Verifying Simulink Stateflow Model: Timed Automata Approach

Yixiao Yang[1], Yu Jiang[1,2], Ming Gu[1], Jiaguang Sun[1]
School of Software, Tsinghua University, TNLIST, KLISS, Beijing, China[1]
Department of Computer Science, University of Illinois at Urbana-Champaign, USA[2]

ABSTRACT

Simulink Stateflow is widely used for the model-driven development of software. However, the increasing demand of rigorous verification for safety critical applications brings new challenge to the Simulink Stateflow because of the lack of formal semantics. In this paper, we present STU, a self-contained toolkit to bridge the Simulink Stateflow and a well-defined rigorous verification. The tool translates the Simulink Stateflow into the Uppaal timed automata for verification. Compared to existing work, more advanced and complex modeling features in Stateflow such as the event stack, conditional action and timer are supported. Then, with the strong verification power of Uppaal, we can not only find design defects that are missed by the Simulink Design Verifier, but also check more important temporal properties. The evaluation on artificial examples and real industrial applications demonstrates the effectiveness.

The abstract demo video address is :
https://youtu.be/TmsU1WRwSqo
The tool and code can be downloaded:
https://www.dropbox.com/sh/374gcfjfei4ywlt/AACF9xqijvY-8nteIhcShIy9a?dl=0

CCS Concepts

•Software and its engineering → Model-driven software engineering;

Keywords

Simulink Stateflow, Uppaal Timed Automaton, Verification

1. INTRODUCTION

Simulink Stateflow is widely used 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 [3]. It has been successfully applied to various industry and livelihood areas, where Simulink Design Verifier [10] are taking the responsibility to uncover design defects of the Stateflow model.

Motivation: However, for those safety-critical applications such as medical devices and avionics, the model validation technique used in Simulink Design Verifier is still insufficient to ensure the correctness. Specifically, the verification capability of Simulink Design Verifier is limited to basic properties. It detects errors in the model that result in the integer overflow, array access violations, division by zero, and violation of requirement assertions described by Simulink verification block. Handling complex temporal properties (e.g. something has to hold at the next state) of those applications is currently infeasible because of the limited descriptive ability of Simulink verification block. More rigorous formal techniques such as model checking should be applied to check the correctness of the Stateflow model.

Challenge: The major challenge for applying those formal verification techniques to support a wider range of properties is that the execution semantics of Stateflow is too complex, which is described in a 1366 pages user guide informally [12]. Advanced modeling feature such as event stack, event interruption, complex state activating and deactivating mechanism, boundary transition, and transitional action etc., are non-straightforward to formalize for verification. Although there are some existing works on translation based verification of Stateflow model, most are efficient and work well covering the most related modeling features within their own domains [4], and few address the temporal part and complex event interrupt mechanism, which are hard to formalize but really important in real model of applications.

Approach: We present STU, to automatically translate the Simulink Stateflow model into the Uppaal timed automata [1, 13] for a more comprehensive formal analysis. Timed automata is chosen because it 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. The advanced Stateflow modeling features (*Composite State, Boundary Transition, Junction, Event, Conditional Action, Transitional Action, Timer and implicit event stack*) are addressed in the tool. With a wider coverage of Stateflow modeling features captured in STU, and the strong verification capability of Uppaal, more comprehensive validation can be accomplished. Potential errors contained in the Stateflow that are missed in simulation or Simulink Design Verifier verification will be detected through Uppaal verification.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970293

2. BACKGROUND

2.1 Simulink Stateflow

The model in Fig. 1 is an example of a Simulink Stateflow diagram which covers most advanced modeling features. The model realizes a counter task that, for every 2 seconds, state A dispatches a *'switch on'* event, and for every *'switch on'* event, state B will increase the variable x by 1. The statement $x = x + 1$ is a conditional action, so it will be executed immediately when the event *'switch on'* is dispatched. On the other hand, the statement $y = y + 1$ is a transitional action which can only be executed when a valid path between two states is detected. So at the end of execution, the value of y is only increased for one time to 1 and the value of x is 3. At the same time, the boolean variable *result* is set to be true, because the activation of state *B2* will trigger the activation of parent state *Count* first. During the activation of state *Count*, the entry action *result = true* is executed.

