STU: Stateflow to Uppaal Technique Report

Yixiao Yang[1,2],Yu Jiang[1,2],Ming Gu[1],Jiaguang Sun[1]
School of Software, Tsinghua University,TNLIST, KLISS, Beijing, China[1]

ABSTRACT

This paper offers the technique report of our tool paper. Many details are offered.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

computer-aided software engineering

Keywords

stateflow , timed automaton, translation , verification

1. INTRODUCTION

So far, at present there are already some tools to do the validation of Matlab Stateflow.However,almost all of the current existing tools did not solve the problem of time and the complete semantic conversion.Stateflow is a flexible modeling tool which has stack in its execution process.The main difficulty of simulating the logic of Stateflow using the traditional automaton is that stateflow is executed on stack.It can dispatch a new event when handling the current event and after the new event executed over the current event should go on.What's more , Stateflow also has the concept of active/deactive.When doing activation, It can de-active itself without any restraint.Stateflow also support cross boundary connection which means the active/deactive process should consider very detailed environment to make decision which state should be activated or deactivated.Stateflow has two kinds of states:Serial state and Parallel state.The parallel state has priority which determine exactly which state should run at order when more than one state is being invoked.Stateflow also supports time , so the designer can decide which action to be executed depends on time.To verify the property with the concern of time is somewhat very difficult.That's why we convert the stateflow to timed automaton to gain the ability of time verification.

Then we will introduce what we have done.what is the advantage of our tool.

ESEC/FSE '4, November 16-22, 2014, Hong Kong

2. RELATED WORK

Although, there are many existing models similar to stateflow such as Statecharts[5] and SSM[6], but the semantics are very different. The Stateflow semantics are completely deterministic which means the parallel states or outgoing transitions in a model are executed at a fixed order. Furthermore, Stateflow has it's own features:a condition action occurs before the source state becomes inactive, the time can be used to determine when a transition can be executed, the entry action, during action and exit action are executed when the state is entered, maintained and exited and so on.

There already exists various methods to apply different kinds of verification tools to Stateflow. Recently, Paula Herber[7] used SMT Solver to verify Discrete-Time MATLAB/Simulink Models. However his work hasn't covered Simulink/Stateflow yet. Chunqing Chen[8] applied stateflow to PAT Verification Tool which has covered some specific features of Stateflow, but his work didn't cover the feature of time in Stateflow and his work is more like writing C code. Circus notation was discussed by Cavalcanti [16] , but the event dispatch machanism was not handled properly. In earlier work, Banphawatthanarak et al. [9] used the SMV model checker to check Stateflow model. However, in his work, hierarchical states and events were not supported and only Boolean-valued variables were allowed. In 2001, an invariant checker was used to verify Stateflow by Sims et al. [10]. Transformation to communication pushdown automata was reported by Tiwari [11], but the sequential execution order among parallel states is not taken into account. Hamon [12] and Hamon and Rushby [13] proposed Denotational and operational semantics of Stateflow, respectively. Their definitions were too abstract to describe features of stateflow. Lustre, a synchronous programming language with restrictions on the use of recursion was used by Scaife et al. [14]. Stateflow's execution is based on the event stack, so the lack of recursion support leads to a depressing result. Toyn and Galloway [15] proposed to model Stateflow diagrams using Z notation from the perspective that they interpreted Stateflow as Statecharts.

Comparing to the above previous work, Our work covers

nearly all the range of the stateflow features which means we have done a completely conversion work including the Time Mechanism which had never been done before. What's more, the tool we apply to verify the stateflow is a formal semantic based timed automaton which is introduced by Uppaal. Similar to our work, Tiwari also used automata to verify Stateflow, but Tiwari's work couldn't handle the sequential execution order among parallel states. In contrast, our work solved the problem by designing an entirely new interaction mechanism. Thanks to the uppal's simulation ability, we could not only do verification but also simulate the converted model to check whether or not our conversion is right.

3. SEMANTICS OF MATLAB STATEFLOW AND UPPAAL AUTOMATON

Stateflow is a finite state machine which can contain sequential decision logic based on state machines. A finite state machine is a representation of an event-driven (reactive) system. In an event-driven system, the system makes a transition from one state (mode) to another, if the condition defining the change is true. Stateflow is just a module of Simulink. Each Stateflow block corresponds to a single Stateflow chart. The Stateflow block interfaces to its Simulink model. The Stateflow block can interface to code sources external to the Simulink model (data, events, custom code).There are mainly six kinds of elements in Stateflow namely State, Junction, Transition, Event, Guard, Action. A state describes an operating mode of a reactive system. In a Stateflow chart, states are used for sequential design to create state transition diagrams. States can be active or inactive. The activity or inactivity of a state can change depending on events and conditions. The occurrence of an event drives the execution of the state transition diagram by making states become active or inactive. At any point during execution, active and inactive states exist. To manage multilevel state complexity, there is a mechanism called hierarchy in Stateflow chart. Any level state can contain two types of child states which are parallel states and serial states respectively. The technical term of serial state in stateflow is called 'Cluster Decomposition', On the other hand the technical term of parallel state is called 'Parallel Decomposition'. Serial state must have at least one default transition and only one child state can be active at one time. Parallel state can't have any default transition and all the child states are active at one time. Though it seems to be a parallel execution in parallel state, however every child states in parallel state has an execution order which make the parallel mechanism turn into the serial mechanism actually. The edge between states are called 'transition'.The transition has four parts and the grammar could be written as EVENT[CONDITION]CONDITIONAL ACTION/COMMON ACTION The EVENT is a standard data type which drive the execution of the stateflow. The diagram could move from one state to another state or just remain still in the life cycle of one event. The CONDITION is the guard on transition. A transition can be executed if and only if the condition is met. The CONDITIONAL ACTION is immediately executed if the condition is met. The COMMON ACTION is executed if there is a valid path between the current active state and the target state which means COMMON ACTION is executed only when there exists a state transfer. In addition to these elements, there also exists Junction in stateflow. A Junction is not a state so the execution flow won't stop when it meets a Junction. If there exists many Junctions between states, the execution flow will do the following. 1.If the execution flow finally find a path which satisfied all the condition exists on the transition , the execution flow will transfer from the current active state to the target state. 2.If the execution flow doesn't find a path between states at last, It will execute all the conditional action on the transition the execution flow has walked and then back to the active state. What's more, There are time operations which are aimed to handle the time based modeling. There has three types of time operations which are before, after, at respectively. The grammar of the three operations is of the same form: TMOP(NUM,EVENT) TMOP is just the three operation:before, after, at. NUM is a double type number. EVENT is mentioned above except for three system reserved keywords:sec,msec,usec which represents second , millisecond ,microseconds. Using the three time operation mentioned above, the time-critical system can be modeled without any difficulty. The action can not only be written on the transition but can also be written on the state itself.There are three kinds of actions which are entry action,during action,exit action respectively. Entry action is executed when the state is first activated. During action is executed when the state is already active. Exit action is executed when the state changes from active to inactive. These three actions and the conditional action are executed without the state changed so if a new event is dispatched during the execution of the previous mentioned four types of actions, the state may change due to the newly dispatched event causing the rest of the previous unfinished work to be ignored.

