From Stateflow Simulation to Verified Implementation: A Verification Approach and A Real-Time Train Controller Design

Yu Jiang[1,2], Yixiao Yang[2], Han Liu[2], Hui Kong[3], Ming Gu[2], Jiaguang Sun[2], Lui Sha[1]

Department of Computer Science, University of Illinois at Urbana-Champaign, USA[1]
Institute of Science and Technology, Austria[3]
School of Software, Tsinghua University, China[2]

***Abstract*—Simulink is widely used for model driven development (MDD) of industrial software systems. Typically, the Simulink based development is initiated from Stateflow modeling, followed by simulation, validation and code generation mapped to physical execution platforms. However, recent industrial trends have raised the demands of rigorous verification on safety-critical applications, which is unfortunately challenging for Simulink.**

In this paper, we present an approach to bridge the Stateflow based model driven development and a well-defined rigorous verification. First, we develop a self-contained toolkit to translate Stateflow model into timed automata, where major advanced modeling features in Stateflow are supported. Taking advantage of the strong verification capability of Uppaal, we can not only find bugs in Stateflow models which are missed by Simulink Design Verifier, but also check more important temporal properties. Next, we customize a runtime verifier for the generated non-intrusive VHDL and C code of Stateflow model for monitoring. The major strength of the customization is the flexibility to collect and analyze runtime properties with a pure software monitor, which opens more opportunities for engineers to achieve high reliability of the target system compared with the traditional act that only relies on Simulink Polyspace. We incorporate these two parts into original Stateflow based MDD seamlessly. In this way, safety-critical properties are both verified at the model level, and at the consistent system implementation level with physical execution environment in consideration. We apply our approach on a train controller design, and the verified implementation is tested and deployed on a real hardware platform.

***Index Terms*—Model Driven Development, Simulink Stateflow, Formal Verification, Timed Automaton, Runtime Verification.**

I. Introduction

Simulink is a widely used tool for model driven development of industrial software systems, which provides delicate support for graphical Stateflow modeling, interactive model-level simulation, some basic design validation, along with C, C++, and VHDL code generation and verification [1]. In practice, Simulink has been successfully applied across various industry applications such as manufacturing control and signal processing systems, where Simulink Design Verifier [2] and Simulink Polyspace [3] are taking the responsibility to uncover design defects and implementation defects, respectively.

However, with respect of safety-critical applications such as medical devices and avionics, Simulink is still insufficient. 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, dead logic, 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. Moreover, although Simulink Polyspace offers the flexibility to check correctness over the implementation code using abstract interpretation techniques, we still lack the knowledge to analyze the interaction between the target software and dynamic physical execution environment. Consequently, to guarantee the correctness of the whole system stays non-trivial. Hence, supporting tools with more verification power such as Uppaal [4] is expected here to check the properties of Stateflow model and more rigorous formal techniques such as runtime verification [5] should be applied to monitor and ensure the correctness of the automatically implemented systems.

However, 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 [6]. 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 many existing works on translation based verification of Stateflow model, most of them are efficient and work well covering the most related modeling features within their own domains [7]. Few address the temporal part and consistency verification of properties on the generated code running in an unexpected dynamic physical environment, which is essential for safety critical applications, such as anti-missile system [1].

In this paper, we present an approach to address the verification challenge in both model and implementation level of Stateflow based MDD. In terms of model-level verification, we develop a tool STU, to translate Stateflow model into timed automata for formal analysis based on model checking tool Uppaal [8]. Timed automata can be used to model and analyze timing behavior of systems, and the methods for checking both safety and liveness properties of timed automata have been well developed and intensively studied in Uppaal. Most frequently used Stateflow modeling features (*composite*

[1]The Patriot anti-missile system failure during Gulf War was caused by the incongruence between the timer module and the new application environment. http://fas.org/spp/starwars/gao/im92026.htm

state, boundary transition, junction, event, conditional action, transitional action, timer, and implicit event driven stack) are addressed in the translation tool with discussions and validations with engineers from Mathwork.