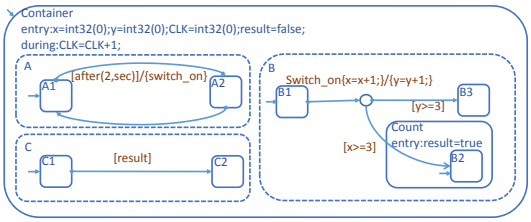

Figure 1: A Stateflow example for counter task which covers most advanced modeling features.

More specifically, Stateflow model is an extended hierarchical state machine which contains sequential decision logic and synchronization events to represent system behaviors. There are mainly six frequently-used modeling elements: *State, Transition, Junction, event, Action* and *Timer*.

State: It represents operating mode of the system. The occurrence of an event will trigger the execution of Stateflow model by making states active or inactive depending on conditions during simulation. The state can be defined hierarchically, and may contain two types of decomposition which are connected in parallel or serial. 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. That is speaking, within a composite state (or a chart), no two exclusive serial sub-states can be active at the same time, while any number of parallel sub-states can be simultaneously activated.

Transition: It is the edge between two states or junctions, 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 explicit or implicit signal that triggers execution of transition, *condition* is a boolean expression that allows the transition to be taken with value true, the *conditional action* is the operation that is immediately executed when the condition is met, and *common action* is the operation that will be executed when the condition is met and there is a non-interrupted valid path between source state and target state. Each transition also has an implicit priority of execution, determined by the information such as hierarchy level of destination state, and position of transition source, etc.

Event: There are two types of event used to trigger execution of a Stateflow diagram. An explicit event is defined by users, and it can be an input from Simulink, an output to Simulink, or local within a diagram. An implicit event is a built-in event that broadcasts automatically during diagram execution. Three commonly used implicit events are system tick, enter(state_name), and exit(state_name): tick indicates the moment when a Stateflow diagram awakens, and the other two occur when the specified state of state_name is entered or exited, respectively. Event broadcasting is a common communication technique in Stateflow.

Action: It contains two kinds of operation attached on transition (*conditional action and common action*), and three kinds of operations attached on state (*entry action, during action and exit action*). *Entry action* is executed when the state is activated, *During action* is executed when the state is already active and stays in, and *Exit action* is executed when the state changes from active to inactive.

Junction: It 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 based on historical data relative to state activity.

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

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

where *TmOp* contains three types of time related operation *before, after*, and *at*, *Num* is the number used to quantify the length of 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

The model in Fig. 2 is an example of a network of timed automata which covers most advanced modelling features. The model consists of three parallel automata *A, B* and *C*. A channel *switch_on* is declared for synchronisation among different automata, and a clock variable t is declared in timed automaton A for time modelling. Every two time units, the action *switch_on!* is synchronized with the action *switch_on?*, and the variable x will increase by 1 in automaton B. If the value of x and y is smaller than 3, automaton B will return to state B_1 immediately for next synchronization from automaton A. After six time units, the transition from state B_4 to B_2 in automaton B would be triggered, and the value of variable *result* should be set to be true, which would immediately trigger the transition from C_1 to C_2 contained in automaton C. Note that the state with the double cycle is the initial state.

Formally, a timed automaton is a finite state machine extended with clock variables. It uses a dense-time model

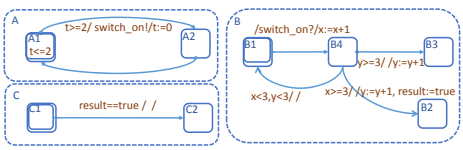

Figure 2: Constructed timed automata for counter.

