Simulator* process, which is part of the environment of the chart process.

The main action of the chart process initialises the state, and recursively offers the services in *AllActions*. There are two nested recursions: the internal one corresponds to the services offered during one particular step of simulation, and can be terminated by a synchronisation over the channel *end_cycle*. The external recursion corresponds to the recursive execution of simulation steps.

The process *Simulator* does not have a state; it declares a series of actions that model the execution of transitions, as well as the procedures for entering, executing and exiting states. These actions capture the operational semantics of Stateflow charts as described in the manual. Its interface consists of a single channel *input_event*, which is used to receive the events for which the chart is to be executed, and a series of internal channels (*read_inputs*, *executeentryaction*, and so on) used solely for interacting with the chart process.

The execution of a chart is then defined in terms of these. We do not provide the details of the (extensive) *Simulator* definition here, which can be found in [MC12]. We describe some parts relevant for our refinement strategy in Section 5.

As already said, we can generate the *Circus* models of Stateflow diagrams described above automatically. In [MC12, Miy12], we define a collection of rules that translate a textual description of a Stateflow diagram to *Circus*. This textual description is that provided by MATLAB Stateflow, and it already resolves tricky issues associated with the graphical rendering of diagrams: order of execution of parallel states. Our work covers a significant subset of the Stateflow notation, including edge-triggered events, data, actions, parallel and exclusive states, connective and history junctions, and all forms of transitions. As far as we know, we provide the widest coverage of Stateflow features.

3 Implementations of Stateflow charts

Our refinement strategy focuses on the implementations of Stateflow charts that may be generated by the Realtime Workshop [Matb] in association with the Stateflow Coder [Matd], but also covers programs that, perhaps, result from modifications of such implementations, but preserve its specific architectural patterns.

Figures 7, 8 and 9 describe the pattern structure of the programs generated by the Realtime Workshop. This is exactly the architectural pattern that we assume in our work, and the only aspect of the Realtime Workshop that is relevant for what we present here.

We distinguish two major aspects of the architecture: data types (represented by slanted boxes in Figure 7) and control flow (rectangular boxes). The first determines how information regarding the status of the states, history junctions, input, output and local data, and events are represented. The second defines how states and transitions are executed. Section 3.1 discusses the data model patterns, and Section 3.2 the execution control patterns.

The programming language used by Stateflow Coder, and that we adopt in examples, is a subset of C. Our strategy, however, is in no way restricted to C, but to the architectural pattern described here, which can be realised by programs written in other languages, like SPARK Ada [Bar03], for instance.

3.1 Architecture: data patterns

The data model of our architectural pattern uses a number of variables to record input, output and local data and events, and execution data used to determine the state of the chart. These variables are grouped in records represented by the slanted boxes in Figure 7. They are used as types of global variables used to control the execution of the chart. We also have an extra global variable that records the event under which the chart is being executed in a particular step. It is called `_sfEvent_C_`, where `C` is the name of the chart. For our example, this global variable is called `_sfEvent_Air_`.

3.1.1 `BlockIO` record

This type groups the variables that store output data and events. For each output data variable, a variable of the same name and type is included in `BlockIO`. For each output event, a variable of the same name and type Boolean is included; it records whether the event has occurred or not.

***Example* 1** *The record in our example is shown below; it contains only one variable that records the value of* `airflow`*. (Its type,* `uint8_T`*, is defined by the code generator for the unsigned integers of 8 bits.)*

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

There are no output events in this example. □

3.1.2 `D_Work` record

This contains the variables that model local data, record the status of the states and history junctions, and output event counters. For each local variable, a corresponding variable is declared.

For each parallel state, that is, for each state in a parallel decomposition, an integer variable whose name is prefixed by `is_active_` is declared. Such a variable is also defined for the chart. For each state with a sequential decomposition, we have an integer variable whose name is prefixed by `is_`. While all variables have type `uint8_T`, the `is_active_` variables are used as boolean variables, and the `is_` variables store values that indicate which substate is active or that no substate is active.

For each state with a history junction, we have an integer variable whose name is prefixed by `was_`. It records the constant for the last active substates.

Finally, for each output event, we have an integer variable whose name is postfixed by `EventCounter`.

***Example* 2** *The record type* `D_Work` *in the implementation of the chart in Figure 2 is shown below. Since the chart has no local data, output events or history junctions, this record encodes only the status of states.*

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

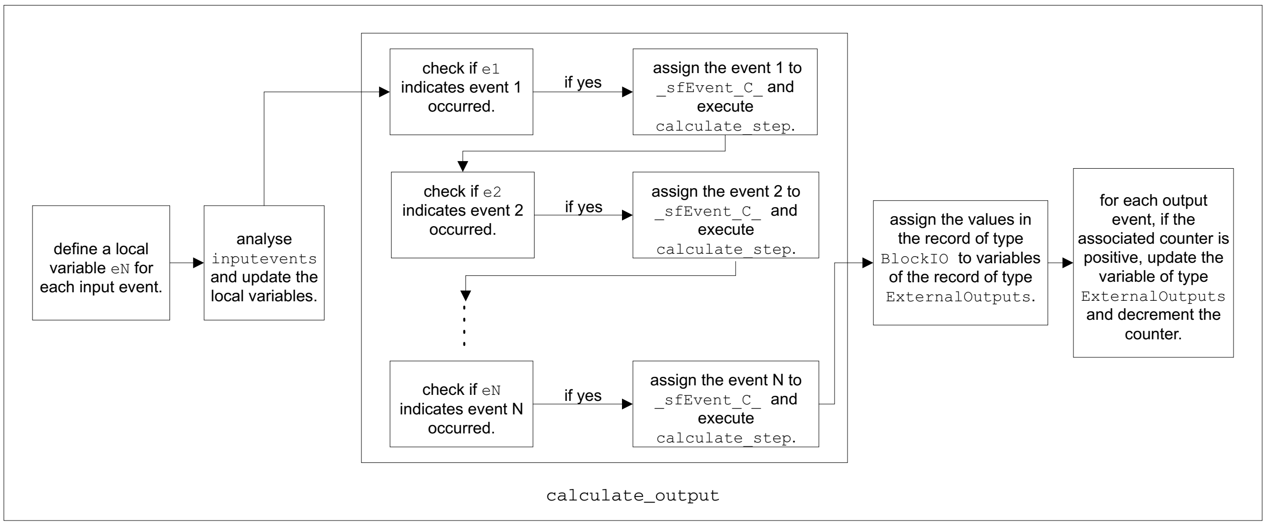
The variable `is_active_c1_Air` *records the status of the chart, and* `is_active_FAN1`*,* `is_active_FAN2` *and* `is_active_SpeedValue` *record, respectively, the status of the parallel states* `FAN1`*,* `FAN2` *and* `SpeedValue`*. The variable* `is_c1_Air` *records the status of the substates of the chart, and* `is_FAN1` *and* `is_FAN2` *record the statuses of the substates of* `FAN1` *and* `FAN2`*. The values of the* `is_` *variables are defined as constants.* □

3.1.3 `ExternalInputs` and `ExternalOutputs` records

The variables of these types are shared to communicate with the implementation of other blocks of the Simulink diagram. As suggested by their names, the record `ExternalInputs` collects the input and the record `ExternalOutputs` the output variables and events.

The input events are represented by an array `inputevents` of real numbers (type `real_T`). Its size is the number of input events, and the real values indicate whether the corresponding event occurred. (Real values are used for compatibility with implementations of Simulink diagrams that include the Stateflow block, but for the Stateflow implementation itself, all that matters is whether the event occurred or not.) The order in which the events are represented in the array is determined by an implicit ordering in the chart. Besides `inputevents`, for each input data `id`, a variable of the same name is declared in `ExternalInputs`.

In the end of each cycle, the values of the output data and events are written to the `ExternalOutputs` record to make them available. It contains, for each output data, a variable of the appropriate type and same name, and for each output event, a boolean variable of the same name. Unlike input events, output events are communicated individually, not through an array.



define a local variable eN for each input event.
analyse inputevents and update the local variables.
check if e1 indicates event 1 occurred.
if yes
assign the event 1 to _sfEvent_C_ and execute calculate_step.
check if e2 indicates event 2 occurred.
if yes
assign the event 2 to _sfEvent_C_ and execute calculate_step.
check if eN indicates event N occurred.
if yes
assign the event N to _sfEvent_C_ and execute calculate_step.
assign the values in the record of type BlockIO to variables of the record of type ExternalOutputs.
for each output event, if the associated counter is positive, update the variable of type ExternalOutputs and decrement the counter.
calculate_output

Figure 8: Structure of the procedure `calculate_output`.

***Example* 3** *Below, we show the record type for our implementation. The variable* `temp` *corresponds to the input data, and the array* `inputevents` *stores values associated to* SWITCH *and* CLOCK*.*

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

The `ExternalOutputs` *record for our example is below.*

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

□

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

3.2 Architecture: control flow

With respect to the execution flow of the chart, the relevant procedures of our architectural pattern are depicted in Figure 7: `MdlInitialize`, `calculate_output`, and `calculate_step`.What we have is an iterative calculation of the outputs.The execution of the chart is initialised by calling `MdlInitialize`, which initialises the components of the records of type `D_Work` and `BlockIO`. The outputs are calculated iteratively by calling `calculate_output`. The calculation of the outputs depends on the procedure `calculate_step`, which implements the execution step of the chart.

***Example* 4** *For our example,* `MDLInitialize` *is sketched below. It initialises the components of the variable* `Air_DWork` *that represent state status to 0 (value* `0U`*, an unsigned 0), indicating that every state is inactive. It also initialises the component of* `Air_B`*; the initial value of output data is defined in the chart.*

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

We omit commands related to aspects of the program that we do not model, like time control. □

Figure 8 defines the programming pattern for `calculate_output`. It processes the array `inputevents`, and calls `calculate_step` for each event that occurred. Once the chart is executed for all events, it shares the values recorded in the `BlockIO` record, by copying them to the `ExternalOutputs` record. Moreover, for each output event raised, it decrements the associated counter in the `BlockIO` record.

Figure 9 shows the pattern for `calculate_step`; it implements one complete execution of the chart. Its structure consists of a number of nested conditionals that evaluate the status of the chart and states and

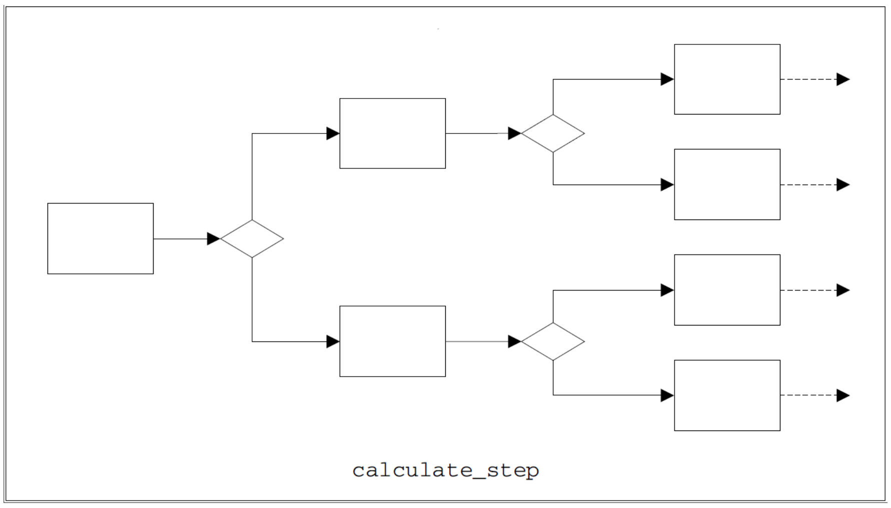


Figure 9: Structure of `calculate_step` and interaction patterns with the servers.

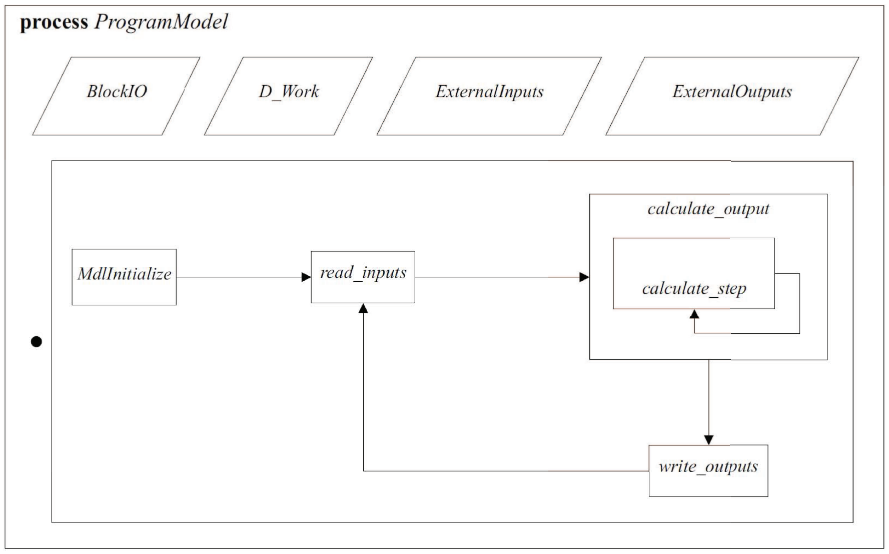


Figure 10: Overview of the models of implementations of Stateflow charts.

the transition guards. In Figure 9, the rectangular boxes correspond to blocks of code that implement the execution of the states, and the diamond-shaped boxes to decision points. For clarity, our figure shows only binary decision points, but they correspond to both `if` and `switch` statements.

