Refinement-based verification of sequential implementations of Stateflow charts

Alvaro Miyazawa
Department of Computer Science
University of York
alvarohm@cs.york.ac.uk

Ana Cavalcanti
Department of Computer Science
University of York
ana.cavalcanti@cs.york.ac.uk

Simulink/Stateflow charts are widely used in industry for the specification of control systems, which are often safety-critical. This suggests a need for a formal treatment of such models. In previous work, we have proposed a technique for automatic generation of formal models of Stateflow blocks to support refinement-based reasoning. In this article, we present a refinement strategy that supports the verification of automatically generated sequential C implementations of Stateflow charts. In particular, we discuss how this strategy can be specialised to take advantage of architectural features in order to allow a higher level of automation.

1 Introduction

MATLAB Simulink [24] is a graphical notation widely used in the automotive and avionics industries; it supports the specification of control systems in a level of abstraction convenient for engineers. A Simulink diagram consists of blocks and wires connecting the inputs and outputs of the blocks.

Stateflow [25] is an extension of Simulink that supports the specification of state transition systems, providing a new Simulink block, namely, a Stateflow chart. It is a variant of Statecharts [12], which extends standard state-transition systems by introducing new features, such as hierarchy and parallelism.

While Simulink diagrams are typically used to specify aspects of a system that can be modelled by differential equations relating inputs and outputs, Stateflow charts usually model the control aspects. There is a wide range of tools that support Simulink and Stateflow. These include a simulation and analysis tool, a verification and validation tool, a code generator and a prototyping tool [24, 25, 23].

The extensive use of Simulink/Stateflow in the development of safety-critical systems, associated with certification standards [3, 9] that recommend the use of formal methods for the specification, design, development and verification of software, makes a formal treatment of these notations extremely useful.

We are concerned with the assessment of the correctness of implementations of Stateflow charts. A frequent approach to this problem is based on the verification of automatic code generators [4, 27, 13]. We propose an orthogonal approach based on the verification of implementations with respect to a model of the chart. An overview of our approach is given in Figure 1.

This approach consists of deriving formal models of a Stateflow chart and its implementation, and applying the refinement calculus to check the correctness of the model of the implementation with respect to the model of the chart. This is particularly suited for situations where automatically generated code is not applicable or convenient, for instance, in situations where hardware and performance requirements require changes in the generated code. Moreover, Simulink and Stateflow are frequently updated, and these updates can have a heavy impact on the cost of the verification of any code generator.

In [16], we propose an operational model of Stateflow charts, provide translation rules for deriving such models, and discuss a tool that automatically generates the model of a Stateflow chart. The model of

J. Derrick , E.A. Boiten, S. Reeves (Eds.):
Refinement Workshop 2011.
EPTCS 55, 2011, pp. 65–83, doi:10.4204/EPTCS.55.5

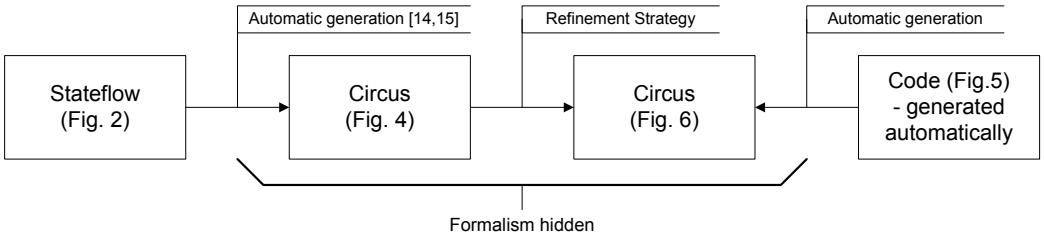

Figure 1: An approach for the verification of implementations of Stateflow charts.

a Stateflow chart is formed by the composition of two processes: the first models the general semantics of Stateflow [25], and the second models specific aspects of a chart. The model of the semantics of Stateflow chart is structured in a way that facilitates the inspection and comparison to the informal semantics found in [25], as no formal analysis can be made because the semantics of Stateflow is available in an informal way [25] or is hidden in the simulation tool.

The refinement calculus is enough for the purpose of verifying such models. However, the expertise required for such verification 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. We aim in our approach to hide as much of the formalism as possible, to allow it to be used in real scenarios by engineers and programmers. For that to be achieved, we must provide means for the refinement to be established at least in a semi-automatic way.

We propose a verification strategy for sequential automatically generated implementations of discrete-time Stateflow charts with respect to models of Stateflow constructed (automatically) as described in [16]. This technique is closely related to that proposed in [5] for verification of implementations of Simulink diagrams. Our work extends those results to cover a larger class of diagrams and implementations.

The implementations that we consider are those that follow the architectural pattern employed by the code generator provided by MATLAB. There are other code generators [22, 27], but as far as we know, they all cover a limited subset of the Stateflow notation. Fixing the architecture of the implementation allows us to specialise the details of the strategy to increase its level of automation.

Our models for Stateflow charts are specified in *Circus* [29], a formal notation that integrates Z [30], CSP [21], Dijkstra's language of guarded commands [8], and the refinement calculus [17]. These models are particularly adequate for refinement-based verification techniques. Our technique uses the *Circus* refinement laws to provide a tactic of refinement that can be used to prove the correctness of an implementation in a highly automated way. Soundness of the technique stems from soundness of the laws.

This article is structured as follows. Section 2 introduces the background material necessary for the presentation of our strategy. Section 3 discusses the architecture of automatically generated implementations of Stateflow charts and provides general guidelines for deriving *Circus* models of these implementations. Section 4 describes our refinement strategy for the verification of implementations of Stateflow charts. Section 5 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.

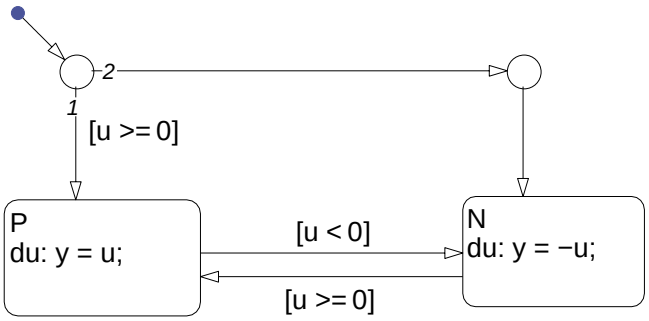

Figure 2: Absolute value chart.

2.1 Stateflow charts

Figure 2 shows our running example: a Stateflow chart adapted from an example supplied with the tool. The chart has one input variable (`u`) and one output variable (`y`); it outputs in `y` the absolute value of `u`.

A Stateflow chart is built from a series of components, such as states, transitions, junctions, data and events. States are represented by rectangles with round corners; in our example, the boxes marked with `P` and `N` are states. States, as well as charts, can have substates, which are arranged in a sequential or parallel decomposition. A state with a sequential decomposition has at most one substate active at any given time, while a state with a parallel decomposition has all of its substates active or inactive at once.

A state has a set of actions associated with it, namely, entry, during, exit, on, and binding actions. Entry, during and exit actions are executed when the state is entered, executed, and exited, respectively; on actions are executed in the same situations as during actions, with the additional requirement that a particular event is being processed; and binding actions bind a particular event or data to the state. In our example, both `P` and `N` have a during action. In `P`, `u` is assigned to `y`, and in `N`, `-u` is assigned to `y`.