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 $(l \ \overrightarrow{g,a,r} \ l')$ 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 timed automata 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*. Automata in the network execute concurrently. They can communicate via shared variables, as well as via events over those synchronous channels. In the general case, an edge from location l_1 to location l_2 can be described in a form $(l_1 \ \overrightarrow{g,\phi,r} \ l')$, if there is no synchronization over channels (ϕ denotes an "empty" action), or $(l_1 \ \overrightarrow{g,ch*,r} \ l')$. Here, $ch*$ denotes a synchronization label over channel *ch* with $* \in \{!, ?\}$, g represents a guard for the edge and r denotes the reset operations performed when the transition occurs.

3. MODEL TRANSFORMATION

The key challenges ot the semantics gap 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 transitional and conditional actions very closely. While the Uppaal timed automata supports single state and non-interrupt transition and action.

Since the semantics of timed automata is simpler than that of Stateflow, we need to deal with the priority, event stack, transitional action, etc. with some simple constructs in Uppaal timed automata, which is highly challenging than the reverse translation from timed automata to Stateflow [9]. To simulate the complex model and execution semantics of Simulink Stateflow, an array based data structure for event and an entirely new cooperative mechanism are designed and introduced.

3.1 Dynamic Event Stack Construction

In Stateflow, the event dispatching and processing mechanism is interruptible. However, in timed automata, there is only synchronous channel among parallel automata and no stack at all. The key idea to simulate Stateflow event stack mechanism is to build a virtual stack in Uppaal. We use a structured array in Uppaal to build the event virtual stack. The element of the array is a data structure defined in the listing 1 below, which records all information related to an event in Stateflow. Each element in the structure node is described as:

Listing 1: The Definition of the Event Structure

```
Structure Event {
  int  Event;
  int  Dest;
  int  DestCrossPosition;
  int  AutomatonType;
  bool Valid;
}
```

1. *Event* is the variable used to label and distinguish different events in Stateflow. We assign a unique integer number to this variable for each Stateflow event.

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

3. *DestCrossPosition* is the variable used to imply the corresponding Uppaal *controller automata* state originated from Stateflow cross-boundary transition.

4. *AutomatonType* is the variable used to map the event to the four types of corresponding Uppaal automata.

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

The virtual stack is the basic element to simulate Simulink Stateflow semantics. It is initialized as empty in the translated Uppaal timed automata, and is dynamically pushed and popped during runtime simulation. When Simulink Stateflow generates an event within a transition or a state operation, the translated Uppaal timed automata will take a corresponding transition with an attached action to dispatch and push an *Event* element into the stack dynamically. Each transition starting from an active state of *controller automata* will check whether the *Dest* of the top element of event stack equals to the label of automata or not. If yes, the transition will be triggered, and the *Event* element will also be popped corresponding to the end of a simulation cycle of Simulink Stateflow. The procedures are mainly accomplished through five encoded functions *DispatchEvent(), PushEvent(), PopEvent(), EventSentToMe(),* and *StackTopEvent()* of timed automata.

3.2 State Transformation

For a regular simple state without decomposition or attached actions, the transformation is straightforward. We just directly map simple Stateflow state s^f to Uppaal timed automata state s^u. But for those complex Stateflow state with decomposition or attached actions, we need to translate it to four cooperative parallel automata:

1. *Controller automata* is used to simulate the event processing mechanism within this complex Stateflow state. It controls how to dispatch the hierarchical active and deactive related event by initializing, popping, and pushing elements of the virtual stack.

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

3. *Condition automata* is used to execute the conditional action, handle the junction, test the guard and priority on each transition contained in this composite state, and store the boolean results.

4. *Common automata* is used to execute the transitional action, and read the guard related array initialized by *condition automata* to execute the satisfied transition contained in this composite state.

Controller automata: For the activation of state s^f in Stateflow, it should estimate whether its upper-level state s^{lf} is activated or not. If not, s^{lf} should be activated first, this is especially true for cross-boundary transitions. In order to simulate this semantics, the corresponding *controller automata* should push an activation event corresponding to state s^f itself onto the stack first, and recursively pushe the activation event associated with the automata originated from s^{lf} onto the stack, until the top composite state arrives. The deactivation of Stateflow state, is a reversal of activation procedure. In *controller automata*, these two tasks are translated to two self-cycle transitions attached with actions *StateActivationLogic()* and *StateDeactivationLogic()* of timed automata.

Action automata: For detail execution of *entry, during*, and *exit* action attached on Stateflow state, it will be captured by the translated *action automata* with three self-cycle transitions. After the execution of *controller automata* on the logic of state active or deactivate, *action automata* will continually read the stack top event for the test of the guard. The guard on the three transitions are StackTop().Event == ActivationEvent, StackTop().Event == DuringEvent and StackTop().Event == DeactivationEvent. Then, the transition with satisfied guard will take, and corresponding action statements in Stateflow are translated to action statements attached on the three transitions.

An example for the translated *controller automata* and *action automata* for a composite state A is presented in Figure 3. For *condition automata* and *common automata*, they are mainly used for Stateflow transitions contained in composite state, and will be described in the following paragraph.

3.3 Transition Transformation

Within Stateflow, each *transition* is attached with four characterizations: *event, condition, conditional action*, and

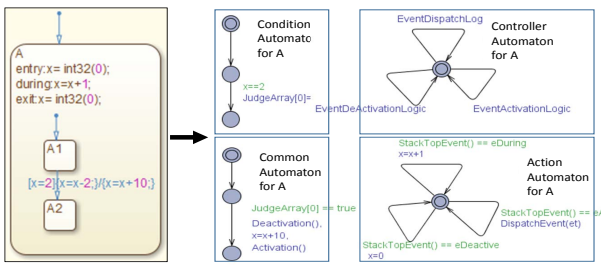

Figure 3: The controller and action automata for a composite state transformation, capturing activation and deactivation.

transitional action. We incorporate them into the *condition and common automata* of the high-level composite state that contains this transition 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 transition in the corresponding *condition automata*.

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

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

When there are multiple transitions starting from a Stateflow state, we should maintain the determinism execution sequence of Stateflow in timed automata. First, we initialize an int array *PathSelect[ ]* to store the priority of transition, where the array index represents the depth of source state or junction node of transition. As presented in Figure 4, the depth of state or junction is defined as the minimum transition number to a pre-state. Besides, a boolean array *PathGuard[ ]* is initialized to store the *condition* test result of every transition, where the array index is the *id* of Stateflow transition.

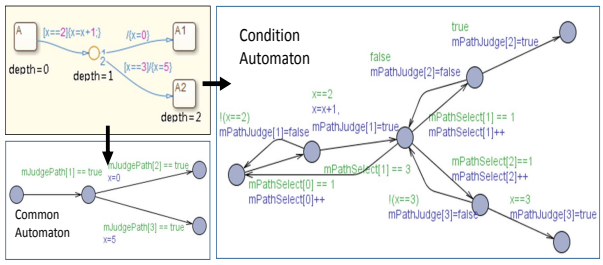

Figure 4: The common and condition automata for a composite state transformation, capturing internal transition.

Condition automata: For a Stateflow transition $t_1^f : s_1^f \rightarrow s_2^f$ with *conditional action* a_c^f and *condition* g^f, we build *condition automata* as below. An intermediate state s_i^u is added between the corresponding timed automata state s_1^u and s_2^u. Based on which, three automata transitions are defined, $t_1^u : s_1^u \rightarrow s_i^u$, $t_2^u : s_i^u \rightarrow s_2^u$ and $t_3^u : s_i^u \rightarrow s_1^u$. The guard on transition t_1^u is $PathSelect[i] == Priority$, which ensures that the transition is executed by its priority order. The guard on transition t_2^u is the *condition* g^f from Stateflow transition t_1^f. The action on transition t_2^u is from *conditional action* a_c^f of the Stateflow transition t_1^f, and an additional assignment of the boolean array element *PathJudge[i]* with value *true*. In this way, *conditional action* can be executed immediately whether there is a legal transition path between two Stateflow states or not. Transition t_3^u is used to roll back to the source state for further test of transitions with lower property, and *PathGuard[i]* is set as *false* to show that this transition could not be taken in *common automata*. Also, if s_2^f is a Stateflow junction node, a transition is added $t_4^u : s_2^u \rightarrow s_1^u$ for roll back of non-complete path. This roll back transition is controlled by the guard *pathSelect[i]* == *n*, where i is the depth of the junction node, n is the number of outgoing transitions from the

junction, and each negative test of the guard on outgoing transition will increase the value of pathSelect[i] by 1. The *timer* of Stateflow is also captured in *condition automata*. Time operation is based on event and is usually used as a time related condition on transition.

Common automata: For a Stateflow transition $t_1^f : s_1^f \rightarrow s_2^f$, we build *common automata* to capture its *transitional action* a_t^f, based on the array *PathGuard[ ]* initialized in *condition automata*. Stateflow transition t_1^f is directly mapped to an automata transition $t_1^u : s_1^u \rightarrow s_2^u$. The guard and action on automata transition t_1^u are from the expression *PathGuard[ ] == true* and *transitional action* a_t^f respectively. It is almost the same as the graphical structure of Stateflow model, with abbreviated guard and transitional action. An example for the translated *common automata* and *condition automata* of the composite state A is presented in Figure 4.

3.4 Tool Implementation

Based on above transition rules, we implement a tool for automatically translation from Stateflow to Uppaal timed automata. The tool **STU** consists of a parser, translator, and storer, and is implemented in 14590 lines of java code with two supporting libraries (JDOM used for read and write XML file, and Antlr used for abstract syntax tree construction and update), as presented in Figure 5. The parser extracts Stateflow model from Simulink project file into memory. The translator transfers Stateflow model and reconstructs the abstract syntax tree in memory according to transition rules. The storer outputs the updated abstract syntax tree to Uppaal model file. The three parts are seamlessly integrated in **STU** to support the formal analysis of Stateflow model based on Uppaal, and can be downloaded in the website presented in abstract.

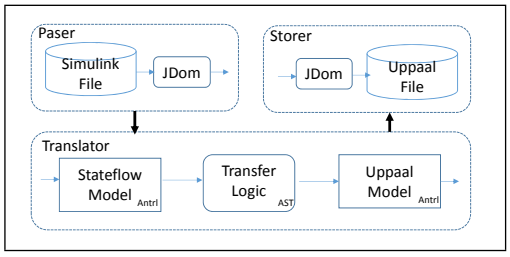

Figure 5: Translation tool design and integration.

4. TOOL EVALUATION

In order to evaluate the tool, we apply it to some artificial and real industrial Stateflow models. The presented Steteflow models, translated timed automata, and properties specifications could be downloaded in web-site presented in footnote 1. Some implicit bugs in Stateflow model that can not be detected in Design Verifier are detected in Uppaal verification based on the translated timed automata.

The first artificial example is the *switch_on counter* example designed to count how many times the event *switch_on* happens. As presented in Figure 6, when the Stateflow model enters the composite 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 passes the verification. But according to manual analysis, the value of y would be zero after 6 seconds. Design Verifier failed to detect this implicit but general bug contained in the model.

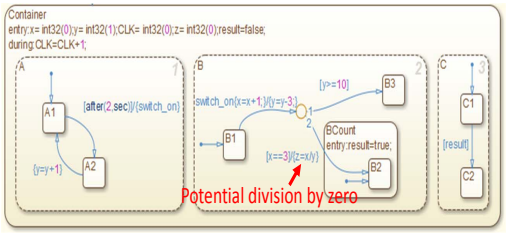

Figure 6: Manual model for validation testing

Then, we translate the Stateflow model to timed automata through the developed tool STU. The translation is accomplished within 0.01 seconds. In the translated timed automata, the integer variable y in Stateflow is mapped to an integer variable *Chart_y*, and the junction node in Stateflow is mapped to a state with the name *Process_Chart _Container_B.SSID49*. Then, property about error of division by 0 within this model can be described as in Table 1.