When the chart, or part of it, is reexecuted as a result of a local event broadcast, `calculate_step` is called recursively. When the broadcast is directed at a particular state, only the block of code that implements the execution of that state is reexecuted. In this case, this code is used to define an auxiliary procedure, and a call to the new procedure implements the broadcast.

In the next section, we describe how we can construct *Circus* models of programs that follow the architectural, data, and control patterns just presented.

4 *Circus* models of implementations

The architecture of the *Circus* models is close to that of the corresponding programs; Figure 10 gives an overview (*cf* Figure 7). The only difference is that the *Circus* model has actions *read_inputs* and *write_outputs* that dot not correspond to a program component; they encode the Stateflow block behaviour. These are simple actions that are determined by the input events and data of a chart.

The *Circus* model is composed of a single process. Schemas $BlockIO\_C$, D_Work_C, $ExternalInputs\_C$, and $ExternalOutputs\_C$, where as before C stands for the name of the chart, are used to model the record types of the program. The state of the process includes components C_B, C_DWork, C_U, and C_Y of these types, and a component $sfEvent\_C$, all corresponding to global variables of the program. The main

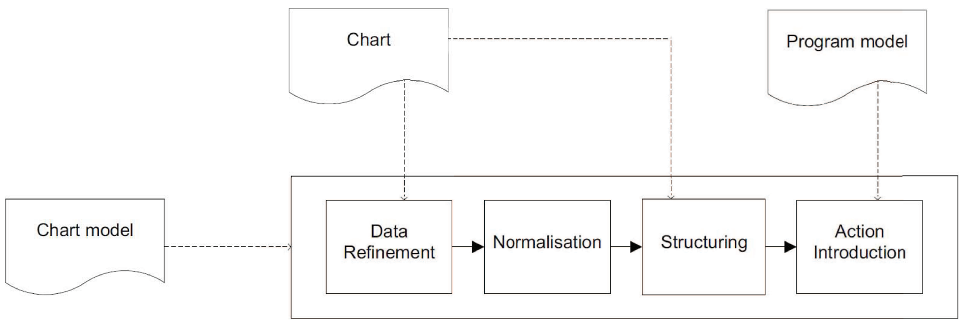


Figure 11: Overview of our refinement strategy.

action reflects the control pattern of the program (see Figure 7).

The generation of *Circus* models of implementations involves two different aspects: straightforward translation and abstraction. The statements are translated into *Circus* actions, and aspects of the program that are not covered by our models of Stateflow chart (for instance, time control) are abstracted.

Firstly, the calculations of the time steps that determine the execution of a chart are abstracted by a synchronisation over the channel *end_cycle* that marks the end of a cycle. Secondly, as illustrated in (the first two boxes of) Figure 8, the treatment of input events in the implementation involves calculations that identify which events occurred according to values supplied by the Simulink model in an array `inputevents`. In our models, we abstract from this calculation by assuming that this is an array of booleans that indicate the occurrence of each event, not its associated value. The translation to C embedded in the MATLAB tool does not make this abstraction, and includes an algorithm that decides, based on the real values of the events, whether an event has been triggered or not. Our models of C programs abstract this algorithm away, and so cannot be used to check its correctness.

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

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

The implementation (see Appendix A) of our example and its model are in `cs.york.ac.uk/circus/s2c`.

5 Refinement strategy

In this section, we present a new refinement strategy suited for the verification of implementations of Stateflow charts that follow the architectural pattern presented in Section 3. It is a tactic of refinement, which we define as a procedure for systematic application of refinement laws.

Our strategy is organised in four phases: data refinement, normalisation, structuring, and action introduction; an overview is provided in Figure 11. The main input is the model of the Stateflow chart, which as already explained, can be automatically generated. We also use the chart itself to guide our calculation of a concrete state in the data-refinement phase, and to identify the transition loops in the structuring phase. We also need to extract from the model of the program information like the actions that model the execution of the states. The strategy is a tactic that proves that the model of the chart is refined by that of the implementation by transforming the former into the latter using the algebraic refinement laws of *Circus*.

In general terms, the data-refinement phase introduces the data model of the implementation, the normalisation phase removes the structure of the abstract chart model, which reflects the Stateflow semantics, and the next two phases introduce the control aspects of the implementation architecture. The structuring phase introduces the control pattern, and the last phase introduces the appropriate naming of actions as adopted in the program model to reflect its functions (procedures).

As usual in refinement techniques, data refinement is carried out first, since typically the algorithms to be used and, therefore, the control flow of the program, are determined by the concrete data representations. To

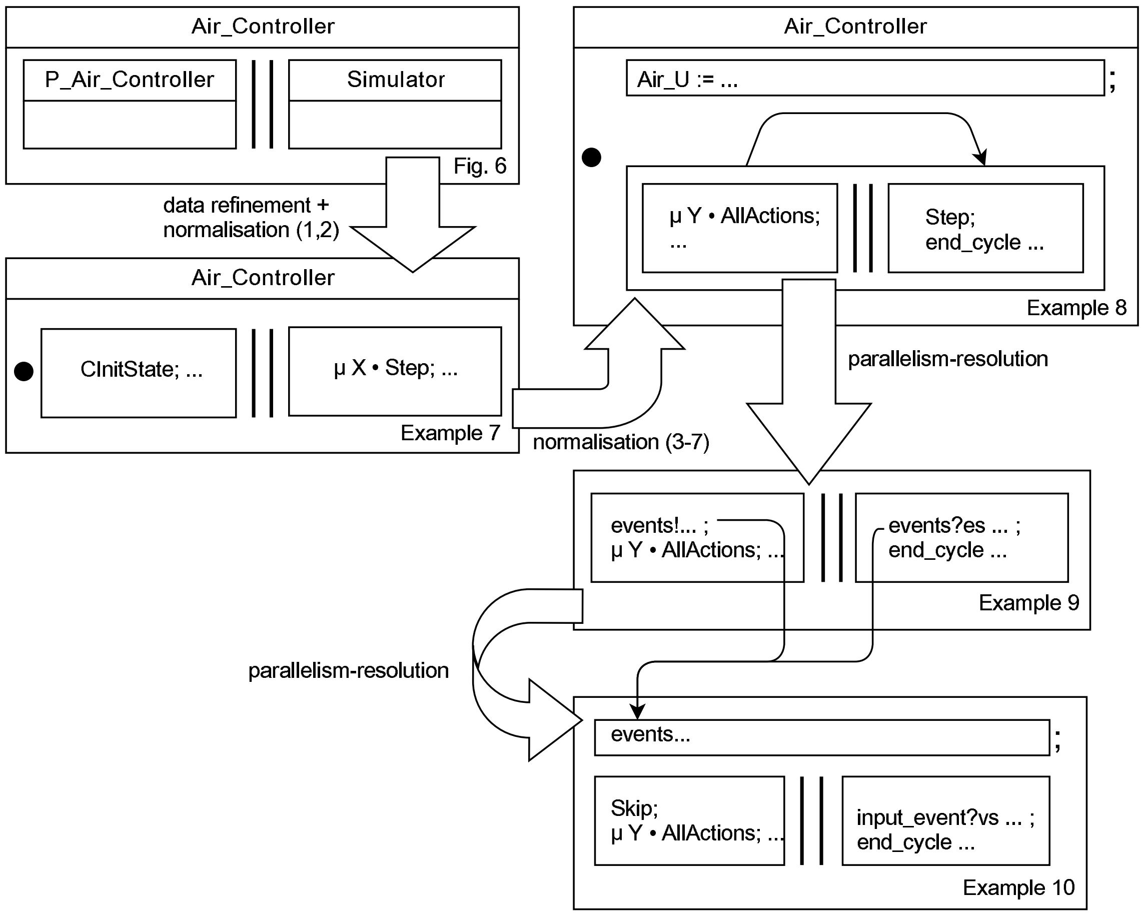


Figure 12: Initial steps of the application of the strategy to the example.

introduce the control structure, normalisation first removes all structure that stems from the modularisation of the abstract chart model. As shown in Figure 5, it embeds a parallelism that reflects the operational semantics of Stateflow, not the design of a program. We, therefore, remove that parallelism to establish a simplified starting point to introduce the structure of the program in the next structuring phase.

The normalised process that is obtained is described using the same pattern of specification used in [CCO11] as part of the verification technique for implementations of Simulink diagrams. In this way, we create the possibility of exploring the combination of our techniques to verify programs that implement Simulink diagrams that involve Stateflow blocks. The structuring phase takes the normalised process and (1) adds variables that are used in the (concrete) data model of the program but have no corresponding variable in the (abstract) data model of the chart; (2) removes any remaining parallelism; and (3) introduces the sequential structure of the program control flow pattern.

The final phase of action introduction is concerned with the introduction of the structure of the procedures of the program. The bodies of these procedures come from the components of the refined action obtained in the structuring phase. It would be possible to precede the structuring phase with the introduction of the procedures, each with an appropriate abstract specification. We could then carry out the structuring to use these procedures, based on their specifications, and refine these specifications independently. This would, however, require the availability of the procedure specifications, which is not necessarily a trivial matter.

In the sequel, we present each individual phase of our refinement strategy. For clarity and conciseness, in some steps, we omit the specification of the precise refinement laws to be applied. All details and missing definitions can be found in [Miy12]. It is worth emphasising that all steps of all phases amount only to the application of one or more of the *Circus* refinement laws. The only exceptions are Steps 1 and 2 of the data-refinement phase; these are, however, just calculations of definitions (of a concrete state and of a retrieve relation) to be used in a subsequent functional data refinement justified by simulation laws.

Figure 12 shows how the initial steps of the refinement strategy transform the model of our example.

Step 1: Calculate the concrete state. Use a template.

Step 2: Calculate the retrieve relation. Use a template.

Transform the chart process as follows.

Step 3: Introduce the abstract state invariant as an assumption after the initialisation and distribute it until all assignments are reached.

Step 4: Convert any actions of the form $\{inv\} ; v := e$. Apply Law assign-schema-conv to all of them.

Step 5: Calculate the simulation. Apply the *Circus* laws of action simulation to the chart process.

Figure 13: Refinement strategy: data-refinement phase.

5.1 Data refinement

The data-refinement phase transforms the state of the chart process. The result is a process whose concrete state already includes many of the components of the implementation model. The exceptions are the components that correspond to output events in C_B, the component *inputevents* of C_U, and the components C_Y and $sfEvent\_C$, which are related to the treatment of input and output events, and output data, and are introduced later in the structuring phase. Figure 13 describes the steps of the data-refinement phase. The laws for which we do not give a reference in this figure, and in others to follow, can be found in Appendix B.

We calculate the concrete state of the implementation model, and a retrieve relation that allows us to calculate a data refinement of the chart process using the *Circus* refinement calculus. It preserves the structure of the process, and transforms the assignments, operation schemas, and communications.

Step 1. We calculate, besides the concrete state, properties of its components that become the concrete state invariant. Namely, we define three schemas: $BlockIO\_C$, D_Work_C, and $ExternalInputs\_C$, where, as before, C stands for the name of the chart. Their specifications can be calculated by instantiating general templates based on the definition of the chart. They define that, for instance, in the schema $BlockIO\_C$, for each output variable in the chart, we declare a component of the same name and type.

In D_Work_C, for each chart local variable, we declare a component of the same name and type; all other components have type $\mathbb{N}$. We declare, for each parallel state S and the chart, a component is_active_S; for each S with a sequential composition, is_S; for each history junction within a state S, was_S; and for each output event e, a component $eEventCounter$. For each state S with a sequential decomposition, possibly including the chart, the invariant requires that the value of is_S is restricted to an identifier of a substate of S, or $C\_IN\_NO\_ACTIVE\_CHILD$, when none of them are active. Similarly, the value of a was_S variable is restricted to a substate of S, or $C\_IN\_NO\_ACTIVE\_CHILD$, if none of them have been active yet.

Finally, in $ExternalInputs\_C$, for each input data, we declare a component of the same name and type.

***Example* 5** *The D_Work_C schema for our example is as follows.*

$$
\begin{array}{|l}
\hline
D\_Work\_Air \\
\quad is\_active\_c1\_Air : \mathbb{N} \\
\quad is\_active\_FAN1, is\_active\_FAN2, is\_active\_Speedvalue : \mathbb{N} \\
\quad is\_c1\_Air, is\_FAN1, is\_FAN2 : \mathbb{N} \\
\hline
\quad is\_c1\_Air \in \{Air\_IN\_NO\_ACTIVE\_CHILD, Air\_IN\_PowerOn, Air\_IN\_PowerOff\} \\
\quad is\_FAN1 \in \{Air\_IN\_NO\_ACTIVE\_CHILD, Air\_IN\_Off, Air\_IN\_On\} \\
\quad is\_FAN2 \in \{Air\_IN\_NO\_ACTIVE\_CHILD, Air\_IN\_Off, Air\_IN\_On\} \\
\hline
\end{array}
$$

□

The concrete state is defined by a schema $ConcreteState$ with three components C_B, C_DWork, and C_U, with an appropriate schema type as defined above.

Step 2. The retrieve relation maps the components of $BlockIO\_C$ that correspond to input and output variables and the components of D_Work_C that correspond to local variables to components of the schema $SimulationInstance$. The $eEventCounter$ components of D_Work_C are mapped to the counters in

SimulationInstance. The components of D_Work_C that record the status of the states and the history junctions are mapped to the components *state_status* and *state_history* of *SimulationData*.

