Masarykova univerzita
Fakulta informatiky

Modelling Stateflow Diagrams for Verification Purposes

Bachelor's thesis

Pavla Kratochvílová

Brno, jaro 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. RNDr. Jiří Barnat, Ph.D.

Acknowledgement

First, I would like to thank doc. RNDr. Jiří Barnat, Ph.D. for advising my thesis and Mgr. Tomáš Kratochvíla for suggesting this topic and discussing technical details. I would also like to thank Bc. Pavel Holica for advice concerning programming in Python.

Last but not least, I would like to thank my family for supporting me in my studies.

Abstract

The thesis presents transformation from Stateflow diagrams into the native input language of DIVINE model checker to make formal verification of safety critical systems possible. The automatic transformation tool was implemented and compared with other approaches.

Keywords

Formal verification, model checking, DiVinE, model based development, transformation, Simulink, Stateflow

Contents

Introduction

Stateflow [Mat13b] is widely used in industry for modelling and simulating systems, which are often safety-critical and their failure may cause injury, death, environmental damage or material loss, hence their correctness has to be guaranteed. However, Stateflow provides only very limited formal verification by Simulink Design Verifier [Mat]. Several transformations of Stateflow diagrams to input languages of various formal verification tools have been created, but they often do not support desired features of Stateflow or cannot be obtained (are either proprietary, or not implemented). Moreover, no transformation tool supports the new Simulink default SLX format [Mat14].

The objective of this thesis is to create a tool for transforming Stateflow diagrams into a native input language of model checker DIVINE [BBČ+06]. The goal was to enable formal verification using high-performance scalable model checker, hence the DIVINE was chosen.

Chapter 1 describes Harel statecharts, their variant, Stateflow diagrams, formal verification and the DIVINE model checker. In Chapter 2, other transformations of Stateflow diagrams to input languages of various formal verification tools are listed. Principles of transformation of Stateflow diagrams into native input language of DIVINE are introduced in Chapter 3 and implementation details are presented in Chapter 4. Chapter 5 describes randomized testing, which was performed to evaluate the work, and also comparison with other related work.

Chapter 1

Preliminaries

1.1 State Diagrams

A *state diagram* [Boo67] is a type of diagram used for describing behaviour of finite state systems. Many forms of state diagrams exist and they may differ in sets of elements used for describing system behaviour. A classic form of a state diagram is extended version of directed graph containing set of nodes denoting *states* and directed edges denoting *transitions* between states. In addition, states can be flagged as *default*, *accepting*, *rejecting*, etc. A *default state* denotes where the system starts and is often indicated by a *default transition*, which is a transition without a source state. To model reactive systems, state diagram needs to contain *events* and means to react to them. When particular event occurs, a specified action is taken.

Graphical representation of finite state machine is an example of a simple state diagram. Events are represented by symbols and their occurrence is expressed by input sequence. Reactions to events are evaluations of state-transition function.

Although classic state diagrams are sufficiently expressive for modelling complex reactive systems, they are not very suitable for such usage. They lack elements providing structure and the number of states and transitions grows exponentially depending on number of parameters, because of the need to have distinct state for every valid combination. To deal with these problems a form of state diagram called Harel statechart [Har87] has been created.

1.1.1 Harel Statechart

Harel statecharts (hereinafter just statecharts) extend classical state diagrams with concepts such as hierarchy of the states, parallelism and history, in order to improve readability of the resulting diagram.

Events, Conditions, Actions

Transitions in a statechart are labelled with an event, triggering of which enables execution of transition. Optionally the label can also contain a *condition* determining which transition will be performed in case of ambiguity. Additionally, transitions and states can also be labelled with instructions, enabling event generation and assignments to variables that are used in conditions. Those instructions can be either instantaneous *actions* associated with a transition or with entering or exiting a state, or persisting *activities* which remain in effect the whole time while being in specified state.

Hierarchy

Instead of "flat" arrangement of states, it is often useful to group states with common properties into a *superstate* to make the diagram more comprehensible. Grouping can be applied repeatedly, resulting in diagram with more granular hierarchy. When the system is in one of the nested states, it is also in respective superstate (and their superstates recursively), thus any actions and activities of the superstates take place.

Transitions can lead from and to any level of hierarchy. If there is an outgoing transition from a superstate, then upon taking this transition the system also leaves current substate of the superstate. If there is an incoming transition to a superstate, then upon taking this transition the system enters specified default state for this superstate.

The notion of *default state* is now extended to denote both an initial state of a statechart and an initial state of a superstate. Both types of default states are indicated by a *default transition*.

The closest superstate of a state is also called *parent* and its closest substate is called *child*. Parent of a transition is defined as the lowest-level state that contains both the source and the destination of the transition.

History

Another way of entering a group of states, besides entering the default state, is utilizing history of the group. To any level in the state hierarchy a *history junction* can be added. When entering such a junction the last visited substate is entered (instead of the default one). Entering by history can be either shallow or deep. In case of the shallow history junction, the last visited substate is entered and if it has nested states the default one is entered. In case of the deep history junction, the history applies all the way down to the states of the lowest level.

Junctions

If more transitions share source or destination, compound transition using a *junction* can be created.

Parallelism

Another important concept of statecharts is *AND decomposition of states*. The state is divided into components (distinct groups of substates) and the system is in all of those components at the same time. When an event occurs, all transitions in those components that can take place (start in active state, are labelled with respective event and any possible conditions hold) apply concurrently. Any AND component with no valid transition stays in its current state nonetheless.

Due to the nature of AND decomposition, behaviour of transition entering or leaving AND decomposed states ought to be described. When entering an AND decomposed state, for each component it can be specified what state (or junction) is entered, otherwise the default state is chosen. Similarly when leaving AND decomposed state, for each component it can be specified in what state (or junction) is the transition enabled, otherwise it is enabled in all states.

1.2 Stateflow

Stateflow is a tool developed by the MathWorks, Inc, extending Simulink [Mat13a] with environment for modelling and simulating reactive systems. They are modelled as Stateflow diagrams, which are a variant of aforementioned statecharts. A Stateflow diagram can be included in Simulink model as one of the blocks interacting with other Simulink components using input and output signals.

The syntax and semantic of Stateflow diagrams is similar to the one used in statecharts, but differs in details and contains additional elements. As there is a great number of minor differences, only the most important will be covered.

Determinism

To begin with, Stateflow is purely deterministic, i.e. has exact order (either implicit or explicit) of transitions for execution. Events in transition labels are optional, and if missing, occurrence of any event enables transition execution.

No Concurrency

Since Simulink simulation ensures repeatability and Stateflow diagrams run on a single thread, everything is performed sequentially. For example, when system enters a state and its substate at once, all *entry actions* should take place concurrently in statecharts. In Stateflow diagram, this is done by executing *entry actions* of the superstate prior to executing *entry actions* of the substate. Greater impact of this limitation is AND decomposed state processing. In Stateflow diagrams this is resolved by ordering AND components.

Early Return Logic

Another two semantic differences are worth mentioning, not for their significance in state modelling, but for their counter-intuitive behaviour, especially when combined:

1. There are two different types of actions within transitions, which differ in execution time: *condition action* and *transition action*. The syntax for transition label is:

   `event[condition]{condition_action}/transition_action`

   Condition action executes immediately after the transition is determined to be performed, but before the active state is left, whereas *transition action* executes after leaving the active state. This is important for transitions leading to a connective junction, since *condition action* is executed even if the whole compound transition is not performed.

2. There can be events generated by actions. Due to the single-threaded design of Stateflow, if an event is generated, Stateflow diagram interrupts current activity in order to process the event and for example executes transition triggered by this event. Depending on where the event is generated, the following can happen:

   - If the event is generated in *entry* or *during action*, this simply results in executing a valid transition from respective state.
   - If the event is generated in *exit action* or *condition action*, the transition is not completed and another transition from the same state can be triggered. Provided the new transition and the interrupted transition happen to be the same, this leads to recursive behaviour.