Two states (within a state or chart with sequential decomposition) can be connected by one or more transitions; they are indicated by arrows and can be guarded by events and conditions. There are two types of actions associated with a transition: condition and transition actions. Condition actions are executed when the guard of the transition (event and condition) is true, and transition actions are executed when the transition leads to a state being exited.

Transitions are classified according to the relative position between its source and target states; inner transitions have the target state as a substate of the source state, and outer transitions do not. There is a special type of transition, called default transition, that has no source; it is used to indicate the default path to be taken when a state or chart is first entered. In our example, we have one default transition, and five outer transitions. Three of the transitions are guarded by a condition: `u>=0`, `u<0`, or `u>=0`.

A transition path is formed by a series of transitions linked by junctions which are represented by circles. There are two junctions in our example; they form two transition paths. A transition path is completed only when a state is reached by following all the transitions in the path. When a transition path is completed, the source of the path is exited, the transition actions of the path are executed, and the target state is entered. Additionally, there is a special type of junction, called history junction, which records the most recently activated substate of the state that contains it.

In the example in Figure 2, initially, the chart is inactive; the first time it is executed, it is activated

$$
\begin{array}{l}
\textbf{channel}\, in, in1, in2, out : \mathrm{seq}\,\mathbb{N} \\
\textbf{process}\, Merger \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, S == [y : \mathrm{seq}\,\mathbb{N}] \\
\quad InitS == [S' \mid y' = \langle\rangle] \\
\quad Merge \mathrel{\widehat=} x1, x2 : \mathrm{seq}\,\mathbb{N} \bullet \\
\qquad \left(\begin{array}{l}
\textbf{if}\, \#x1 = 0 \longrightarrow y := y \frown x2 \\
[\!]\, \#x2 = 0 \longrightarrow y := y \frown x1 \\
[\!]\, \#x1 \neq 0 \wedge \#x2 \neq 0 \longrightarrow \\
\quad \left(\begin{array}{l}
\textbf{if}\, head\, x1 \leq head\, x2 \longrightarrow y := y \frown \langle head\, x1 \rangle \,;\, Merge(tail\, x1, x2) \\
[\!]\, head\, x1 > head\, x2 \longrightarrow y := y \frown \langle head\, x2 \rangle \,;\, Merge(x1, tail\, x2) \\
\textbf{fi}
\end{array} \right) \\
\textbf{fi}
\end{array} \right) \\
\quad \bullet\, InitS \,;\, in1?x1 \longrightarrow in2?x2 \longrightarrow Merge(x1, x2) \,;\, out!y \longrightarrow \textbf{Skip} \\
\textbf{end} \\
\textbf{process}\, SplitSorter \mathrel{\widehat=} \ldots \\
\textbf{process}\, ParallelSorter \mathrel{\widehat=} \left(\begin{array}{c} SplitSorter \\ [\![\{\!| in1, in2 |\!\}]\!] \\ Merger \end{array} \right) \setminus \{\!| in1, in2 |\!\}
\end{array}
$$

Figure 3: The *ParallelSort* specification.

and one of its two states is entered depending on whether `u` is greater than or equal to zero (state `P`) or not (state `N`). In the next execution, if the sign of `u` has changed, a transition takes place from one state to the other. If there is no change, the during action of the active state assigns the absolute value of `u` to `y`.

Before presenting the *Circus* model of this chart, we give, in the next section, an overview of *Circus*.

2.2 *Circus*

We present the main *Circus* features using the example in Figure 3. 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 input sequence. A detailed presentation of *Circus* can be found in [29].

A *Circus* specification is a sequence of paragraphs: Z paragraphs (axiomatic definitions, schemas, and so on), channel and channel set declarations, and process definitions. The first paragraph of our example defines four channels *in*, *in*1, *in*2, and *out*, which communicate sequences of natural numbers, that is, elements of the type $\mathrm{seq}\,\mathbb{N}$. The second paragraph is a basic process definition. It provides the name of the process (*Merger*), the definition of its state using a schema *S*, an action *InitS* defined by an operation schema, an action *Merge*, and a main action (after a •), which defines, using the previously defined actions, the overall behaviour of the process.

In general, *Circus* actions are written using a mixture of Z and CSP constructs, and guarded commands. In our example, the main action initialises the state using *InitS*, reads a value $x1$ through the channel *in*1, reads a value $x2$ through *in*2, calls *Merge* with the values $x1$ and $x2$ as parameters, and outputs the state variable y through *out*.

The schema *S* has only one component y of type $\mathrm{seq}\,\mathbb{N}$, that is, the set of sequences of natural numbers. The schema *InitS* specifies an initialisation operation over *S* that 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* takes two sequences $x1$ and $x2$ of natural numbers, and appends them to the state variable y, so that if both input sequences are ordered, the final sequence in y is also ordered. The specification of *Merge* uses a conditional and assignments from the guarded commands language. If one of the sequences is empty, the non-empty sequence is appended. When both sequences are not empty, *Merge* compares the first element of each sequence ($head\, x1 \leq head\, x2$), appending the smallest of them to y, 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.

Processes encapsulate their state and interact with other processes through channels. The usual CSP operators can be used to combine processes. The fourth paragraph in Figure 3 defines the process *ParallelSorter* as the parallel composition of the processes *SplitSorter* and *Merge*, communicating over the channels $in1, in2$. The process *SplitSorter* in the third paragraph is omitted; it splits a sequence of natural numbers in two, sorts each sequence in parallel, and outputs them through channels $in1$ and $in2$. In the definition of *ParallelSorter*, the channels $in1$ and $in2$ are hidden, thus yielding a process whose interface contains only the channels *in* and *out*.

In the next section, 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* operational models of Stateflow charts that we can generate automatically. In these models, the execution of one step of the chart is initiated by reading inputs, and concluded by writing outputs, and synchronising on a channel called *end_cycle*. A more detailed description can be found in [15, 16].

Our models consist of two *Circus* processes in parallel. The first, *Simulator*, represents the simulator, and is the same for every chart. The second, the chart process, represents a particular chart. The simulator and the chart processes communicate over the channels in the set *interface* plus the channel *end_cycle*, with the channels in *interface* hidden. Figure 4 shows the structure of the automatically generated model of the chart in Figure 2.

The chart process *P_AbsoluteValue* uses a data model that defines the state, transition, and junction identifiers, as well as the states, transitions, and junctions as bindings of specific schemas. These are constants that capture information about the structure of the chart. They are represented by the first rectangle in Figure 4. These constants 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 binding. These constants are declared using a schema *StateflowChart* whose definition is omitted in Figure 4. Their values are fixed in the process chart.

Next, the chart process defines a series of schemas that specify components of the state and corresponding initialisation operations. Information about which states are active and which states are recorded in the history junctions is recorded in the schema *SimulationData*, and chart variables are recorded in the schema *SimulationInstance*. These schemas are conjoined to define the schema *State* that specifies the state of the process. We adopt the convention of prefixing a $v\_$ to the name of the chart variable to clarify the nature of the name.

Next, the chart process defines a series of *Circus* actions that can be divided into four groups: actions that correspond to state and transition actions, actions that correspond to calculation of triggers and conditions, actions that read inputs and write outputs, and actions that output the structure of the chart. All these actions are grouped to define *AllActions*, which is used in the main action as shown in Figure 5.