***Example* 6** *For our example, we have the following retrieve relation.*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \ \mathit{RetrieveFunction}\ \ }} \\
P\_Air\_S \\
ConcreteState \\
\hline
v\_airflow = Air\_B.airflow \\
state\_status = \{s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet \\
\quad s = s\_PowerOn \land active = \left(\begin{array}{l}\mathbf{if}\ Air\_DWork.is\_Air = Air\_IN\_PowerOn \\ \mathbf{then\ True} \\ \mathbf{else\ False}\end{array}\right) \lor \\
\quad \ldots \\
\quad s = c\_Air \land active = \mathbf{if}\ Air\_DWork.is\_active\_Air \neq 0\ \mathbf{then\ True\ else\ False} \\
\} \\
state\_history = \{\} \\
v\_temp = Air\_U.temp \\
\hline
\end{array}
$$

In the definition of state_status, we use a set comprehension to define how each s in states is associated to a boolean active in state_status. The condition for the state `PowerOn`*, for instance, equates s to s_PowerOn, and active to* **True** *or* **False** *depending on the value of is_Air, which corresponds to the chart. Since our example does not contain history junctions, the* `D_Work_Air` *record in its implementation has no* `was_` *field. The model of the implementation, therefore, has no component that models the state component state_history. In this case, in the retrieve relation, we equate state_history to the empty set.* □

In general, the correspondence between variables is trivial. It is obtained by equating each of the concrete variables to the corresponding abstract variable whose name is the same except for a prefix $v\_$. For event counters, each *eEventCounter* component is equated to the corresponding *counter_e*.

The function *state_status* is related to the *is_active_* and *is_* variables. We characterise how state identifiers s are related to booleans *active* in *state_status*, using the *is_active_* and *is_* components of *C_DWork*, the record of type *DWork_C* in the concrete state. For each *is_S* variable, we require $s = s\_SS \land active = (\mathbf{if}\ C\_DWork.is\_S = C\_IN\_SS\ \mathbf{then\ True\ else\ False})$, for each substate SS of S. For each *is_active_S*, we require $s = S \land active = (\mathbf{if}\ C\_DWork.is\_active\_S \neq 0\ \mathbf{then\ True\ else\ False})$. Similarly, the relation between the *was_* prefixed variables and *state_history* is specified using a set comprehension that defines the value of *state_history* in terms of the value of each of the *was_* variables.

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

Steps 3, 4, and 5. The ultimate goal of these steps is to carry out a data refinement. Steps 3 and 4 carry out transformations to allow the application of the *Circus* data refinement techniques in Step 5.

Step 3 introduces the invariant of the abstract state as an assumption after the initialisation and distributes it through the action until all assignments are preceded by the invariant.

Some of the assignments in the chart, and therefore, in the (abstract) chart process are implemented as assignments to components of records. In the implementation model, they become assignments to records themselves. For instance, the assignment $v\_airflow := 0$ in the action *entryaction_PowerOff* in the chart process shown in Figure 6, corresponds to an assignment $Air\_B := \langle\!|\, airflow == 0 \,|\!\rangle$ in the model of the implementation. The simulation law for assignment, however, does not handle records directly, and therefore, in Step 4 we transform the assignments to schema operations.

To obtain schema actions to which simulation laws can be applied, however, we need to include the state invariant in these schemas. This is achieved in Step 4 by applying the novel law Law assign-schema-conv to the assignments preceded by the assumptions introduced in the previous step.

The data refinement carried out (in Step 5) is standard [CSW03, WD96]. Using the retrieve relation, we apply the *Circus* laws of simulation to obtain a *Circus* process with the concrete state by data refinement. The only aspect worth of note is the treatment of records.

5.2 Normalisation

As already said, the objective of this phase is to remove the top (parallel) structure of the chart model, which reflects the operational semantics of the Stateflow notation (see Figure 5). This results in a model

Step 1 Remove the parallelism between the chart and the *Simulator* processes. Apply the definition of process parallelism.

Step 2 Move the hiding to the main action. Apply the definition of process hiding.

Transform the main action of the resulting process as follows.

Step 3 Isolate the initialisation operation. Apply Law sch-par-distr from right to left.

Step 4 Distribute the hiding. Apply Law hid-distr-seq.

Step 5 Eliminate the hiding over the initialisation. Apply Law hid-ident.

Step 6 Evaluate the parallel recursions. Apply Law rec-par-merge.

Step 7 Refine the initialisation. Apply Law seq-assign-conv.

Figure 14: Refinement strategy: normalisation phase.

whose monolithic, but simple, process structure is adequate as a starting point for us to introduce the control structure of the architectural pattern of implementations in the next phases of the refinement strategy.

The steps of this phase are described in Figure 14. We first eliminate the parallelism between the chart and *Simulator* processes, and then rewrite the main action of the resulting new process to a normal form: an initialisation action, followed by a recursive action that captures each step of execution of the chart.

Following the *Circus* definition of process parallelism, in Step 1, we construct a new process with the same state components of the chart process (since the *Simulator* process is stateless). For its main action, we combine the main actions of the chart and *Simulator* processes in parallel in the same way the processes are combined: with the same synchronisation set. The action parallelism operator (see Table 1) associates with each action a disjoint set of the names of variables in scope that it can modify. In the definition of process parallelism, the name sets that define the partitions of the parallel actions in the main action of the resulting process list the state components of the original parallel processes. In Step 2, we use the definition of process hiding to move the hiding of the set *interface* of channels to the main action of the resulting process.

***Example* 7** *For our example, the main action of the new process resulting from applying Steps 1 and 2 to the process Air_Controller in Figure 6 is as follows.*

$$\bullet \left(\begin{array}{c} (CInitState \ ; \ \mu X \bullet (\mu Y \bullet (AllActions \ ; \ \ Y \Box end\_cycle \longrightarrow \mathbf{Skip})) \ ; \ X) \\ \llbracket \{Air\_B, Air\_DWork, Air\_U\} \mid interface \cup \{\!| end\_cycle |\!\} \mid \{\} \rrbracket \\ (\mu X \bullet Step \ ; \ end\_cycle \longrightarrow X) \end{array} \right) \setminus interface$$

The parallel action $(\mu X \bullet Step \ ; \ end\_cycle \longrightarrow X)$ is the main action of the Simulator process. The action Step (omitted here) encodes the operational semantics of one step of execution of an arbitrary chart. The end of a step is marked by a synchronisation on the channel end_cycle. □

The following Steps 3 to 7 transform the parallelism of recursions in the main action into a single recursion.

***Example* 8** *The result of Steps 3 to 7 for our example is shown below.*

$$\bullet \left(\begin{array}{l} Air\_U := \langle\!| temp == 0 |\!\rangle \ ; \ Air\_B := \langle\!| airflow == 0 |\!\rangle \ ; \ Air\_DWork := \ldots ; \\ \left(\mu X \bullet \left(\begin{array}{c} (\mu Y \bullet AllActions \ ; \ \ Y \Box end\_cycle \longrightarrow \mathbf{Skip}) \\ \llbracket \ldots \rrbracket \\ Step; \ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) ; \ X \right) \setminus interface \end{array} \right)$$

For conciseness, we omit the name and synchronisation sets in the parallelism, which do not change. □

The Steps 3 to 7 are very simple and use existing *Circus* laws of general use. The only interesting novelty is the specific Law rec-par-merge used in Step 6, which transforms a parallelism of recursions into a recursion of parallelisms; it is presented below. The channel *end* is instantiated to *end_cycle* in our strategy, and the actions *A* and *B* are instantiated to *AllActions* and *Step*.

Law [rec-par-merge]

$$(\mu X \bullet (\mu Y \bullet A \; ; \; Y \,\square\, end \longrightarrow \mathbf{Skip}) \; ; \; X) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mu X \bullet B \; ; \; end \longrightarrow X)$$
$$=$$
$$(\mu X \bullet ((\mu Y \bullet A \; ; \; Y \,\square\, end \longrightarrow \mathbf{Skip}) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B \; ; \; end \longrightarrow \mathbf{Skip}) \; ; \; X)$$

provided *$end \in cs$; $end \notin usedC(A, B)$ $wrtV(A) \cap usedV(B) = \varnothing$; and $usedV(A) \cap wrtV(B) = \varnothing$.*

The two parallel recursions proceed in synchrony. This is enforced by the first proviso of the law, which states that the channel *end* is in the synchronisation set *cs* of the parallelism and is only used where explicitly shown. The syntactic function $usedC(A)$ gives the set of channels used in the definition of the action A; also, we use $usedC(A, B)$ as an abbreviation for $usedC(A) \cup usedC(B)$.

In the parallel recursions, a communication over *end* terminates the inner recursion of the first parallel action, and, therefore, one step of its outer recursion, and one step of the recursion in the second parallel action. In the recursion of parallelisms, this synchronous behaviour is captured as a single recursion.

We use $usedV(A)$ to denote the set of variables used (read, but not written) by A, and $wrtV(A)$ the set of variables written by A. The second proviso of the Law rec-par-merge guarantees that each parallel recursion does not use the variables written by the other. This is necessary because, after each step of the recursion of parallelisms, the parallel actions have access to the new values of variables updated in the previous step. This is not the case in the parallel recursions, because the parallelism does not terminate.

In our verification, the application of Law rec-par-merge allows us to take advantage of the fact that, in both the chart and the *Simulator* process, each step of execution of the chart is marked by a synchronisation on *end_cycle*. Moreover, for each of the steps of the simulator, an arbitrary number of executions of *AllActions* may be necessary to provide and update information about the chart components.

5.3 Structuring

In this phase, we introduce the control structure of the implementation: that in Figure 10.

Starting point The steps of this phase are to be applied to the normalised process obtained in the previous phase. The general form of its main action was illustrated in the previous section and is described in Figure 15, where *MdlInitialize* stands for a sequence of assignments that initialise the state. The action *Step* of the *Simulator* process is also reproduced in Figure 15.

Step requests from the chart a sequence *es* of input events, reads a sequence *vs* (of the same size) of boolean values associated to these events, requests the chart to read the input data, executes the chart for each event by calling another action $ExecuteEvents(es, vs)$, and requests the chart to write its outputs. $ExecuteEvents(es, vs)$ is defined by an iterated sequence of calls $ExecuteEvent(es(i), vs(i))$ that executes the chart for each of the events in *es* (and their associated values). The sequence $id(1 \mathrel{..} \# es)$ gives the indices in the set $1 \mathrel{..} \# es$ in order. $ExecuteEvent(e, v)$ models the execution of the chart for the event e; the boolean parameter v indicates whether it occurred or not. A conditional calls the action $ExecuteChart(e)$ if e did occur, that is, $v = \mathbf{True}$. It is $ExecuteChart(e)$, omitted here, that models the execution of the chart for e.

Target In this phase, the state of the normalised process is extended to include components that do not have a corresponding component in the (abstract) chart model, and, therefore, are not introduced in the data-refinement phase. We introduce the component *sfEvent_C*, which records the event being handled, and extend the record in *C_U* to include the component *inputevents*, which keeps the input values associated with each event. In the chart model, these values are read from a channel and not recorded anywhere. We also declare *C_Y*, which records the final value of the output data and events in each step for sharing. We also extend the record in *C_B* to include the components that keep the value of the output events as they are calculated during a step of execution also for sharing. In summary, in this phase the global variables and the components of the record-valued global variables of the program that are used in the treatment of inputs and outputs are introduced and allocated in the right record of the program data model.

In addition, the main action of the normalised process is transformed in this phase to become completely sequential. Its overall structure is formalised in Figure 16; it matches the structure of the program model previously characterised in Figure 10, but is here sketched in *Circus*. As already said, this main action initialises the state components, using an action that follows the pattern *MdlInitialize*, and starts a recursion whose iterations model the implementation of the execution of one step of the chart. In each iteration, (1) the inputs are read using an action that follows the pattern *ReadInputs*; (2) some calculations are carried out, as defined by the sequence of conditionals in the *CalculateOutputs* pattern; (3) the outputs are written using an action whose pattern is *WriteOutputs*; and (4) the end of the step is signalled using *end_cycle*.

$$\left(\begin{array}{l} \mathit{MdlInitialize}; \\ \left(\mu X \bullet \left(\begin{array}{l} \mu Y \bullet \mathit{AllActions}\ ;\ \ Y \Box \mathit{end\_cycle} \longrightarrow \mathbf{Skip} \\ \qquad\qquad [\![ns_1 \mid \mathit{interface} \cup \{\!| \mathit{end\_cycle} |\!\} \mid ns_2]\!] \\ \mathit{Step};\ \ \mathit{end\_cycle} \longrightarrow \mathbf{Skip} \end{array}\right);\ X \right) \setminus \mathit{interface} \end{array}\right)$$

where

$$\mathit{Step} \mathrel{\widehat=} \left(\begin{array}{l} \mathit{events}?\mathit{es} \longrightarrow \mathit{input\_event}?\mathit{vs} : (\#\, \mathit{vs} = \#\, \mathit{es}) \longrightarrow \mathit{read\_inputs} \longrightarrow \\ \mathit{ExecuteEvents}(\mathit{es}, \mathit{vs}); \\ \mathit{write\_outputs} \longrightarrow \mathbf{Skip} \end{array}\right)$$

$$\begin{array}{l} \mathit{ExecuteEvents} \mathrel{\widehat=} \mathit{es} : \mathrm{seq}\, \mathit{EVENT};\ \ \mathit{vs} : \mathrm{seq}\, \mathbb{B} \bullet \\ \quad (\,;\ \ i : \mathit{id}(1 \mathrel{..} \#\, \mathit{es}) \bullet \mathit{ExecuteEvent}(\mathit{es}(i), \mathit{vs}(i))) \end{array}$$

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

Figure 15: Pattern: structuring starting point.