With a wider range of Stateflow modeling features captured in STU, and the strong verification capability of Uppaal, more comprehensive validations are feasible. Potential errors that may not be detected in simulation or Simulink Design Verifier would be found via Uppaal verification. If errors are detected, Stateflow model needs to be analyzed and revised with the help of mapping dictionary for translation. As a result, the code generated from the verified model will be more reliable and can be analyzed through Simulink Polyspace. Furthermore, because Simulink Polyspace checks the implementation code using abstract interpretation techniques which provide little support for temporal properties, we customize the runtime verification to monitor properties on code integration and system deployment running in a dynamic physical execution environment, which contributes many safety hazards for system failure. Within this part, we are able to not only translate some safety critical properties verified by Uppaal to the property descriptions of runtime monitor for consistency checking, but also add some system level properties such as platform-dependent delay that couldn't be described based on the abstract Stateflow model. The overall procedure about the proposed extended approach is presented in Figure 1.

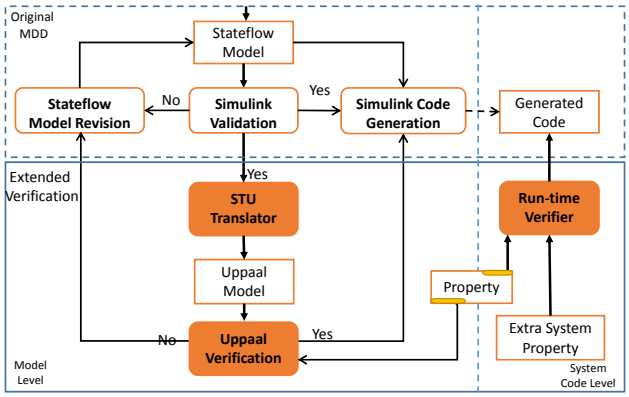

Fig. 1. Integrate STU and runtime verification into Stateflow based MDD.

The rest of paper is organized as follows. Some backgrounds about Stateflow, timed automata and runtime verification are introduced in Section 2. Related works about the verification of Stateflow and runtime verification are discussed in Section 3. Section 4 shows the design and implementation of the proposed approach, including Stateflow to Uppaal timed automata translation, customization of runtime verification and interfaces among them. Evaluation results on artificial examples and real train controller system design are presented in Section 5, and we conclude in Section 6 with more discussions about the approach.

II. Background

In this section, we present background information on the elements and semantics of Stateflow and timed automata, and some introduction to runtime verification.

A. Simulink Stafeflow

The model in Fig. 2 is an example of a Stateflow diagram which covers most advanced modeling features. The outmost composite state *Container* is parallelly decomposed into three sub-composite states *A, B* and *C*. State *A* and *C* is further serially decomposed in two automatic states, respectively, where the initial automatic state such as A_1 is attached with an arrow. State *B* is further serially decomposed into two automatic states (B_1 and B_3), a sub-composite state (*Count*) and a junction denoted by a small cycle. There is a cross-boundary transition from the junction into the initial automatic state of the sub-composite state (*Count*). Statements attached on state such as *Container* and *Count* are *entry, during* and *exit* actions. Statements attached on transitions includes *guard, common* action and *conditional* action. 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.

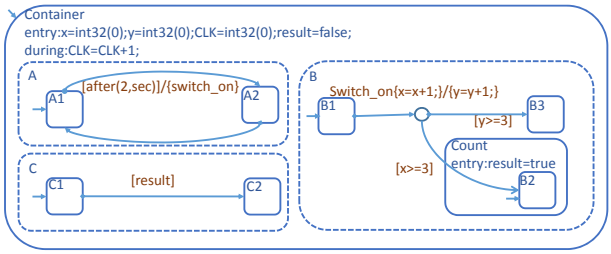

Fig. 2. A Stateflow example for counter task which covers most advanced modeling features.

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. When an event is globally broadcast, the evaluation of the event starts from a Stateflow diagram that is the root of all its components and follows the hierarchy of states in a top-down manner. An event can also be directly broadcast from one state to another to synchronize parallel states, and the evaluation of the event is within the destination state.

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.

B. Uppaal Timed Automata

The model in Fig. 3 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.

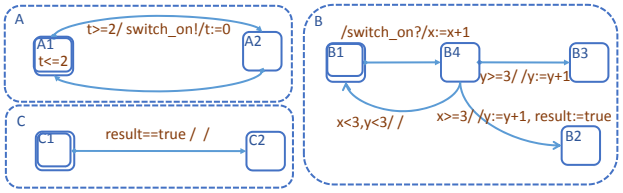

Fig. 3. Manually constructed timed automata for counter.

Formally, 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 $(l\ \ \overrightarrow{g,a,r}\ \ l')$ 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.