- If the event is generated in *transition action*, the transition is not completed and another transition from an active state can be triggered. However, in this case the source of the interrupted transition is no longer active state, and therefore only transitions from superstates or parallel states can take place.

New Elements

Stateflow diagram can contain graphical functions, MATLAB functions, Simulink functions, truth tables, etc.

1.2.1 Stateflow XML Document

Simulink stores its models in MathWorks proprietary file formats MDL and SLX [Mat14]. SLX was introduced in Simulink version R2012a and it has been the default file format since version R2012b. It is compressed file archive containing information about model in XML format and other auxiliary files. The XML document contains all elements of a Stateflow diagram including information about their mutual relations.

For example, state hierarchy is represented in a tree structure inside the XML document. Sources, destinations and labels of transitions are stored as child elements of corresponding transitions. States are stored in a similar way, containing their label as a child. In addition, there is also non-functional information about Stateflow elements stored, for example their position and shape. With an exception of action language setting (MATLAB or C syntax), they affect neither the model nor its representation and therefore are not relevant for verification of Stateflow diagrams.

All Simulink files contain only the syntax of the system and no semantics of the system behaviour is provided.

1.3 Formal Verification

Formal verification is a process of proving or disproving correctness of a system in respect to formal specifications or properties by providing formal proof on an abstract model of the system. In contrast to simulation and testing, the formal verification considers every possible behaviour of the system and in case of constructing a proof, the respective property of system is guaranteed.

Model checking [CGP99a] is one of the formal verification methods. It is a technique for verifying if a finite state concurrent system satisfies given

properties and this can be automatically performed by a *model checker*, software tool for model checking. Both the system and the properties need to be specified in a formalism accepted by a model checker. It is common to formalize properties as a set of formulae in some temporal logic. Model checker then exhaustively explores all the reachable states of the system and either proves given property or provides a counterexample.

1.3.1 DIVINE

The DIVINE model checker is a tool for parallel verification of concurrent systems. It is capable of parallel computation in both single-host and multi-host environment utilizing shared memory or network communication respectively.

Properties are specified as formulae of linear temporal logic [CGP99b].

Systems can be modelled in several different formats, one of which is DVE modelling language. DVE was designed to describe systems made of processes communicating via buffered or synchronous channels. Processes contain local variables, states, transitions and assertions. Transitions can be guarded by a condition (also called *guard*), which determine whether the transition can be executed, and if there are more available transitions, one is chosen nondeterministically. Transitions can also have *effects*, which are assignments to the variables.

Formal definition of DVE 2.0 syntax is given in [Kri13].

Chapter 2

Related Work

There are several attempts to transform Stateflow diagrams to an input language of some formal verification tool. Automatic code generation is often implemented, however, the verification of implementations is usually not supported, as the semantics of Stateflow is only informally and partially described in Stateflow User's Guide [Mat13c]. Also all the automatic transformations found require old version of the MDL file format as an input.

List of the transformations found:

- The mdl2smv tool [BBK99] translates Stateflow diagrams into an input language for Cadence SMV model checker [MCBL]. The translation does not support: *condition actions* in transitions, actions within states, connective junctions, graphical or MATLAB functions, temporal conditions, and event broadcasting.

- NuSMV GUI [FCoGoT] translates Stateflow diagrams into an input language of NuSMV model checker. The tool provides interface to enable editing of the translated systems graphically. The translation does not support: hierarchical states, events, and non-boolean variables.

- The translation to communication push-down automata for Symbolic Analysis Laboratory (SAL) is presented in [Tiw02]. The translation cannot be obtained and does not support: sequential execution order of AND decomposed states.

- The sf2lus tool [SSC+04] can translate Stateflow diagrams into a synchronous programming language Lustre, which can be model checked by Lesar [HLR92]. They provide well written examples. The translation does not support: inter-level transitions, MATLAB version 14 (MATLAB 7.0) or newer. This is crucial, since newer Matlab versions cannot convert Simulink models into more then 10 years old version.

- Simulink/Stateflow Analyser [TG05] can translate Stateflow diagrams into ISO Standard Z notation. The tool makes use of typechecker and theorem prover CADiZ [TM93].
- Formal specification of Stateflow diagrams in Circus notation is proposed in [Miy12].
- The translation of Stateflow diagrams into CSP# modelling language is presented in [CSL+12]. Model checker PAT [NUoS] is used for verification.
- The ForReq tool [BBB+12] translates Simulink models into Common Explicit-State Model Interface for DiVinE model checker, and SMV for NuSMV model checker. The tool is Honeywell proprietary and cannot be publicly obtained.
- Manual construction of Simulink models in an invariant checker is informally presented in [SCBR01]. The translation is not implemented.
- The translation into Promela [Lei08] require Stateflow diagram to be manually redrawn in VIP tool. Model checker SPIN [H+] is used.

Experimental capability evaluation, which shows practical usage of these transformation tools, is in section 5.3.

Chapter 3

Transformation

Since the Stateflow diagrams contain features that DVE modelling language lacks, direct transformation cannot be used. Such features are for instance: actions on states, events, hierarchical states, connective and history junctions, ordering of transitions, graphical functions etc. While some of these features can be easily emulated in DVE language, others are difficult to model in DVE and yet are rarely used in industry. Like other works that provide transformations of Stateflow diagrams into various formalisms (mentioned in Chapter 2), the transformation in this thesis supports only a subset of Stateflow that covers majority of industrial applications.

The covered subset of Stateflow includes: states with *entry*, *during* and *exit actions*, transitions with conditions, *condition actions* and *transition actions*, state hierarchy, labelled default transitions, implicit `tick` event, and variables of integer types.

Unsupported subset of Stateflow includes: *bind actions*, connective and history junctions, AND decomposition of states, events (except for implicit `tick`), variables of real number types, functions, and boxes.

Hereinafter, let the Stateflow diagram be considered as a diagram with the aforementioned limited functionality.

Since the specification of Stateflow semantics is available only as an informal description in the Stateflow User's Guide, the transformation into DVE modelling language will be also described in informal manner. The semantics that is not clearly covered by the Stateflow User's Guide were deduced from observation of Simulink simulator results for dedicated models. The transformation algorithm is based also on these observations.

3.1 Definition of Stateflow Diagram

For the purpose of the transformation description let the Stateflow diagram be defined as a triple (S, T, V), where:

- S is a finite set of states.

- T is a finite set of transitions.
- V is a finite set of variables.

State is a quadruple $(A_{en}, A_{du}, A_{ex}, par)$, where:

- A_{en}, A_{du} and A_{ex} are finite lists of actions.
- par is a parent state (or null if the state is on the highest level in the hierarchy).

Transition is a septuple $(src, dst, C, A_{cond}, A_{trans}, ord, par)$, where:

- src is a source state (or null in case of a default transition).
- dst is a destination state.
- C is a finite set of conditions.
- A_{cond} and A_{trans} are finite lists of actions.
- ord is an execution order of the transition.
- par is a parent state (or null).

Let a default transition be called *initial* when the transition has no parent.

3.2 SF2DVE Transformation

This section describes proposed transformation from Stateflow SLX format into DVE language accepted by DIVINE model checker. Each Stateflow diagram in given Stateflow model is transformed into single process in DVE language.

The transformation starts with importing XML elements (and their attributes) into internal data structures that correspond to aforementioned Stateflow diagram definition. Variable declarations are removed from labels and are added to global variables of the diagram. In order to preserve their scope, unique prefix is generated and every occurrence of the variable is substituted with the new name.

The main algorithm is summarized in 3.1, where its functions are detailed in following sections.

The text "system async;" is printed at the end of the DVE model, since only asynchronous system is supported by DVE 2.0 language (specified in [Kri13]).