The pattern *ReadInputs* is for an action that reads a sequence of events *vs* of a known size N via the channel *input_event*, and assigns it to the component *inputevents* of C_U. Afterwards, it inputs via channels *i_vi* the value v of each of the input variables *vi* and assigns it to the corresponding component *vi* of C_U.

In *CalculateOutputs*, for each input event represented by the i-th element of the $C\_U.\mathit{inputevents}$ sequence, if it occurred ($C\_U.\mathit{inputevents}(i) = \mathbf{True}$), the value of *sfEvent_C* is updated to the corresponding event (E_i), and the chart is executed. The order of the events is determined by the chart.

In the pattern *ChartExecution* for actions that execute the chart, we have a recursion due to the possibility of local event broadcasts leading to reexecution. In each step, a local variable *c_previousEvent* is used to record the current event if a broadcast takes place (and changes the event in *sfEvent_C*). This variable is needed to allow continuation after reexecution. If the chart is not active ($C\_DWork.is\_active\_C = 0$), a *ChartExecution* action changes the value of *is_active_C* in *C_DWork* to 1. Afterwards, it executes the sequential actions required to enter states of a chart; this is omitted in Figure 16. If the chart is active, the action executes its states. The pattern of execution of states is as described in Figure 9: conditionals for those with sequential decomposition, and sequences for those with a parallel decomposition.

Finally, the *WriteOutputs* pattern describes actions that communicate the output events and variables. Through a channel *o_E* corresponding to an event E, we communicate a boolean, indicating whether E occurred or not. This is determined by a preceding conditional that checks the counter for E in *C_DWork*, and stores the result in the E component of C_B. (Assignments to components of records are not valid in *Circus*. We use them here as an abbreviation for an assignment to the whole record-valued variable of a record that differs only in the value of the indicated component.) If the counter is positive, its value is decremented. The assignment $C\_Y := C\_B$ records the calculated values of the output variables and events in C_B in C_Y, which corresponds to a shared variable of the program. The interleaving of communications realises the sharing by outputting the value of the components of C_Y. Through *o_E* we communicate the value of E as stored in C_Y, and through a channel *o_v* we communicate the value of the variable v stored in C_Y.

Refinement steps Figure 17 shows the steps of the structuring phase; each of them is the application of a separate refinement procedure discussed later in this section. The first step introduces local variables that later become part of the concrete state, the following four steps introduce different elements of the control structure of the implementation architecture, and the last step simplifies the resulting actions. In the sequel we give an overview of these steps and of the refinement procedures.

The order of the steps simplifies the overall structuring. The introduction of the new variables for recording events is performed in Step 1 because they are used in the *CalculateOutput* action introduced in Step 2. Specifically, they are used in the actions that implement the state executions (omitted in Figure 16). Without the global variables, Step 2 would generate local variables throughout the model. This would make the introduction of the global variables harder, since it would require unification of the multiple local variables.

The implementation of local event broadcasts and transition loops in Step 3 requires the identification of

$MdlInitialize\ ;\ (\mu X \bullet ReadInputs\ ;\ CalculateOutputs\ ;\ WriteOutputs\ ;\ end\_cycle \longrightarrow X)$

where *MdlInitialize* is the pattern

$$C\_U := \langle\!| \ldots |\!\rangle\ ;\ C\_B := \langle\!| \ldots |\!\rangle\ ;\ C\_DWork := \langle\!| \ldots |\!\rangle$$

ReadInputs is the pattern

$$\begin{array}{l} input\_events?vs : (\# vs = N) \longrightarrow C\_U := \langle\!| inputevents == vs\ ;\ \ldots |\!\rangle; \\ (i\_v1?v \longrightarrow C\_U.v1 := v \mathbin{|||} i\_v2?v \longrightarrow C\_U.v2 := v \mathbin{|||} \ldots) \end{array}$$

CalculateOutputs is the pattern

$$\begin{array}{l} \left(\begin{array}{l} \mathbf{if}\ C\_U.inputevents(1) = \mathbf{True} \longrightarrow sfEvent\_C := E_1\ ;\ ChartExecution \\ [\!]\ C\_U.inputevents(1) = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right); \\ \ldots; \\ \left(\begin{array}{l} \mathbf{if}\ C\_U.inputevents(n) = \mathbf{True} \longrightarrow sfEvent\_C := E_n\ ;\ ChartExecution \\ [\!]\ C\_U.inputevents(n) = \mathbf{False} \longrightarrow \mathbf{Skip} \\ \mathbf{fi} \end{array}\right) \end{array}$$

ChartExecution is the pattern

$$\left(\mu Y \bullet \left(\begin{array}{l} \mathbf{var}\ c\_previousEvent : \mathbb{N} \bullet \\ \quad \left(\begin{array}{l} \mathbf{if}\ C\_DWork.is\_active\_C = 0 \longrightarrow \\ \quad C\_DWork := \langle\!| is\_active\_C == 1, \ldots |\!\rangle\ ;\ \ldots \\ [\!]\ C\_DWork.is\_active\_C \neq 0 \longrightarrow \ldots \\ \mathbf{fi} \end{array}\right) \end{array}\right)\right)$$

and *WriteOutputs* is the pattern

$$\begin{array}{l} \left(\begin{array}{l} \mathbf{if}\ C\_DWork.counter\_E > 0 \longrightarrow \\ \qquad C\_DWork.counter\_E := C\_DWork.counter\_E - 1)\ ;\ C\_B.E := \mathbf{True} \\ [\!]\ C\_DWork.counter\_E = 0 \longrightarrow E := \mathbf{False} \\ \mathbf{fi} \end{array}\right); \\ \ldots; \\ C\_Y := C\_B; \\ o\_E!(C\_Y.E) \longrightarrow \mathbf{Skip} \mathbin{|||} \ldots \mathbin{|||} o\_v!(C\_Y.v) \longrightarrow \mathbf{Skip} \mathbin{|||} \ldots \end{array}$$

Figure 16: Pattern: structuring target.

Step 1 Introduce input event variables. Apply procedure input-event-var-introduction.

Step 2 Introduce conditionals in *CalculateOutput*. Apply the procedure parallelism-resolution to the body of the outer recursion in the main action.

Step 3 Introduce recursions that implement event broadcast and transition loops. Apply the procedure recursion-introduction to the body of the outmost recursion in the main action.

Step 4 Introduce assignments. Apply procedure assignment-introduction to the body of the outmost recursion in the main action.

Step 5 Introduce update of outputs. Apply procedure update-output to the second action in the sequence that defines the body of the outmost recursion in the main action.

Step 6 Simplify. Apply the procedure simplification to the recursion in the main action.

Figure 17: Refinement strategy: structuring phase

actions that arise in the refinement calculations in Step 2 and are written in a particular pattern. Namely, these are parallel actions whose structure is similar to that of the starting point of this phase (see Figure 15).

Step 4 refines schema operations to assignments, and Step 5 introduces the shared variable C_Y that is used to communicate outputs. These are the schemas and outputs whose specifications arise as a result of Steps 2 and 3. Steps 4 and 5 can, therefore, occur in any order, but after Step 3, and also before Step 6 because the actions introduced by them may require simplification.

Step 1 The procedure input-event-var-introduction is very simple. It uses standard laws for introduction of variables to declare *inputevents* and *sfEvent_C* as local variables in the parallel action *Step*. Additionally, it uses simple laws of *Circus* to extend their scope, and then promote them to state components. As a result of these steps, *inputevents* and *sfEvent_C* are added to the name set ns_2 of the parallelism (see Figure 15). The details of this procedure input-event-var-introduction are omitted here.

Step 2 In general terms, the procedure parallelism-resolution systematically applies, to the body of the outer recursion in the main action, step laws to resolve the parallelism between the recursion offering *AllActions*, and *Step*. As it does so, it unravels the structure embedded in *AllActions* and *Step*: conditionals that check the status of chart states, the occurrence of events, and guards of transitions. The result already has a structure similar to that of the body of the recursion in Figure 16, but the actions that model a state execution may still retain some parallelism. This arises if the chart has local event broadcasts or transition loops (involving just junctions). These parallelisms are the target of Step 3.

When compared to the pattern of our target in Figure 16, after Step 2 (1) the recursions have not been introduced; (2) the conditionals in *CaculateOutputs* are nested (rather than in sequence); (3) there are spurious internal communications arising from the removal of the parallelism in the previous phase; (4) the actions that model the execution of a state may contain parallelisms; and (5) the outputs in *WriteOutputs* are communicated in the conditionals, rather than at the end, but the conditionals are interleaved rather than in sequence. All these issues are tackled in the next steps.

The details of the procedure parallelism-resolution are presented in Section 5.3.1. It is rather extensive, as it has to consider the several forms of parallelism that can arise from the application of the step laws.

Step 3 This deals with local event broadcasts and transition loops, which are modelled by the parallelisms left in the previous step, if any. The procedure recursion-introduction defines new recursive actions and proves their equivalence to the existing parallel actions. The result of this step transforms the main action so that the variable blocks that model the chart execution include recursive actions whose bodies are similar to the variable blocks themselves. We present the procedure recursion-introduction in Section 5.3.2.

Step 4 This step refines the main action so that all schema operations are refined to assignments. The procedure assignment-introduction distinguishes two types of schema operations: (1) those that activate or deactivate a state, which are specified using one of the schemas *Activate* or *Deactivate*, originally defined in the chart model; and (2) those introduced in the data-refinement phase plus the initialisation schema. The assignments to be introduced are determined in assignment-introduction according to the kind of schema operation (*Activate* or *Deactivate*), to the type of the state S (parallel or sequential), and to whether the parent of a sequential state being activated has a history junction or not. The predicates in the schemas of the second group are conjunctions of equalities, and so we convert them into assignments directly. The refinement laws used are simple Z laws for refining schemas to assignments.

Step 5 As already mentioned, as a result of Step 2, output events and data are communicated in interleaving. The objective of this step is to gather together the construction of the values to be output, before making all outputs available (still in interleaving). The procedure update-output is described in Section 5.3.3.

Step 6 To conclude the structuring phase, we have a final simplification step. The operational semantics of Stateflow charts, as specified in the *Simulator* process of the chart models, considers all possible paths of execution that might arise in the execution of an arbitrary chart. Typically, the semantics of a particular chart, as defined by the chart process, does not involve all these paths. We, therefore, in carrying out the Steps 2 and 4 above, which basically evaluate the semantics of the chart in a systematic way, may introduce unnecessary assignments, and conditionals. They are eliminated in this step.

Unnecessary assignments arise, for example, when a transition loop leads to a path that exits and subsequently enters the same state. In this case, during this step we remove the sequence of two assignments that record the state as inactive and then active. Unnecessary conditionals arise, for example, when absence of local event broadcast makes it unnecessary to check early return logic conditions. In this case, during this step we remove conditionals whose conditions can be shown to be always true or false.

In addition, we remove unnecessary variables (that arise from the action that models early return logic). We also eliminate internal channels originally used for communication between the chart and *Simulator* processes. The elimination of the parallelism between them makes these communications unnecessary.

We also simplify the control structure in this step. Nested conditionals that execute sequential states and check a transition's trigger and condition are flattened into a single conditional with multiple branches, unfolded recursions are folded again, and assumptions and redundant **Skip** actions are eliminated.

Finally, the process is data refined to include *inputevents* in the binding of type *ExternalInputs*.

The details of this extensive but simple procedure are omitted here.

5.3.1 Procedure parallelism-resolution

This procedure has three parameters: a set $loopT$ of transitions that start loops, a set $visitedT$ of such transitions whose implementations have already been refined, and a parallel action A to be refined. In the Step 2 of the structuring phase, parallelism-resolution is applied with the following arguments: the set of all transitions of the chart that start a loop (which can be extracted from the chart itself), the empty set, and the body of the outer recursion in the main action (see Figure 15).

This is a recursive procedure defined in terms of the syntactic structure of A. In the sequel, we describe how each form of parallelism that the argument action A may take is refined.

A. Parallel composition unit (base case) The rather trivial first case is the base case of our procedure: a parallelism $\mathbf{Skip} \llbracket ns_1 \mid cs \mid ns_2 \rrbracket \mathbf{Skip}$. It requires the application of Law par-unit to obtain **Skip**.

B. Prefixing over channel in the synchronisation set on the right-hand side This case covers the situation where the right-hand side (originally, the process *Simulator*) is requesting the left-hand side (originally the chart process) to execute an action. This is characterised as follows, where c is in cs.

$$(\mu Y \bullet AllActions \ ; \ \ Y \,\Box\, end\_cycle \longrightarrow \mathbf{Skip}) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket\, c \longrightarrow A$$

In our refinement strategy, for every model, this is the first case applicable, by definition of *Step*.

The refinement has to evaluate the communication. Since the synchronisation is offered by *AllActions*, we need to unfold the recursion. The precise steps are omitted here. The result takes the following form, where $c \longrightarrow B$ is an action included in the external choice of *AllActions*.

$$c \longrightarrow B \ ; \ (\mu Y \bullet AllActions \ ; \ \ Y \,\Box\, end\_cycle \longrightarrow \mathbf{Skip}) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket\, c \longrightarrow A$$

We recursively apply parallelism-resolution to the remaining parallel action. The parameters of the recursive call are $loopT$ and $visitedT$ unchanged, and the whole action above.