Then, the state of the system is defined by the locations of all automata, and the values of clocks and discrete variables. Every automaton may fire a transition separately, or synchronize with another automaton with the channel action $ch!$ and $ch?$ as below:

- $(\bar{l},\ u) \rightarrow (\bar{l}[l_i'/l_i],\ u')$,
  $if\ (l_i\ \ \overrightarrow{g,a,r}\ \ l_i'),\ u \in g, u' = [r \mapsto 0]u, u' \in I(\bar{l}[l_i'/l_i])$.
- $(\bar{l},\ u) \rightarrow (\bar{l}[l_i'/l_i,\ l_j'/l_j],\ u')$,
  $if\ \exists\ i \neq j,\ (l_i\ \ \overrightarrow{g,ch?,r}\ \ l_i'),\ (l_j\ \ \overrightarrow{g,ch!,r}\ \ l_j'),\ u \in g_i \wedge g_j,\ u' = [r_i \cup r_j \mapsto 0]u, u' \in I(\bar{l}[l_i'/l_i],\ \bar{l}[l_i'/l_i])$.

where $\bar{l}$ denotes a vector of current locations of the automata network, u is as usual a clock assignment recording the current

values of the clocks in the system, and $\bar{l}[l_i^{'}/l_i]$ denotes the vector $\bar{l}$ with l_i being substituted with $l_i^{'}$. 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 real-time embedded applications.

C. Runtime Verification

Runtime verification can be used for many purposes, such as debugging, or safety policy monitoring, verification, behavior modification, etc. It aims to be a lightweight verification technique complementing other verification techniques such as model checking and theorem proving, by analyzing only one or a few execution traces and by working directly with the actual system, thus scaling up relatively well and giving more confidence in the results of the analysis. Following the descriptions in [9], it can be defined as:

> *"Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property."*

Technically speaking, in runtime verification, a correctness property, is typically automatically translated into a monitor. Such a monitor is then used to check the current execution of a system or a finite set of recorded execution with respect to the property. Moreover, through its reflective capabilities, it can be made as an integral part of the target system, monitoring and guiding its execution. Researchers usually use Aspect-oriented Programming as a technique for defining program instrumentation in a modular way for the specified monitor.

III. Related Work

Last decades, a variety of computation models and the corresponding toolkits have been proposed to facilitate the design of embedded system, among which the two most widely used are SCADE suite based on safety state machine [33] and Simulink toolkit based on Stateflow [1]. Both of them have been successfully applied in a variety of applications.

The underlying computation model safety state machine of SCADE is formally defined and provides a mathematical basis for the complete formal analysis of systems. Hence, the SCADE suit, including graphical modelling, test automation, SAT-based verifier, and certified code generator, provides a systemic solution for developing extremely safety critical systems such as avionics. Accompanied with the certified code generator, the SAT-based SCADE Design Verifier (DV) plays a very important role in ensuring the correctness of the model, to formally express and assess safety requirements and find bugs early in the development process. Properties to verify are defined with SCADE observer itself. The boolean outputs are the proof objectives for DV that then automatically produces counter-examples. However, while SCADE verifier performs very well for certain verification tasks, it can fail badly for others due to complexity problems and the descriptive limitation of the observer. There are many efforts trying to enhance the verification ability of SCADE [34], [35]. Besides, while mainly focusing on embedded software, the certificated code generator currently has few support for the synthesis of hardware with 20,0000 US dollars for a single licence.

Similar to SCADE suit, Simulink also supports system design with Stateflow modelling, simulation, validation and code generation. 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. Simulation based technique is adopted widely, while the main challenge is to solve the coverage of simulation patterns. Many researchers have developed test generation tools for Simulink designs including Reactis [10], T-VEC [11], Beacon Tester [12], and AutoMOTgen [13] 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 [14].

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 [15], Lustre [16], SMV [17], PAT [18], hoare logic and SAL [19], 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. In PAT [7] based verification technique, they covered most advanced features of Stateflow, while with limited support of event interrupt dispatch mechanism and time operation support. Besides, there is also some nice work translating Uppaal timed automata to Simulink Stateflow for simulation and code generation [20], [21]. Since the semantics of timed automata is simpler than that of Stateflow, the translation procedure is different from our setting, because we need to deal with the priority, event stack, transitional action etc of Stateflow during our reverse transformation. Also, based on their tool, we can build an interface to connect to our transformation to form a closed loop.