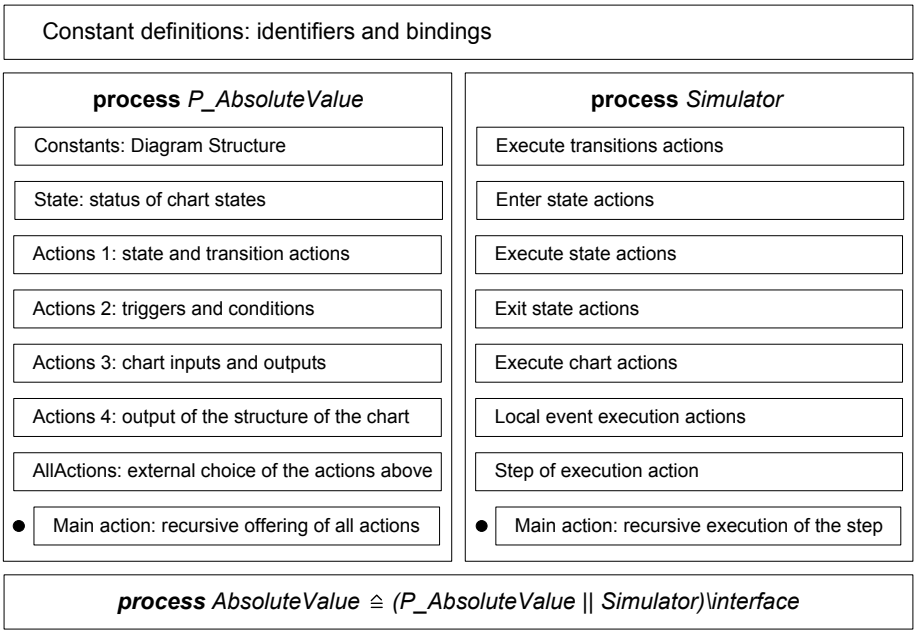

Figure 4: Structure of the model of the chart in Figure 2

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

end

Figure 5: Main action of the chart process shown in Figure 4

The main action of the chart process is shown in Figure 5; it initialises the state, and recursively offers the actions in *AllActions*, with the additional possibility that any of these actions can be interrupted at any time by a communication over the channel *interrupt_chart*. The possibility of interruption accounts for the occurrence of early return logic in the chart, that is, the interruption of the execution of the chart brought about by a state inconsistency produced by a local event broadcast. The main action has two nested recursions: the internal one corresponds to the actions that are offered to 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 processes of entering, executing and exiting states. The execution of a chart is then defined in terms of the previous actions. A chart can be executed multiple times in the same time step due to the occurrence of multiple input events or the broadcast of a local event. In the first case, an action encodes the execution of the chart for each active event in the appropriate order and is used to define the step of execution. In the second case, we define an action that captures the occurrence of a broadcast and executes the chart under the appropriate setting. This action is called whenever a state or transition action is executed. All these actions are combined to define the step of execution of the

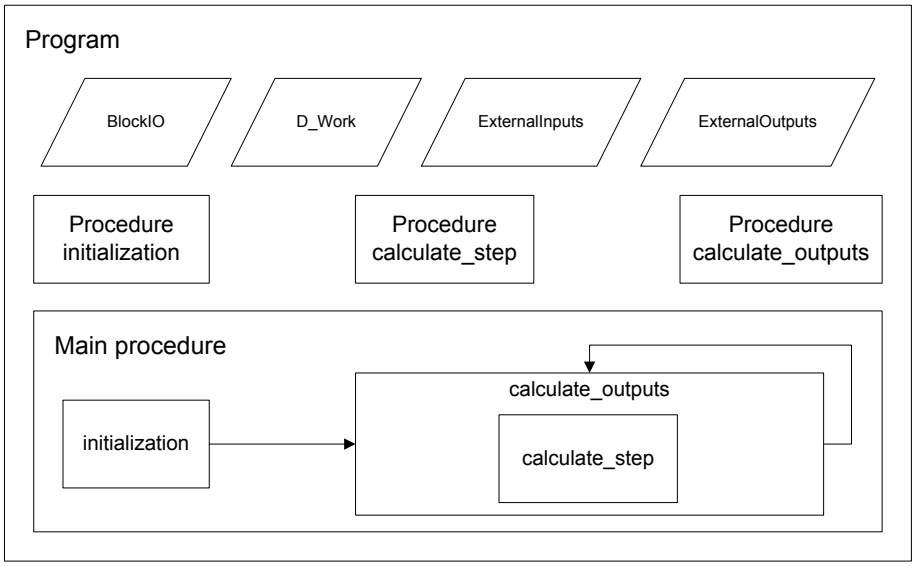

Figure 6: Architcture of the implementations of Stateflow charts

simulator, which is called recursively in the main action of the process *Simulator*.

In the next section we discuss an approach to modelling implementations of charts.

3 Implementations of Stateflow charts

Figure 6 shows the structure of implementations of Stateflow charts generated automatically by Real Time Workshop/Stateflow Coder. In general, the implementations produced consist of a series of structures that define the state of the chart (inputs, outputs, local variables, events, execution state, and so on) and a series of procedures. The procedures can be divided into those that implement the execution of the chart, which are relevant for our verification, and those that calculate the next time step. Since we capture time using synchronisation, we restrict our attention those of the first kind depicted in Figure 6 as `calculate_outputs`, `initialization` and `calculate_step`.

The procedure `calculate_outputs` implements the execution of the chart, `initialization` initialises the variables of the program, and `calculate_step` implements the control of the execution of the chart according to the number of active events. The main procedure of the implementation initializes the program and repeatedly calculate the outputs using the procedure `calculate_step`. In the case of a C implementation, the procedures shown in Figure 6 are implemented as C functions whose names reflect the name of the chart. In our example, for instance, the procedure *calculate_outputs* is implemented as the C function `AbsoluteValue_output`.

Of particular interest to us is how the implementation models information regarding the status of the states. This is done in two different ways, according to the type of decomposition of the states. In this discussion, we regard the chart as a state. If the state has a parallel decomposition, its status is modelled by a single variable in the structure *D_Work_X*, where *X* is the name of the chart. This variable is called *is_active_S*, where *S* is the name of the state; it has a numerical type (`uint8_T`), but, in fact, it is used as a boolean variable, that is, if its value is zero the state is not active, otherwise, it is active.

$$
\begin{array}{l}
D\_Work\_AbsoluteValue == [is\_active\_c1\_AbsoluteValue, is\_c1\_AbsoluteValue : \mathbb{N}] \\
\ldots \\
\textbf{process}\, AbsoluteValue \mathrel{\widehat=} \textbf{begin} \\
\textbf{state}\, AbsoluteValue\_state == [AbsoluteValue\_DWork : D\_Work\_AbsoluteValue;\ \ldots] \\
AbsoluteValue\_DWork\_is\_c1\_AbsoluteValue \mathrel{\widehat=} x : \mathbb{N} \bullet AbsoluteValue\_DWork := \\
\quad \langle is\_active\_c1\_AbsoluteValue == AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue, \\
\quad is\_c1\_AbsoluteValue == \_y \rangle \\
\ldots \\
AbsoluteValue\_output \mathrel{\widehat=} tid : \mathbb{Z} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{if}\, AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue = 0 \longrightarrow \\
AbsoluteValue\_DWork\_is\_active\_c1\_AbsoluteValue(1); \\
\left(\begin{array}{l}
\textbf{if}\, AbsoluteValue\_B.SineWave1 \geq 0 \longrightarrow \\
AbsoluteValue\_DWork\_is\_c1\_AbsoluteValue(AbsoluteValue\_IN\_P) \\
[]\neg\ (AbsoluteValue\_B.SineWave1 \geq 0) \longrightarrow \\
AbsoluteValue\_DWork\_is\_c1\_AbsoluteValue(AbsoluteValue\_IN\_N) \\
\textbf{fi}
\end{array} \right) \\
[]\neg\ (AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue = 0) \longrightarrow \\
\ldots \\
\textbf{fi}
\end{array} \right) \\
AbsoluteValue\_Y\_y(AbsoluteValue\_B.y)
\end{array} \right) \\
\ldots \\
\bullet\, AbsoluteValue\_initialize\,;\ \mu X \bullet Input\,;\ AbsoluteValue\_output\,;\ Output\,;\ end\_cycle \longrightarrow X
\end{array}
$$