Uppaal is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications. The tool is designed to verify systems that can be modelled as networks of timed automata extended with integer variables, structured data types, user defined functions, and channel synchronisation. The model-checker Uppaal is based on the theory of timed automata [4] (see [42] for automata theory) and its modelling language offers additional features such as bounded integer variables and urgency. The query language of Uppaal, used to specify properties to be checked, is a subset of TCTL (timed computation tree logic) [39,3]. Networks of Timed Automata A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously. In Uppaal, a system is modelled as a network of several such timed automata in parallel. The model is further extended with bounded discrete variables that are part of the state. These variables are used as in programming languages: They are read, written, and are subject to common arithmetic operations. A state of the system is defined by the locations of all automata, the clock values, and the values of the discrete variables. Every automaton may fire an edge (sometimes misleadingly called a transition) separately or synchronise with another automaton1, which leads to a new state.

Committed and Urgent Locations There are three different types of locations in Uppaal: normal locations with or without invariants (e.g., x<=3 in the previous example), ur-

gent locations, and committed locations. Figure 8 shows 3 automata to illustrate the difference. The location marked u is urgent and the one marked c is committed. The clocks are local to the automata, i.e., x in P0 is different from x in P1. An urgent location is equivalent to a location with incoming edges reseting a designated clock y and labelled with the invariant y<=0. Time may not progress in an urgent state, but interleavings with normal states are allowed. A committed location is more restrictive: in all the states where P2.S1 is active (in our example), the only possible transition is the one that fires the edge outgoing from P2.S1. A state having a committed location active is said to 11 be committed: delay is not allowed and the committed location must be left in the successor state (or one of the committed locations if there are several ones).

4. ARCHITECTURE DESIGN AND IMPLEMENTATION

This tool is mainly a conversion tool which converts Matlab Stateflow to Uppaal Automaton.The Input of the tool is the file ended with .slx, a kind of compressed binary format. The output of the tool is the file ended with .xml which can be read and run by Uppaal Verfication Tool. Besides the input and output, the vast majority of the work of the tool is the conversion. The most important and the most difficult task is to transform the stateflow execution mechanism composed of states, events, transitions, junctions fully into the uppaal timed automaton without the loss of semantics. As can be seen from the previous chapter, the most difference between stateflow and uppaal is that stateflow is driven by event and the execution order of every step of the event is deterministic. As stateflow is executed on the stack, this leads to a recursive happened situation that the newly dispatched event will block the current event's execution to gain the ability to run and after the newly dispatched event executed over , then the previous blocked event gain the ability to proceed. To make the problem of conversion become even more difficult is that the stateflow has hierarchy structure which combined with activation/deactivation mechanism very closely. As doing activation,a state should recursively invoke it's parent's activation logic if it's parent is in the inactive state. Similarly,in the deactivation a state should recursively invoke it's children's deactivation logic at order if it's children are in the active state. A new event may be dispatched when handling activation/deactivation recursively. All the mechanism described above is very complex. To simulate the complex mechanism in uppaal which only supports very simple data structure, an entirely new mechanism was designed.

5. MODEL IMPLEMENTATION

Our conversion work of stateflow covers the following elements: state, junction, transition, cross boundary transition, conditional action, transitional action, event and the most importantly element : time. Here is an example of a little complex stateflow diagram which covers all the elements mentioned above.
In this graph, the execution is driven by time, we set that the total runtime is 10 seconds. The solver which is used to run the model is a kind of 'discrete solver'. The 'discrete solver' run the model at a fixed time gap. The time gap here is set to 1 second.

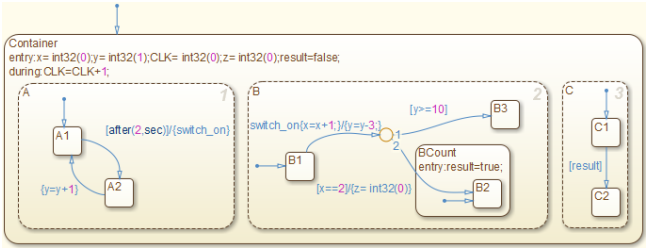


Figure 1: Count switch on event

So the final result of the chart after the 10 seconds' execution is shown as below:

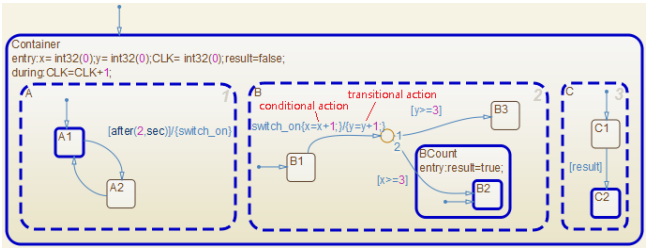


Figure 2: Count switch on event

The comments in the graph is noted with a red font.

The model in the graph above just do one simple thing : for every 2 seconds, the state A dispatches a 'switch on' event. Then for every 'switch on' event, x increases by 1. In the graph, statement x=x+1 is a kind of conditional action, so it can be executed immediately when a event 'switch on' is dispatched. On the other hand, statement y=y+1 is a kind of transitional action which can only be executed when a path between two states is find. So at the end of execution of the model, the value of y is 1 and the value of x is 3. The boolean variable 'result' is set to true at last due to the activation of the state 'B2' causing the activation of B2's parent : 'BCount'. In the activation of 'BCount', the entry action 'result=true;' is executed.

As mentioned above, the execution of stateflow is sequencial. However, the automatons in Uppaal run in parallel. So, a synchronization mechanism is needed. In this section, we will introduce the six main parts of our design and the reason of the design. We will show the the discoveries from our rigorous modeling and validation procedure.

5.1 Event Stack

The event's distributing and processing mechanism in stateflow is much like the function invokation mechanism in program which means if a new event is dispatched, the current executing event must be hang up until the newly dispatched event's execution is over. However in Uppaal, there only exists asynchronous parallel automatons and no stack at all. The idea of our approach to simulate the Stateflow's stack mechanism in Uppaal is to build a stack by ourselves.We use the c-like structured array in uppaal to build a stack with finite space.

Every slot of the stack in uppaal is a data structure noted as EVENT which records all the information related to an event in stateflow.

Stack Slot(Event).

It needs to use six datas of int/bool types in uppaal to simulate the event in stateflow.

Data1. Event Identification : In uppaal, we just use an unique integer number to distinguish different events in stateflow.

Data2. Event's Destination(State) : As described, Every complex state corresponds to four uppaal automaton.In uppaal, we assign each state an unique number and the event must be executed by exactly one state at one time.This data implies which state should handle this event. However, a state may corresponds to four automatons, which automaton should handle this event? the following data will solve this problem.

Data3. Event's Destination(Type Of Automaton) : this is an enum of four value {controller, ctrl action, common, condition}, Data2 and Data3 together will determine exactly one automaton. These four kinds of automata have different functions which we will describe later.

Data4. Event's Destination(Node In Automaton) : We also assign each node in each automaton an unique number. In some cases, the event should only be handled in some particular node in the automaton. So this data implies which node in specified automaton should handle this event.

Data5. Is Valid : This data just implies whether this event is valid now. If the event is on the top of stack and is invalid, the event will be deleted by a daemon uppaal automaton.

Data6. Extra Info : This data record some simple flag in order to exchange some very simple information between different automata.

Event Structure.

Summarize the facts mentioned above, the structure EVENT contains the information shown in Algorithm 1. The stack

Algorithm 1 structure of EVENT

```
int mEvent;      //Event Identification
int mDest;       //Event's Destination
int mDestCPosition; //Event's Destination
int mAutomatonType; //Event's Destination
bool mValid;     //Whether event should be deleted
int mInfo;       //Some simple extra infomation
```

is an array with every element of type EVENT.