Compared to previous works, we try to cover most Stateflow advanced modeling features, including the timing mechanism that has never been addressed before, and make use of the strong verification tool Uppaal to diagnose more properties. We also formalize the complex event stack and executions interrupt mechanism which is limitedly supported before. Uppaal is chosen because timed automata can be used to model and analyze timing behavior of systems, and the methods for checking both safety and liveness properties of timed automata have been well developed and intensively studied, which has been successfully applied in verification of many safety-critical systems. Besides, because Simulink Polyspace mainly detects common errors such as overflow, division by zero, and out-of-bound pointers, and provides little support for temporal properties, we customize runtime verification on the generated

code to assist Simulink Polyspace, so that the properties during model validation and extra runtime related properties can be consistently verified and monitored on the executable system. They are integrated into Stateflow based MDD lifecycle to acquire higher confidence in safety critical applications. Then, we apply the enhanced MDD to the implementation of a real train controller, which is premier studied in [31, 32]. In [31], the author proposes a heterogeneous modeling language to model both data-oriented and control-oriented behavior of train controller. In [32], the author uses timed automata to model and verify the real time protocol used for communication of controller. Based on their description about the train controller system, we will show how the enhanced Stateflow MDD construct a Stateflow model, find bugs through translated verification, generate code for real platform implementation, and insert runtime monitor to the system.

IV. Extended MDD Approach

In this section, we introduce the kernel components presented in Figure 1 : the transformation rules and implementation of STU, and the customization of runtime verification.

A. Formal Verification of Stateflow

When translating Simulink Stateflow to Uppaal timed automata for verification, the most important and difficult task is to overcome the gap between their execution semantics. As introduced in background, key differences between Stateflow and timed automata are :

(1) Stateflow transition is driven by event. Execution of every event is in deterministic sequential order, and interruptible with stack. While timed automata is executed in parallel, and driven by the channel synchronization without the support of stack.
(2) Stateflow supports hierarchy structure which is combined with recursive activation-deactivation mechanism, transitional action, and conditional action very closely. While timed automata support single state.

To bridge the gaps above and simulate complex execution semantics of Simulink Stateflow, an array based data structure for event and some cooperative mechanisms are designed and introduced for Uppaal timed automata.

1) Event Stack Basis: 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:

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

Listing 1. Definition of the Event Structure

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 Stateflow semantics. It is initialized as empty in the translated Uppaal timed automata, and is dynamically pushed and popped during runtime simulation. When 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 Stateflow. The procedure above is mainly accomplished through five encoded functions *DispatchEvent(), PushEvent(), PopEvent(), EventSentToMe(),* and *StackTopEvent()*, with an example of *DispatchEvent()* presented as below.

```
void DispatchEvent(Event E)
{
   PushEvent(E.Event, E.Dest, -1,
             E.AutomatonType, true);
}
void PushEvent(int Event, int Dest,
               int Cross, int Automaton,
               bool Valid)
{
   Top++ ;
   Stack[Top].Event = Event;
   Stack[Top].Dest = Dest;
   Stack[Top].DestCrossPosition = Cross;
   Stack[Top].Valid = Valid;
   Stack[Top].AutomatonType = Automaton;
}
```

Listing 2. Dispatch and push an event

Daemon automata have two duties. The first is to delete invalid event on the top of the virtual stack, and the second is to dispatch *System Event* to keep the automata running when the virtual stack is empty. *System Event* is reserved in Stateflow which refers to the default event generated by Simulink to drive the suspended model periodically. To delete an invalid event, *daemon automata* needs a self-cycle transition with attached action to continuously check whether the value *Event.Valid* of the element of the top stack is false or not. If

yes, *Stack[Top]* will be deleted through function *PopEvent()*, as encoded in function *DeleteInvalidEvent()*. To generate a *System Event*, *daemon automata* needs a self-cycle transition with attached action to continuously check the whether the stack is empty or not. If yes, a predefined event element will be pushed onto the top of the empty stack, as presented in the function *GenerateSystemEvent()*.

```
void  DeleteInvalidEvent()
{
   if (Stack[top].Valid == false){
        PopEvent(Stack[]);
   }
}
```

Listing 3. Delete an invalid event

```
void  GenerateSystemEvent()
{
   if (Top==0){
        PushEvent(SE.Event, 1, -1,
                  SE.AutomatonType, true);
   }
}
```

Listing 4. Generate a system event

Based on the structured virtual stack, we translate Stateflow into Uppaal timed automata automatically. As introduced in the background, there are six most frequently-used elements in Stateflow, where the *event* and *action* elements are attached on *state*, and the *transition*, *junction* and *timer* element can be scanned through *transition*. Hence, we demonstrate the transition rules for *state* and *transition*, with other elements embedded into them.