Figure 7: *Circus* model of the implementation of the chart in Figure 2

If the state has a sequential decomposition, its status is modelled by two variables in *D_Work_X*. The variable *is_active_S* is as described above. The variable *is_S* records a number that identifies which substate is active at the time, its value is zero if there are no active substates. If a state is a child of a state with a sequential decomposition, no variable of the form *is_active_S* is created, as the information about its status is already recorded in the variable *is_P*, where *P* is the name of its parent state.

In our example, we have only states with sequential decompositions. The status of the chart is recorded by `is_active_c1_AbsoluteValue` and `is_c1_AbsoluteValue`. There is no need for variables that record the status of `P` and `N`, as this information can be obtained from `is_c1_AbsoluteValue`.

We model the implementation as a series of schemas and a single process. Figure 7 gives a partial view of the model of the implementation of our example. Schemas model the records in the implementation. For instance, `D_Work_AbsoluteValue` is modelled by the schema *D_Work_AbsoluteValue*.

For each of the relevant C function in the implementation, we define a *Circus* action that models it. In Figure 7, we present the *Circus* action that models the function `AbsoluteValue_output`. The main action of the process is fixed and consists of calling the initialisation action (*AbsoluteValue_initialize* for our example), and recursively reading the inputs (with the action *Input*), producing the outputs (with the action *AbsoluteValue_output*), offering the outputs (with the action *Output*), and signalling the end of the "time-step" by synchronising on *end_cycle*. The actions *Input* and *Output* abstract as communications the shared variables used to implement inputs and outputs.

In the modelling of the functions, we map C constructs to similar *Circus* constructs. Loops are

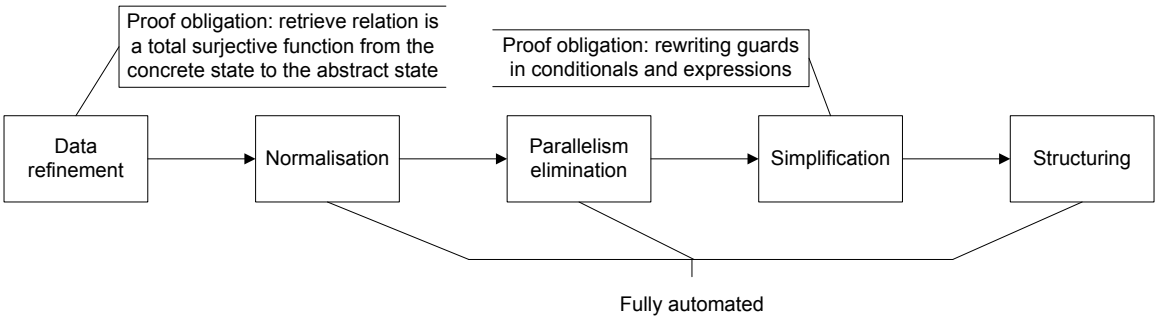

Figure 8: Overview of the refinement strategy

modelled using recursion. In general, the translation of implementation constructs is direct, except for the assignment to a variable of a structure. Since we cannot write $b.f := v$ (as a translation of `b.f = v`), for a variable b of type binding, we define *Circus* actions that specify the assignment of a binding to the variable, as the action *AbsoluteValue_DWork_is_c*1*_AbsoluteValue* in Figure 7. This action takes one parameter x of type $\mathbb{N}$, and assigns a binding of the schema *D_Work_AbsoluteValue* to the state component *AbsoluteValue_DWork*. The binding is formed by associating each component of the schema to a value, the component *is_active_c*1*_AbsoluteValue* is associated to the value of the same component, and *is_c*1*_AbsoluteValue* is associated to the value of the parameter.

In the next section, we discuss a refinement strategy that supports the verification of the models of implementations just described with respect to the models discussed in Section 2.3.

4 Refinement Strategy

Our refinement strategy consists of five phases: data refinement, normalisation, parallelism elimination, simplification, and structuring. Figure 8 illustrates the strategy; it identifies the fully automated phases and the proof obligations that stem from the other phases.

In the data refinement phase, we modify the state of the chart process in order to conform to the state of the implementation model. The normalisation phase transforms the parallel composition of the chart and simulation processes into a single process whose main action initialises the state and recursively offers an action that encodes a step of execution of the chart. The parallelism elimination phase collapses the parallel actions that occur in the resulting process. This is necessary because the parallelism in the diagram model reflects the operational semantics of Stateflow, not a parallel design for a program. In the simplification phase, we simplify expressions and predicates, and move assumptions through the model to eliminate unreachable branches of alternations. Finally, in the structuring phase, we rewrite the main action to match the functions of the implementation, and therefore the actions of its model.

The strategy proposed in this section is generic enough to be applied to a large class of charts and implementations. It takes advantage of restrictions on the architecture of the implementation to support a high degree of automation. We consider here only sequential implementations of Stateflow diagrams. The steps of the strategy are, however, useful in the refinement to parallel implementations as well.

In the sequel, we describe the details of each phase. They define procedures to apply existing and novel *Circus* refinement laws whose soundness guarantees the soundness of our verification strategy.

ConcreteState

$$AbsoluteValue\_state$$

$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \in \{0,1,2\}$$

Figure 9: Restricted concrete state

4.1 Data refinement

In this phase, we construct a retrieve relation between the abstract state of the chart process and the concrete state of the implementation model. With that, we use the *Circus* calculus to construct a refinement of the chart process, and so preserve its structure, and transform the assignments, operation schemas, and communications. Precisely, we follow the procedure below for constructing retrieve relations that are total surjective functions from concrete to abstract states. They allow us to proceed with the data refinement in a calculational fashion.

The state components of the implementation model belong to one of four groups: execution specific data, like `AbsoluteValue_DWork` in our example, local variables (like `AbsoluteValue_B`), input variables (like `AbsoluteValue_U`), and output variables (`AbsoluteValue_Y`). Additionally, we can restrict the components of the concrete state to take values only over the appropriate sets. For example, Figure 9 shows the state of our implementation model with one additional invariant; it requires that *is_c*1*_AbsoluteValue* takes values from $\{0,1,2\}$.