Algorithm 3.1 SF2DVE($stateflow$)

1: **for each** diagram d in $stateflow$ **do**
2: $\quad\overline{d} \leftarrow$ FLATTEN(d)
3: $\quad$PRINTPROCESS($\overline{d}$)
4: **end for**
5: $print$("system async;")

3.3 Flattening

Since DVE language does not support states hierarchy, each diagram needs to be flattened first, i.e. transformed into an equivalent diagram without hierarchy. The flattening is performed in the FLATTEN function, which also makes several other modifications. These modifications are required for the purposes of the modelling in DVE language and do not change the model behaviour.

The FLATTEN function is detailed in the following sections and its pseudocode is in the algorithm 3.2.

3.3.1 States

During the Stateflow execution, exactly one *leaf state* (state with no children) of the state hierarchy is active at a time. The exceptions are Stateflow diagrams containing AND decomposed states, which are not supported by the transformation. The superstates cannot be modelled in DVE language as standalone state, hence all leaf states need to carry the properties of their superstate. Therefore, the flattened diagram will contain only the leaf states of the original diagram.

Both Stateflow diagram and DVE process can contain conditions and actions. However, there are several types of actions in Stateflow diagram. In DVE process, only transitions can carry out actions, hence all types of Stateflow actions need to be associated with some transition in DVE. Therefore, the states in the flattened diagram have no *entry* or *exit actions* and its incoming and outgoing transitions will perform the actions instead.

However, the states in the flattened diagram have *during actions* that are composed of the *during actions* of the original leaf state along with the *during actions* of all its superstates. These *during actions* are sorted the same way as they are executed in Stateflow, starting with the *during actions* of the highest-level superstate and ending with the *during actions* of the leaf state. These *during actions* will be emulated by special loop

transitions. The priority of these transitions cannot be determined during the FLATTEN function as all other transitions need to have higher priority. Therefore, these transitions are created later, during the PRINTPROCESS function.

Two newly generated states $start$ and $error$ have to be added to the flattened diagram in order to cover special Stateflow states, which are not explicitly displayed in Stateflow diagram. The $start$ state is hidden initial state of the Stateflow diagram as there can be actions taken before entering any default state of the Stateflow diagram. The $error$ state corresponds to specific Stateflow error state that is reached when some default transition shall be performed and none is valid.

Algorithm 3.2 FLATTEN$(diagram)$

1: $\overline{diagram} \leftarrow$ new diagram
2: **for each** leaf state s in $diagram.S$ **do**
3: $\quad \overline{s} \leftarrow$ new state
4: $\quad \overline{s}.A_{du} \leftarrow s.A_{du}$
5: $\quad parent \leftarrow s.par$
6: $\quad$ **while** $parent$ is not null **do**
7: $\quad\quad \overline{s}.A_{du} \leftarrow parent.A_{du} + \overline{s}.A_{du}$
8: $\quad\quad parent \leftarrow parent.par$
9: $\quad$ **end while**
10: $\quad \overline{diagram}.S.add(\overline{s})$;
11: **end for**
12: Add new state $start$ to $\overline{diagram}.S$
13: Add new state $error$ to $\overline{diagram}.S$
14: **for each** transition t in $diagram.T$ **do**
15: $\quad \overline{diagram}.T = \overline{diagram}.T \cup$ PROCESSTRANSITION$(t, diagram)$
16: **end for**
17: $\overline{t} \leftarrow$ new transition
18: $\overline{t}.src \leftarrow start$
19: $\overline{t}.dst \leftarrow error$
20: $\overline{t}.ord \leftarrow (1, 0, 0)$
21: $\overline{diagram}.T.add(\overline{t})$
22: $\overline{diagram}.V \leftarrow diagram.V$
23: **return** $\overline{diagram}$

In addition, a transition from the $start$ state to the $error$ state is created to model the situation when no initial default transition is available. Therefore, this transition is assigned with an *order* that is lowest among initial

default transitions. The transition priorities in general are covered in section 3.3.4.

3.3.2 Transitions in General

One Stateflow transition is generally transformed into several transitions in the flattened diagram. The transformation of a Stateflow transition is performed by the PROCESSTRANSITION algorithm, which is divided into two parts:

- Algorithm 3.3 that solves transitions in general is described in this section.
- Algorithm 3.6 that solves transition sources and destinations, priorities, and composing of transformed transitions is described in following sections.

The default transitions of a superstate indicate which of its substates shall be entered. Since Stateflow stops only in the leaf states, the transitions entering the superstate are extended with the behaviour of the default transitions. Details are covered in section 3.3.3.

When a transition is performed, its source state is exited and its destination state is entered. Moreover, as the source is no longer active, its superstates are also exited and the superstates of the destination are entered. However, the superstates that are common to both the source and the destination stay active the whole time. Additionally, before the transition is performed and the active state is exited, all superstates of the source perform their *during actions*, and then the *condition actions* of the transition are executed.

Upon taking a transition, *exit actions* of all exited states are executed first, followed by the *transition actions* of the transition and finally the *entry actions* of all entered states.

The exceptions are transitions from superstates to one of their substates or vice versa. In both cases, the superstate is a parent of the transition and stays active the whole time, hence its *entry* or *exit actions* are not executed. However, the *during actions* of the superstate are performed. In the first case, the transition is called *inner transition*.

When the transition in the figure 3.1 is transformed, the resulting transition in the figure 3.2 has to preserve all associated actions of the original unflattened states. In Stateflow, the *exit actions* of a state are executed prior to the *exit actions* of its superstate, whereas *entry* or *during actions* are executed other way around.

Algorithm 3.3 PROCESSTRANSITION$(t, diagram)$ part 1

1: **if** t is default transition and not initial **then**
2: **return** empty set
3: **end if**
4: $A_{cond} \leftarrow t.A_{cond}$
5: $A_{trans} \leftarrow t.A_{trans}$
6: **if** t is not default transition **then**
7: **if** $t.src = t.par$ **then**
8: $A_{du} \leftarrow t.src.A_{du}$
9: $A_{ex} \leftarrow$ new list
10: **else**:
11: $A_{du} \leftarrow$ new list
12: $A_{ex} \leftarrow t.src.A_{ex}$
13: **end if**
14: $parent \leftarrow t.src.par$
15: **while** $parent$ is not null **do**
16: $A_{du} \leftarrow parent.A_{du} + A_{du}$
17: **if** $parent$ is not superstate of t **then**
18: $A_{ex} \leftarrow A_{ex} + parent.A_{ex}$
19: **end if**
20: $parent \leftarrow parent.par$
21: **end while**
22: $A_{cond} \leftarrow A_{du} + A_{cond}$
23: $A_{trans} \leftarrow A_{ex} + A_{trans}$
24: **end if**
25: **if** $t.dst = t.par$ **then**
26: $A_{en} \leftarrow$ new list
27: **else**:
28: $A_{en} \leftarrow t.dst.A_{en}$
29: **end if**
30: $A_{en} \leftarrow t.dst.A_{en}$
31: $parent \leftarrow t.dst.par$
32: **while** $parent$ is not null **do**
33: $A_{en} \leftarrow parent.A_{en} + A_{en}$
34: $parent \leftarrow parent.par$
35: **end while**
36: $A_{trans} \leftarrow A_{trans} + A_{en}$