2) State Transformation Rule: 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()*.

Let us look at *StateDeactivationLogic()* presented in Alg 1 in detail, there are two sub-functions cooperating to accomplish the task. The first sub-function *DispatchDeactivationToChild()* is used to deactivate refined sub-states contained in current state. If the current state is refined in parallel sub-states, the deactivation event for sub-state with the lowest priority is pushed into stack through *DispatchEvent()* function first, then the parallel sub-states with higher priorities are handled sequentially. If the current state is refined in serially connected sub-states, the deactivation event for current active sub-state is pushed into the stack directly. The second sub-function *HandleDeactivation()* is used to handle the logic of action attached on the current state. If there is *exit action* attached on the state, it will dispatch an event to the corresponding *action automata*. If the current state is also a sub-state, it will also generate an event to notify its upper-level state that current state has been exited. The algorithm *StateActivationLogic()* for activation can be encoded and interpreted in the same way.

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.

Algorithm 1: Composite State deactivation logic

```
Void StateDeactivationLogic(int state_s^f)
{
  DispatchDeactivationToChild(state_s^f);
  HandleDeactivation(state_s^f);
}

void DispatchDeactivationToChild(int state_s^f)
{
SubStates[ ] ← Substate(s^f);
if (SubStates[ ] are active) then
  if (SubStates[ ] are in parallel) then
    while (i ≤ length.PrioritySort(SubStates[ ])) do
      DispatchDeactivationToChild(SubStates[i]);
      i++;
    end while
  else
    DispatchEvent(Event DeactivationEvent);
  end if
end if
}
void HandleDeactivation(int state_s^f)
{
if (s^f has attached exit action) then
  Event DeactivationEvent.AutomatonType = action;
  DispatchEvent(Event DeactivationEvent).
end if
if (s^f is a sub-state) then
  int UpperLevelDest = AutomatonID(Parstate(s^f));
  Event DeactivationEvent.Dest = UpperLevelDest;
  DispatchEvent(Event DeactivationEvent);
end if
}
```

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 4. 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.

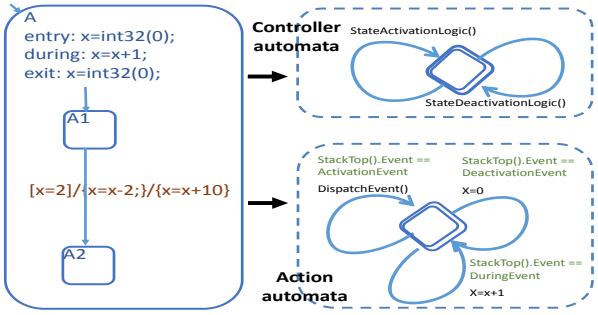

Fig. 4. The controller and action automata for a composite state transformation, capturing activation and deactivation.

3) Transition Transformation Rule: Within Stateflow, each *transition* is attached with four characterizations: *event, condition, conditional action*, and *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 5, 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.

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. As described in the background section, it is described as *[TmOp(Num, Event)]*. We count the appearance times *(Num)* of the event *(Event)*, and store the value using an int array *Times[ ]*. The index of the array is the integer ID assigned to the event. When an event is dispatched, the value of *Times[Event]* is increased by 1. The translation rules for the four types of time operations are below. Then, each translated guard is attached on the corresponding transition contained in *condition automata*.

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

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 5.

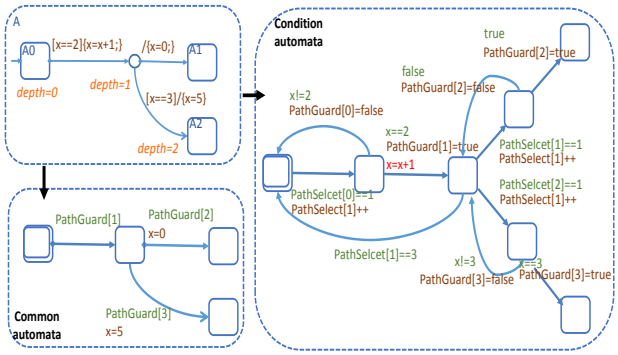

Fig. 5. The common and condition automata for a composite state transformation, capturing internal transition.