The retrieve relation maps the execution specific data to the components of *SimulationData*, and the local, input and output variables to components of *SimulationInstance*. The correspondence between the input and output variables is trivial. It is obtained by equating the concrete variables to the abstract variable whose name is the same except for a prefix *v_*. The specification of the relation between the execution specific data of the concrete state and the function *state_status* in *SimulationData* is obtained by using a set comprehension where, for each state identifier *s*, we define a boolean *active* that determines its status from the concrete state. These conditions can be calculated as follows. For a chart name *C*, and each component of *D_Work_C* (in our example, the schema *D_Work_AbsoluteValue*) named *is_active_name* (*is_active_c*1*_AbsoluteValue*, for example), where *name* is the name of state (or chart), we have the following condition.

$$s = name \land active = (\textbf{if } C\_DWork.is\_active\_name > 0 \textbf{ then True else False})$$

For instance, the condition for *is_active_c*1*_AbsoluteValue* equates *s* to *c_AbsoluteValue*, and *active* to **True** or **False** depending on whether the value of *is_active_c*1*_AbsoluteValue* is greater than zero or not.

For each component of the schema *D_Work_C* of the form *is_name*, where *name* is the name of a state (or chart), we formulate a condition in the following way. For each substate *X* of *name*, we define the condition below.

$$s = s\_X \land active = (\textbf{if } C\_DWork.is\_name = C\_IN\_X \textbf{ then True else False})$$

In our example, the condition for the state `P` equates *s* to *s_P*, and *active* to **True** or **False** depending on whether the value of *is_c*1*_AbsoluteValue* is *AbsoluteValue_IN_P* or not. All these conditions are composed in a disjunction as shown in the definition of the retrieve relation for our example in Figure 10.

RetrieveFunction

$$
\begin{array}{l}
P\_AbsoluteValue\_S \\
ConcreteState \\
\hline
state\_status = \{s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet \\
\quad s = s\_N \land active = \textbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = \\
\qquad AbsoluteValue\_IN\_N\ \textbf{then True else False} \lor \\
\quad s = s\_P \land active = \textbf{if}\ AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = \\
\qquad AbsoluteValue\_IN\_P\ \textbf{then True else False} \lor \\
\quad s = c\_AbsoluteValue \land active = \\
\qquad \textbf{if}\ AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue > 0\ \textbf{then True else False}\} \\
state\_history = \{\} \\
v\_u = AbsoluteValue\_U.u \\
v\_y = AbsoluteValue\_B.y
\end{array}
$$

Figure 10: Total surjective functional retrieve relation

Since our example does not contain history junctions, the implementation has no state components that model the state component *state_history* of the chart process. The model of the chart establishes that this state component is the empty partial function, therefore we equate *state_history* to the empty set.

The retrieve relation in Figure 10 is functional because each abstract state component is defined by a function of a component of the concrete state. Since no restriction is imposed on the concrete state for the applicability of the function, the relation is also total. Moreover, for every abstract state A, it is possible find a concrete state that is related to A by the retrieve relation because the functions are invertible.

Using this retrieve relation, we apply the laws of simulation [6, 19] to obtain a *Circus* process *C_P_AbsoluteValue* by data refinement of *P_AbsoluteValue*, and to refine the process *AbsoluteValue* to a process *CAbsoluteValue*, as shown in Figure 11. We define the constant *ss* to increase the readability of *C_P_AbsoluteValue*. This function is defined as the characterisation of *state_status* in Figure 10.

The main action does not change, but components of the actions that it uses are transformed. For example, *Activate* and *InitState* are data refined to operate over the concrete state. Figure 11 shows part of the definition of *CInitState* (the data refinement of the schema *InitState*); it shows the part of the predicate that defines the operation. The actions *condition_P_N*, *Inputs*, and *Outputs* are also data refined; the first has the component *v_u* rewritten to *AbsoluteValue_U.u*, the second has the assignment transformed into an action that assigns a value to a component of a schema binding (as mentioned in Section 3), and the third has the component *v_y* substituted by *AbsoluteValue_B.y*, in accordance with the retrieve relation.

In the next section, we describe how to collapse the parallel composition in *CAbsoluteValue*.

4.2 Normalisation

In this phase, we first collapse the parallelism between the chart and simulator processes in the process *CAbsoluteValue*, and rewrite the main action of the resulting new process to a normal form: an initialisation action, followed by a recursive action. This allows us to focus on the body of the recursion that characterises one step of execution.

process $C\_P\_AbsoluteValue \mathrel{\widehat=}$ **begin**

$StateflowChart$

$identifier = c\_AbsoluteValue$
$states = \{(s\_P, S\_P), (s\_N, S\_N), (c\_AbsoluteValue, C\_AbsoluteValue)\}$
$transitions = \{(t\_P\_N, T\_P\_N), \ldots, (t\_6\_N, T\_6\_N)\}$
$junctions = \{(j5, J5), (j6, J6)\}$

$ss : SID \pfun \mathbb{B}$

$ss = \{s : \mathrm{dom}\, states;\ active : \mathbb{B} \bullet$
$\quad s = s\_N \land active =$ **if** $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N$
$\qquad$ **then True else False** $\lor$
$\quad s = s\_P \land active =$ **if** $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P$
$\qquad$ **then True else False** $\lor$
$\quad s = c\_AbsoluteValue \land active = AbsoluteValue\_DWork.is\_active\_c1\_AbsoluteValue\}$

state $ConcreteState$

$CActivate == [\Delta ConcreteState;\ x? : SID \mid \ldots]$
$CInitState == [ConcreteState' \mid AbsoluteValue\_DWork'.is\_active\_c1\_AbsoluteValue =$ **False** $\land \ldots]$

$\ldots$
$condition\_P\_N \mathrel{\widehat=}$ **if**$((AbsoluteValue\_U.u <_{\mathscr{A}} 0) \neq 0) \longrightarrow \ldots\ldots$
$Inputs \mathrel{\widehat=} (read\_inputs \longrightarrow (i\_u?x \longrightarrow AbsoluteValue\_U\_u(x)))$
$Outputs \mathrel{\widehat=} (write\_outputs \longrightarrow (o\_y!(AbsoluteValue\_B.y) \longrightarrow$ **Skip**$))$

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

end

process $CAbsoluteValue \mathrel{\widehat=} Simulator \llbracket interface \cup \{\!| end\_cycle |\!\} \rrbracket C\_P\_AbsoluteValue$

Figure 11: Data refinement of the processes shown in Figure 4

$$\bullet \left(\begin{array}{l} \left(\begin{array}{l} \big(CInitState\big); \\ \mu X \bullet \left(\mu Y \bullet \left(\begin{array}{l} \left(\begin{array}{l} AllActions \\ \triangle \\ (interrupt\_chart \longrightarrow \mathbf{Skip}) \end{array} \right) ; Y \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) ; X \end{array} \right) \\ [\![\{AbsoluteValue\_B, AbsoluteValue\_DWork\} \mid interface \cup \{\!| end\_cycle |\!\} \mid \{\}]\!] \\ (\mu X \bullet Step ; X) \end{array} \right) \setminus interface$$

Figure 12: Combined main action after merging the two processes.

$$\bullet \left(\begin{array}{l} \big(CInitState\big); \\ \mu X \bullet \left(\mu Y \bullet \left(\begin{array}{l} \left(\begin{array}{l} AllActions \\ \triangle \\ (interrupt\_chart \longrightarrow \mathbf{Skip}) \end{array} \right) ; Y \\ \Box \\ end\_cycle \longrightarrow \mathbf{Skip} \end{array} \right) \right) \setminus interface ; X \\ [\![\ldots]\!] \\ Step \end{array} \right)$$

Figure 13: Main action after the normalisation phase (We abbreviate the parallelism).

We construct the new process by taking the state of the chart process (the simulator process is stateless), and combining the main actions of the chart and simulator processes in the same way the processes were combined, as shown in Figure 12. This is a direct application of the definition of the semantics of process parallelism in *Circus*.