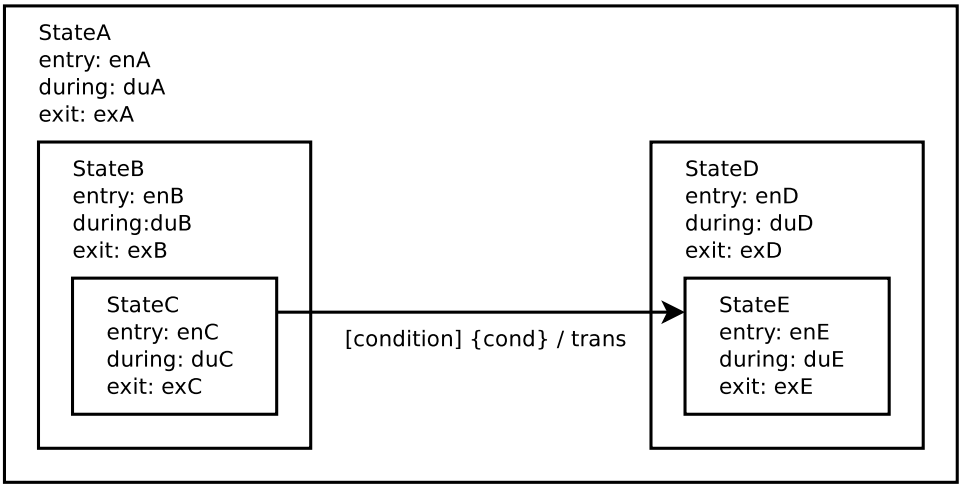

Figure 3.1: Actions upon taking transition

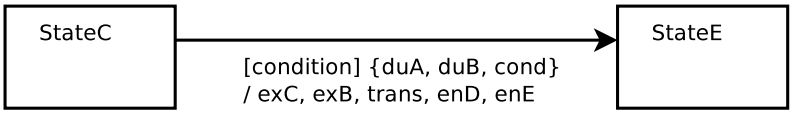

Figure 3.2: Actions upon taking transition (Flattened)

3.3.3 Sources and Destinations

Generally, the source and the destination of a transition can be a state at any level of the state hierarchy. If there is an outgoing transition from a superstate, it can be performed whenever one of its leaf states is active. In case of incoming transition, one of its default transition is performed. In the flattened diagram, a transition for every pair of possible sources and destinations, is created and its *transition actions* are extended with additional actions.

To determine all possible sources and destinations, recursive functions FINDSRCPATHS and FINDDSTPATHS are called.

The FINDSRCPATHS function searches all substates of the source and returns list of the leaf states. For each leaf state the list of its *exit actions* is attached, along with *exit actions* of its supestates on the path to the original source. The function is shown in the algorithm 3.4.

The FINDDSTPATHS function recursively searches the default substates of the destination. For each leaf state reachable via default transitions from

Algorithm 3.4 FINDSRCPATHS(src, A, P)

1: **if** src is a leaf state **then**
2: **return** $P + [(src, A)]$
3: **end if**
4: **for each** state s that is child of src **do**
5: $A \leftarrow s.A_{ex} + A$
6: $P \leftarrow$ FINDSRCPATHS(s, A, P)
7: **end for**
8: **return** P

the destination a list of corresponding actions and set of corresponding conditions is attached. The actions consist of the *entry actions* of the leaf state and all its superstates on the path to the original destination and also *condition* and *transition actions* of the default transitions on the path. The attached set of conditions consists of the conditions of all default transitions on the path.

In Stateflow, if two or more transitions are valid, the one with the highest priority is performed, whereas in DVE the transition is chosen nondeterministically. To emulate the ordering of transitions, conditions of a transition need to be extended with negated conditions of all transitions with higher priority. The ordering also applies for default transitions, hence the set of conditions needs to be extended accordingly.

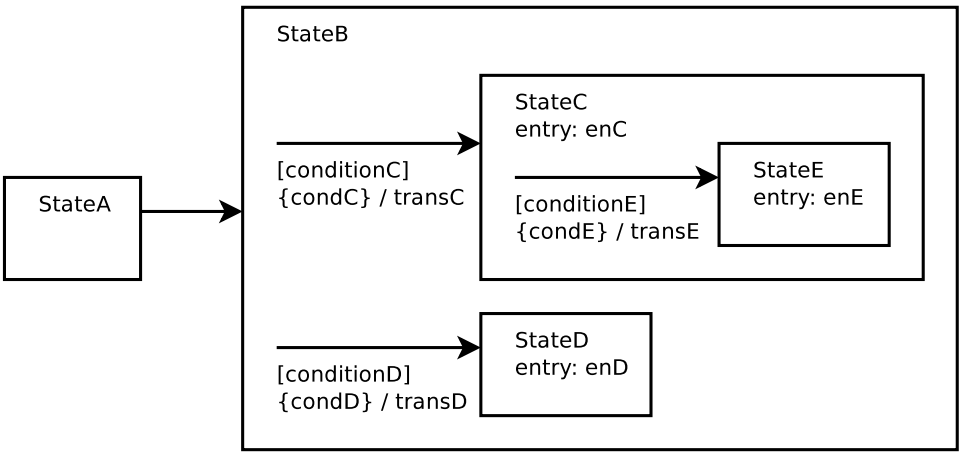

Figure 3.3: Destination paths

When the FINDDSTPATHS function searches the Stateflow diagram depicted in the figure 3.3, the following list is produced:

```
[(StateE, {conditionC, conditionE}, [condC, transC, enC,
   condE, transE, enE]),
(error, {conditionC, not conditionE}, [condC, transC, enC]),
(StateD, {conditionD}, [condD, transD, enD]),
(error, {not conditionC, not conditionD}, [])]
```

The pseudocode of the FINDDSTPATHS function is shown in the algorithm 3.5.

Algorithm 3.5 FINDDSTPATHS(dst, C, A, P)

1: **if** dst is a leaf state **then**
2: **return** $P + [(dst, C, A)]$
3: **end if**
4: $T_{def} \leftarrow$ default transitions for dst sorted by execution order
5: $C_{neg} \leftarrow$ new list
6: **for each** transition t in T_{def} **do**
7: $C \leftarrow C + t.C + C_{neg}$
8: $C_{neg}.add(\text{NEGATE}(t.C))$
9: $A \leftarrow A + t.A_{cond} + t.A_{trans} + t.dst.A_{en}$
10: $P \leftarrow$ FINDDSTPATHS$(t.dst, C, A, P)$
11: **end for**
12: $P.add((error, dst, C + C_{neg}, A))$
13: **return** P

3.3.4 Priority

The priority of transitions is based firstly on the hierarchy level of the source state, secondly on the type of the transition (default, inner or others) and lastly on the execution order of the transition. Since the information about hierarchy level of the source and the type of transition is lost when the diagram is flattened, it needs to be added to the execution order. Therefore, the transformed execution order of a transition is a triple *(srcHierarchy, transitionType, order)*.

The ordering of transitions is resolved while printing the DVE process, in section 3.4, and it is done in the same way as in FINDDSTPATHS function – conditions of each transition are extended with negated conditions of all transitions with the same source and higher priority.

3.3.5 Composing of Transformed Transitions

After new sources and destinations for transformed transitions are determined, one transition for each combination is created. Its conditions consist of the former set of conditions united with the conditions from the default transitions on the path to the new destination. Its actions consist of the *condition actions* (previously extended with *during actions* of superstates), actions on the path from the source, *transition actions* (previously extended with *entry* and *exit* actions of original source and destination and part of their superstates) and actions on the path to the new source.

The second part of the PROCESSTRANSITION function is shown in the algorithm 3.6.

3.4 Printing the DVE Process

The PRINTPROCESS function is shown in the algorithm 3.7.

Variables and states are declared in DVE syntax.

Conditions of each transition are extended with negated conditions of all transitions with the same source and higher priority. The extended conditions form the guard of the transition. The effect consists of the list of the *transition actions*.

Additionally, loop transition is created for every state with *during actions*. Its guard consist of negated conditions of all transitions leaving the corresponding state and the effect consists the list of the *during actions*.

3.5 Transformation of an Example

To demonstrate the transformation, Stateflow model Microwave [Sba10], shown in the figure 3.4, was chosen for its simplicity.

As mentioned in section 1.2.1, Stateflow models are stored in XML documents contained in SLX file archives. Since the XML document for the Microwave model is too large, the excerpt of the Microwave XML model omits the variables and all parts that are irrelevant for the model behaviour. While the input variable *clear* was declared in the Simulink model, is was not used in the Stateflow diagram.

```
<state SSID="5">
     <P Name="labelString">RUNNING</P>
     <Children>
          <state SSID="8">
               <P Name="labelString">COOKING
```

Algorithm 3.6 PROCESSTRANSITION($t, diagram$) part 2

1: **if** t is initial default transition **then**
2: $srcPaths \leftarrow$ new list
3: $srcPaths.append((start, \text{new list}))$
4: **else**
5: $srcPaths \leftarrow$ FINDSRCPATHS($t.src$, new list, new set)
6: **end if**
7: $dstPaths \leftarrow$ FINDDSTPATHS($t.dst$, new set, new list, new set)
8: **if** t is initial default transition **then**
9: $SrcHierarchy \leftarrow 0$
10: **else**
11: $SrcHierarchy \leftarrow 1 +$ number of superstates or $t.src$
12: **end if**
13: **if** t is initial default transition **then**
14: $transitionType \leftarrow 0$
15: **else**
16: **if** t is inner transition **then**
17: $transitionType \leftarrow 2$
18: **else**
19: $transitionType \leftarrow 1$
20: **end if**
21: **end if**
22: $order = t.ord$
23: $T \leftarrow$ new set
24: **for each** path p_s in $srcPaths$ **do**
25: **for each** path p_d in $dstPaths$ **do**
26: $\bar{t} \leftarrow$ new transition
27: $\bar{t}.src \leftarrow s.state$
28: $\bar{t}.dst \leftarrow d.state$
29: $\bar{t}.C \leftarrow t.C + p_d.C$
30: $\bar{t}.A_{trans} \leftarrow t.A_{cond} + p_s.A + t.A_{trans} + p_d.A$
31: $\bar{t}.ord \leftarrow (srcHierarchy, transitionType, order)$
32: $T.add(\bar{t})$
33: **end for**
34: **end for**
35: **return** T

Algorithm 3.7 PRINTPROCESS($diagram$)

1: $print($"process" $+ diagram.name +$ "{"$)$
2: Print variables from $diagram.V$
3: $print($"state"$)$
4: Print names of states from $diagram.S$
5: **for each** transition t in $diagram.T$ **do**
6: $print(t.src +$ "->" $+ t.dst +$ "{"$)$
7: $C \leftarrow t.C$
8: **for each** transition t_2 in $diagram.T$ **do**
9: **if** $t.src = t_2.src$ and $t.ord < t_2.ord$ **then**
10: $C.add($NEGATE$(t_2.C))$
11: **end if**
12: **end for**
13: $print($"guard"$)$
14: Print conditions from C
15: $print($"effect"$)$
16: Print actions from $t.A_{trans}$
17: $print($"}"$)$
18: **end for**
19: **for each** state s in $diagram.S$ **do**
20: **if** $s.A_{du}$ is not empty **then**
21: $print(s.name +$ "->" $+ s.name +$ "{"$)$
22: $C \leftarrow$ new set
23: **for each** transition t in $diagram.T$ **do**
24: **if** $t.src = s$ **then**
25: $C.add($NEGATE$(t.C))$
26: **end if**
27: **end for**
28: $print($"guard"$)$
29: Print conditions from C
30: $print($"effect"$)$
31: Print actions from $s.A_{du}$
32: $print($"}"$)$
33: **end if**
34: **end for**
35: $print($"}"$)$

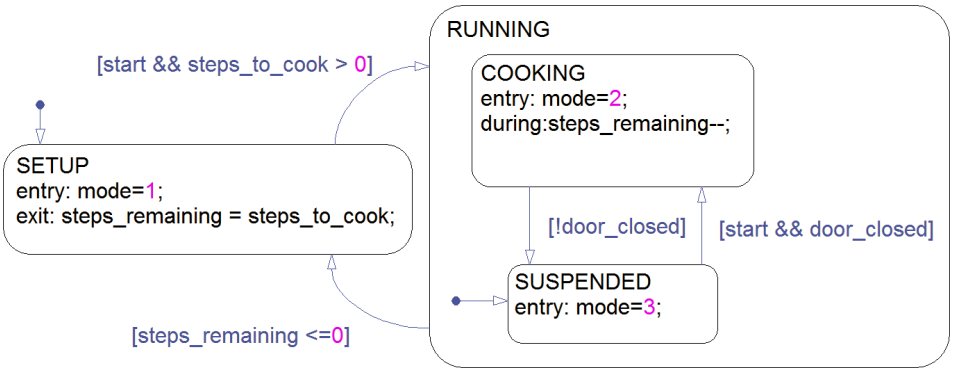

Figure 3.4: Model Microwave

```
            entry: mode=2;
            during:steps_remaining--;</P>
        </state>
        <state SSID="9">
            <P Name="labelString">SUSPENDED
            entry: mode=3;</P>
        </state>
        <transition SSID="13">
            <P Name="labelString">[!door_closed]</P>
            <src> <P Name="SSID">8</P> </src>
            <dst> <P Name="SSID">9</P> </dst>
            <P Name="executionOrder">1</P>
        </transition>
        <transition SSID="24">
            <P Name="labelString">[start &amp;&amp;
               door_closed]</P>
            <src> <P Name="SSID">9</P> </src>
            <dst> <P Name="SSID">8</P> </dst>
            <P Name="executionOrder">1</P>
        </transition>
        <transition SSID="42">
            <src> </src>
            <dst> <P Name="SSID">9</P> </dst>
            <P Name="executionOrder">1</P>
        </transition>
    </Children>
</state>
<state SSID="1">
    <P Name="labelString">SETUP
    entry: mode=1;
    exit: steps_remaining = steps_to_cook;</P>
```

```
</state>
<transition SSID="6">
     <P Name="labelString">[start &amp;amp;&amp;amp; steps_to_cook &
        gt; 0]</P>
     <src> <P Name="SSID">1</P> </src>
     <dst> <P Name="SSID">5</P> </dst>
     <P Name="executionOrder">1</P>
</transition>
<transition SSID="4">
     <src> </src>
     <dst> <P Name="SSID">1</P> </dst>
     <P Name="executionOrder">1</P>
</transition>
<transition SSID="16">
     <P Name="labelString">[steps_remaining &lt;=0]</P>
     <src> <P Name="SSID">5</P> </src>
     <dst> <P Name="SSID">1</P> </dst>
     <P Name="executionOrder">1</P>
</transition>
```

The transformation is executed using the command:

```
sf2dve.py --state-names name microwave.slx microwave.dve
```

and the following DVE model is generated:

```
process process_Chart {
     input byte door_closed;
     input byte start;
     input byte clear;
     input int steps_to_cook;
     int steps_remaining = 0;
     int mode = 1;
     state state_start, state_SETUP, state_SUSPENDED,
        state_COOKING, state_error;
     init state_start;
     trans
          state_start -> state_SETUP { effect mode = 1; }
          state_start -> state_error { guard false; }
          state_SETUP -> state_SUSPENDED { guard start and
             steps_to_cook > 0; effect steps_remaining =
             steps_to_cook, mode = 3; }
          state_SETUP -> state_error { guard start and
             steps_to_cook > 0, false; effect
             steps_remaining = steps_to_cook; }
          state_SUSPENDED -> state_SETUP { guard
             steps_remaining <= 0; effect mode = 1; }
          state_SUSPENDED -> state_COOKING { guard start and
```

```
            door_closed, not (steps_remaining <= 0); effect
             mode = 2; }
         state_COOKING -> state_SETUP { guard
            steps_remaining <= 0; effect mode = 1; }
         state_COOKING -> state_SUSPENDED { guard not
            door_closed, not (steps_remaining <= 0); effect
             mode = 3; }
         state_COOKING -> state_COOKING { guard not (not
            door_closed), not (steps_remaining <= 0);
            effect steps_remaining = steps_remaining - 1; }
}

system async;
```

Chapter 4

Implementation

The transformation algorithm from Stateflow to DVE described in Chapter 3 was implemented by sf2dve.py tool. The tool is written in the programming language Python3 [PSF13] (version 3.2 or newer is required) and uses python3-lxml [BF+14] and python3-ply [B+11] libraries. The tool accepts two arguments: path to existing SLX or XML file to be transformed and path to DVE file to be generated. Python call graph of the sf2dve tool can be found in the attachments. This graph was generated by the pycallgraph library [K+13].