***Example* 9** *For our example, this case applies to the following main action.*

$$\left(\begin{array}{l} (\mu Y \bullet AllActions \ ; \ \ Y \,\Box\, end\_cycle \longrightarrow \mathbf{Skip}) \\ \qquad \llbracket \{Air\_B, Air\_DWork, Air\_U\} \mid interface \cup \{\!| \, end\_cycle \, |\!\} \mid \{inputevents, sfEvent\_Air\} \rrbracket \\ (events?es \longrightarrow input\_event?vs : (\# vs = \# es) \longrightarrow inputevents := vs \ ; \ readinputs \longrightarrow \ldots) \end{array}\right)$$

When this case is applied to the above action, we obtain the action below (before recursing in the application of parallelism-resolution).

$$\left(\begin{array}{l} events!\langle e\_SWITCH, e\_CLOCK\rangle \longrightarrow \mathbf{Skip} \ ; \ (\mu Y \bullet AllActions \ ; \ \ Y \,\Box\, end\_cycle \longrightarrow \mathbf{Skip}) \\ \qquad \llbracket \{Air\_B, Air\_DWork, Air\_U\} \mid interface \cup \{\!| \, end\_cycle \, |\!\} \mid \{inputevents, sfEvent\_Air\} \rrbracket \\ (events?es \longrightarrow input\_event?vs : (\# vs = \# es) \longrightarrow inputevents := vs \ ; \ readinputs \longrightarrow \ldots) \end{array}\right)$$

Since the first communication of Step is an input events?es, the matching output through events in AllActions is revealed. The assignment to inputevents now on the right-hand side of the parallelism was introduced as a result of the Step 1 of the structuring phase described previously. □

C. Synchronisation This case occurs as a result of the refinement in the previous case; it evaluates the parallel prefixed actions that are unfolded using standard step laws.

Step 1 Apply Law seq-distr-cond.

Step 2 **Distribute parallelism.** Apply Law par-distr-cond.

Step 3 If g refers to a state component, recursively apply parallelism-resolution to each branch. Otherwise, apply Law cond-elim, before recursing.

Figure 18: parallelism-resolution(**H**): steps for conditional followed by sequence, on either side.

***Example* 10** *Proceeding with our example, we obtain the action below before recursing.*

$$events.\langle e\_SWITCH, e\_CLOCK\rangle \longrightarrow \left(\begin{array}{l} \mathbf{Skip} \; ; \; (\mu Y \bullet AllActions \; ; \; Y \,\Box\, end\_cycle \longrightarrow \mathbf{Skip}) \\ \qquad [\![\{Air\_B, Air\_DWork, Air\_U\} \mid interface \cup \{\!|\, end\_cycle \,|\!\} \mid \{inputevents, sfEvent\_Air\}]\!] \\ (input\_event?vs : (\# vs = 2) \longrightarrow inputevents := vs \; ; \; readinputs \longrightarrow \ldots) \end{array}\right)$$

The communication on events is evaluated and extracted from the parallelism. Additionally, in the input side of the parallelism, the value of es is determined by the output: # es, for instance, can be resolved to 2. □

D-F. Leading Skip, prefixing over channel not in the synchronisation set, assignment or schema on either side In these cases we have a simple action on either side of the parallelism, and we apply a simple law that removes it or extracts it from the parallelism. For a leading **Skip** on the left parallel action, for instance, $(\mathbf{Skip} \; ; \; A) \; [\![\; ns_1 \mid cs \mid ns_2 \;]\!] \; B$, we use a unit law of sequence to remove the **Skip**. In a case like $(c \longrightarrow A) \; [\![\; ns_1 \mid cs \mid ns_2 \;]\!] \; B$, of a prefixing over a channel not in the synchronisation set, this involves a communication c with the environment. A simple step law par-prefix-step extracts the communication from the parallelism. Its provisos always hold, since the structure of the process we are refining is quite restricted. For instance, for all communications that are relevant to this case, the first possible communication in the other parallel action is over a channel in the synchronisation set.

***Example* 11** *Proceeding with our example, we obtain the action below.*

$$events.\langle e\_SWITCH, e\_CLOCK\rangle \longrightarrow input\_event?vs : (\# vs = 2) \longrightarrow inputevents := vs; \\ \left(\begin{array}{l} \mu Y \bullet AllActions \; ; \; Y \,\Box\, end\_cycle \longrightarrow \mathbf{Skip} \\ \qquad [\![\{Air\_B, Air\_DWork, Air\_U\} \mid interface \cup \{\!|\, end\_cycle \,|\!\} \mid \{inputevents, sfEvent\_Air\}]\!] \\ (read\_inputs \longrightarrow \ldots) \end{array}\right)$$

The communication with the environment on input_events and its associated assignments are extracted from the parallelism. Moreover, further recursive applications of previous cases extract the internal communication read_inputs, and the external communication in AllActions over the channel i_temp corresponding to the input variable `temp`. *In general, however, there may be several input variables and, consequently, several input communications in interleaving. This is the object of the next case.* □

G. Leading interleaving on the left-hand side This case covers actions characterised as follows.

$$((A_1 \; [\![\!| \; ns_1 \mid ns_2 \; |\!]\!] \; A_2)) \; ; \; B) \; [\![\; ns_x \mid cs \mid ns_y \;]\!] \; C$$

We have on the left an interleaving of actions A_1 and A_2, followed by an action B, all in parallel with another action C. As in previous cases, we apply a step law to extract the interleaving from the parallelism. As before, the provisos of the step law always hold by the construction of the model.

H. Conditional followed by sequence, on either side As already mentioned, the structure of *AllActions* and *Step* involves a number of conditionals. This case covers their treatment, considering actions of the form $((\mathbf{if} \; g \longrightarrow A_1 \; [\!] \; \neg \; g \longrightarrow A_2 \; \mathbf{fi}) \; ; \; B)$, and the similar cases where the conditional is on the right-hand side of the parallelism. All conditionals have mutually exclusive guards g and $\neg \; g$.

In general, the conditionals can involve checks that depend on the current state of the chart, for instance, the verification of the guard of a transition, or checks that are based solely on the structure of the chart. The two types of conditionals can be distinguished by their guards. If a guard refers to state components, it is of the first type. If not, it is of the second type, and can be eliminated using the static information that defines the structure of the chart. This is achieved in this step.

For illustration, the precise refinement steps to be carried out in this case are shown in Figure 18. We first distribute the sequential composition over the conditional, and then the parallelism using the fact that the guards of the conditional are mutually exclusive. Next, if the guard refers to state components, we recursively apply parallelism-resolution to the actions in each branch. If it does not, we remove the conditional using the definitions of constants like s_Off3, $states$, and $transitions$ shown in Figure 6. Finally, we recursively apply parallelism-resolution to the remaining action with the remaining parameters unchanged.

***Example* 12** *Proceeding with our example, at this stage, we have the action below, where the conditionals now shown on the right parallel action are originally part of ExecuteEvent.*

$$
\begin{array}{l}
events.\langle e\_SWITCH, e\_CLOCK\rangle \longrightarrow input\_event?vs : (\# vs = 2) \longrightarrow inputevents := vs \; ; \; readinputs \longrightarrow \\
\left(\begin{array}{l}
\mu Y \bullet AllActions \; ; \; Y \;\Box\; end\_cycle \longrightarrow \mathbf{Skip} \\
\quad [\![\{Air\_B, Air\_DWork, Air\_U\} \mid interface \cup \{\!| end\_cycle |\!\} \mid \{inputevents, sfEvent\_Air\}]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{if}\ inputevents(1) = \mathbf{True} \longrightarrow \\
\qquad sfEvent\_Air := e\_SWITCH \; ; \; ExecuteChart(sfEvent\_Air) \\
[\!]\ inputevents(1) = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
\left(\begin{array}{l}
\mathbf{if}\ inputevents(2) = \mathbf{True} \longrightarrow \\
\qquad sfEvent\_Air := e\_CLOCK \; ; \; ExecuteChart(sfEvent\_Air) \\
[\!]\ inputevents(2) = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
write\_outputs \longrightarrow end\_cycle \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right)
\end{array}
$$

The guards refer to inputevents and so the conditional is not eliminated. If we proceed with the application of the steps in Figure 18, we obtain the following action.

$$
\begin{array}{l}
events.\langle e\_SWITCH, e\_CLOCK\rangle \longrightarrow input\_event?vs : (\# vs = 2) \longrightarrow inputevents := vs \; ; \; readinputs \longrightarrow \\
\left(\begin{array}{l}
\mathbf{if}\ inputevents(1) = \mathbf{True} \longrightarrow \\
\qquad \left(\begin{array}{l}
\mu Y \bullet AllActions \; ; \; Y \;\Box\; end\_cycle \longrightarrow \mathbf{Skip} \\
\quad [\![\ldots]\!] \\
\left(\begin{array}{l}
sfEvent\_Air := e\_SWITCH \; ; \; ExecuteChart(sfEvent\_Air); \\
\left(\begin{array}{l}
\mathbf{if}\ inputevents(2) = \mathbf{True} \longrightarrow \\
\qquad sfEvent\_Air := e\_CLOCK \; ; \; ExecuteChart(sfEvent\_Air) \\
[\!]\ inputevents(2) = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
write\_outputs \longrightarrow end\_cycle \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right) \\
[\!]\ inputevents(1) = \mathbf{False} \longrightarrow \\
\qquad \left(\begin{array}{l}
\mu Y \bullet AllActions \; ; \; Y \;\Box\; end\_cycle \longrightarrow \mathbf{Skip} \\
\quad [\![\ldots]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{if}\ inputevents(2) = \mathbf{True} \longrightarrow \\
\qquad sfEvent\_Air := e\_CLOCK \; ; \; ExecuteChart(sfEvent\_Air) \\
[\!]\ inputevents(2) = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\mathbf{fi}
\end{array}\right); \\
write\_outputs \longrightarrow end\_cycle \longrightarrow \mathbf{Skip}
\end{array}\right)
\end{array}\right) \\
\mathbf{fi}
\end{array}\right)
\end{array}
$$

At this stage, recursive calls to parallelism-resolution *lead to a number of applications of the previous cases, until we reach a parallelism whose right-hand parallel action is a call to ExecuteChart.*

It is worth mentioning that the innermost conditional in the first branch of the outermost conditional accounts for the possibility of executing the chart with the event e_CLOCK after it has been executed under the event e_SWITCH in the previous action.

□

I. Call action on either side Whenever a call action is the leading action in one of the sides of the parallelism, and none of the other cases apply, we expand it using a procedure copy. It replaces a call to an action with its definition, with the appropriate parameters substituted using standard laws.

If the action that is called is the *Simulator* action *ExecuteTransition*, however, the refinement to be carried out is different. *ExecuteTransition* models the execution of a sequence of transitions, so, when executing a transition loop through a call to *ExecuteTransition*, the uncontrolled application of the procedure copy leads to nontermination. To avoid that, we use the parameters *loopT* and *visitedT* of parallelism-resolution.

In a call to *ExecuteTransition*, the first argument t is the identifier of the first transition to be executed. If it starts a loop ($t \in loopT$), there are two possibilities: a call with t as argument has already been expanded

Step 1 Use state component $\mathit{sfEvent\_C}$ Apply Law use-loc-var to $\mathit{TreatLocalEvent}(e, s)$ to introduce a local variable $\mathit{previousEvent}$ of type $\mathbb{N}$, with initial value $\mathit{sfEvent\_C}$, whose temporary value becomes e.

Step 2 Extend and distribute. Apply Laws var-seq-ext-right and var-par-ext-right, and apply par-seq-step twice.

Step 3 Expand action call. Apply the copy procedure to $\mathit{TreatLocalEvent}(e, s)$.

Step 4 Simplify the conditional. Try to apply Law cond-elim to eliminate each of the branches of the conditional that defines $\mathit{TreatLocalEvent}$.

Step 5 Distribute the parallelism. Apply Law par-seq-dist.

Step 6 Recurse. Apply the procedure parallelism-resolution to the second parallelism.

Figure 19: parallelism-resolution($\mathbf{L}$): steps for local event broadcast.

by a previous application of this case ($t \in \mathit{visitedT}$), or not. If not, it is expanded as usual and parallelism-resolution is applied recursively with parameters loopT and $\mathit{visitedT} \cup \{t\}$, that is, t is marked as "visited". If t is in $\mathit{visitedT}$, we leave the parallelism unresolved: it is refined in the Step 3 of the structuring phase.

We observe that this treatment of $\mathit{ExecuteTransition}$ is not an artefact of our example. This is a *Simulator* action, and the *Simulator* process is part of all chart models and is the same in all models. Its structure, just like that of the chart itself, is used to guide our refinement strategy, which is general.

J. Explicit recursion on the right-hand side In this case, we unfold the recursions. Due to the structure of the chart model, we know that this does not lead to nontermination of our refinement strategy.

Recursions come from two sources. The first are the *Simulator* actions that offer a choice between treating a local event and recursing, or signalling the end of the chart execution. The second are calls to the *Simulator* action $\mathit{transitionActionCheck}$, which checks the status of substates as part of the check of the early return logic condition. Since any chart action involves only a finite number of local event broadcasts, and every state has a finite number of substates, there can only be a finite number of applications of this case.

K. Leading prefixing over channel in the synchronisation set on the left-hand side A *Circus* action that models a local event broadcast uses internal channels to control the *Simulator*. In this case, we consider a parallelism where the left-hand action is such a communication. In the right-hand action, at these points, *Step* always offers a(n external) choice that accepts the communication. The general pattern is $c \longrightarrow A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c \longrightarrow B_1 \Box d \longrightarrow B_2) ; \ C$, where both c and d are in cs.