Next, we move the schema action *CInitState* out of the parallel composition, distribute the hiding over the sequential composition, and eliminate the hiding over *CInitState*. The external recursion in the first action of the parallel composition, and the recursion in the second action are then merged. This is possible because *Step* necessarily terminates in a synchronisation over the channel *end_cycle*, since this channel is in the synchronisation set, and this synchronisation stops the inner recursion, and starts a new cycle of the external recursion. We are left with the action in Figure 13, which calls *CInitState*, and recursively executes the parallel composition of a recursion (the recursion over *Y* in Figure 12), and the action *Step*, with the channels in *interface* hidden.

4.3 Parallelism elimination

In this phase, we eliminate the parallelism still embedded in the main action. The parallel action inside the recursion of Figure 12 reads an input event, reads the input variables, executes the chart, writes the outputs, and ends the cycle. In general, the step of execution involves a series of communications over channels that may or may not be in the synchronisation set. Communications over channels not in the synchronisation set are moved outside the parallel composition, and the others are used to select an action from *AllActions* (or trigger an interruption). We proceed as follows to evaluate the communications and remove the parallelism.

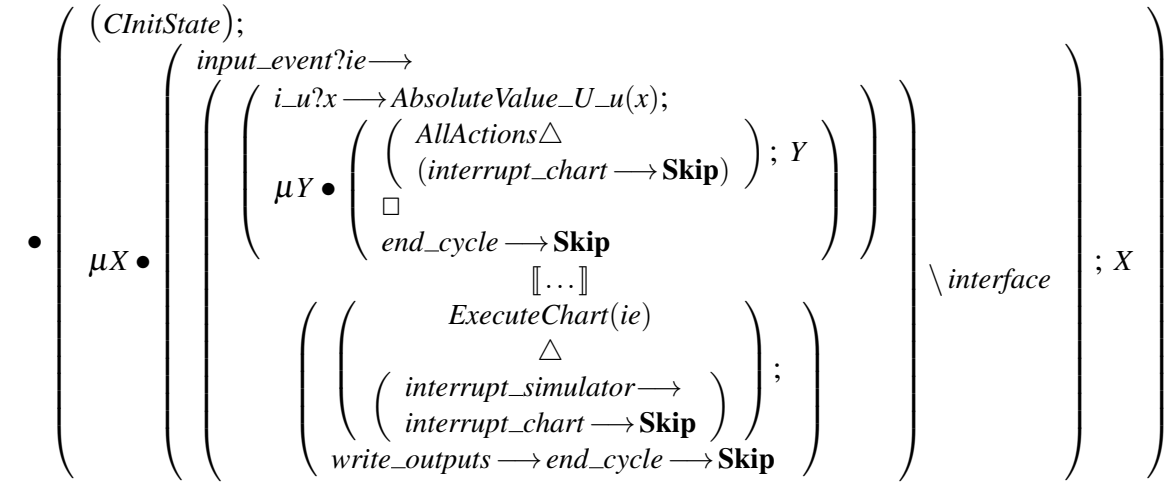

Figure 14: Main action after the communications over *input_event* and *read_inputs* are treated.

By expanding the definition of *Step* in Figure 13, we have two communications one after the other. The first is a communication over the channel *input_event* that is not in the synchronisation set, and the second is a synchronisation over *read_inputs*, which is in the synchronisation set. We move the communication outside the parallel composition, unfold the recursion over Y, and resolve the communication on the channel *read_inputs*. This produces a prefixing action identical to the body of the action *Inputs* in Figure 11. Because the communication is hidden, we eliminate it and obtain the action in Figure 14.

The actions that can be selected can be an atomic information request, as exemplified by *read_inputs*, a non-atomic information request, or a Stateflow action request. In the first case, as already shown, the parallelism can be removed by resolving the communication. In the case of a non-atomic request (for instance, a trigger action of Figure 4), the action is composed of two communications. To eliminate the parallelism we resolve them; this potentially involves resolving a conditional expression that selects the appropriate value to communicate. A Stateflow action request consists of a communication that identifies the appropriate action, a series of *Circus* actions that encode the Stateflow action, and a synchronisation that indicates completion. In this case, the initial communication and the final synchronisation are treated as usual. The encoding of the Stateflow action contains assignments and local event broadcasts. Assignments are moved out of the parallel composition, and broadcasts produce a recursive execution of the chart, which can be treated using the same strategy.

The strategy for eliminating parallelism can be seen as a two level strategy. The first level is guided by the structure of the simulator process, which potentially leads to the execution of an action of the chart process, and the second level is guided by a chart process action that has been executed. Since the simulator process is the same for all charts, we explore its structure to define the refinement strategy.The same is not true for the chart process, but due to the simple structure of the actions in the chart process, we can explore the limited patterns that occur.

At the end, we obtain a main action whose structure is as shown in Figure 16. There, we have already simplified expressions, which is the objective of the next phase.

$$\left(\begin{array}{l}
\left(\mu Y \bullet \left(\begin{array}{l} AllActions\,;\ Y \\ \Box \\ end_cycle \longrightarrow \mathbf{Skip} \end{array}\right)\right) \\
\qquad [\![\ldots]\!] \\
\left(\begin{array}{l} status!(states(c_AbsoluteValue).identifier)?active \longrightarrow \\ \left(\begin{array}{l} \mathbf{if}\, active = \mathbf{True} \longrightarrow \\ \qquad \left(\begin{array}{l} ExecuteActiveChart(states(c_AbsoluteValue), ie); \\ write_outputs \longrightarrow end_cycle \longrightarrow \mathbf{Skip} \end{array}\right) \\ [\!] active = \mathbf{False} \longrightarrow \\ \qquad \left(\begin{array}{l} ExecuteInactiveChart(states(c_AbsoluteValue), ie); \\ write_outputs \longrightarrow end_cycle \longrightarrow \mathbf{Skip} \end{array}\right) \\ \mathbf{fi} \end{array}\right) \end{array}\right)
\end{array}\right) \setminus interface$$

Figure 15: Parallel action of the main action in Figure 14 after further refinement steps.

4.4 Simplification

In the simplification phase, we transform expressions and eliminate unreachable branches of conditional statements. The simplification of expressions takes advantage of the constants that model the structure of the chart, as well as state invariants. For instance, Figure 15 contains an expression that appears frequently in communication resolutions: $states(c\_AbsoluteValue).identifier$. It evaluates to the identifier of the state whose identifier is *c_AbsoluteValue*, but this is exactly *c_AbsoluteValue*, thus we simplify it.

After these simplifications are carried, we obtain a main action as in Figure 16. The last branch of the second conditional has the guard: $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue \neq AbsoluteValue\_IN\_N$. Since *AbsoluteValue_IN_N*=2, the invariant of the concrete state implies that

$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = 1 \vee AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = 0$$

Therefore, since *AbsoluteValue_IN_P* is a constant defined as 1, we replace the above guard with

$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P \vee$$
$$AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = 0$$

This guard is then broken in two, and a new branch is added to the conditional statement. We proceed in this way for every guard defined by a disjunction that checks the status of a state. This simplification is applied whenever the status of a state in a sequential decomposition is checked because our model always includes a branch whose guard is an inequality. It is necessary because, while our model contains only binary conditionals, the implementations may have conditionals with more than two branches.