Figure 3: Stack

Figure 3 shows the array of stack.

We give name 'mExecutionStack' to the stack array. To denote the index of the top of the stack, we use an integer variable noted as 'mExecutionStackTop'.

The stack operation can be simulated by array operation. The poping an element on the top of stack can be taken as deleting an element at the tail of the array. Similarly,

Structure 2 Stack Top Index

```
int mExecutionStackTop = -1;
```

Structure 3 Stack Declaration

```
EVENT mExecutionStack[MaxStackSize+1];
```

pushing an element onto stack can be achieved by adding an element at the tail of the array.

Algorithm 4 Push One Event

```
void PushOneEvent(int event,int dest,int destCPosition,bool valid,int automatonType,int info)
{
mExecutionStackTop++;
mExecutionStack[mExecutionStackTop].mEvent = event;
mExecutionStack[mExecutionStackTop].mDest = dest;
mExecutionStack[mExecutionStackTop].mDestCPosition
= destCPosition;
mExecutionStack[mExecutionStackTop].mValid = valid;
mExecutionStack[mExecutionStackTop].mAutomatonType
= automatonType;
mExecutionStack[mExecutionStackTop].mInfo = info;
}
```

To pop one event, there is no need to really delete the event on the tail of the array, just decrease the stack top index 'mExecutionStackTop' by 1.

Algorithm 5 Pop One Event

```
void PopOneEvent()
{
mExecutionStackTop–;
}
```

Every transition in all the parallel automaton in uppaal, will look at the top of the stack to see whether the event is sent to the automaton it belongs to. If the event is sent to the automaton which the transition belongs to, the transition will gain the privilege to run.

Example.

This stack is the base of everything. Let us show an example about how to use it. From figure 3, there are 2 parallel states in stateflow. When transition from state A1 to state A2 in A is executed, the event 'EA' is dispatched, the transition from state B1 to B2 gain the chance to run. The final result is that state B2 is reached earlier than A2. From this simple example, we can see the execution mechanism of event is based on stack.

At the very beginning, we introduce three basic functions which are the fundamental elements used in the transformed Uppaal model everywhere.

We define that the function 'EventSentToMe' on the transition return true if the event on the top of the stack is sent to the Automaton which contains the transition. This function can not be a global function because it must have some identity infomation of state to judge whether event is sent to this automaton. This function is actuallly put into the local

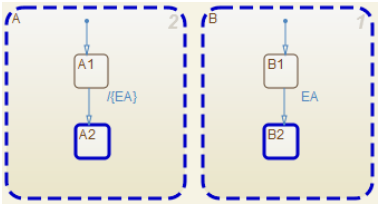


Figure 4: Case to show stack use

data section of an automaton. The function 'DispatchEvent' push an event on the top of the stack. The function 'StackTopEvent()' get the stack top event.

Algorithm 6 Event Sent To Me

```
bool EventSentToMe()
{
bool r1 = false;
bool r2 = false;
r1 = StackTopEvent().mDest == stateid;
r2 = StackTopEvent().mAutomatonType == automaton-
Type;
return r1 && r2;
}
```

Algorithm 7 Dispatch Event

```
void DispatchEvent(EVENT EA)
{
PushOneEvent(EA.mEvent,EA.mDest,-
1,EA.mInfo,true,EA.mAutomatonType);
}
```

Algorithm 8 Stack Top Event

```
EVENT StackTopEvent()
{
return mExecutionStack[mExecutionStackTop];
}
```

Back to our example, if the execution flow reaches state A and transition from A1 to A2 dispatches a new event 'EA' because there is no guard on its transition. Due to B has the highest priority '1' labeled on upper right corner, so 'EA' is dispatched to B at first and 'EA sent to B' which is denoted as 'EA->B' is pushed on stack. after 'EA->B' havign being executed over which means that transition from B1 to B2 is taken, the stack pop up it. Now, the stack is empty. Due to the diagram is a parallel decomposition and A has a lower priority '2' adjacent to B's '1', so the 'EA' is sent to A this time denoted as 'EA->A'. But nothing happens, because A's current active element is not the state but the transition which dispatched 'EA', nothing would happen when a transition is being executed. Without new events dispatching the execution stops when 'EA->A' being pop up and the whole process stops.

The change on the stack is as follows:

The corresponding pseudo uppaal automaton which uses the three fundamental functions mentioned above is as the following:

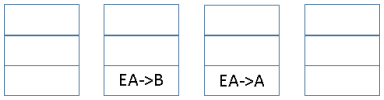


Figure 5: stack change

Note : the brown color text denotes guard(condition) and the light blue text denotes action.

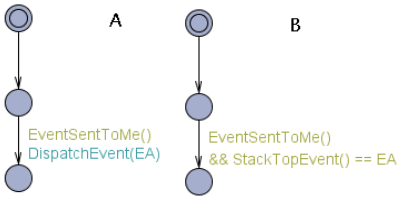


Figure 6: pseudo uppaal automaton

5.2 Daemon Automaton

The Daemon Automaton has two duties. The first duty is to delete invalid event on the top of stack. The second duty is to dispatch 'System Event' to keep the automaton running when the stack is empty.

System Event is a concept in Stateflow which refers to the default event generated by Stateflow's the running environment. In Stateflow it is generated periodically and is used to make the suspended automaton run again.

Duty1 Delete Invalid Event.

To delete an invalid event, the Daemon Automaton needs a self cycle transition. This self cycle transition check if the field : mValid of the stack top event is true. If the field mValid is false, then the stack top event will be pop up. The popping up operation is just done by decrease the variable represents the index of stack top by 1.The function 'JTopStackInvalid()' return true if and only if the stack top event's field : mEvent is false. The function 'JTopStackInvalid()' is showed in the following Algorithm9, To pop up the event, the previous mentioned function 'PopOneEvent()' is used. The funtion 'PopOneEvent()' is adapted to any event popping up situation and even a little change is not needed.

Algorithm 9 If Stack Top Event Invalid

```
EVENT JTopStackInvalid()
{
return mExecutionStack[mExecutionStackTop].mValid
== false;
}
```

Duty2 Generate System Event.

In uppaal, the generation of System Event is achieved by pushing an EVENT structured data onto the top of the empty stack. The six fields of the SYSTEM EVENT in uppaal are all fixed constants.

System Event 10 The Value of System Event

```
mEvent = eTrigger;
mDest = 1;
mDestCPosition = -1;
mAutomatonType = aControllerAutomaton;
mValid = true;
mInfo = -1;
```

eTrigger is a constant integer defined in the golobal data area in Uppaal which represents the id of System Event in Uppaal. In Stateflow, there exists a special state which is the common ancestor of all other states. It contains all the states and transitions in one Stateflow diagram. This special state is called 'Chart'. The id of Chart we allocated is 1. The System Event's mDest is 1 which means the System Event is dispatched to Chart at the very beginning. The fields : mDestCPosition and mInfo are -1. So these two fields are unused in System Event. Every newly created Event's mValid is true so does System Event. Further more, every event should be dispatched to Controller Automaton at first. Because only the Controller Automaton decides where and how to dispatch the event.

We define function 'IsStackEmpty()' to judge whether stack is empty or not. System event can only be dispatch when stack is empty. The function 'GenerateDrivenEvent()' generates the System Event and dispatches it to Chart.