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). 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 website [22].

B. Runtime Verification of System

The key technical ingredient in runtime verification is to specify dynamic runtime environment related properties that couldn't be easily described based on the abstract Stateflow model, and choose proper run-time monitoring tools according to adopted programming language. Over the past decade, tremendous of efforts have been invested in developing program runtime verification systems [23]. Most of these works can be regarded as an extension of AspectJ [24]. Some exceptions are ARACHNE [25] and RMOR [26]. ARACHE performs runtime weaving into the binary code of C programs with a limited form of regular expressions, while RMOR monitors the execution of C programs against state machines using aspect-oriented pointcut language to connect events to code fragments. For hardware runtime verification, the property specification is usually translated into a hardware description such as VHDL and Verilog, which is then synthesized and loaded into reconfigurable blocks of the FPGA [27].

Within Stateflow based MDD, we can generate VHDL and C code from the verified Stateflow model with the code generator of Simulink. Those two languages are widely used in industrial system design. For the generated C code, we can apply RMOR directly. For the generated VHDL code, we can also customize tools for hardware runtime verification. But, the separation of hardware monitor and software monitor may increase the complexity of proposed approach and bring challenges to verify properties related with their interactions.

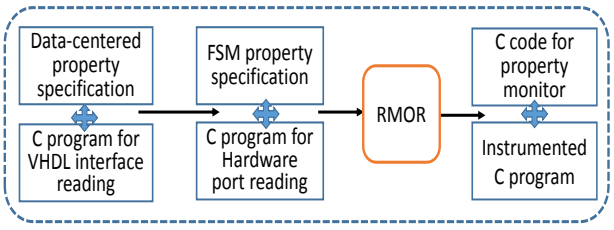

Fig. 6. Hardware runtime verification customisation on Software

Based on the complexity reduction idea presented in our previous work [28] and the observation that VHDL has well-defined interface of input and output data ports, we customize a data-centered runtime verification technique, into software monitor for runtime verification of VHDL. In [29], we have designed data-centered domain description language *DRTV* and translated the data-centered model to property monitors, based on which, main customization is presented in Figure 6. From the data description part of *DRTV*, which is easy to be derived from the VHDL interface, we derive the additional C program to read the data value of pin bounded to the interface of VHDL. From the property description part of *DRTV*, which is defined on events based on values of data, we derive the event definition and state machine property definition in the format of RMOR. Then, those specifications and accompanied C program are input to RMOR to generate the software monitor and instrumented C program. In this way, we make use of the monitor running on the software processor to verify the behavior of hardware.

V. Experiment Results

In order to evaluate the proposed approach, we apply it to some artificial Stateflow based MDD and a real-time train controller design. The presented Steteflow models, translated timed automata, and properties specifications could be downloaded in website [22]. 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, and some violations that can not be supported in Polyspace can be specified and detected in runtime verification monitor.

Artificial Examples: 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 7, 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.

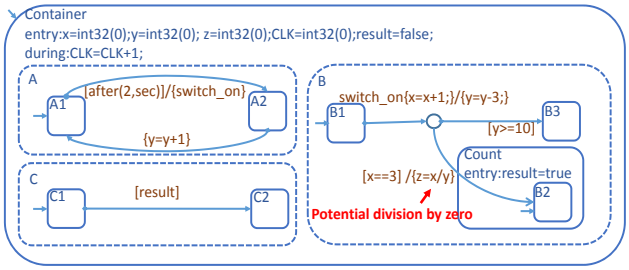

Fig. 7. Manual model for validation testing

Then, we translate the Stateflow model to timed automata through the developed tool STU. The translation is accomplished within 0.1 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 I.

TABLE I
Property List

Property	Formula	Time(second)
P1	E<>Process_Chart_Container_B.SSID49 and Chart_y == 0 and Chart_x == 3	0.01

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. Then, we can define a reverse property using temporal keyword "$A[\ ]$" to get the counter example to help locate the bug. Hence, we need to return back to the original Stateflow model to correct the bug by adding an additional condition $y! = 0$ in front of the action. From the verification of this property, we can see that Uppaal also checks the reachability of state, which can be used to detect deadlock of Stateflow model.

For runtime verification, we specify the FSM property in the input format of RMOR, as abstracted in listing 5. The property specification is based on the generated C code of Stateflow model. The three pointcut expressions mean the value for the current state, variable related to guard, and variable related to action of the potential transition, respectively. If all of them are satisfied, an error of division by 0 will be triggered. The property description above is used to generate executable C monitor, which will also be encoded into the generated code of Stateflow model with RMOR. Some other safety and liveness temporal properties are also supported, and overall procedure is similar to Figure 6.

Real-time Train Controller : We apply the proposed approach to a real industrial application of Stateflow based MDD of train communication control system. According to IEC Standard 61375 [30], the control system consists of many multifunction vehicle bus (MVB) controllers which interconnect devices within a vehicle. MVB master controller broadcasts a master frame, which carries an identifier of process data frame for the rest of MVB slave controllers. At the end of a predefined macro period, current MVB master controller will give up control ability, and an MVB slave controller will be rotated as the new master to control message communications.

Traditionally, the world most widely used MVB controller D113 is developed by implementing the underlying C and VHDL codes manually, according to the discussion with the engineers from Duagon company. Recently, China CNR corporation and Tsinghua University cooperate to develop their MVB controller based on our proposed approach, and the result controller is named TiMVB. First, we build Stateflow model strictly according to the description of IEC Standard 61375. The overall structure of the model is presented in Figure 9, and we get permission to make the module master rotation and part of memory traffic control public. 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, the main logic and accompanied timer of which are presented in [22]. After preliminary Stateflow validation on two MVB controller instances, we translate the main logic and some accompanied Stateflow models into 32 corresponding parallel timed automata within 0.3 seconds and verify some properties described in table II. Those properties are 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. In the communication process, there may be inconsistencies such that the frame sequences are out of order or not satisfied with time requirements.

TABLE II
PROPERTY LIST

Property	Formula	Time(second)
P1	A[ ] Process_Chart_OneMVB1(2)_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master and Process_Chart_OneMVB2(1)_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	32.93
P2	A[ ] not (Process_Chart_OneMVB1_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master) and Process_Chart_OneMVB2_LOGIC .Chart_OneMVB2_LOGIC_Rrgular_Master	29.34
P3	A[ ] not (Process_Chart_OneMVB1_LOGIC .Chart_OneMVB1_LOGIC_Standby_Master) and Process_Chart_OneMVB2_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	33.02

The first 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 throughout train communication network. Then, we design the second and third property to differentiate the counter example of the two situations, respectively.

Through manual analysis of counter examples demonstrated in Uppaal, we trace back to Stateflow model. For the counterexample of the first situation, initially, there is one MVB controller in state "Regular_Master" and the other in state "Standby_Master". If the "Standby_Master" MVB controller receives no master frame because of packet loss on bus, it will trigger a timeout T_standby_event and go to state "Regular_Master". While the other MVB controller is still in state "Regular_Master", there will be two masters at the same time. For the counterexample of the second situation, if both MVB controllers are in state "Regular_Master", they will send master frame separately, and master collision event would be triggered. They will transit to the state "Standby_Master" when they receive the master collision event, and there will be no master within network.

Furthermore, these two problems can be traced back to the handling logic of timeout event and master collision event described in Figure 105 of IEC standard. For the first problem, we propose to add a handshake before standby master changes to regular master because of the timeout. For the second problem, when a collision happens, we propose to withdraw the responsibility of MVB master controller that is the slave in the previous cycle. Those changes are captured in the revision of Stateflow model, and the translated Uppaal timed automata of the revised Stateflow model passes verification.

In master and slave frame communication process, there may be inconsistencies such that the frame sequences are out

of order. Part of the master frame generator logic described from page 236 of IEC 61375 are modelled as Stateflow model in [22]. Properties about master and slave communication process defined on the translated Uppaal timed automata of other parts of Stateflow model are verified, and the violations such as incorrect packet retransmission presented in our previous work [31], [32] are reproduced in this approach. We revise Stateflow model as well as the backend IEC standard according to analysis results of counter examples. These bugs and ambiguousness have already been submitted, proved and would be revised in the new version of IEC standard.