We traverse the resulting action and for each conditional statement found, we attempt to simplify the guard. If we can simplify a guard to false, we eliminate the branch. If the guard is true, we reduce the whole conditional to the action it guards; this is correct, because all our conditionals are of the form $\mathbf{if}\, b \longrightarrow \ldots [\!] \neg b \longrightarrow \ldots \mathbf{fi}$. If a conditional statement cannot be simplified, it should match a conditional statement in the model of the implementation. For example, the first conditional statement in Figure 16 corresponds to the first conditional statement of the action *AbsoluteValue_output* in Figure 7.

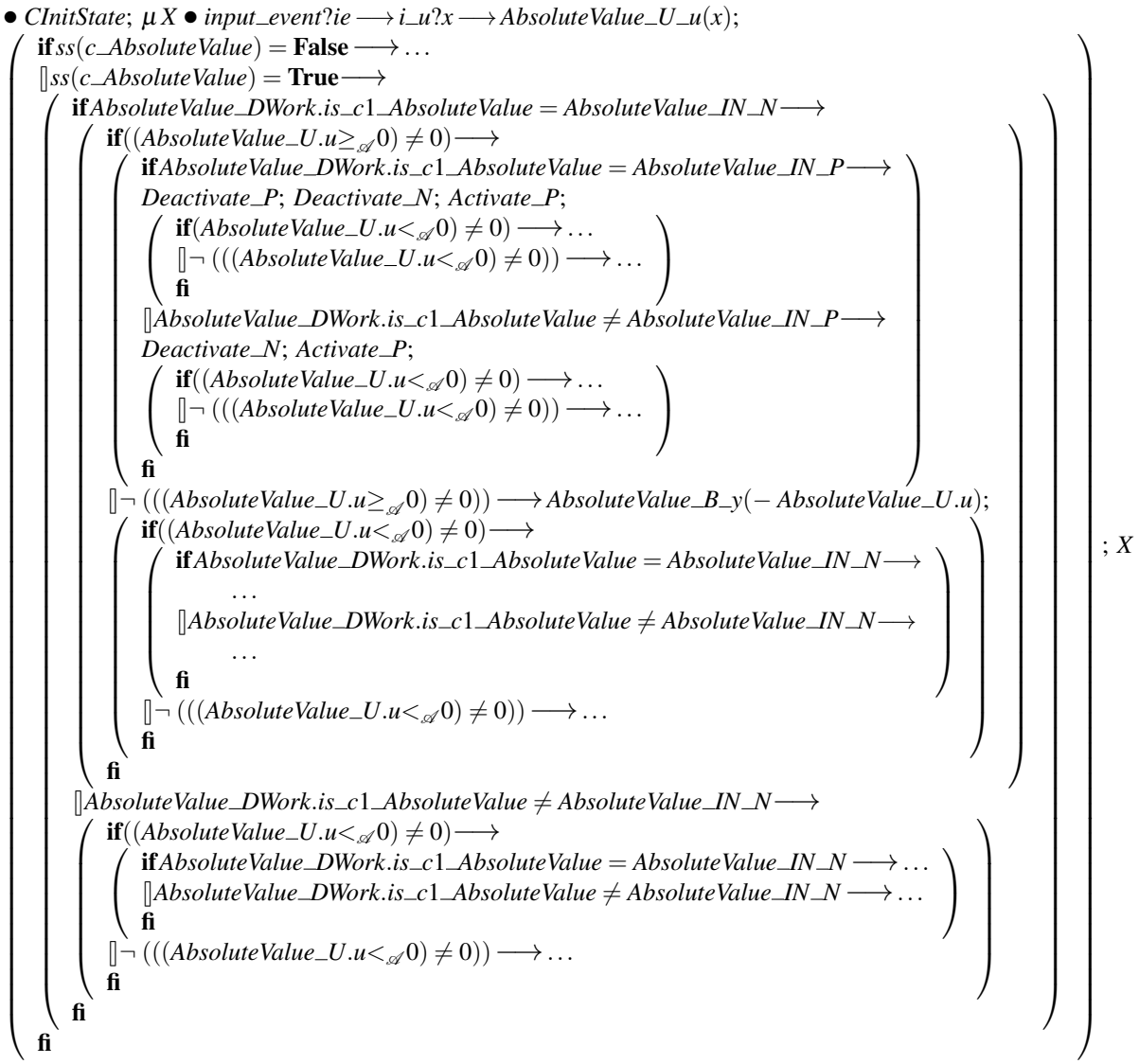

Figure 16: Partially simplified main action of process *AbsoluteValue*.

The fourth conditional statement in Figure 16 is an example of a statement that can be simplified. The guard of the first branch is $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_P$, but this is inside a branch whose guard is $AbsoluteValue\_DWork.is\_c1\_AbsoluteValue = AbsoluteValue\_IN\_N$, and since $is\_c1\_AbsoluteValue$ cannot have both values (and no statement modifies this component between the two branches), the first guard is false, and that branch can be eliminated. In this way, we put the model of the chart in the same shape as the model of the implementation, and once this is achieved, the structuring phase takes place. Formalisation of this strategy using refinement requires a law to introduce assumptions based on the guards of the conditional, laws to distribute and use assumptions, and finally a law to remove an assumption when it is no longer needed. These are standard laws that are valid in *Circus* as shown in [19].

4.5 Structuring

The structuring phase identifies each component of the main action that corresponds to an auxiliary action in the model of the implementation. It introduces this extra action in the process *CAbsoluteValue*, and uses the copy rule to replace the component of its main action by a call to it. The rationale behind this phase is to match the main action to that of the model of the implementation.

For instance, we compare the action *AbsoluteValue_output* in the model of the implementation to the subactions of the main action obtained from the previous phase. We identify a subaction that is a match, introduce its definition, and substitute the name *AbsoluteValue_output* in the main action. The result of all this should be exactly the model of the implementation (as shown in Figure 7 for our example). If this is not the case, the verification has failed: either the program is wrong, or it does not conform to the architectural pattern that we can handle.

The detailed application of this strategy to our example can be found in [14].

5 Conclusion

We have proposed a refinement-based verification strategy for implementations of Stateflow charts. This strategy is guided by the structure of the models of Stateflow charts described in [16]. We have also discussed how such a strategy can take advantage of the architecture imposed on generated code.

We have provided a procedure for obtaining retrieve relations that support the data refinement of the specification in a calculational style, thus rendering the data refinement phase also suitable for automation. In the case of the normalisation and parallelism elimination phases, the possibility of automation stems from the fixed structure of our models. The simplification phase can be semi-automated because the main action consists of a number of nested if-statements, and assumptions generated by the guards of the conditional statements (among others) can be moved into the associated action, potentially falsifying some of the conditions in an internal conditional statement. Finally, the structuring phase can be guided by matching actions from the model of the implementation to subactions of the action being refined.

The refinement strategy for Simulink presented in [5] consists of four steps that systematically collapse the massive parallelism of the diagram specification to match the processes of the implementation model, prove that each of the procedures in the implementation refine the action that specifies it in the corresponding process, and finally, prove that the parallel programs refine the process that specifies the system. The main actions of component processes are put in a normal form where they are defined as the iterative execution of a step that consists of reading the inputs in interleaving, calculating the outputs and updating the state, writing the outputs in interleaving, and synchronising on the channel *end_cycle*.