In this step, refinement resolves the external choice using simple and standard laws. We distribute the sequence and the parallelism over the external choice and eliminate the deadlocked parallel actions. The provisos of the laws applied follow from the fact that c and d are both in the synchronisation set, and are different. The result is the action $c \longrightarrow A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket c \longrightarrow (B_1 ; \ C)$.

L. Local event broadcast The pattern for this case is as shown below.

$$((\mu Z \bullet \mathit{AllActions} ; \ Z \Box \mathit{end\_local\_execution} \longrightarrow \mathbf{Skip}) ; \ A) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mathit{TreatLocalEvent}(e, s) ; \ B)$$

The *Simulator* action $\mathit{TreatLocalEvent}$ is used in all actions that handle local event broadcast. It decides whether to reexecute the whole chart or a specific state, depending on the kind of broadcast. Similarly to calls to $\mathit{ExecuteTransition}$ singled out in the case **I**, calls to $\mathit{TreatLocalEvent}$ cannot be expanded indiscriminately. Moreover, calls to $\mathit{TreatLocalEvent}$ occur in parallel with the actions of the chart process that model local event broadcasts: a recursion that provides the services of $\mathit{AllActions}$; termination is by synchronisation on $\mathit{end\_local\_execution}$.

Figure 19 presents the steps to be carried out. The objectives of the refinement are twofold. First, we resolve the decision embedded in $\mathit{TreatLocalEvent}$, which is based on the structure of the chart: namely, the target of the broadcast. Second, we split the parallelism to isolate the encoding of the execution of the broadcast from that of the continuation of the execution of the chart represented by A and B in our pattern.

The resulting action is below; it contains a sequence of parallelisms: the first corresponds to the execution of the local event broadcast and is further refined in the Step 3 of the structuring phase, and the second is the target of a recursive application of parallelism-resolution.

$$\left(\begin{array}{l}\mathbf{var}\, previousEvent : \mathbb{N} \bullet previousEvent := sfEvent\_C \;;\; sfEvent\_C := e;\\ \quad\left(\begin{array}{l}\left(\begin{array}{l}\mu X \bullet AllActions \;;\; X \,\Box\, end\_local\_execution \longrightarrow \mathbf{Skip}\\ \qquad [\![ns_1 \mid cs \mid ns_2]\!]\\ ExecuteChart(sfEvent\_C) \;;\; end\_local\_execution \longrightarrow \mathbf{Skip}\end{array}\right);\\ \\ (A \;[\![\; ns_1 \mid cs \mid ns_2 \;]\!]\; sfEvent\_C := previousEvent \;;\; B)\end{array}\right)\end{array}\right)$$

In splitting the parallelism, a local variable *previousEvent* is used to store the current event in $sfEvent\_C$, before it is updated to the broadcast event, so that later, the value of $sfEvent\_C$ can be restored.
Step 1 of the refinement procedure for this case applies the novel, but simple, Law use-loc-var below.

Law [use-loc-var]

$$A(e) = (\mathbf{var}\, x : T \bullet x := v;\; v := e;\; A(v);\; v := x)$$

where *e, of type T, is a value argument of A.*
provided *$v \notin FV(A)$ and x is fresh.*

This law applies to an action call $A(e)$, where e is a value argument. (A more general version explicitly allows for more arguments, as is the case of the call $TreatLocalEvent(e, s)$, but for simplicity, we only indicate e above). It declares a fresh local variable x (of a type T), which is initialised with the value of a variable v not used in A, which is then updated to hold the value of e temporarily for the execution of A.

In the refinement carried out in this case, we use Law use-loc-var to substitute $TreatLocalEvent(e, s)$ with the declaration of a local variable *previousEvent* of type $\mathbb{N}$, record the value of the state component $sfEvent\_C$ in *previousEvent*, store the argument e of the call to *TreatLocalEvent* in $sfEvent\_C$, call *TreatLocalEvent* with the e substituted with $sfEvent\_C$, and restore the value of $sfEvent\_C$. In the Step 2, we extend the scope of *previousEvent* and extract the assignments from the parallelism.

In Step 3, we expand $TreatLocalEvent(sfEvent\_C, s)$; this results in a conditional whose guards do not refer to state components. It establishes whether the destination of the broadcast is a state or the chart, in order to call the appropriate action. Step 4 simplifies this conditional to one of its branches by attempting to apply Law cond-elim to eliminate the first branch, and then the second, if unsuccessful. One of the applications necessarily succeeds, since the constants of the model that record the structure of the chart can be used to determine that one of the guards of the conditional is **True** and the other is **False**.

***Example* 13** *For the sake of example, we assume that the conditional simplifies to the first branch, and we obtain the result below.*

$$\left(\begin{array}{l}\mathbf{var}\, previousEvent : \mathbb{N} \bullet previousEvent := sfEvent\_C \;;\; sfEvent\_C := e;\\ \left(\begin{array}{l}(\mu Z \bullet AllActions \;;\; Z \,\Box\, end\_local\_execution \longrightarrow \mathbf{Skip}) \;;\; A\\ \qquad [\![ns_1 \mid cs \mid ns_2]\!]\\ ExecuteChart(sfEvent\_C) \;;\; end\_local\_execution \longrightarrow \mathbf{Skip} \;;\; sfEvent\_C := previousEvent \;;\; B\end{array}\right)\end{array}\right)$$

□

Finally, Step 5 applies the novel Law par-seq-dist below to separate the parallelism into a sequence of two parallel compositions. It considers a parallelism of sequences (and relates it to a sequence of parallelisms). The action of the left-hand side of the parallelism is the second component of a pair (M, N) of actions defined by mutual recursion. The first component M may offer a communication over a channel l and start the second component N, and N offers a choice between calling M and recursing on N or synchronising on a channel el and terminating. The first action of the sequence on the right-hand side of the parallelism is a simple recursion that communicates on l, conditionally recurses, and synchronises on el afterwards. The second action of the right-hand side starts with a synchronisation on el.

Law [par-seq-dist]

$$\begin{array}{l}N \;;\; B_1 \;[\![\; ns_1 \mid cs \mid ns_2 \;]\!]\; (\mu X \bullet A_2[l \longrightarrow (\mathbf{if}\; b \longrightarrow X \;[\!]\; \neg\, b \longrightarrow C\,\mathbf{fi}) \;;\; el \longrightarrow \mathbf{Skip}]) \;;\; el \longrightarrow B_2\\ =\\ \left(\begin{array}{l}(N \;[\![\; ns_1 \mid cs \mid ns_2 \;]\!]\; (\mu X \bullet A_2[l \longrightarrow (\mathbf{if}\; b \longrightarrow X \;[\!]\; \neg\, b \longrightarrow C\,\mathbf{fi}) \;;\; el \longrightarrow \mathbf{Skip}]) \;;\; el \longrightarrow \mathbf{Skip});\\ (B_1 \;[\![\; ns_1 \mid cs \mid ns_2 \;]\!]\; B_2)\end{array}\right)\end{array}$$

where $(M, N) \mathrel{\widehat=} \mu X, Y \bullet (F[l \longrightarrow Y], (X \;;\; Y) \,\Box\, el \longrightarrow \mathbf{Skip})$
provided

- $\{\!|\, l, el \,|\!\} \subseteq cs$; $\{\!|\, l, el \,|\!\} \cap usedC(F, A_2) = \varnothing$; $initials(M) \in cs$; *and*
- $usedV(B_1) \cap ns_2 = usedV(B_2) \cap ns_1 = \varnothing$.

While there are remaining parallelisms p in the main action

Step 1 Calculate a possibly recursive action. Applying the procedure parallelism-resolution to p.

Step 2 Refine parallel action.

- **(a)** If the calculated action is $\mu X \bullet F(X)$, apply Law unique-fixed-point to p and F.
- **(b)** Else, substitute the calculated action for the parallelism.

Figure 20: Refinement strategy:structuring phase - recursion-introduction.

In our application of this law, N is the recursion offering *AllActions* with el as *end_local_execution*. The action M is *AllActions* itself, which accepts communications on a channel *local_event* and then starts a new recursion identical to the one that called it. So, l is the channel *local_event*. The recursion in the right parallel action is instantiated to the action *ExecuteChart*: it treats local event broadcasts, and that involves recursive calls. The action that carries out the local event execution is a prefixing on *local_event*, followed by a conditional that checks the type of broadcast, followed by a synchronisation on *end_local_execution*. All recursive calls to *ExecuteChart* follow this pattern, as required in Law par-seq-dist.

In the parallelism of sequences in the above law, l and el are in the synchronisation set and are only used in the recursions as explicitly shown, as stated in the first two provisos. Each communication on l, therefore, triggers a recursive call Y to N on the left-hand side, and, on the right-hand side, a recursive call or a call to C. When either of those calls on the right-hand side terminates, we have a synchronisation on el. Since N is offering to synchronise on el or reexecute X (that is, M, which is waiting for a communication on a channel in the synchronisation set as stated by the third proviso) both sides synchronise on el and the most recent recursive call to N terminates. When the recursion on the right-hand side terminates, a second synchronisation on el prompts the mutual recursion to terminate. Since there is no possibility of B_2 communicating with the mutual recursion, or B_1 communicating with the simple recursion, because both recursions terminate synchronously, B_2 does not use variables written by the mutual recursion, and B_1 does not use variables written by the simple recursion (last proviso), we can separate the parallel action.

Step 6 applies the procedure parallelism-resolution to the second parallel action.

M. Leading local variable declaration on either side The actions to which this case applies follow a pattern where either side of a parallel action is a local variable block declaring a single variable of type boolean; for example $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mathbf{var}\ b : \mathbb{B} \bullet B)$. We apply standard laws to give b a fresh name and expand its scope out of the parallelism, before recursing. The name set associated with the parallel action declaring the variable is extended to include the new name of the variable.

Since parallelism-resolution is recursive, termination is an issue. A detailed argument based on the structure of the parallel actions found in our chart models is provided in [Miy12].

5.3.2 Procedure recursion-introduction

This procedure is used in Step 3 of the structuring phase. It receives the same parameters *loopT* and *visitedT* as parallelism-resolution, which it uses to call that procedure. It acts on the body of the outermost recursion in the main action to transform the remaining parallel actions into recursions.

For each parallel action p of the general form $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B$, in Step 1 we calculate, a (possibly recursive) sequential action. For that, we apply the procedure parallelism-resolution to p. The result may be of the form $F[A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B]$ or F, that is, it may or may not contain the same parallelism. If it does, the calculated action is a recursion $\mu X \bullet F[X]$ whose body is the result obtained with parallelism-resolution, where all parallelisms $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B$ are replaced with a recursive call X. If it does not contain the parallelism, the result of this calculation is the action F itself.

If the calculated action is recursive, we refine the parallel action to that recursion using a standard fixed-point law in Step 2(a). Otherwise, we simply replace the parallelism with the calculated action in Step 2(b), since the procedure used to calculate it establishes equivalence.

As mentioned previously, we use the procedure recursion-introduction with the assumption that there are no mutually recursive actions to be introduced. This means that the calculated actions do not contain themselves any further parallelisms, which would generate a recursion.

$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\textbf{if } C\_DWork.counter\_E > 0 \longrightarrow \\ \quad C\_DWork := \langle\!|counter\_E == (C\_DWork.counter\_E - 1), \ldots |\!\rangle \; ; \; o\_E!(\textbf{True}) \longrightarrow \textbf{Skip} \\ [\!] \, C\_DWork.counter\_E = 0 \longrightarrow o\_E!(\textbf{False}) \longrightarrow \textbf{Skip} \\ \textbf{fi}\end{array}\right) \\ \qquad |\![\{C\_DWork.counter\_E\} \mid \ldots]\!| \\ \ldots \end{array}\right) \\ \qquad |\![\ldots \mid \ldots]\!| \\ (o\_v!(C\_B.v) \longrightarrow \textbf{Skip} \; ||| \; \ldots)\end{array}\right)$$

Figure 21: update-output: interleavings to be refined.

5.3.3 Procedure update-output

As a parameter, update-output takes the sequence *output_events* of output events in the order in which they are defined in the chart. This procedure introduces a new state component, namely, C_Y, which, as previously explained, records the values of the output variables and events to be communicated at the end of the step. The procedure update-output also expands the definition of the schema type of C_B, to include boolean components that record whether the output events have occurred or not. The concern in update-output is mostly with interleavings of the general form shown in Figure 21; each is an interleaving of conditionals communicating events, and communications of output variables. These interleavings occur repeatedly at the end of the innermost branches of the nested conditionals in the main action.

The procedure update-output first extracts the interleavings to the end of the action. This is possible because the interleavings, followed by a synchronisation on *end_cycle*, are the final action in all innermost branches of all conditionals. (The procedure parallelism-resolution pushes them inside the conditionals, and recursion-introduction may introduce tail recursions that terminate with them.)

Next, for each event E in the sequence of output events, we identify the conditional that communicates the output event: that with a communication through o_E. We introduce a local variable E of type $\mathbb{B}$, assign to it the value v that is being communicated through o_E, and communicate E instead. Afterwards, we extend the scope of the local variable over the main action, and promote it to a state component.

***Example* 14** *The result of these steps on the conditional in Figure 21 is shown below.*

$$\left(\begin{array}{l}\textbf{if } C\_DWork.counter\_E > 0 \longrightarrow \\ \quad C\_DWork := \langle\!|counter\_E == (C\_DWork.counter\_E - 1), \ldots |\!\rangle \; ; \; E := \textbf{True} \; ; \; o\_E!(E) \longrightarrow \textbf{Skip} \\ [\!] \, C\_DWork.counter\_E = 0 \longrightarrow E := \textbf{False} \; ; \; o\_E!(E) \longrightarrow \textbf{Skip} \\ \textbf{fi}\end{array}\right)$$

□

We now extract the prefixings $o\_E!(E) \longrightarrow \textbf{Skip}$ that communicate the local variables from the conditional, and then extract the conditional from the interleaving.

***Example* 15** *Following on from our example, we obtain the action below.*

$$\begin{array}{l}\left(\begin{array}{l}\textbf{if } C\_DWork.counter\_E > 0 \longrightarrow \\ \quad C\_DWork := \langle\!|counter\_E == (C\_DWork.counter\_E - 1), \ldots |\!\rangle \; ; \; E := \textbf{True} \\ [\!] \; C\_DWork.counter\_E = 0 \longrightarrow E := \textbf{False} \\ \textbf{fi}\end{array}\right) ; \\ \left(\left(\begin{array}{l} o\_E!(E) \longrightarrow \textbf{Skip} \\ \qquad |\![\{\} \mid \ldots]\!| \\ \ldots \end{array}\right) |\![\ldots \mid \ldots]\!| \; (o\_v!(C\_B.v) \longrightarrow \textbf{Skip} \; ||| \; \ldots)\right)\end{array}$$

□

After all output events have been considered, a simple data refinement includes the newly added state components in the component C_B, and standard variable introduction laws are used to declare the variable C_Y and initialise it with C_B. (This is now possible because the bindings of *ExternalOutputs_C* and *BlockIO* have the same components). Next, we substitute C_Y for C_B in the interleaved communications, and distribute the local variable over the main action. Finally, we promote it to a state component. The result, as probably expected, is an action following the pattern *WriteOutputs* shown in Figure 16.

5.4 Action introduction

At the structuring phase, the main action of the refined process should be the same as that of the model of the implementation, except that it is decomposed into a number of local actions. In this phase, we refine our process to match exactly the process that models the implementation. First, its local actions are introduced in the process being refined. Next, we exhaustively apply the copy-rule from right to left to replace occurrences of the definitions of the actions with a call to the appropriate action.

***Example* 16** *The main action of the process resulting from the application of this phase to our example is the action ExecuteChart, which is specified as follows.*

$$MdlInitialize \ ; \ \mu X \bullet read\_inputs \ ; \ Air\_output \ ; \ write\_outputs \ ; \ end\_cycle \longrightarrow X$$

ExecuteChart calls the action MdlInitialize to initialise the state, and recursively reads the inputs using the action read_inputs, executes the chart using Air_output, writes the outputs using write_outputs, and signals the end of the step by synchronising on end_cycle.

This is exactly the action of the implementation model, and completes the verification of our example. □

By starting from the (abstract) chart model and deriving the implementation model purely as the result of applications of algebraic refinement laws (as dictated by our refinement strategy), we prove that (the abstract model of) the chart is refined by the (model of the) implementation.

5.5 Automation

Two industrial case studies were used to validate the semantics of Stateflow charts, but they are too large to be manually verified. In order to support semi-automatic verification, we must first formalise the strategy in a tactic language for refinement, such as ArcAngel [OC08], and then use a refinement tool. A version of ArcAngel for *Circus* is already available [OZC11], and its recent mechanisation [ZOC12] in a theorem prover that supports the use of *Circus* is a robust basis for the future automation of our refinement strategy.

The main challenge in achieving full automation is the verification of the provisos of the refinement laws. We now consider the proof obligations raised by these provisos in each step of our refinement strategy.

The data-refinement phase has five steps. Since we have patterns for the definition of both the concrete state and the retrieve relation, the first two steps can be fully automated. In Step 3, the application of the refinement laws requires mainly the verification of syntactic restrictions; which can be automated without difficulties. There are, however, a few proof obligations raised. The majority of them result from the application of laws for assumptions, conditionals and schemas, and due to the nature of our models are simple properties involving equalities and inequalities that can be easily discharged by automated provers.

Some of them, on the other hand, are action refinements. To distribute an assumption $\{p\}$ through an action is in general straightforward, except for the case where the action has the form $\mu X \bullet F(X)$. In this case, the proof obligation raised has the form $\{p\} \ ; \ F(X) \sqsubseteq F(\{p\} \ ; \ X)$. To prove this, we need to distribute $\{p\}$ through $F(X)$ up to the calls X. For that, we can proceed automatically by exhaustive application of the assumption distribution laws used in Step 3 itself and the simplification Step 6 of the structuring phase.

Finally, Steps 4 and 5 of the data-refinement phase raise no proof obligations.

The normalisation phase can also be fully automated as all the refinement laws yield relatively simple provisos. Steps 1, 2 and 4, in particular, do not yield any proof obligations. The remaining steps require the proof of properties described by simple propositions involving again simple equalities and inequalities. Due to the structure of our abstract chart models, they can be easily proved.

The structuring phase is the most complex phase of our refinement strategy, but due to the fixed structure of our models and target architecture, it is possible to reach a high degree of automation. The first step of this phase introduces new state components; it can be fully automated because information about the components can be extracted from the chart, and the location of their introduction in the model is fixed by the strategy. The proof obligations generated by this step are, as in previous steps, simple propositions.

In the Step 2 of the structuring phase, the structure of the model defines the applicable step laws needed to eliminate the parallelism. This is basically a normalisation step. Just as before, the proof obligations generated are propositions involving simple equalities and inequalities, but in a few instances they require the verification of determinism and divergence freedom.

***Example* 17** *In Example 12, the application of the Law* seq-distr-cond *to transform the sequence of conditionals into nested conditionals generates the following proof obligations.*

$$inputevents(1) = \mathbf{True} \lor inputevents(1) = \mathbf{False}$$
$$inputevents(1) = \mathbf{True} => \neg \ (inputevents(1) = \mathbf{False})$$

They follow from simple properties of booleans. □

In general, our refinement strategy makes a number of assumptions about the automatically generated models of Stateflow chart: deadlock free, divergence free, and deterministic. All of these properties can be checked on a model-by-model basis using a theorem prover for example. Alternatively, since these properties follow from the structure of the *Circus* models, rather than from details of a particular chart, it is possible to prove that every model of a well-formed chart satisfies these properties.

Step 3 of the structuring phase introduces recursions; it reuses the procedure used in Step 2 to eliminate parallelism and applies a law that requires checking determinism. This step relies on information that can be calculated from the chart, and the automation that can be achieved is comparable to that of Step 2.

In Step 4, schema operations are converted into assignments. Since the strategy provides the patterns for the assignments to be introduced, this step can be automated, with its provisos being again simple predicates. Step 5 yields only more of the simple predicates involving equalities and can be fully automated.

The final Step 6 of the structuring phase simplifies the model by distributing assumptions and eliminating unnecessary constructs. This step raises proof obligations in the form of simple predicates and refinements themselves, and also requires the verification of determinism. The refinement proof obligations can be discharged as suggested for the similar provisos generated in Step 3 of the data-refinement phase.

The final phase, action introduction, can be completely automated since it does not raise proof obligations.

In summary, all the information required by the refinement strategy can be calculated automatically from the models of the charts, and the application of the laws themselves can be automated via tactics of refinement. Whilst most of the proof obligations generated by the refinement laws are simple and can be easily discharged, some involve the verification of determinism, divergence freedom, and refinement itself. The refinement proof obligations can be discharged using procedures already defined in the strategy. The others require the use of theorem provers or model checkers. Out of all the refinement laws used in our refinement strategy, 60% of them produce no proof obligations, 30% produce easily dischargeable proof obligations and 10% produce proof obligations that require theorem proving or model checking.

Proof of general properties about our models that support the automatic discharge of such obligations is part of our agenda for future work.

6 Conclusions

In this paper, we have identified a simple, but general architectural pattern for the implementation of Stateflow charts (Figures 7, 8, and 9); it is the architecture adopted by MATLAB's automatic code generators. Based on this pattern, we have presented a refinement strategy for the verification of implementations. While parts of the strategy are dependent on the architecture of the implementation (namely, the data-refinement and structuring (except Steps 2 and 3) phases), other parts are useful in general for strategies that may target different architectures. In particular, the normalisation phase and the procedure parallelism-resolution are central to any strategy, since they support the collapsing of the process parallelism.

Additionally, the architecture-dependent procedures can also be a starting point for other strategies. While their details may require revision, the fundamental underlying principles are bound to remain the same. For instance, procedures similar to those used in the structuring phase, where appropriate assumptions are introduced and distributed through the action to simplify the structure, are likely to be widely applicable.

Our refinement strategy should produce the process that models the implementation. If this is not true, because a law application prescribed by the strategy is not valid, then there are two possibilities. Either the implementation is incorrect or it does not follow the architectural patterns assumed by this strategy. In the first case, the comparison between the two process may shed some light into what is the actual problem. Although the issue of error traceability is interesting, we leave it as future work. In the latter case, we can directly apply the refinement calculus to the model, or identify the architectural patterns used in the implementation, and adapt our refinement strategy to explore them.

Overall, our strategy is a general approach for the verification of implementations of Stateflow charts. It is also a substantial starting point for the development of other strategies, tailored for other architectural patterns, as it tackles aspects of the refinement process that are fundamental to any verification based on our automated technique for generation of chart models. The strategy uses the *Circus* refinement calculus and derives its soundness from the soundness of the refinement laws. Our refinement strategy also stands as validation of the operational approach to formal semantics of graphical notations exposed in [MC12].

While the reliance of our strategy on particular architectures implies that new strategies must be developed to tackle more general implementations, it also leads to a higher degree of automation. Moreover, the simpler proof obligations that need to be discharged during refinement increase automation and scalability. Use of the strategy does not require an understanding of its details or even of *Circus*. As indicated in Figure 1, automation of the generation of models and of the strategy hides the formalism.

Related work. The many varieties of state diagram notations have been the focus of much work on formal semantics of graphical notations [HPSS87, PS91]. Whilst some of these works can form the basis for the verification of implementations, like [MLPS97, TG05], which use Z, and [SZ02], which uses B, none of them explore this possibility, which has been the main focus of the work we presented here. Formalisations based on automata [LMM99, vdB02, Tiw02] are not appropriate for reasoning based on refinement. Works that target model checkers [LP99, BK00, CSL$^+$12] provide interesting complementary results on analysis of diagrams. So far, we have not explored the use of our *Circus* models to analyse diagrams, although that is possible.

UML state machines are considered in a number of works close to ours in their choice of formal modelling language: CSP [NB03], *Circus* itself [RSM05], and Event-B [Abr10] (for UML-B [SSB11]). These models are not as comprehensive as ours in terms of coverage of Stateflow notation, and are structured in a different way. Yet, it should be possible to reuse parts of our refinement strategy to verify implementations against these models as long as we can identify refinement laws of CSP and Event-B similar to the *Circus* laws that we use. In particular, the data-refinement phase and the procedures `parallelism-resolution` and `simplification` could be reused and extended in a rather direct way.

CSP is also used in [CSL$^+$12] to give semantics to a subset of the Stateflow notation. Our strategy can potentially be modified to tackle the models produced in this work as well. In this case, Step 2 of the structuring phase (`parallelism-resolution`) can be eliminated as they do not use parallelism.

In [MWC10], an approach to model-based development based on Simulink and Stateflow is proposed; it is based on SCADE and its formal notation, Lustre. It uses a tool to translate Simulink and Stateflow diagrams, via Lustre, to the models accepted by third-party tools (model checkers and theorem provers) and to generate C or Ada code. Similarly, [SSC$^+$04] uses Lustre to give semantics to Stateflow. The approach focuses on model checking and analysis of diagrams. Code generation, as already discussed, is extremely attractive and cost-effective, but not adequate in all situations. In our approach, instead of Lustre, we use *Circus*, and address verification of implementations rather than analysis of diagrams or code generation.

Our work is a natural extension of [CCO05, CC06, CCO11]. In [CCO05], we have given a semantics of Simulink diagrams in *Circus*. It extends the results of the Z technique and tool in [AC05] to cover a larger subset of the Simulink notation, but it still does not cover Stateflow blocks. In [CC06, CCO11], the *Circus* semantics is used to define a refinement strategy to verify parallel implementations of Simulink diagrams. Since it uses the models in [CCO05], it does not consider Stateflow blocks, which is our main contribution here. Combination of that refinement strategy and ours is an interesting avenue for future work. Both strategies have a normalisation phase that generates processes written using the same pattern of specification.

The use of Simulink and Stateflow in the development of safety-critical systems has yielded a number of guidelines that restrict their use to include only features considered safe. For instance, [Mata] proposes guidelines for improving readability. These guidelines have been further investigated in [FFBZ09], where they are applied in the development of railway signalling systems. Whilst we believe that such guidelines do not affect our models of Stateflow and, therefore, the refinement strategy, it is reasonable to assume that they can be used to simplify our refinement strategy to tackle the more restrictive subset of Stateflow charts.

Future work. The current refinement strategy targets sequential implementations. In [Miy12], we present an extended strategy that deals with parallel implementations for charts in which (some) parallel states that do not share variables are run in parallel. As future work, we would like to support the verification of a wider variety of implementations. Of particular interest are architectures that use cyclic executive scheduling; they are popular in control system implementations, and are adopted in the strategy in [CCO11].

Additionally, as already mentioned, we do not treat programs that include mutual recursions. Our strategy, however, can be extended to treat mutual recursions by modifying the procedure recursion-introduction to extract information about mutual recursions from the implementation, calculate the appropriate recursive actions, and apply a version of the fixed point laws that refines actions to mutual recursions.