Table 1: Property List

Formula	Time
E<>Process_Chart_Container_B.SSID49 and Chart_y == 0 and Chart_x == 3	0.43s

Where "$E <>$" is a temporal keyword which means eventually, "Process_Chart_Container_B.SSID49" is automata state name corresponding to the Stateflow junction node, "Chart_x == 3" is automata value test corresponding to the guard "x==3" of Stateflow transition from junction node to state B_2, and "Chart_y == 0" is also automata value test corresponding to the Stateflow action $z = x/y$ attached on the transition from junction node to state B_2. The property consists of a serial combination of previous predicates, and means that y may be set to be 0 when the transition is enabled, which will cause the error of division by 0. Verification result shows that the property is satisfied and the error can be triggered within 0.43 second.

Then, we apply STU to a real industrial Stateflow model of the train communication control system and do some verification. The system consists of many multifunction vehicle bus (MVB) controllers which interconnect devices within a vehicle, and the rotated MVB master controller broadcasts a master frame[6, 7, 15]. Given master rotation as an example, the master transfer logic described in page 260 and Figure 105 of IEC 61375 are modeled as Stateflow model. After preliminary Stateflow validation on two MVB controller instances, we translate the main logic and some accompanied Stateflow models into 151 corresponding parallel timed automata within 0.1 seconds and verify the property described in table 2 within 3 seconds. This property is derived from real potential hazards of system failure. For example, in the MVB master and slave rotation process, there may be inconsistence such that two masters appear at the same time.

The property is violated during verification, which means that there exists a path that two MVB controllers may simultaneously reach "Regular_Master" state, or simultaneously reach "Standby_Master" state. The first situation will lead to master collision and the second will lead to no master

Table 2: Property List

Formula	Time
A[ ] Process_Chart_OneMVB1(2)_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master and Process_Chart_OneMVB2(1)_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	2.349s

throughout train communication network. Through manual analysis of counter examples demonstrated in Uppaal, we trace back to the design defects of Stateflow model, which can be further traced back to the bugs in the standard.

Currently, any models that consist of the advanced modeling features mentioned in the introduction can be translated by our tool. Because 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 correctness by carefully compare simulation results of the translated model, including the value and state sequence step by step, in the same way as previous works.

5. RELATED WORK

Because Stateflow has no formal semantics for rigours formal verification, plenty of attempts have touched the topic to assist Simulink Design Verifier in acquiring correctness of Stateflow model, which can be classified into two categories, simulation-based techniques and verification-based techniques. Many researchers have developed simulation based tools for Simulink designs including Beacon Tester [11], and AutoMOTgen [5] etc. For 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 model to pushdown automata [2], SMV [8], PAT [4], Hybrid automata, hoare logic and SAL [14], 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, they focus and provide a well-defined framework to ensure the function correctness, while the hierarchical states and events are out of their considerations.

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. The ongoing work mainly focus on strengthening the useability of STU in the following two aspects: (1) the conversion of randomized function in Stateflow is not supported yet. (2) the layout of the translated Uppaal timed automata needs to be improved. (3) the automatic trace back tool should be developed.

7. ACKNOWLEDGMENT

This research is sponsored in part by NSFC Program (No. 91218302, No. 61527812), National Science and Technology Major Project (No. 2016ZX01038101), Tsinghua University Initiative Scientific Research Program (20131089331), MIIT IT funds (Research and application of TCN key technologies) of China, and the National Key Technology R&D Program (No. 2015BAG14B01-02), Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23.

8. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Y. Jiang and Y. Yang. From stateflow simulation to verified implementation: A verification approach and a real-time train controller design. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[7] Y. Jiang and H. Zhang. Design and optimization of multi-clocked embedded systems using formal techniques. *IEEE Transactions on Industrial Electronics*, 62(2):1270–1278, 2015.

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

[9] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. Safety-critical medical device development using the upp2sf model translation tool. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):127, 2014.

[10] SimulinkDesignVerifier. http://www.mathworks.com.

[11] B. Tester. Applied dynamics international.

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

[13] R. Wang and M. Gu. Formal modeling and synthesis of programmable logic controllers. *Computers in Industry*, 62(1):23–31, 2011.

[14] 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.

[15] H. Zhang and H. Zhang. Design of mixed synchronous/asynchronous systems with multiple clocks. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2220–2232.