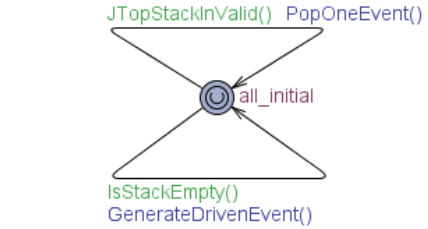
Algorithm 11 Is Stack Empty?

```
bool IsStackEmpty()
{
    return mExecutionStackTop < 0;
}
```

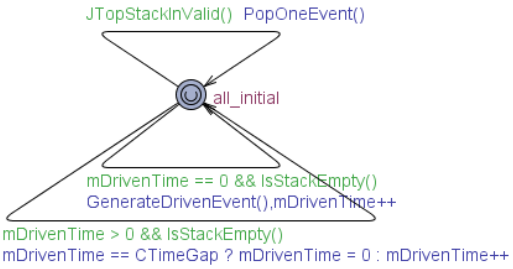
Algorithm 12 Generate Driven Event

```
void GenerateDrivenEvent()
{
                                PushOneEvent(eTrigger,1,-
1,true,aControllerAutomaton,-1);
}
```

For the time dependent Stateflow model, one more judgement that whether total run time is divisible by time gap is needed which means that only the two conditions first is stack empty, second is total run time divisible by time gap be met at the same time can the System Event be dispatched. The 'time gap' is a parameter set by Stateflow to dispatch System Event periodically. So another self cycle needs to be added to handle the total run time increment and total run time judgement. The mod operation is time consumed, in actual implementation we don't use mod to judge whether total run time is divisible by time gap, instead, we use a global int variable which represents the mod result of total run time. We give this variable a name 'mDrivenTime'. Every time, when 'mDrivenTime' is greater or equal than time gap. The 'mDrivenTime decrease by time gap'. So 'mDrivenTime' incrementally updated to ensure it is the mod result of total run time. The design is shown in following.



(a) Daemon Without Time



(b) Daemon With Time

Figure 7: Daemon Automaton

5.3 State

At last, having taken overall consideration, we are forced to convert it into four types of uppaal automaton namely 'Controller Automaton', 'Controller Action Automaton' or 'Ctrl Action Automaton', 'Common Automaton', 'Condition Automaton' respectively.

The Controller Automaton is aimed to simulate the event dispatch mechanism in stateflow. It controls and determines how and where to dispatch the event.
The Controller Action Automaton is responsible for doing entry/during/exit actions written on the state.
The Condition Automaton executes the guard on transition, put the boolean results into an array and does conditional action.
The Common Automaton uses the boolean results generated by conditional automaton to determine which path can be satisfied to run and does the transitional action.

Example.

As an example,we give a very simple case as shown below.

From figure 8, we can see that when a state is about to leave, the deactivation should be invoked. Similarly when a state is about to enter, the activation should be invoked.

For activation, it should first to judge whether it's parent is activated, if it's parent is not activated, activate it's parent first. To solve this problem, The controller must first push a activation event sent to self onto stack and then push a similar event sent to it's parent onto stack. Due to stack is last-in first-out, the activation sent to parent will be executed at first. This procedure recursively proceed until an already activated controller automaton no longer go on dispatching activation events to its parent, instead, it begin to handle the real activation logic. The activation logic con-

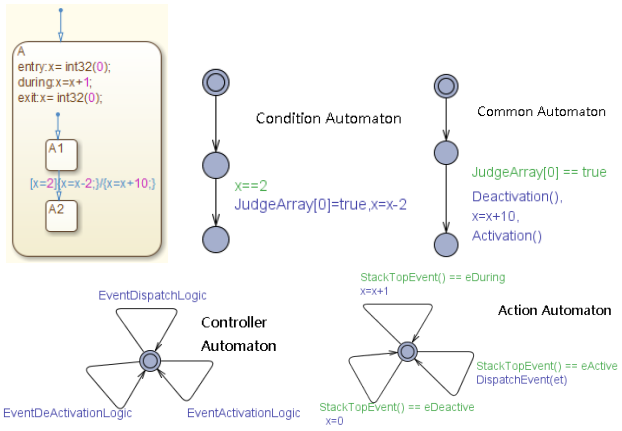


Figure 8: state conversion framework

sists of two parts: 1.do entry action. 2.activate child state.

For deactivation, the procedure is similar. A state should de-active it's active child state first. By achieving this, the controller must first push a deactivation event sent to self onto stack and then push a similar event sent to it's active child onto stack. The real deactivation logic also contains two parts: 1.do exit action 2.deactive parent state.

Deactivation Logic.

Deactivation Logic handles the exit action and recursively deactive it's avtive child states. If the state's child states are parallel states, then these childs states must be deactived in ascending order of priority.

The deactivation logic of every state must be splited into two opposite parts. These two opposite parts are translated into two self cycle transition of the initial state of the Controller Automaton. The first part is to dispatch the event to its active child state. Before dipatching a newly created Deactivation Event to its child state, the current Deactivation Event must be deleted first. So at one time, there is only one Deactivation Event on stack. The second part is to handle the exit action and make decision to choose between dispatching the event to parent state or dispatching the event to parallel state with higher priority. Most importantly, the recursion procedure of deactivation must stop at the state which contains the transition. This state's id is recorded in the field : mInfo of EVENT type. To distinguish these two opposite parts, mInfo is set to negative number to indicate that the first part should be handled and set to positive number to indicate the second part should be handled.

We define the function 'DispatchDeactivationToChild' to handle the first part's event dispatching work. We define the function 'HandleDeactivation' to handle second part's work. The function 'DeactiveFirstPartJudge' return true if the stack top event is sent to this state and the field : mEvent is eDeactivation and the field mInfo is negative number. The function 'DeactiveSecondPartJudge' return true if the stack top event is sent to this state and the field : mEvent is eDeactivation and the field mInfo is positive number.

Whether event is sent to this automaton is a very common judgement nearly appeared in all the guard of transition. Besides this basic judgement, the type of stack top event being eDeactivation, and mInfo being negative, imply that the stack top event is a deactivation event and first part's logic of deactivation should be executed.

Algorithm 13 Deactive First Part Judge

```
bool DeactiveFirstPartJudge()
{
bool r1 = EventSentToMe(StackTopEvent());
bool r2 = (StackTopEvent()).mEvent == eDeactivation;
bool r3 = (StackTopEvent()).mInfo < 0;
return r1 && r2 && r3;
}
```

Algorithm 14 Deactive Second Part Judge

```
bool DeactiveSecondPartJudge()
{
bool r1 = EventSentToMe(StackTopEvent());
bool r2 = (StackTopEvent()).mEvent == eDeactivation;
bool r3 = (StackTopEvent()).mInfo > 0;
return r1 && r2 && r3;
}
```

This only difference between previous function is that mInfo is greater than 0, it implies second part's logic of deactivation should be executed. So it can be concluded that the field mInfo here play a role of flag that which part(first or second) should be executed when receiving a deactivation event.

Algorithm 15 Dispatch Deactivation To Child

```
void DispatchDeactivationToChild()
{
PopOneEvent();
if NowState's child state is active then
  if NowState's children are parallel states then
    Dispatch Deactivation Event to it's lowest priority
    child state.
  else
    Dispatch Deactivation Event to it's active child state.
  end if
end if
}
```

This function is actually the handling logic of first part of deactivation. If a state has serial childs, the deactivation event should be dispatched to its active child state exactly and if a state has parallel childs, the deactivation event should be dispatched to the lowest priority child state.

This function is just the logic of second part of deactivation. In summary, this algorithm only does three things : first, do exit action, second, dispatch deactivation event of first part logic to it's adjacent higher priority brother state, third, if no brother to dispatch, dispatch deactivation event of second part logic to its parent.

Algorithm 16 Handle Deactivation

```
void HandleDeactivation()
{
PopOneEvent();
if NowState is active then
  if NowState has exit action then
    Dispatch Deactivation Event to it's corresponding action automaton.
  end if
  if NowState has parallel childs then
    Dispatch Deactivation Event to it's last priority parallel.
  end if
  if NowState has parent then
    EVENT evt = Generate Deactivation Event;
    evt.mInfo = -evt.mInfo;
    //here mInfo is positive number.
    //only here dispatch positive info Deactivation Event.
    Dispatch evt to it's parent.
  end if
else
  if NowState has brother state then
    Dispatch Deactivation Event to it's higher priority parallel.
  end if
  if NowState has parent then
    Dispatch Deactivation Event to it's parent.
  end if
end if
}
```

The deactivation part of Controller Automaton is shown below:

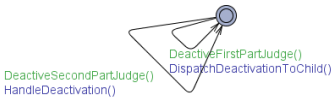


Figure 9: Deactive Logic Automaton

Event Dispatch Logic.

When receiving an event, the controller automaton needs to decide where and how to dispatch the event. When event is dispatched, the first destination must be Chart even the event has specified destination such as send(Event, A). The code send(Event, A) means Event must be sent to state A. However, as described in Section Action, every event must be dispatched to Chart at first and record the real destination in field : mInfo, then let Chart dispatch the event down. The reason of this strange design is that every already active state may have during action which must be done when an event is dispatched. In such a consideration, an event must be dispatched from very beginning which refers to Chart. So the Event Dispatch Logic is created to handle the during action execution and event dispatching.

We define function 'EventDispatchJudge' to judge whether Event Dispatch Logic should be executed. The function 'HandleEventDispatch' is to handle the Event Dispatch Logic. The field mInfo become a flag. If mInfo == -1 then this event will dispatch to its corresponding common automaton. If mInfo != -1 then this event will not dispatch to Common Automaton. As will be described later, Common Automaton executes the transition. Do not dispatch to Common Automaton means that the transition's executions will be ignored. It is easy to understand this design because if the state is not the event's real destination, only during action could be executed.

Algorithm 17 Event Dispatch Judge

```
bool EventDispatchJudge()
{
bool r1 = EventSentToMe(StackTopEvent());
bool r2 = (StackTopEvent()).mEvent >= eValidEventBase;
return r1 && r2;
}
```

This function is to judge whether the stack top event id is greater than eValidEventBase which is actually 40.

Algorithm 18 Handle Event Dispatch

```
void HandleEventDispatch()
{
if NowState's id == StackTopEvent().mInfo then
  StackTopEvent().mInfo = -1;
end if
Dispatch During Event to Action Automaton;
//When Action Automaton executed over, During Event will be deleted.
if StackTopEvent().mInfo == -1 then
  Dispatch this event to Action Automaton;
end if
if NowState has child then
  if NowState's children are parallel states then
    Dispatch this event to NowState's highest priority child;
  else
    Dispatch this event to NowState's currently active priority child;
  end if
end if
if NowState has brother state then
  Dispatch this event to NowState's lower priority brother node;
end if
}
```

Event id less than 40 is reserved for internel use such as eDeactivation=-10 and eActivation=-8. So the event dispatch logic only handle the event corresponding to Stateflow's event which will be mapped to id more than 40. This algorithm shown above does three things : first, do entry action, second, dispatch this event to its corresponding common automaton to run the real state transfer logic, third, dispatch event to child, fourth, dispatch event to its lower priority brother state.

The event-dispatch part of Controller Automaton is shown below:

Activation Logic.

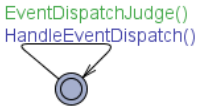


Figure 10: Event Dispatch Logic Automaton

Activation is similar to Deactivatoin, it also has two opposite parts. The first is to dispatch the event to its inactive parent state. However, before dipatching a newly created Activation Event to its parent state, the current Activation Event must not be deleted. This is different from deactivation. So at one time, there may be several Aactivation Events on stack. The second part is to handle the entry action and make decision to choose between running default transition or dispatching the event to child state or dispatching the event to parallel state with higher priority. Most importantly, the recursion procedure of activation must stop at a already activated ancestor state.

We define the function 'DispatchActivationToParent' to handle the first part's event dispatching work. We define the function 'HandleActivation' to handle second part's work. The function 'ActiveFirstPartJudge' return true if the stack top event is sent to this state and the field : mEvent is eActivation and the field mInfo is negative number. The function 'ActiveSecondPartJudge' return true if the stack top event is sent to this state and the field : mEvent is eDeactivation or is eDefaultActivation and the field mInfo is positive number. The const integer value eDefaultActivation represents the event driven a state's common automaton to run default transitions.

Algorithm 19 Active First Part Judge

```
bool ActiveFirstPartJudge()
{
bool r1 = EventSentToMe(StackTopEvent());
bool r2 = (StackTopEvent()).mEvent == eActivation;
bool r3 = (StackTopEvent()).mInfo < 0;
return r1 && r2 && r3;
}
```

Here, the algorithm is very similar to deactivation and the only difference is that it is to judge whether mEvent equals eActivation instead of judging mEvent equals eDeactivation. The field mInfo again is used to distinguish which part(first or second) should be executed.

Algorithm 20 Active Second Part Judge

```
bool ActiveSecondPartJudge()
{
bool r1 = EventSentToMe(StackTopEvent());
bool r2 = (StackTopEvent()).mEvent == eActivation;
bool ro2 =(StackTopEvent()).mEvent == eDefaultActivation;
bool r3 = (StackTopEvent()).mInfo > 0;
return r1 && (r2 || ro2) && r3;
}
```

This function is also very similar to its corresponding deactivation logic. However, in this function, event 'eDefaultActivation' appears at the first time. In activation, there are two types of activation need to be handled : eActivation and eDefaultActivation. The event eActivation is the Activation Event in general sense and the event eDefaultActivation is an event created for dealing with only one special circumstance. The only one special circumstance is that when the current stack top is not an activation event, but the just activated state has serial childs and there are no active child states in this state. In this situation, this state's default transitions should be executed to reach a valid child state. The event eDefaultActivation is just designed for this situation. What's more, when entering a valid child state, the activation for this child state will be dispatched, so this is a recursive procedure and all the related states will be activated properly.

Algorithm 21 Dispatch Activation To Parent

```
void DispatchActivationToParent()
{
if NowState is active then
  PopOneEvent();
else
  StackTopEvent().mInfo = -StackTopEvent().mInfo;
  Dispatch Activation Event to its parent;
end if
}
```

Algorithm 22 Activation Dispatch To Child State

```
void ActivationDispatchToChildState()
{
if NowState has parallel child states then
  PopOneEvent();
  Dispatch Activation Event To NowState's highest priority child state;
else
  if StackTopEvent().mEvent != eActivation then
    EVENT evt = GenerateEvent();
    evt.mDest = StackTopEvent().mDest;
    evt.mInfo = StackTopEvent().mInfo;
    evt.mEvent =eDefaultActivation;
    PopOneEvent();
    Dispatch evt To NowState's Common Automaton.
  end if
end if
}
```

The above two functions just dispatch the activation event to its parent and child. In particular, function 'void DispatchActivationToParent()' is the handling logic the activation's first part. It is worth noting that when dispatching to parent, mInfo is set to negative letting the parent make a fresh start handle the activation which means the parent should start from handing the activation logic by handing the first part logic of activation at first and then the second part. When dispatching to child, eDefaultActivation is generated to handle the previous mentioned 'the only one special circumstance'.

Algorithm 23 Handle Activation

```
void HandleActivation()
{
if StackTopEvent().mEvent == eDefaultActivation
then
  int info = StackTopEvent().mInfo;
  //info stores the real event which causes the activation
  occuring.
  PopOneEvent();
  if NowState has parallel child states then
    Dispatch Activation Event To NowState's highest
    priority child.
  else
    EVENT evt = GenerateEvent(); evt.mInfo = info;
    Dispatch evt To NowState's Common Automaton.
  end if
else
  PopOneEvent();
  if NowState is active then
    ActivationDispatchToChildState();
  else
    if NowState has entry action then
      Dispatch Activation Event To Action Automaton.
    end if
    ActivationDispatchToChildState();
    if NowState has brother state then
      Dispatch Activation Event To brother state with
      lower priority.
    end if
  end if
end if
}
```

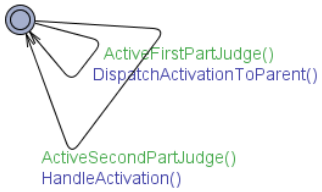


Figure 11: Active Logic Automaton

The deactivation part of Controller Automaton is shown above:

The function 'void HandleActivation()' is the handling logic of the second part of activation. In summary, ignoring the many if-block judgement, the function only does two things : first, do entry action, second, dispatch first part activation event to its brother state with lower priority or dispatch eDefaultActivation event to its corresponding common automaton to run crrresponding Stateflow's default transitions.

5.4 Action

There are two types of Actions which are namely Conditional Action and Transitional Action. The way we treat the two types of actions are the same. The only difference between Conditional Action and Transitional Action is the automaton to which they belongs. The conditional action is executed in the Condition Automaton whereas Transitional Action runs in Common Automaton.

For action which is just algebraic calculation, we transform the action into one Uppaal Transition whose action is the same as the action in Stateflow.

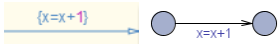


Figure 12: Algebraic Action Transform

For action which is a kind of event dispatch. We need to use an immediate Uppaal state between the start state and the end state. Before the immediate Uppaal state, the event is dispatched and after the immediate Uppaal state, the transition wait the event to be executed over. To achieve this, before dispatching the event, we dispatch a sentry event called 'EventOver'. After the event 'EventA' executed over, 'EventA' is deleted from the stack. So the sentry event 'EventOver' is on top of the stack. The transition after the immediate state gain the ability to run.

Figure 13: Event Action Transform

5.5 Entry/During/Exit Action

The entry/during/exit actions are executed in the controller action automaton. These three actions are transformed to three self cycles in the Action Automaton. When the execution is over, the Action Automaton pop up the stack top event which dirves this Action Automaton to run.

Generate Action conversion using the method described in the previous section.

Then, The add guard on three self cycle transtions.

Entry corresponding transition guard:
EventSentToMe() && StackTopEvent().mEvent == eActivation

During corresponding transition guard:
EventSentToMe() && StackTopEvent().mEvent == eDuring

Exit corresponding transition guard:
EventSentToMe() && StackTopEvent().mEvent == eDeactivation

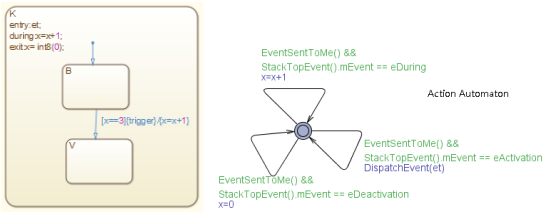


Figure 14: Action automaton

5.6 Transition

The label on transition in Stateflow is of the following form:

event[*condition*]{*condition action*}/{*transition action*}

We put the execution of 'event[*condition*]{*condition action*}*' into Condition Automaton.*

We put the execution of '/{*transition action*}*' into Common Automaton.*

Condition Automaton.

Condition Automaton is responsible for handling judgement on transition and executing the condition action. Besides, every transition in Stateflow has priority, every transition should be executed in the ascending priority order. So, There is another duty for Condition Automaton : Executing the transition by its priority order.

To solve the execution order of the transition, we introduce an array of integer type called 'mPathSelect'. The index of the array is correspond to the depth from the current active state.

For a state or junction, The depth of this state or junction is defined as the minimum transition num to one specific state. From this specific state, the state or junction can be reached.

For every index i, mPathSelect[i] is an integer. The transition which source is the state at depth i could be executed only when the transition's priority is equal to mPathSelect[i].

We also introduce an array which is of type bool called 'mPathJudge'. This array stores the condition's execution result of every transition and will be used by Common Automaton. The index of the array is the id of the transition.

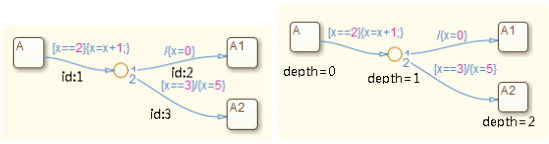


Figure 15: Transition Example

First, we map each state or junction in Stateflow to exactly one Uppaal state, we call these Uppaal states 'original state'. Second, between any two states in Uppaal, we add an immediate state given name 'immediate state'. Third, we connect 'original state' to 'immediate state', the guard on transition is of the form : 'mPathSelect[i] == Stateflow-Transition's Priority'. This ensures the transition is executed by it's priority order. Then for each 'immediate state', it must have a precursor node and a successor node. We connect 'immediate state' to its precursor node, the guard of this transition is the negation of the corresponding Stateflow-Transition's guard. This corresponds to the situation that Stateflow-Transition's condition is not satisfied and we need to go back and try next transition. When coming back, mJudgePath[id] is set false to show that this transition could not be taken. Similarly, We connect 'immediate state' to its successor node which corresponds to the situation that Stateflow-Transition's condition is satisfied and we should proceed and set the mJudgePath[id] to true to show that this transition can be taken.

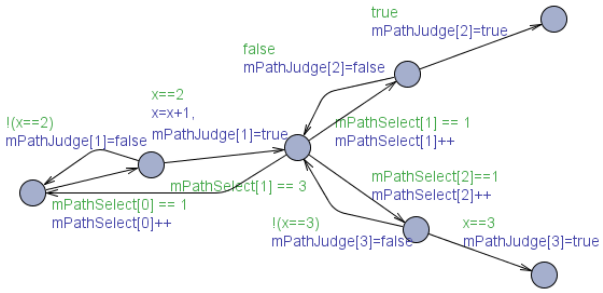


Figure 16: Condition Automaton

From the above figure, .

Common Automaton.

Due to Condition Automaton has done all the judgement and stored the result into a bool array namely 'mJudgePath'. The Common Automaton should just use this bool array 'mJudgePath' directly. The transition action is not handled yet, so in Common Automaton, transition action is executed.

5.7 Time Operation

The time operation in stateflow in just a time related judgement, such as 'after(3,sec)'.
In stateflow, time is also a kind of event. All we need to do is to count the happening times of the event.Such as the example 'after(3,sec)'. The converted uppaal judgement is 'mHappenedTimes[sec] >= 3 '. 'mHappenedTimes' is an array which record the happened times of all the event. The

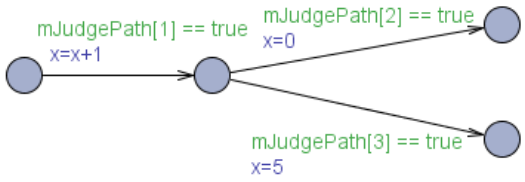


Figure 17: Common Automaton

Figure 18: Time conversion

array's index is the event's id.
The time operation is of four kinds and the translation rule is shown below.
after : after(3,sec) → mHappenedTimes[sec] >= 3
before : before(3,sec) → mHappenedTimes[sec] <= 3
at : at(3,sec) → mHappenedTimes[sec] == 3
every : every(3,sec) → mHappenedTimes[sec] % 3 == 0

5.8 Tool transformation

From a higher view, our tool can be split into three parts : 1.Read simulink file into memory; 2.Transfer the model in the memory; 3.Output the modified memory model into file; This tool is implemented in java. To read and write xml file,

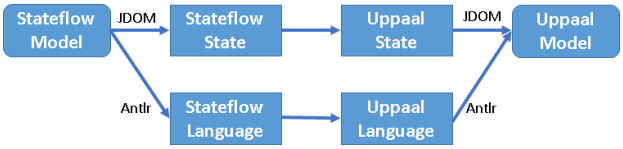


Figure 19: Tool Design

the jdom jar library is used. The technology used to parse the text language in stateflow into abstract syntax tree is Antlr. Due to reason that the input file of stateflow is a compressed binary file format which is ended with '.slx'. We first take it as zip formatted file and unzip the compressed file. Then a folder which contains all the information about the stateflow model appears. The output file of the uppaal model is just a simple xml file restricted with a well defined DTD structure. The total java code of our tool is 14590 LOC. The uppaal meta function code is 527 LOC. The total code includes some library is 22380LOC.

6. VERIFICATION RESULTS

The 'switch_on' counter is designed to count how many times the event 'switch_on' happens. In this diagrams, there is a divide by zero error marked in red. To speak in an other word, the integer variable y can not be zero. So we have the first property : y!=0 in any time and any step. The statement y=y-3 is only statement which can cause y to be 0 and it is executed before enter the junction in state B. So we also need to verify that the junction in B is reachable. This is the second property : junction in B is reachable.

In Uppaal , there is a corresponding map to objects in Stateflow. the integer variable 'y' in Stateflow is mapped to

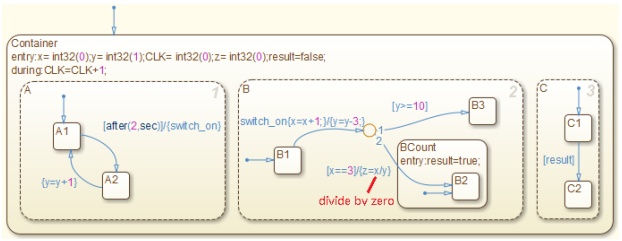


Figure 20: Experiment Case

an integer variable 'Chart_y' in Uppaal. The junction of B in Stateflow is mapped to state with name 'Process_Chart_Container_B.SSID49'.

The Stateflow diagram also desmonstrates that y is set to 0 before reaching the junction in state B and y is not equal to 0 before reaching the state B1 in state B. So the y is set 0 if and only if the transition between B1 and junction is executed.

So our properties can be defined as follows:

Table 1: Property List

Property1	A[] Chart_y != 0
Property2	E<>Process_Chart_Container_B.SSID49
Property3	E<>Process_Chart_Container_B.SSID49 and Chart_y == 0
Property4	E<>Process_Chart_Container_B.Chart _Container_B_B1 and Chart_y != 0

The verification result shows that y is set to 0 if and only if the transition between B1 and the junction named SSID49 is executed which is consistent with previous expection.

Table 2: Verification Result

Property	Verdict	Memory(MB)	Time(SEC)
Property1	No	414	0.002
Property2	Yes	414	0
Property3	Yes	189	0.373
Property4	Yes	289	0.19

The experiment above shows the transformation and verification of Stateflow's basic elements. The dividing by zero error is checked by the property : In all states, y can not be zero. What's more, Uppaal can not only check the value of variable but also check the reachability of state. In the following case of temperature control, we check whether the ruinous state 'Destroyed' can be reached.

An int typed data 'temperature' is set to 60 at first and decrease to 50 when entering state 'Stopping'. If temperature is less or equal than 50, the diagram leaves the state 'stopping' and enter state 'Running' to increase the temperature. There is a constraint in warming up, that is : if the temperature is greater or equal than 90, the ruinous state 'Destroyed' will be reached which means the whole logic is wrong. Besides, the warming up process has another

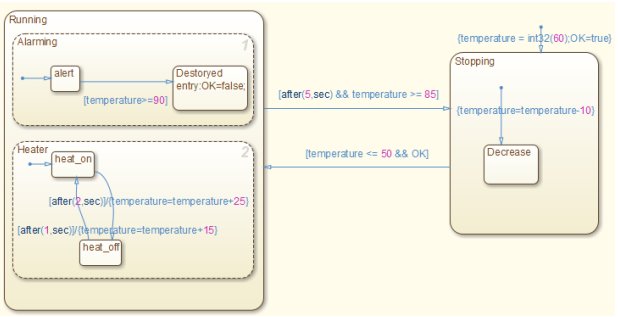


Figure 21: Temperature Control Stateflow Model

constraint : the heating process must last for at least 5 seconds. The condition '[after(5,sec) && temperature >= 85]' on transition from 'Running' to 'Stopping' guarantees this constraint.

Our goal is to verify that whether the ruinous state 'Destroyed' can be reached. If the ruinous state 'Destroyed' can be reached the totoal logic is wrong.

The partial transformed uppaal automaton is shown, our transformed uppaal models preserve the hierarchical structure of corresponding Stateflow diagrams. The aim of verification is to check whether the Stateflow state 'Destroyed' in parent state 'Alarming' can be reached. Stateflow State 'Alarming' is mapped to an uppaal automaton named 'Process_Chart_Running_Alarming' and 'Destroyed' is mapped to an uppaal state called 'Chart_Running_Alarming_Destoryed' in the automaton just mentioned.

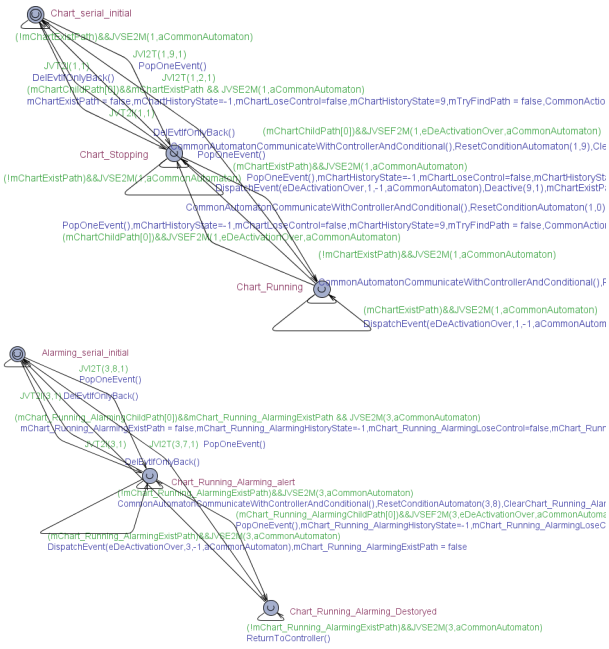


Figure 22: Temperature Control Uppaal Model

The verification is that the state 'Destroyed' can be reached, so the logic is wrong.

Table 3: Property List

Property1	E<> Process_Chart_Running_ Alarming.Chart_Running_Alarming_Destoryed

Table 4: Verification Result

Property	Verdict	Memory(MB)	Time(SEC)
Property1	Yes	29	0.013

Now, we apply our case to an industrial Stateflow model. This case is a MVB control system. There are parallel distribued MVB machines which are connected by the central control bus. When there are mutiple MVB machines on the central bus, they need to elect a master and the rest becomes slaves. What's more, there are time limits for being master which means a master must surrender its privileges after a fixed time. In the interaction process, there will be inconsistencies such as two master may appear at the same time. Without loss of generality, we model two MVBs connected

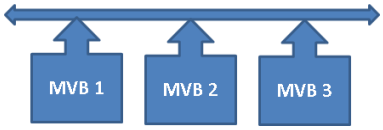


Figure 23: MVB Overview

together in Stateflow.

The logic of MVB1 and MVB2 are almost same. Due to the complexity of the Stateflow model, our uppaal model is complex too.

The final aim of our verification is to check whether MVB1 and MVB2 can be master at the same time. If there is a possibility that two MVBs are master at the same time, the problem called 'RAC Brain Split' arises. What's more, to verify that our transformation is right, another three randomly selected properties are proposed. These three properties are the run time result of Stateflow. If the verification shows that these properties are satisfied, then our transformation is right at a very high rate.

As the verification result shows that there is really a chance that two MVBs become master at the same time. This can be considered as the bug of the MVB international standard agreement. This bug has already been be submitted before and our tool find this bug by verification.

7. CONCLUSION AND FUTURE WORK

We have introcduced a systematic method to convert and validate the Stateflow model by Uppaal Verfication Tool. Our conversion from Stateflow to Uppaal covers many advanced features such as conditional action, deactivation and activation of a state. The conversion preserves the execution semantics of Stateflow. The converted Uppaal model is executable and this simulation ability enables us to check whether our conversion is right. Users can use Uppaal to check many safety and liveness properties. Serveral exam-

Figure 24: MVB Stateflow Model Overview

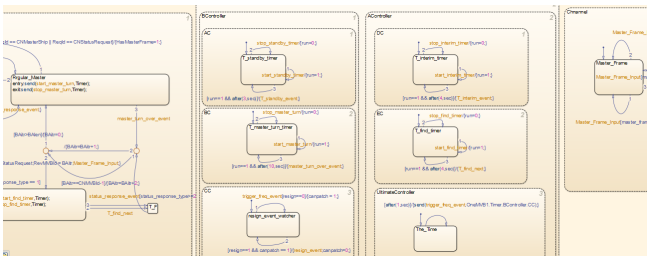


Figure 25: MVB1

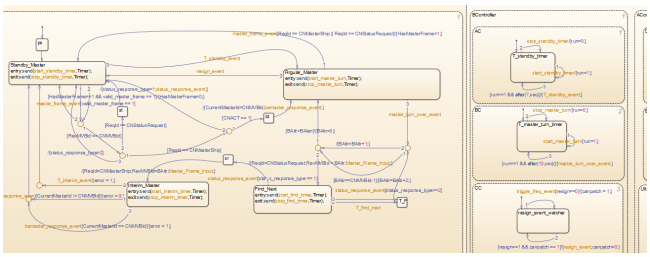


Figure 26: MVB2

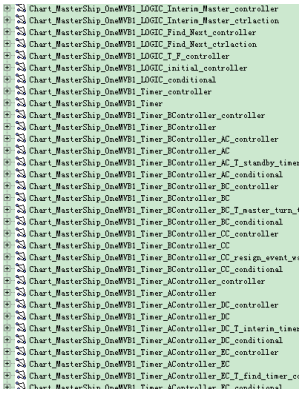


Figure 27: MVB Uppaal Overview

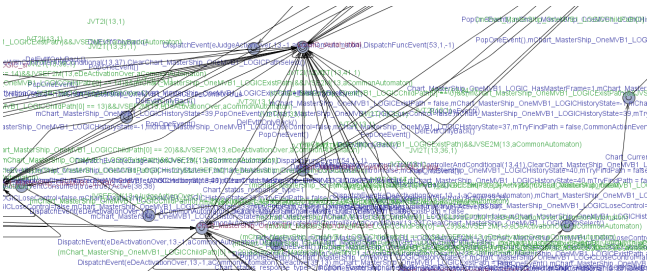


Figure 28: MVB Uppaal LOGIC

Table 5: Property List

Property1	E<> Process_Chart_MasterShip_OneMVB1_LOGIC .Chart_MasterShip_OneMVB1_LOGIC_Rrgular_Master and Process_Chart_MasterShip_OneMVB2_LOGIC .Chart_MasterShip_OneMVB2_LOGIC_Rrgular_Master
Property2	E<> Process_all_controller.mTotalTime == 2 and ProcessChart_MasterShip_OneMVB1_ LOGIC.ChartMasterShip_ OneMVB1_LOGIC_Standby_Master and Process_Chart_MasterShip_OneMVB2_LOGIC.Chart_ MasterShip_OneMVB2_LOGIC_Standby_Master
Property3	E<> Process_all_controller.mTotalTime == 14 and Process_Chart_MasterShip_ OneMVB1_LOGIC.Chart_MasterShip_OneMVB1_ LOGIC_Find_Next and Process_Chart_ MasterShip_OneMVB2_LOGIC.Chart_MasterShip _OneMVB2_LOGIC_st
Property4	E<> Process_all_controller.mTotalTime == 16 and Process_Chart_MasterShip_ OneMVB1_LOGIC.Chart_MasterShip_OneMVB1_ LOGIC_Standby_Master and Process_Chart_ MasterShip_OneMVB2_LOGIC.Chart_ MasterShip_OneMVB2_LOGIC_Rrgular_Master

Table 6: Verification Result

Property	Verdict	Memory(MB)	Time(SEC)
Property1	Yes	628	2.349
Property4	Yes	325	0.01
Property3	Yes	326	0.02
Property2	Yes	327	4.42

ples have been applied to show whether some properties in these examples hold. The model of Uppaal is a timed automaton, so the location property can also be checked. The verification of location property such as whether location1 and location2 can be reached at the same time is the flash point of our work.

Until now, the conversion of randomized function in Stateflow is not supported. Randomization is a very important aspect in modeling. Some rarely happened situation can be modeled by randomization such as the loss of packet in network transportation. We will convert the randomized element in Stateflow to Uppaal in our next work. Uppaal doesn't support floating points and bit wise integer numbers such as int8, int16 and uint8. Our next step is to convert these types into Uppaal. What's more, the structured data in Stateflow is not covered in our conversion work yet, so in our next work, Stateflow's structured data will also be included.