The Stateflow XML document is parsed using lxml library and queried using XPath expressions. The XML document is first checked for presence of unsupported features that are listed in chapter 3. When the Stateflow model is supported, a flattened representation of each diagram is created and based on these representations DVE processes are constructed.

Since states can have conflicting names in Stateflow, their Stateflow identifier (with prefix `state_`) is used in the resulting DVE document. For human readable output, names or hierarchical names of states can be used instead of the Stateflow identifiers. This can be achieved by using switch `--state_names`. However, only the default naming, which uses unique Stateflow identifiers, ensures that no name conflict occurs.

All state and transition labels are parsed using LALR parsers generated by python3-ply library. Parsing of labels has two stages. In the first stage, all types of state actions, transition conditions and transition actions are extracted from the label, using corresponding grammars. In the second stage, actions and conditions are parsed, using simplified C grammar that complies with Stateflow syntax (defined in the Stateflow User's Guide [Mat13c]).

4.1 Variables

Stateflow allows using various types of numeric variables, including real number types, while DVE allows `int` and `byte` types only. Therefore, Stateflow data types `boolean` and `int8` are translated as `byte` and other Stateflow integer types are translated as `int`. Stateflow data types `single` and `double` are not supported. Since the range of integer types differ in Stateflow and DVE, a problem arises when a Stateflow variable overflows. Such behaviour cannot be statically detected during the transformation and needs to be accounted for when using sf2dve.

One way to overcome non-matching data type ranges in DIVINE is to adjust all effects in DVE to ensure that the resulting variable values are the same as in the Stateflow models, even when the data type ranges differ and integer overflow occurs in Stateflow. For example, consider a variable `x` having data type with range $< min, max >$ ($size = max - min + 1$) and a transition:

```
effect x = expression;
```

The transition shall be adjusted as follows:

```
guard expression > max or expression < min;
effect x = (expression - min) % size + min;
```

This approach could increase the robustness of the transformation. However, it was not implemented, since it would greatly increase the complexity of the resulting DVE model.

Stateflow also supports declaration of local variables in state and transition labels. These variables are renamed using identifier of the corresponding state or transition as their prefix (making them practically local) and declared as local variables of the process.

4.2 Environment Modelling

Reactive systems are usually and verified with surrounding environment that models how inputs behave. The tool can generate additional process `feed_inputs` that models simplified environment by generating all possible input values within given range at any instant ($\langle 0, 1 \rangle$ for booleans and for example $\langle 0, 7 \rangle$ for integers). Example of such a process for two inputs, boolean `b` and integer `i`:

```
process feed_inputs {
   state start;
   init start;
```

```
trans
   start -> start { effect b = 0; i = 0; }
   start -> start { effect b = 1; i = 0; }
   start -> start { effect b = 0; i = 1; }
   start -> start { effect b = 1; i = 1; }
   start -> start { effect b = 0; i = 2; }
   start -> start { effect b = 1; i = 2; }
   start -> start { effect b = 0; i = 3; }
   start -> start { effect b = 1; i = 3; }
   start -> start { effect b = 0; i = 4; }
   start -> start { effect b = 1; i = 4; }
   start -> start { effect b = 0; i = 5; }
   start -> start { effect b = 1; i = 5; }
   start -> start { effect b = 0; i = 6; }
   start -> start { effect b = 1; i = 6; }
   start -> start { effect b = 0; i = 7; }
   start -> start { effect b = 1; i = 7; } }
```

Such a process can be generated using switch `--feed-inputs`. To set custom range for the integer types, switch `--input-values` can be used together with the range in format: `--input-values min,max`

When a model is extended with the environment process, it is useful for the verification to require that the `feed_inputs` process cannot be performed twice in a row. Otherwise, values that were assigned to the input variables in the first `feed_inputs` transition cannot be processed by the actual process, when instantly reassigned in the second transition of the `feed_inputs` process.

This behaviour is enforced by the `--force-alternation` switch that executes the actual process in the odd steps and the `feed_inputs` process in the even steps.

This is achieved by adding a special variable `sf2dve_alt` and adjusting all transitions as follows:

- The guard `sf2dve_alt;` and the effect `sf2dve_alt = 0;` are added to the transitions of the `feed_inputs` process.
- The guard `not sf2dve_alt;` and the effect `sf2dve_alt = 1;` are added to the transitions of any other process.

The environment modelling approach (adding the alternating `feed_inputs` process) unfortunately complicates verification and simulation of the DVE model. The fact that every second transition is made by the `feed_inputs` process must be taken into consideration. While this is clear for the simulation, the LTL formulae for the verification have to be adjusted. For

example, when an event lasts n time steps in a source Stateflow model, it lasts $2n$ time steps in the generated DVE model extended with the alternating `feed_inputs` process.

Chapter 5

Evaluation

This chapter provides answers for the following questions:

- Is the behaviour of the resulting model equivalent to the behaviour of the original model? The answer is detailed in chapter 5.1.
- Can the resulting model be formally verified using DIVINE? The answer is detailed in chapter 5.2.
- Is there a transformation tool that supports more Stateflow features? The answer is detailed in chapter 5.3.

5.1 Transformation Testing

Verification whether the sf2dve transformation creates a DVE model equivalent to the given Stateflow model was performed by randomized testing. Initially, randomized sequence of input values was generated for each input of the system. Then both models were executed with these random values and their outputs were compared.

In order to execute the Stateflow diagram with the given input values, all `inport` blocks were replaced with *Repeating Sequence Stair* blocks, as depicted in the figure 5.1. All `outport` blocks were replaced with *Bus Creator* block that merges all Stateflow outputs into a vector. This output vector is routed to newly created *To File* block in order to save the outputs for comparison.

In order to simulate the DVE model with given input values, the model was extended with the `feed_inputs` process, described in chapter 4, and the `divine simulate` command was used. The input values were combined to match the transitions in the `feed_inputs` process, extended with values driving the actual system, and passed using the switch `--trace`. Odd simulation steps drive the `feed_inputs` process, whereas even simulation steps drive the actual system. Since Simulink systems are deterministic, only one successor is possible for even simulation steps. Also, switch

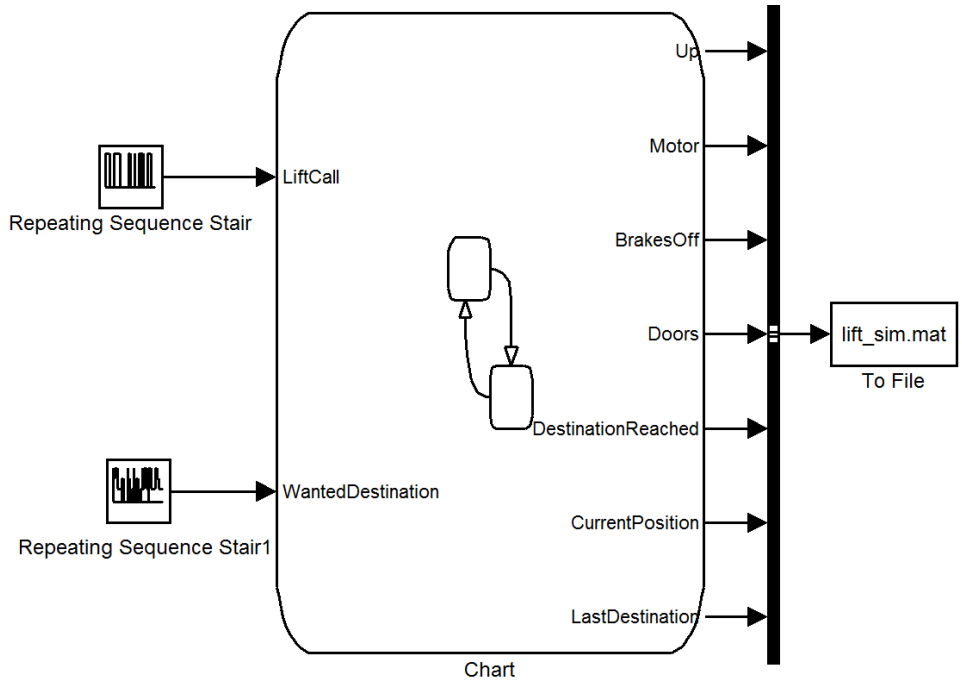

Figure 5.1: Model Lift adjusted for testing

`--no-reduce` had to be used, since there was a DIVINE bug in partial order reduction [Št15] at the time of the testing.

The length of one test was limited to about 52 000 time steps, because DIVINE simulate command with switch `--trace` accepts input only from command line argument which length is limited.

The resulting outputs from the Stateflow diagram and the DVE model were compared. Since MATLAB output from *To File* block is a binary file, the comparison had to be done in the MATLAB environment. The Stateflow model Lift [ABB], shown in the figure 5.2, was used for the testing and the outputs from both the original Stateflow model and the transformed DVE model were identical.

Second system that was tested using randomized input values was ControlIO system [Hon]. Also the second test passed without any dissimilarity. While implementing the sf2dve tool, similar randomized tests were useful to reveal some minor differences between the Stateflow and DVE models.

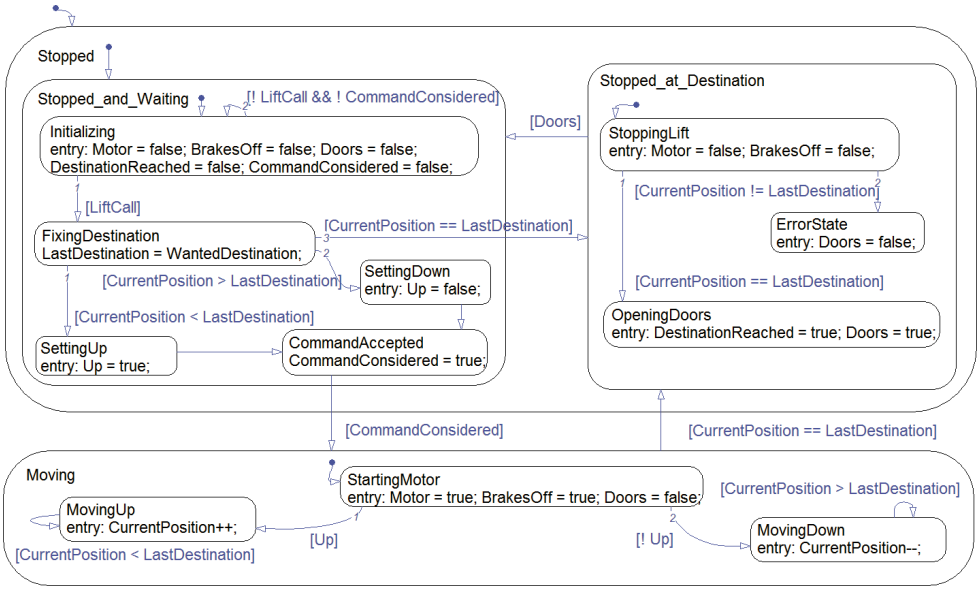

Figure 5.2: Model Lift

5.2 Verification of the Lift System

Since the goal of this thesis is to enable formal verification of Stateflow models, example model Lift was transformed into DVE and formally verified using DIVINE with respect to the given requirements. The requirements were specified by the ABB together with the Lift model as follows:

- When the elevator stopped, brake must be set in order to keep the lift in the position. Before the lift moved, the brakes are deactivated and the engine is started.

  These two informal requirements can be written as one formula:

  $\mathbf{G}\ (motor \Leftrightarrow brakesoff)$

- A door can be opened only if the lift is at the Current floor. Otherwise, the door is blocked.

  This informal requirement can be written as follows:

  $\mathbf{G}\ (destinationreached \Leftrightarrow doors)$

- When a button is pressed, the elevator must join the desired floor.

Since it is needed to be accounted for the alternating environment process `feed_input` as described at the end of the section 4.2, this informal requirement can be written as follows:

$\mathbf{G}\ (liftcall \wedge \mathbf{X}\, liftcall \Rightarrow \mathbf{F}\, destinationreached)$

Based on these formulae the following file lift.ltl was created:

```
#define motor (process_Chart->Motor)
#define brakesoff (process_Chart->BrakesOff)
#define liftcall (LiftCall)
#define destinationreached (process_Chart->
   DestinationReached)
#define doors (process_Chart->Doors)

#property G(motor <-> brakesoff)
#property G(destinationreached <-> doors)
#property G(liftcall && X liftcall-> F destinationreached)
```

The Lift model was generated with the following switches:

```
--feed-inputs --force-alternation --state-names name
```

The verification of the properties was performed with the commands:

```
divine combine -f lift.ltl lift.dve
divine verify -p LTL lift.prop1.dve
divine verify -p LTL lift.prop2.dve
divine verify -p LTL lift.prop3.dve
divine verify -p deadlock lift.dve
```

All four properties hold.

5.3 Capability Evaluation

The main purpose of this section is to show which types of Stateflow models are supported by various transformation tools.

The set of Stateflow models was gathered from examples of the transformation tools and converted into the Simulink version, which shall be supported by the tool, in order to achieve the maximum compatibility. Moreover, the model Microwave had to be adjusted, since it contained several invalid elements, such as the text *conditionaction* in place of a condition action. The models used for capability evaluation together with their origin and original Simulink version are listed in the table 5.1.

While most of the transformation tools from Chapter 2 cannot be obtained, some of them are available for the capability evaluation. The table 5.2 shows the capabilities of the available transformation tools.

Model Name	Origin	Simulink Version
AC [JD+]	mdl2smv tool example	6.6
ControlIO	Honeywell proprietary	7.9
Lift	ABB public use case	7.9
Microwave	NuSMV GUI tool example	7.6
SetReset [S+]	sf2lus tool example	5.0

Table 5.1: Stateflow Models

	AC	ControlIO	Lift	Microwave	SetReset
sf2dve	Edited[1]	Ok	Ok	Ok	Edited[1]
ForReq CESMI	Edited[1]	Ok	Ok	Edited[3]	Edited[1]
ForReq NuSMV	Edited[1]	Ok	Ok	Edited[3]	Edited[1]
mdl2smv	Ok	Error[4]	Error[4]	Ok	Edited[2]
NuSMV GUI	Error[5]	Error[5]	Error[5]	Ok	Error[5]
PAT	Error[6]	Error[6]	Error[6]	Error[6]	Error[6]
sf2lus	Error[7]	Error[7]	Error[7]	Error[7]	Ok

Table 5.2: Capability Evaluation

1. When the events were changed into inputs, the model was automatically translated.
2. When all the actions were finished with a semicolon, the model was automatically translated.
3. When two conditions were changed, the model was automatically translated.
4. The tool resulted in segmentation error during transformation.
5. Model was not recognized as a Stateflow model, even after conversion into corresponding MDL file version 7.6.
6. Not successful in finding a way how to import any Stateflow model. Question how to do that was raised at the PAT forum.
7. Current Matlab cannot convert Simulink models into version 5.0.

Chapter 6

Conclusion

The goal of this thesis, to create a tool for translating Stateflow diagrams to the DVE modelling language, was successfully completed.

The transformation supports the following Stateflow features: states with *entry*, *during* and *exit actions*, transitions with conditions, *condition actions* and *transition actions*, state hierarchy, labelled default transitions, implicit `tick` event, variables of integer data types. The transformation handles even proper parsing of Stateflow labels, actions and conditions, which some other tools, including ForReq tool from Honeywell, fail to implement.

The verification that the behaviour of the transformed DVE models is equivalent to the behaviour of the Stateflow models was performed by randomized testing. The verification of the Lift system, which was enriched with generated environment process, against its requirements was successfully performed.

Since the proposed tool is the only one which supports the new SLX format, Honeywell is planning to incorporate the tool for formal verification of Stateflow models.

In future work, the tool should be extended to support the following Stateflow features: connective junctions, AND decomposition of states, and non-broadcasting events. Moreover, it would be beneficial to support Simulink blocks that can be directly translated into DVE modelling language (for example: constant, gain, switch).

Bibliography

[ABB] ABB. Lift. Stateflow model and requirements; publically presented by ABB, unavailable online.

[B+11] David Beazley et al. Ply, 2011. Version 3.4. Python library. `http://www.dabeaz.com/ply/`.

[BBB+12] Jiří Barnat, Jan Beran, Luboš Brim, Tomáš Kratochvíla, and Petr Ročkai. Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In *Formal Methods for Industrial Critical Systems (FMICS 2012)*, volume 7437 of *LNCS*, pages 78–92. Springer, 2012.

[BBČ+06] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. DiVinE – A Tool for Distributed Verification. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 278–281, Berlin, Heidelberg, 2006. Springer-Verlag. `http://dx.doi.org/10.1007/11817963_26`.

[BBK99] Chonlawit Banphawatthanarak, Ken Butts, and Bruce H. Krogh. Symbolic Verification of Executable Control Specifications. pages 581–586. IEEE, August 1999.

[BF+14] Stefan Behnel, Martijn Faassen, et al. lxml, 2014. Version 3.3.6. Python library. `http://lxml.de/`.

[Boo67] Taylor L. Booth. Sequential Machines and Automata Theory, 1967.

[CGP99a] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CGP99b] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. Temporal Logics. In *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[Che10] Chunqing Chen. Formal Analysis for Stateflow Diagrams. In *Fourth International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C 2010)*, pages 102–109. IEEE, 2010.

[CSL+12] Chunqing Chen, Jun Sun, Yang Liu, JinSong Dong, and Manchun Zheng. Formal Modeling and Validation of Stateflow Diagrams. *International Journal on Software Tools for Technology Transfer*, 14(6):653–671, 2012.

[Deg12a] Jutta Degener. ANSI C grammar, Lex specification, 2012. `http://www.quut.com/c/ANSI-C-grammar-l-2011.html`.

[Deg12b] Jutta Degener. ANSI C Yacc grammar, 2012. `http://www.quut.com/c/ANSI-C-grammar-y.html`.

[FCoGoT] FBK-irst, CMU, The University of Genova, and The University of Trento. NuSMV GUI. `http://www.dsi.unifi.it/~fantechi/Collaborazionegets/tools.html`.

[H+] Gerard J. Holzmann et al. SPIN. `http://spinroot.com/spin/whatispin.html`.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming 8*, pages 231–274, 1987.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and Verifying Real-time Systems by Means of the Synchronous Data-flow Programming Language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[Hon] Honeywell. ControlIO. Proprietary Stateflow model.

[JD+] Alma L. Juarez-Dominguez et al. AC. Stateflow model; available as an example within mdl2smv tool. `https://cs.uwaterloo.ca/~aljuarez/Docs/mdl2smv_v3.zip`.

[K+13] Gerald Kaszuba et al. Python Call Graph, 2013. Version 1.0.1. `http://pycallgraph.slowchop.com/en/master/`.

[Kri13] Jan Kriho. Enhanced parser for DVE modelling language. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2013.

[Lei08] Florian Leitner-Fischer. Evaluation of the Matlab Simulink Design Verifier versus the model checker SPIN. Master's thesis, 2008. http://kops.ub.uni-konstanz.de/volltexte/2008/6125.

[Mat] The MathWorks, Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier/.

[Mat13a] The MathWorks, Inc. Simulink, 2013. Version R2013a. http://www.mathworks.com/products/simulink/.

[Mat13b] The MathWorks, Inc. Stateflow, 2013. Version R2013a. http://www.mathworks.com/products/stateflow/.

[Mat13c] The MathWorks, Inc. Stateflow User's Guide, 2013. Version R2013b.

[Mat14] The MathWorks, Inc. Saving Models in the SLX File Format, 2014. [Online; accessed 6. 12. 2014] http://www.mathworks.com/help/simulink/ug/saving-a-model.html#btbr7kx-1.

[MCBL] Ken McMillan and Cadence Berkeley Labs. Cadence SMV Model Checker. http://www.kenmcmil.com/smv.html.

[Miy12] Alvaro Heiji Miyazawa. *Formal verification of implementations of Stateflow charts*. PhD thesis, University of York, 2012. http://etheses.whiterose.ac.uk/2353/.

[NUoS] National University of Singapore. PAT. http://pat.sce.ntu.edu.sg/.

[PSF13] Python Software Foundation. Python, 2013. Version 3.3. https://www.python.org/.

[S+] Norman Scaife et al. SetReset. Stateflow model; available as an example within sf2lus tool. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/SimulinkStateflow2Lustre/sf2lus_user_manual/sf2lus002.html#toc1.

[Sba10] Daniele Sbaraccani. Modellomicrowave, 2010. Stateflow model; available as an example within NuSMV GUI tool. http://www.dsi.unifi.it/~fantechi/Collaborazionegets/nusmv_gui_tool/modellomicrowave.mdl.

[SCBR01] Steve Sims, Rance Cleaveland, Ken Butts, and Scott Ranville. Automated Validation of Software Models. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 91–96. IEEE, 2001.

[SSC+04] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT 2004)*, pages 259–268, New York, NY, USA, 2004. ACM.

[TG05] Ian Toyn and Andy Galloway. Proving Properties of Stateflow Models Using ISO Standard Z and CADiZ. In *Proceedings of the 4th International Conference on Formal Specification and Development in Z and B*, ZB'05, pages 104–123, Berlin, Heidelberg, 2005. Springer-Verlag.

[Tiw02] Ashish Tiwari. Formal Semantics and Analysis Methods for Simulink Stateflow Models. Technical report, SRI International, 2002. `http://www.csl.sri.com/\discretionary{-}{}{}$\sim$tiwari/\discretionary{-}{}{}stateflow.html`.

[TM93] Ian Toyn and John A. Mcdermid. CADiZ: An Architecture for Z Tools and its Implementation. *SOFTWARE – Practice and Experience*, 25:305–330, 1993.

[Š06] Pavel Šimeček. DIVINE - Distributed Verification Environment. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2006.

[Št15] Vladimír Štill. DVE: POR is broken., 2015. DIVINE bug #321. `https://divine.fi.muni.cz/trac/ticket/321`.

Attachments

- sf2dve.zip – Archive with the source codes, containing the tool for the transformation of the Stateflow diagrams to the DVE language and the tool for running simulation of a DVE model (enhanced with input-feeding process).
- examples.zip – Archive with Stateflow diagram examples, that were used in the capability evaluation and with the lift.ltl file.