Our strategy has a similar nature, however, it is worth mentioning some important differences. Our models of Stateflow charts owe their parallelism to the separation between the structure of the model and the operational semantics of the simulator, not to any implicit or explicit parallelism in the chart. Therefore, while in [5] collapsing the parallelism is guided by the implementation model, in our strategy it is performed until there are no parallel actions left. The beginning of our parallelism elimination phase is similar to the process of putting the main action in a normal form in the strategy for Simulink diagrams. The equivalent step in our model is, however, not as linear as in [5]. The decision to end the cycle comes from the action inherited from the simulator process. Nevertheless, by the end of the parallelism elimination phase, we have a process whose main action is in the normal form of [5]. This suggests that not only our models can be integrated to the models of Simulink diagrams, but also that our strategy can be used to put a Stateflow block in the normal form prescribed in [5]. This will support the integrated use of these verification techniques for Simulink diagrams involving Stateflow blocks.

There are several approaches to the formal analysis of Stateflow diagrams. These works aim at the analysis of diagrams, not of their implementations. Operational and denotational semantics are proposed in [11] and [10]; these support static analysis, interpretation, and compilation of Stateflow charts. Translations of Stateflow into notations that support model checking are presented in [1], [26], [22], and [7]. Verification in these approaches is based on temporal logics and bisimulation, rather than refinement, thus verification of implementations is not the objective. An approach based on Z to verify that the chart satisfies a set of requirements of the system being modelled is presented in [28]. However, it places strong restrictions on the Stateflow notation.

Olderog [18] integrates three views of a system (trace specification, process algebra and Petri nets) by formalising a relation between them. While this approach ends in a graphical notation, namely Petri nets, we take the opposite direction: from a graphical notation to a program. In [2], the refinement calculus is adapted to Simulink diagrams, but they do not cover Stateflow charts, and their goal is not the verification of implementations, but the development of diagrams from contracts. In [20], a semantics for μ-Charts is constructed in Z, and a notion of refinement of μ-Charts is derived from the existing Z refinement calculus. This approach is similar to that presented in [16], where we define a semantics of Stateflow charts in *Circus*, thus allowing the *Circus* refinement calculus to be applied, but it differs in the sense that we focus in a industrial non-formal notation, while the μ-Charts notation is a simplification of Statecharts mainly used in academia. Moreover, our approach goes beyond the application of the refinement calculus to Stateflow charts; it addresses the problem of automation of the refinement process.

As far as we know, this is the first work to address the issue of verification of implementations of Stateflow charts. Moreover, as explained in detail [15], our models of Stateflow charts used as the base for the verification eliminate many of the restrictions imposed in other formalisations.

Given the generality of our refinement strategy, we believe it scales well to modified implementations. In particular, an implementation that does not modify the use of the variable *AbsoluteValue_DWork* should still be amenable to the specialisation of the refinement strategy discussed in Section 4.1. Moreover, our strategy can be used as a preliminary phase in the verification of parallel implementations, as all the parallelism eliminated by the strategy derives from the structure of the model.

As future work, we will address the issue of verification of parallel implementations of Stateflow charts. Parallel implementations are not common, and as far as we know there are no code generators that produce parallel implementations. We will extend the current strategy to allow the verification to be carried out after the introduction of parallelism in the implementation using fixed design patterns.

References

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

[2] P. Boström et al. (2007): *Stepwise Development of Simulink Models Using the Refinement Calculus Framework*. In: *ICTAC 2007*, *LNCS* 4711, pp. 79–93, doi:10.1007/978-3-540-75292-9_6.

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

[4] P. Caspi et al. (2003): *From Simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications*. *ACM SIGPLAN Notices* 38(7), pp. 153–162,doi:10.1145/780753.780754.

[5] A. L. C. Cavalcanti et al. (2011): *From control law diagrams to Ada via Circus*. *FAC* , pp. 1–48, doi:10.1007/s00165-010-0170-3.

[6] A. L. C. Cavalcanti et al. (2003): *A Refinement Strategy for Circus*. *FAC* 15(2 - 3), pp. 146–181, doi:10.1007/s00165-003-0006-5.

[7] C. Chen (2010): *Formal Analysis for Stateflow Diagrams*. In: *SSIRI-C '10*, pp. 102–109, doi:10.1109/SSIRI-C.2010.29.

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

[9] DO-178b (1992): *Software Considerations in Airborne Systems and Equipment Certification*.

[10] G. Hamon (2005): *A denotational semantics for Stateflow*. In: *EMSOFT*, ACM, pp. 164–172, doi:10.1145/1086228.1086260.

[11] G. Hamon & J. Rushby (2004): *An Operational Semantics for Stateflow*. In: *FASE*, *LCNS* 2984, Springer-Verlag, Barcelona, Spain, pp. 229–243, doi:10.1007/s10009-007-0049-7.

[12] D. Harel (1987): *Statecharts: A Visual Formalism for Complex Systems*. *SCP* 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.

[13] R. Lublinerman et al. (2009): *Modular code generation from synchronous block diagrams: modularity vs. code size*. In: *POPL' 09*, ACM, pp. 78–89, doi:10.1145/1480881.1480893.

[14] A. Miyazawa & A. L. C. Cavalcanti: *Refinement of the AbsoluteValue chart*. Available at `www.cs.york.ac.uk/~alvarohm/refinement`.

[15] A. Miyazawa & A. L. C. Cavalcanti (2011): *A formal semantics of Stateflow charts*. Technical Report YCS-2011-461, University of York.

[16] A. Miyazawa & A. L. C. Cavalcanti (2011): *Refinement-oriented models of Stateflow charts*. *SCP* (to appear).

[17] C. C. Morgan (1994): *Programming from Specifications*. Prentice Hall International Series in Computer Science.

[18] E.-R. Olderog (1991): *Nets, terms and formulas: three views of concurrent processes and their relationship*. Cambridge University Press, New York, NY, USA, doi:10.1017/CBO9780511526589.

[19] M. V. M. Oliveira (2006): *Formal Derivation of State-Rich Reactive Programs using Circus*. Ph.D. thesis, Department of Computer Science - University of York, UK. YCST-2006-02.

[20] G. Reeve & S. Reeves (2006): *Logic and refinement for charts*. ACSC '06, Australia, pp. 13–23.

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

[22] N. Scaife et al. (2004): *Defining and translating a "safe" subset of simulink/stateflow into lustre*. In: *EMSOFT*, ACM, pp. 259–268, doi:10.1145/1017753.1017795.

[23] The MathWorks,Inc.: *Real-Time Workshop*. `www.mathworks.co.uk/products`.

[24] The MathWorks,Inc.: *Simulink*. `www.mathworks.co.uk/products`.

[25] The MathWorks,Inc.: *Stateflow and Stateflow Coder 7 User's Guide*. `www.mathworks.co.uk/products`.

[26] A. Tiwari (2002): *Formal semantics and analysis methods for Simulink Stateflow models*. Technical Report, SRI International. `www.csl.sri.com/~tiwari/stateflow.html`.

[27] A. Toom et al. (2008): *Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos*. In: *ERTS'08*.

[28] I. Toyn & A. Galloway (2005): *Proving properties of Stateflow models using ISO Standard Z and CADiZ*. In: *ZB-2005, vol. 3455 of LNCS*, pp. 104–123,doi:10.1007/11415787_7.

[29] J. C. P. Woodcock & A. L. C. Cavalcanti (2002): *The Semantics of Circus*. In: *ZB 2002: Formal Specification and Development in Z and B*, *LCNS* 2272, Springer-Verlag, pp. 184–203,doi:10.1007/3-540-45648-1_10.

[30] J. C. P. Woodcock & J. Davies (1996): *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc.