Unpublished Report
Converting Simulink Stateflow To Uppaal Timed Automata

Yixiao Yang[1],Yu Jiang[1,2],Ming Gu[1],Jiaguang Sun[1]

Department of Computer Science, University of Illinois at Urbana-Champaign, USA[2]
School of Software, Tsinghua University, TNLIST, KLISS, Beijing, China[1]

ABSTRACT

Simulink is a widely used tool for the model-driven development of control systems in industry, which is started with the stateflow model construction, simulation, validation and followed by the code generation to a physical implementation. However, the increasing demand of verification for safety critical applications brings new challenge to the stateflow because of the lack of formal semantics.

In this paper, we present STU, a self-contained toolkit to address the verification challenge. The tool translates the stateflow model into the formal timed automata model. It reads the stateflow model file of Simulink, and generates the semantics equivalent timed automata model file of Uppaal, which can be verified directly. Most of the advanced modeling features in stateflow such as the event stack, composite state, junction, transitional action, and timer etc., are supported. Then, with the strong verification power of Uppaal, we can not only find bugs in stateflow that will not be detected by the Simulink Design Verifier, but also check more temporal properties. The evaluation on artificial examples and real industrial applications demonstrates the effectiveness and scalability of STU.

The example, video demo and tool are available at:
https://sites.google.com/site/jiangyu198964/home

Categories and Subject Descriptors

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

General Terms

computer-aided software engineering

Keywords

Simulink Stateflow, Uppaal Timed Automaton, Verification

1. INTRODUCTION

Simulink is a widely used tool for the model driven design of software systems, which provides well support for the graphical stateflow model construction, interactive graphical model simulation, some basic design validation, and the C, C++, and VHDL code generations [6]. It has been successfully applied to various industry and livelihood areas such as medical devices and avionics development. For those safety-critical applications, the model validation technique is not enough to ensure the correctness. More rigorous formal techniques such as model checking and theorem proving should be applied to check the correctness of the model.

However, Simulink provides limited support for the formal verification of stateflow with the toolbox Design Verifier, which mainly checks some basic static properties such as divide by zero. The main challenge for the verification is that the execution semantics of stateflow is too complex, which is described in a 1366 pages user guide informally. The advanced modeling feature such as event stack, event interruption, complex state active and deactive mechanism, boundary transition, and transitional action etc., are not easy to formalize for verification. Although there are many existing works on the translation based verification of stateflow, most of them works well within their own domain while abstracting some domain unrelated modeling features.

In this paper, we present STU, to translate the stateflow model into the timed automata for formal analysis based on the model checking tool Uppaal [1]. Timed automata can be used to model and analyze the timing behavior of systems, and methods for checking both safety and liveness properties of timed automata have been well developed and intensively studied in Uppaal. Most of the advanced stateflow modeling features as well as the implicit event driven and interrupt stack are addressed in the tool, which consists of three parts:

- First, a parser is designed and implemented to extract the stateflow model from the Simulink project file. The stateflow model is stored in the project file with many other accompanied project contents, we have to analysis the file and locate the keywords and information used to store the model elements.
- Second, a translator is implemented according to the well designed mapping rules from the stateflow to the timed automata, where most of the advanced modeling features are captured in the formalized rules.
- Finally, a storer is implemented to organize the translated timed automata into the model file that can be read by Uppaal for verification directly.

The three parts are seamlessly integrated together in STU to support the formal analysis of stateflow model. With a

ESEC/FSE '5, November 16-22, 2015, Italy

wider coverage of stateflow modeling features captured in STU, and the strong verification capability of Uppaal, more comprehensive validation can be accomplished. Potential errors that may not be detected in simulation or Simulink Design Verifier will be found during the verification, and the code generations on the verified model will be more reliable for safety critical applications.

2. BACKGROUND

In this section, we present some background on the elements and semantics of stateflow and timed automata.

2.1 Simulink Stafeflow

Stateflow is an extended hierarchical state machine which contains sequential decision logic and synchronization events to represent the behavior of systems. There are mainly six frequently-used elements in stateflow : *State, Transition, Junction, event, Action* and *Timer.*

State describe the operating mode of the system. The occurrence of an event triggers the execution of the model by making states active or inactive depending on events and conditions during simulation. Each state may contain two types of decomposition which are parallel or serial connected. The serial decomposing state must have at least one default transition with only one sub-state activated, while the parallel decomposing state does not have any default transition with all sub-states activated at one time.

Transition is the edge between two states, representing the mode change from the source state to the destination state. Each transition is attached with four characterizations:

[*event*] [*condition*] [*conditional action*] / [*common action*]

Where *event* specifies the explicit or implicit event that triggers the execution of this transition, *condition* is a boolean expression that allows the transition to be taken with value true, *conditional action* is the operations that is immediately executed when the condition is met, *common action* is the operation that will be executed when the condition is met and there is a non-interrupted valid path between the source and target state. Each transition also has a implicit priority of execution, determined by information such as the hierarchy level of the destination state, etc.

Action contains the operations attached on the transition, and the three kinds of operations attached on the state, *entry action*, *during action*, and *exit action.* During action is executed when the state is already active. Exit action is executed when the state changes from active to inactive.

Junction contains two types, *connective junction* and *history junction*, where the former enables the representation of different possible transition paths for a single transition, and the later represents historical decision points bade on historical data relative to state activity.

Timer is used to specify the time related behaviors of the system, which is characterized as:

$$[TmOp\ (Num, Event)]$$

where *TmOp* is the three types of time related operation *before, after*, and *at*, *Num* is the number used to quantify the length of the time period, and *Event* consists of three system reserved keywords: *sec, msec*, and usec which represents second , millisecond , and microseconds, respectively.

2.2 Uppaal Timed Automata

A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where clock variables evaluate to real numbers, and all clocks progress synchronously. It can be defined as a tuple consists of six elements: (L, L_0, C, A, I, E), where L is a set of locations, l_0 is the initial location, C is a set of clocks, A is a set of actions, $B(C)$ is a set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$ ($x, y \in C$, and $\bowtie \in \{<, \leq, =, \geq, >\}$), I is a set of invariants on the location, and $E \subseteq L \times A \times B(C) \times 2^C \times L$ denotes a set of transition edges. The edge connects two locations with an action, a guard and a set of clocks, formalized as $(l \xrightarrow{g,a,r} l')$ when $(l, a, g, r, l') \in E$. The transition represented by an edge can be triggered when the clock value satisfies the guard labeled on the edge. The clocks may reset when a transition is taken. A system can be modeled as a network of several timed automaton in parallel with synchronous actions defined on channel *ch*. The input action *ch*? represents receiving an event from the channel *ch*, while the output action *ch*! stands for sending an event on the channel *ch*. 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 a transition separately, or synchronize with another automaton.

The model-checker Uppaal jointly developed by Uppsala University and Aalborg University is based on the theory of timed automata, and the query language used to specify properties to be checked, is a subset of TCTL (timed computation tree logic). It has been applied successfully ranging from communication protocols to multimedia applications.

3. RELATED WORK

Lots of works have been done to ensure the correctness of the Simulink stateflow model, which can be classified into two categories, simulation based techniques and verification based techniques. The simulation based technique is adopted widely, while the main challenge is to solve the coverage of the simulation patterns. Many researchers have developed test generation tools for Simulink designs including Reactis [11], T-VEC [4], Beacon Tester [13], and AutoMOTgen [8] etc. These tools use combinations of randomization and constraint solving techniques to generate test cases, to guarantee that coverage goals over model elements are satisfied. Recently, the symbolic analysis has also been successfully applied to improving the simulation coverage of Simulink stateflow model [2].

For the verification based techniques, the main challenge is that Simulink Stateflow lacks a formal and rigorous definition of its semantics. Many researchers have defined several types of formal semantics for Stateflow, and developed many specialized tools for translating subsets of the model to pushdown automata [5], Lustre [9], SMV [10], PAT [12] and SAL [15], which can be verified through the corresponding supporting tools. Most of them performs well within their own domain while abstracting some domain unrelated modeling features. For example, in SMV based translation [3], they focus and provide well framework to ensure the function correctness, while the hierarchical states and events are abstracted. In the PAT [7] based verification technique, they covered most advanced features of Stateflow, while with lim-

ited support of the event interrupt dispatch mechanism as well as the time operation support.

Compared to the those works that performs well within their domain and focus, our work tries to cover most of the stateflow advanced modeling features, including the time mechanism that has never been addressed before. We also formalize the complex event stack and execution interrupt mechanism which are limitably supported before. Right now, our approach covers all semantic examples in the stateflow user guide [14]. Because the execution semantics of stateflow is described in informal natural languages based on examples, it is not possible to formally prove the equivalence and correctness of the transformation. We acquire the correctness by carefully compare the simulation results of the transformation model, including the value and state sequence step by step, in the same way as previous works.

4. TRANSFORMATION RULES

The most important and difficult task is to overcome the gap between the Simulink stateflow execution semantics and that of the Uppaal timed automata. As introduced in the background, the key differences between Simulink stateflow and Uppaal timed automata are :

(1) Simulink stateflow transition is driven by event, and the execution order of every step of event is in deterministic sequential manner, interruptible and recursive with stack. While the Uppaal timed automata is executed in parallel, and driven by the channel synchronization without the support of stack.

(2) Simulink stateflow supports hierarchy structure which is combined with recursive activation-deactivation mechanism, the interruptible events and actions very closely. While the Uppaal timed automata supports single state and non-interrupt transition.

To simulate the complex model and execution semantics of Simulink stateflow, an array based data structure for event and an entirely new cooperative mechanism for timed automata in Uppaal are designed and introduced.

4.1 Event Stack Basis

In the stateflow, the event dispatching and processing mechanism, that if a new event is dispatched, the current executing event must be hang up until the execution of the newly dispatched event is over, is very similar to the function invocation mechanism in software program. However, in timed automata, there is synchronous channel among parallel automata and no stack at all. The key idea to simulate the stateflow event stack mechanism is to build a virtual stack in Uppaal. We use the c-like structured array in Uppaal to build a stack. Every element of the virtual stack is a data structure noted as **Event**, which records all information related to an event in stateflow. Where each element in the structure node is described as:

1. *mEvent* is the variable used to distinguish different events in stateflow. We assign a unique integer number to this variable for each stateflow event.

2. *mDest* is the variable used to map the event to a corresponding *Uppaal controller automaton* originated from a stateflow composite state. We assign a unique integer number of the automaton number to this variable.

3. *mDestCPosition* is the variable used to imply the corresponding *ordinary Uppaal automaton* state originated from stateflow cross-boundary transition. We assign a unique integer number of the state to this variable.

4. *mAutomatonType* is the variable used to map the event to the corresponding Uppaal *controller automaton* originated from stateflow state with decomposition or attached actions. This kind of state will be translated into four cooperative automata *(controller automaton, action automaton, condition automaton* and *common automaton)*.

5. *mValid* is the variable used to denote whether this event is valid or not at present. If the event is on the top of stack and is invalid, the event will be deleted by a daemon automaton, which is responsible of deleting invalid event on the top of stack, and dispatchomg *System Event* when the stack is empty.

6. *mInfo* is the variable used to record some simple flag.

Listing 1: The Definition of the Event Structure in the Uppaal Timed Automata

```
structure Event {
  /* Event Identification */
  int mEvent;
 /* Event Destination */
  int mDest;
  int mDestCPosition;
  int mAutomatonType;
  /* Whether event should be deleted */
  bool mValid;
  /* Some simple extra infomation */
  int mInfo;
}
```

The virtual stack is the basic element to simulate the Stateflow semantics in Uppaal. It is empty in the initial translated Uppaal automata, and is pushed and popped during the runtime simulation and verification. When the stateflowblack generate an event within a transition or a state operation, the translated Uppaal automaton will take the corresponding transition, then dispatch and push an *Event* element into the stack dynamically. Every transition in the translated Uppaal parallel automata will look at the top of the stack, to check whether the event belongs to the parent automaton of the transition. If yes, the transition will take. When the stateflow finishes an execution cycle, the *Event* element will be popped during the corresponding transition in Uppaal timed automaton. The procedure above is mainly accomplished through four functions *DispatchEvent(), PushOneEvent(), PopOneEvent(), EventSentToMe(),* and the *StackTopEvent()*, where details are demonstrated in the algorithms as below.

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 running environment. In Stateflow it is generated periodically and is used to make the suspended automaton run again.

Algorithm 1: 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;
}
```

Algorithm 2: Pop One Event

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

Algorithm 3: Event Sent To Me

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

Algorithm 4: Dispatch Event

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

Algorithm 5: Stack Top Event

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

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 6: 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.

Algorithm 7: Is Stack Empty?

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

Algorithm 8: Generate Driven Event

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

Take the stateflow model in the Figure 1 as an example to show the dynamic construction and operation of the event stack. There are two parallel composite states A and B, and an event EA communicates between the two states during the execution in Simulink. Then, the abstracted event related *controller automata* is presented at the right side.

During the transition $A1 \rightarrow A2$ contained in the automata for the composite state A, the event EA is pushed into the stack with the function *DispatchEvent()* on the translated pseudo controller automata. The function *EventSentToMe()* on the transition $B1 \rightarrow B2$ returns true if the event on the top of the stack is sent to the parent automaton which contains the transition $B1 \rightarrow B2$. When the transition $B1 \rightarrow B2$ takes, the event EA is popped from the stack with the function *PopOneEvent()*. The graphical representation of the virtual stack is listed in the Figure 2, where the *DrivenEventToA* is the system event.

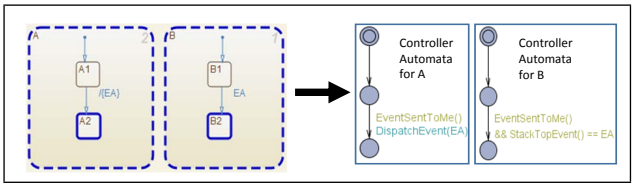

Figure 1: Parallel State with Event Communication

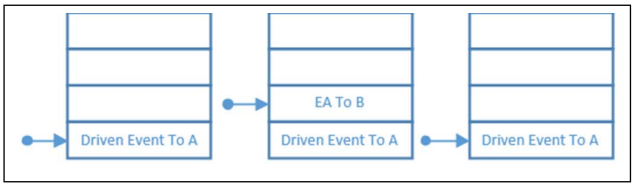

Figure 2: Stack changes during the execution.

Based on the virtual stack and the operations, we we translate the stateflow model. There are mainly six frequently-used elements in stateflow *State, Transition, Junction, Event, Action* and *Timer*. The *Event* and *Action* elements are attached on the *State*, and *Transition*. The junction can be scanned through *Transition*. Hence, we demonstrate the transition rules for the *State*, *Transition*, and *Timer* below.

4.2 State Transformation Rule

For the regular single state without decomposition and attached actions, the transformation is easy and straightforward. We just map the stateflow state to the Uppaal automata state. But for the complex stateflow state with the decomposition or with the attached actions, we need to translate it to four parallel automata:

1. *Controller Automaton* is used to simulate the event dispatch and processing mechanism within this stateflow state. It controls and determines how to dispatch the event by initializing, popping, and pushing the element of the virtual stack.

2. *Action Automaton* is responsible for handling the attached actions (*entry, during, exit*). For the composite state without the attached actions, this automaton will not be generated.

3. *Condition Automaton* is used to test the guard on each transition contained in this composite state, and store the boolean results into the corresponding array. The conditional action is also attached on this automaton.

4. *Common Automaton* reads the array initialized by the *Condition Automaton* and execute the satisfied transition contained in the composite state. The transitional action is also attached on this automaton.

For the activation of state in stateflow, it should judge whether its parent is activated or not. If not, its parent should be activated first. In order to simulate the semantics, the corresponding *controller automaton* will push an activation event corresponding to the state itself onto the stack first, and recursively pushes the activation event associated with the automaton originated from the parent onto the stack, until the top composite state arrives. For the deactivation of state in stateflow, it is similar to the activation procedure. This task are translated to two self cycles in the *controller automaton*, and is attached with the function *EventActivationLogic()* and *EventDeActivationLogic()*, the details of which are presented as below.

Algorithm 9: Event De-Activation Logic

```
bool EventActivationLogic()()
{
DeactiveFirstPartJudge();
DeactiveSecondPartJudge();
DispatchDeactivationToChild();
HandleDeactivation();
}
```

Algorithm 10: 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 11: Deactive Second Part Judge

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

Algorithm 12: 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
}
```

For the execution of the corresponding *entry, during*, and *exit* action in stateflow, it will be captured by the translated *Action Automaton* with three self-cycle transition. After the execution of the *controller automaton*, the *action automaton* will pop the stack top event for test of the guard. The guard on the three transitions are StackTopEvent().mEvent == eActivation, StackTopEvent().mEvent == eDuring and the StackTopEvent().mEvent == eDeactivation, respectively.

Algorithm 13: 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 other two *condition automaton* and *common automaton* are mainly for the stateflow transitions contained in the composite state, and will be described in the following subsection in detail. Taking the example presented in the Figure 3 to detail the transformation. There are three actions attached on the composite state A. The abstracted translated automata for this composite state is presented.

4.3 Transition Transformation Rule

Within the stateflow model, each *transition* is attached with four characterizations: *event, condition, conditional action*, and *transitional action*. We need to incorporate all these elements onto the translated Uppaal timed automata originated from the composite state as below:

1. *event* is transformed into a unique integer as described in the event stack transformation.

2. *condition* is transformed into the guard of the transition in the corresponding *condition automaton*.

3. *conditional action* is transformed into the action of the transition in the corresponding *condition automaton*.

4. *transitional action* is transformed into the action of the transition in the corresponding *common automaton*.

During the transition transformation procedure, an implicit characterization about the transition priority should be addressed in the condition automata. When there are

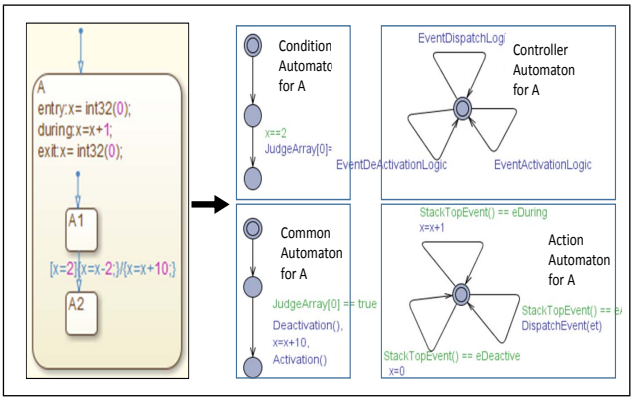

Figure 3: Composite State transformation integrated in the Controller and Action Automaton.

multiple transitions starting from a state, we should keep the determinism execution sequence of Stateflow in Uppaal.

First, we initialize an int array named *mPathSelect[ ]* to simulate the priority. The array index represents the depth of the source state of the transition, and the value is the current priority. As presented in the Figure 4, the depth of the state or junction is defined as the minimum transition number to one original state, from which, the state or junction can be reached. The transition whose priority match with the value of the array will be executed. Besides, a boolean array named *mPathJudge[ ]* is initialized to store the condition test result of every transition, and will be used by *common automaton*. The index of this array is the *id* of the corresponding transition.

Then, we add an intermediate state s_i between any two automata states s_1 and s_2 originated from the stateflow states or junctions. Then, the stateflow transition $s_1 \rightarrow s_2$ is broken to $s_1 \rightarrow s_i$ and $s_i \rightarrow s_2$ in automata. The guard on the automata transition $s_1 \rightarrow s_i$ is : $mPathSelect[i] == Priority\ of\ Stateflow\_Transition$, which ensures the transition is executed by its priority order. The action on the automata transition $s_1 \rightarrow s_i$ is the statement of increasing the value of *mPathSelect[i]* by 1. The guard g_i on the automata transition $s_i \rightarrow s_2$ is the guard from the stateflow transition $s_1 \rightarrow s_2$. The action on the automata transition $s_i \rightarrow s_2$ is the *conditional action* statement of the stateflow transition $s_1 \rightarrow s_2$, and the assignment of the a boolean array element *mPathJudge[i]* with value *true*. When the guard g_i is not true, we should add an additional transition, $s_i \rightarrow s_1$, to roll back to the initial state for the further test of transitions with lower property. In this case, the *mJudgePath[i]* is set as false to show that this transition could not be taken. Also, if s_2 is originated from a stateflow junction, we also need to add a transition $s_2 \rightarrow s_1$ for the roll back of junction.

Based on the array *mJudgePath[]* initialized in the *condition automata*, we construct the *common automata* for *transitional actions*. Each stateflow state and junction is mapped to an automata state without the intermediate state, and the stateflow transition is mapped to an automata transition directly. The guard is the expression *mJudgePath[] == true* and the *transition action* statements on the stateflow are mapped to the action on the automata transition directly. A translation example for the stateflow transitions is presented in the Figure 4.

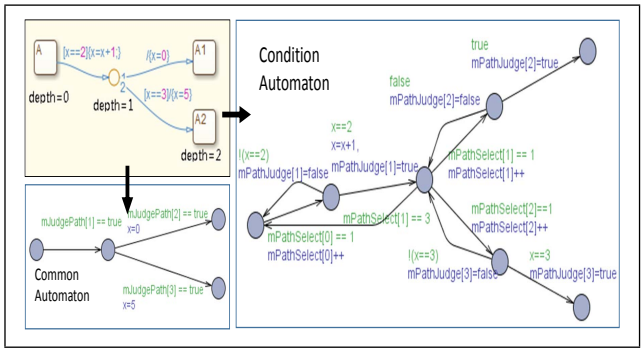

Figure 4: Transition transformation integrated in the Common and Condition Automaton.

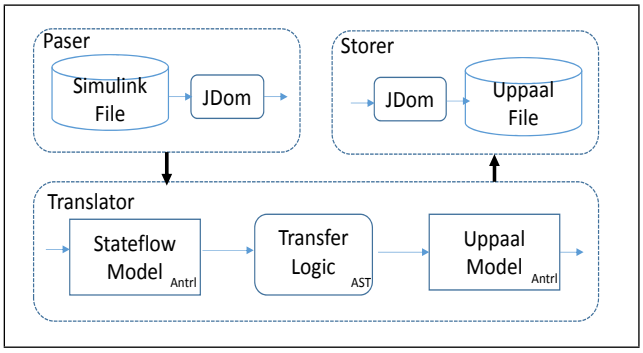

Figure 6: Tool Design

4.4 Timer Operation

Within the stateflow model, *Timer* is characterized as *[TmOp (Num, Event)]*, where *TmOp* is the three types of time related operation (*before, after*, and *at*), and *Num* is the number used to quantify the length of the time period denoted by *Event*. The time operation in stateflow is based on event and is usually used as a time related condition on the transition. We need to count the happening times of the event using an array *mHappenedTimes[ ]* when the event is dispatched with the function *EventDispatch()* , and the index of the array equals to the *id* of the event. Then, the translation rules for the four types of time operations are:

$$
\begin{aligned}
after(Num, Event) &\rightarrow mHappenedTimes[Event] >= Num \\
before(Num, Event) &\rightarrow mHappenedTimes[Event] <= Num \\
at(Num, Event) &\rightarrow mHappenedTimes[Event] == Num \\
every(Num, Event) &\rightarrow mHappenedTimes[Event]\%Num == 0
\end{aligned}
$$

An example of time operation such as *after(3,sec)* is translated as below.

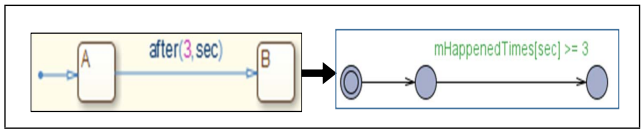

Figure 5: Time transformation integrated in the Condition Automaton

Based on the rules for the event driven semantics, state, transition, and timed operations, the stateflow model can be mapped to a demon automaton, and a set of common automata, action automata, controller automata and condition automata parallel connected in Uppaal.

4.5 Tool Implementation

The tool mainly consists of three parts, a parser, translator, and a storer, and is implemented in Java language with two supporting libraries (JDOM used for read and write xml file, and Antlr used for abstract syntax tree construction and updation). The parser extracts the stateflow model from the Simulink project file into memory. The translator transfers the stateflow model and reconstruct the abstract syntax tree in the memory, according to the transition rules presented in the previous section. The storer organize and output the updated abstract syntax tree to the Uppaal model file. presented in the Figure 6.

The three parts are seamlessly integrated together in STU to support the formal analysis of stateflow model based on Uppaal. Overall, the total java code of our tool is 14590 lines of code, and can be downloaded from the website [].

5. EXPERIMENT RESULTS

In order to demonstrate the efficiency and scalability of the translation tool STU, we apply it to some artificial stateflow models and a real industrial stateflow model for testing. Some implicit bugs in the stateflow model that can not be detected in the Design Verifier can be detected in the Uppaal verification based on the translated timed automata.

The first example is the *switch_on counter* example designed to count how many times the event *switch_on* happens. As presented in the Figure 7, when the stateflow model enters the state B, there is a potential error of division by 0 contained in the transitional action $z = x/y$. So, we may verify the property non-division by zero in Design Verifier, and the model pass the verification. But according to manual analysis, the value of y would be zero after 6 seconds. The Design Verifier failed to detect the bug.

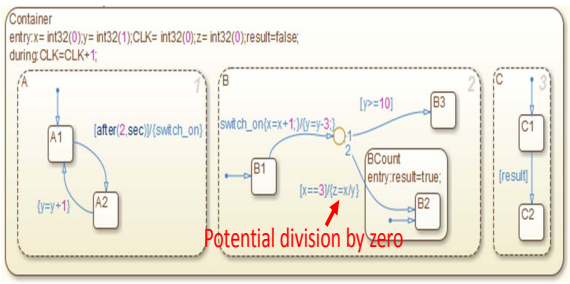

Figure 7: Manual model for validation testing

Then, we translate the stateflow model to timed automata. The translation is accomplished within 0.1 second with 16 parallel connected automata. Due to page limit, the model is not presented in the paper and can be downloaded in the website [16]. In the translated timed automata, the integer variable y in stateflow is mapped to an integer variable *Chart_y*, and the junction in stateflow is mapped to a state with the name *Process_Chart_Container_B .SSID49*. Then, we can verify two properties oriented to the potential error of division by zero as presented in the Table 1.

Table 1: Property List

P1	A[] Chart_y != 0
P2	E<>Process_Chart_Container_B.SSID49 and Chart_y == 0 and Chart_x == 3

Both of the property are verified as yes. The second property, means that y may be set to 0 when the transition is enabled, which will cause the error of division by zero. We can also see that Uppaal also check the reachability of state, which can be used to detect the deadlock of the original stateflow model. Based on the verification, we need to return back to the stateflow model to correct the bug.

Then, we apply our tool to a real industrial stateflow model of the train communication control system [10] and do some verification. The system consists of many multi-function vehicle bus (MVB) controllers which interconnect devices within a vehicle, and the rotated MVB master controller broadcasts a master frame, which carries the identifier of a process data frame for the rest slave MVB. In the rotation process, there will be inconsistencies such that two masters may appear at the same time. First, We build the stateflow model strictly according to the standard, as presented in the figure 8.

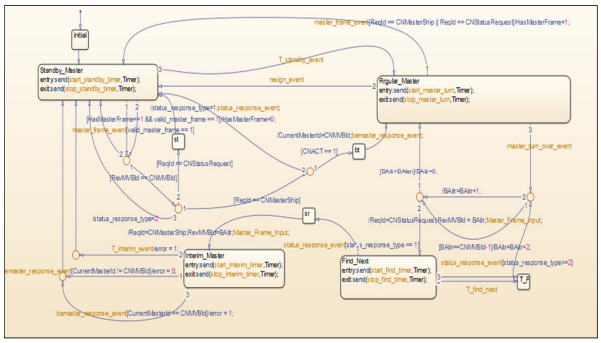

Figure 8: MVB stateflow model for the testing

The constructed model consists of two MVB controllers are translated to timed automata through STU with 32 parallel timed automata, and the properties about the master conflict are defined as below.

The verification result is true, which means that two MVB controllers may reach the regular master state and become master at the same time. Through manually analysis, we find that the bug of the time period configuration in the stateflow model as described in the IEC standard leads to the problem. This bug has been be submitted and proved. More details about the model and other examples can be referred to the support file [16].

6. CONCLUSION

In this paper, we present a tool for the translation of stateflow model to timed automata, which covers many advanced features such as conditional action, activation of composite state, and timer etc. The translated timed automata model can be input to Uppaal for simulation and verification directly. Then, many safety and liveness properties of the original stateflow model can be verified by the Uppaal to acquire higher reliability. Experiment results demonstrate

Table 2: Property List

P1	E<> Process_Chart_MasterShip_OneMVB1_LOGIC .Chart_MasterShip_OneMVB1_LOGIC_Rrgular_Master and Process_Chart_MasterShip_OneMVB2_LOGIC .Chart_MasterShip_OneMVB2_LOGIC_Rrgular_Master
P2	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
P3	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
P4	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 3: 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

the superiority in model validation compared to tools such as Simulink Design Verifier.

The ongoing work mainly focus on strengthening the useability of STU in the following three aspects: (1) the conversion of randomized function in Stateflow is not supported yet. (2) the conversion of floating points and bit wise integer numbers such as int8, int16 and uint8 is not supported yet, we support uint 32 right now. (3) the layout of the translated Uppaal timed automata needs to be improved.

7. REFERENCES

[1] R. Alur. Timed automata. In *Computer Aided Verification*, pages 8–22. Springer, 1999.

[2] R. Alur, A. Kanade, S. Ramesh, and K. Shashidhar. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 89–98. ACM, 2008.

[3] C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 581–586. IEEE, 1999.

[4] M. R. Blackburn and R. D. Busser. T-vec: A tool for developing critical systems. In *Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software*

Safety. Process Security. Proceedings of the Eleventh Annual Conference on, pages 237–249. IEEE, 1996.

[5] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97: Concurrency Theory*, pages 135–150. Springer, 1997.

[6] P. Caspi and etc. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *ACM Sigplan Notices*, volume 38, pages 153–162. ACM, 2003.

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

[8] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. Shashidhar. Automotgen: Automatic model oriented test generator for embedded control systems. In *Computer Aided Verification*, pages 204–208. Springer, 2008.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[10] K. L. McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[11] S. Sims and D. C. DuVarney. Experience report: the reactis validation tool. In *ACM SIGPLAN Notices*, volume 42, pages 137–140. ACM, 2007.

[12] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *Computer Aided Verification*, pages 709–714. Springer, 2009.

[13] B. Tester. Applied dynamics international.

[14] I. The MathWorks. Stateflow user guide.

[15] H. Wernli, M. Paulat, M. Hagen, and C. Frei. Sal-a novel quality measure for the verification of quantitative precipitation forecasts. *Monthly Weather Review*, 136(11):4470–4487, 2008.

[16] J. Yu. In *Uiuc*. `https://sites.google.com/site/jiangyu198964/home`.