There are two kinds of Stateflow events, triggered and function call, and in our work we handle only triggered events. The distinction is at the level of the semantics of the Simulink diagram that includes the Stateflow block, rather than the semantics of the Stateflow chart. For inputs, the distinction is used to control when the block can be executed for that event: either (triggered) just once during the step of the Simulink diagram, or possibly several times. In the case of outputs, a triggered event is only signalled to the Simulink diagram (by the Stateflow chart) at the end of the Stateflow block step. If the same triggered output event occurs several times during the Stateflow block step, that event is signalled just once to the Simulink diagram, but the remaining occurrences are queued and signalled at the end of the subsequent steps of the chart. In the case of an output function-call event, the signalling happens as soon as it occurs. It is very simple to change our model to signal function-call events as they occur. All we need to do is to introduce a synchronisation on the channel that corresponds to the event.

The abstraction of input events mentioned in Section 4 arises from the independent treatment of Stateflow diagrams. When considered in the context of a Simulink diagram, like when code is generated, this abstraction is no longer needed. Integration of our results with the existing modelling and refinement strategy for Simulink can address this issue easily.

While the technique for the generation of *Circus* models of Stateflow charts has been formalised, implemented and extensively validated, the refinement strategy requires further validation. The implementation of our refinement strategy is not a trivial task and is part of our plans for future work. Nevertheless, validation in terms of soundness stems from the validity of the *Circus* refinement laws, not from case studies.

Finally, since the refinement strategy derives its soundness from the refinement laws used, providing mechanised proofs for all the refinement laws is a requirement for any practical use of the strategy. Since *Circus* has theorem proving support [ZC10], these laws can be formalised and verified. This task, however, is not simple, and requires deep understanding of the semantics of *Circus*.

References

[Abr10] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[AC05] 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.

[Bar03] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[BK00] C. Banphawatthanarak and B. H. Krogh. Verification of stateflow diagrams using smv: sf2smv 2.0. Technical Report CMU-ECE-2000-020, Carnegie Mellon University, 2000.

[CC06] A. L. C. Cavalcanti and P. Clayton. Verification of Control Systems using *Circus*. In *11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 269 – 278. IEEE Computer Society, 2006.

[CCM$^+$03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. In R. Alur and I. Lee, editors, *EMSOFT 2003*, volume 2855 of *Lecture Notes in Computer Science*, pages 84 – 99. Springer-Verlag, 2003.

[CCO05] 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.

[CCO11] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465 – 512, 2011.

[CSL$^+$12] C. Chen, J. Sun, Y. Liu, J. S. Dong, and M. Zheng. Formal modeling and validation of Stateflow diagrams. *International Journal on Software Tools for Technology Transfer*, 14(6):653 – 671, 2012.

[CSW03] 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.

[CW99] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.

[FFBZ09] A. Ferrari, A. Fantechi, S. Bacherini, and N. Zingoni. Modeling guidelines for code generation in the railway signaling context. In *NASA Formal Methods*, pages 166 – 170, 2009.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[HH98] C. A. R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, pages 54 – 64. IEEE Press, 1987.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 331–347. Kluwer Academic Publishers, 1999.

[LP99] J. Lilius and I. P. Paltor. The Semantics Of UML State Machines. Technical Report 273, Turku Centre and Computer Science, 1999.

[LST09] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams modularity versus code size. In *36th Symposium on Principles of Programming Languages*, 2009.

[Mata] Mathworks Automotive Advisory Board. MAAB Control Algorithm Modeling. `http://www.mathworks.co.uk/help/simulink/maab-control-algorithm-modeling.html`.

[Matb] The MathWorks,Inc. *Real-Time Workshop*. `www.mathworks.com/products/rtwt`.

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

[Matd] The MathWorks,Inc. *Stateflow and Stateflow Coder 7 User's Guide*. `www.mathworks.com/products`.

[MC11] A. Miyazawa and A. L. C. Cavalcanti. Refinement-based verification of sequential implementations of Stateflow charts. In J. Derrick, E. Boiten, and S. Reeves, editors, *Refinement Workshop*, Electronic Notes in Theoretical Computer Science. Elsevier, 2011. DOI: 10.4204/EPTCS.55.

[MC12] A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10 – 11):1151 – 1177, 2012.

[Miy12] A. Miyazawa. *Formal verification of implementations of Stateflow charts*. PhD thesis, University of York, 2012.

[MLPS97] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On formal semantics of Statecharts as supported by statemate. In *BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, 1997.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[MWC10] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58 – 64, 2010.

[NB03] 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.

[OC08] 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.

[Oli06] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs Using Circus*. PhD thesis, University of York, 2006.

[OZC11] M. V. M. Oliveira, F. Zeyda, and A. L. C. Cavalcanti. A tactic language for refinement of state-rich concurrent specifications. *Science of Computer Programming*, 76(9):792 – 833, 2011.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer Berlin Heidelberg, 1991.

[Ros11] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[RSM05] 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.

[SSB11] C. Snook, V. Savicks, and M. Butler. Verification of UML models by translation to UML-B. In *9th International Conference on Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 251 – 266. Springer, 2011.

[SSC$^+$04] 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.

[SZ02] Emil Sekerinski and Rafik Zurob. Translating Statecharts to B. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 128–144. Springer Berlin Heidelberg, 2002.

[Tar] TargetLink. `http://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm`. Accessed: 05-05-2013.

[TG05] I. Toyn and A. Galloway. Proving Properties of Stateflow Models using ISO Standard Z and CADiZ. In M. C. Henson H. Treharne, S. King and S. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 104 – 123. Springer-Verlag, 2005.

[Tiw02] A. Tiwari. Formal Semantics and Analysis Methods for Simulink Stateflow Models. Technical report, SRI International, 2002. `www.csl.sri.com/~tiwari/stateflow.html`.

[TNP+08] A. Toom, T. Naks, M. Pantel, M. Gandriau, and Indrawati. Gene-Auto: an automatic code generator for a safe subset of Simulink/Stateflow and Scicos. In *4th European Congress ERTS Embedded Real-Time Software*, 2008.

[vdB02] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, V1:130 – 141, 2002.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[ZC10] Frank Zeyda and Ana Cavalcanti. Encoding *Circus* Programs in ProofPower-Z. In Andrew Butterfield, editor, *Unifying Theories of Programming*, volume 5713 of *Lecture Notes in Computer Science*, pages 218 – 237. Springer Berlin Heidelberg, 2010.

[ZOC12] F. Zeyda, M. V. M. Oliveira, and A. L. C. Cavalcanti. Mechanised support for sound refinement tactics. *Formal Aspects of Computing*, 24(1):127 – 160, 2012.

A Implementation of Air controller example

```
static void Air_chartstep_c1_Air(void)
{
  if (Air_DWork.is_active_c1_Air == 0) {
    Air_DWork.is_active_c1_Air = 1U;
    Air_DWork.is_c1_Air = Air_IN_PowerOff;
    Air_B.airflow = 0U;
  } else {
    switch (Air_DWork.is_c1_Air) {
     case Air_IN_PowerOff:
      if (_sfEvent_Air_ == Air_event_SWITCH) {
        Air_DWork.is_c1_Air = Air_IN_PowerOn;
        Air_DWork.is_active_FAN1 = 1U;
        Air_DWork.is_FAN1 = Air_IN_Off;
        Air_DWork.is_active_FAN2 = 1U;
        Air_DWork.is_FAN2 = Air_IN_Off;
        Air_DWork.is_active_SpeedValue = 1U;
      }
      break;
     case Air_IN_PowerOn:
      if (_sfEvent_Air_ == Air_event_SWITCH) {
        Air_DWork.is_active_SpeedValue = 0U;
        Air_DWork.is_FAN2 = (uint8_T)Air_IN_NO_ACTIVE_CHILD;
        Air_DWork.is_active_FAN2 = 0U;
        Air_DWork.is_FAN1 = (uint8_T)Air_IN_NO_ACTIVE_CHILD;
        Air_DWork.is_active_FAN1 = 0U;
        Air_DWork.is_c1_Air = Air_IN_PowerOff;
        Air_B.airflow = 0U;
      } else {
        switch (Air_DWork.is_FAN1) {
         case Air_IN_Off:
          if (Air_U.temp >= 120.0) {
            Air_DWork.is_FAN1 = Air_IN_On;
          }
          break;
         case Air_IN_On:
          if (Air_U.temp < 120.0) {
            Air_DWork.is_FAN1 = Air_IN_Off;
          }
          break;
         default:
          Air_DWork.is_FAN1 = Air_IN_Off;
          break;
        }
        switch (Air_DWork.is_FAN2) {
         case Air_IN_Off:
          if (Air_U.temp >= 150.0) {
            Air_DWork.is_FAN2 = Air_IN_On;
          }
          break;
         case Air_IN_On:
          if (Air_U.temp < 150.0) {
            Air_DWork.is_FAN2 = Air_IN_Off;
          }
          break;
         default:
          Air_DWork.is_FAN2 = Air_IN_Off;
          break;
        }
        Air_B.airflow = (uint8_T)((Air_DWork.is_FAN1 == Air_IN_On) +
          (Air_DWork.is_FAN2 == Air_IN_On));
      }
      break;
     default:
      Air_DWork.is_c1_Air = Air_IN_PowerOff;
      Air_B.airflow = 0U;
      break;
    }
  }
}

static void Air_output(int_T tid)
{
  int32_T c_previousEvent;
  if (Air_U.inputevents[0]) {
      c_previousEvent = _sfEvent_Air_;
      _sfEvent_Air_ = Air_event_SWITCH;
      Air_chartstep_c1_Air();
      _sfEvent_Air_ = c_previousEvent;
  }
  if (Air_U.inputevents[1]) {
      c_previousEvent = _sfEvent_Air_;
      _sfEvent_Air_ = Air_event_CLOCK;
      Air_chartstep_c1_Air();
      _sfEvent_Air_ = c_previousEvent;
    }
  Air_Y.airflow = Air_B.airflow;
}
```

B Novel refinement laws

We present here, in alphabetical order, the novel refinement laws of *Circus* that we need.

Law [assign-schema-conv]

$$\{inv\}\ ;\ v := e \quad = \quad [\Delta S \mid vs' = vs \land v' = e]$$

where

- *$S == [dv, dvs \mid inv]$,*
- *dv is the declaration of v, dvs is the declaration of the remaining variables, and*
- *inv is the state invariant.*

provided *$inv \Rightarrow inv \land (\exists\, d' \bullet inv' \land vs' = vs \land v' = e)$.*

Law [cond-elim]

$$(\mathbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \mathbf{fi}) \quad = \quad A_1$$

provided *$b_1 \lor b_2$; $b_1 \Rightarrow \neg\, b_2$; and $b_1 \Leftrightarrow$ **True**.*

Law [par-distr-cond]

$$\begin{array}{l}(\mathbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \mathbf{fi})\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B \\ = \\ (\mathbf{if}\ b_1 \longrightarrow A_1\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B \ [\!]\ b_2 \longrightarrow A_2\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B\ \mathbf{fi})\end{array}$$

provided *$initials(B) \subseteq cs$ and B is deterministic.*

Law [par-prefix-step]

$$\begin{array}{l}((c \longrightarrow A)\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B) \quad = \quad c \longrightarrow (A\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B) \\ ((c?x \longrightarrow A)\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B) \quad = \quad c?x \longrightarrow (A\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B) \\ ((c.e \longrightarrow A)\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B) \quad = \quad c.e \longrightarrow (A\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ B)\end{array}$$

provided *$c \notin cs$; $x \notin usedV(B)$; $initials(B) \subseteq cs$; and B is deterministic.*

Law [seq-assign-conv]

$$[\Delta S \mid c_1' = e_1 \land \ldots \land c_m' = e_m \land c_{m+1}' = c_{m+1} \land \ldots \land c_n' = c_n] \quad \sqsubseteq \quad c_1 := e_1\ ;\ \ldots\ ;\ c_m := e_m$$

where *$S = [d \mid inv]$ and $c_1, \ldots, c_n$ are state components (elements of αd).*
syntactic restriction *αd and $\alpha d'$ are not free in $e_1, \ldots, e_n$.*
provided *$inv[e_1, \ldots, e_m / c_1, \ldots, c_m]$.*

Law [seq-distr-cond]

$$(\mathbf{if}\ b_1 \longrightarrow A_1 \ [\!]\ b_2 \longrightarrow A_2\ \mathbf{fi})\ ;\ B \quad = \quad (\mathbf{if}\ b_1 \longrightarrow A_1\ ;\ B \ [\!]\ b_2 \longrightarrow A_2\ ;\ B\ \mathbf{fi})$$

provided *$b_1 \lor b_2$ and $b_1 \Rightarrow \neg\, b_2$.*

Law [tail-rec-seq-dist]

$$(\mu X \bullet \mathbf{if}\ p \longrightarrow A\ ;\ X \ [\!]\ q \longrightarrow \mathbf{Skip}\ \mathbf{fi})\ ;\ B \quad = \quad (\mu X \bullet \mathbf{if}\ p \longrightarrow A\ ;\ X \ [\!]\ q \longrightarrow B\ \mathbf{fi})$$

provided *$p \lor q$ and $p \Rightarrow \neg\, q$.*

Law [var-seq-ext-right]

$$(\mathbf{var}\ x : T \bullet A)\ ;\ B \quad = \quad (\mathbf{var}\ x : T \bullet A\ ;\ B)$$

provided *$x \notin FV(B)$.*