Following the implementation style of D113, that data frame processing logic and process data communication logic are implemented in VHDL, and message data communication logic and master transfer logic are implemented in C. We generate C code and VHDL code from the indirectly verified Stateflow model. Before synthesizing those codes in FPGA and ARM processor directly, we encode some lightweight runtime monitors into the generated code first. As described in section IV-B, we use data centered software monitor to verify some dynamic environment related behaviors that are not easy verify in model level. As described in IEC standard, the suggested time constraint on an MVB slave controller between the finish of a master frame receiving and the start of a slave frame responding should be less than 4us, and the time constraint on an MVB master controller between the finish of a master frame sending and the start of a response slave frame receiving should be less than 42.7us. Those two properties are not easy to capture in model level, because it is not easy to model dynamic transmission delay of data on MVB bus in Stateflow, even with a preliminary channel model.

Those constraints are described with data centered runtime verification property below. Variables are related to interfaces of VHDL code, which are configured to pins of the hardware platform. Those variables will be continuously loaded by accompanied C functions. Then, the property and accompanied C functions are transformed and input to RMOR to get the instrumented code. At last, generated VHDL codes, C codes, and monitor codes are synthesized to system platform with eCos for final testing, as presented in Figure 8 and 9.

```
DataCenter Monitor TimeConstraints()
{
   ......
   event TimeoutReply =
    ((T_Master_Receive - T_Slav_Send_)<4)
   event TimeoutResponse =
    ((T_Master_Send - T_Slav_Receive)<42.7)
   event Trigger =
     TimeoutReply || TimeoutResponse;

   state safe{
    When Trigger -> error;
   }
}
```

Listing 5. Runtime Monitoring for Time Interval Between Master and Slave Frames.

Unfortunately, the runtime monitor reports an error because of TimeoutReply event. The time is 6.4us, which is greater than 4us. We solve the problem by changing the time-consuming GPIO operation of notifying the arrival of master frame to direct hardware interrupt, and change the arbitration mechanism for reading access of register pool for slave master data. So the slave MVB controller can response more quickly and the time is about 3.4us for the revised one, as in Figure 9. The implemented TiMVB based on the proposed approach is now deployed in real trains of China and Argentina.

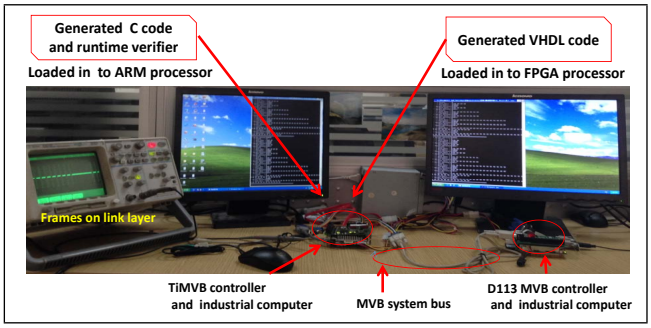

Fig. 8. Real system platform simulation between D113 controller and TiMVB controller. The left is the implemented TiMVB, and the right is D113.

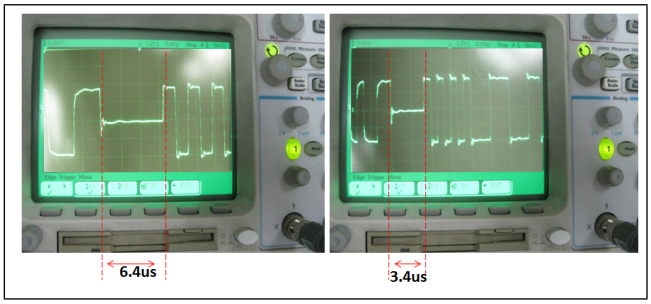

Fig. 9. We use oscilloscope to test the result that we get from the monitor. The left is the system for the GPIO operation which is 6.4us, and the right for the hardware interrupt and higher access priority one which is 3.4us.

VI. Conclusion and Disscussion

In this paper, we present an approach to address the verification challenge of Stateflow based MDD. By translating Stateflow model to timed automata, Uppaal can be incorporated to assist Simulink Design Verifier for more safety and liveness properties verification. With customizing runtime monitors to generated codes of Stateflow model, RMOR can be incorporated to assist Simulink Polyspace for verification of more concrete properties, even related with physical platform. In this way, properties are not only satisfied at the model level, but also consistently verified at the implementation level with dynamic the physical execution environment in consideration.

Discussion and Ongoing Work: Right now, our approach covers all semantic examples in Stateflow user guide [6] except for examples with encoded Matlab function. We plan to capture the translation of function next step. Translated timed automata are about 6 times larger than the original Stateflow model, in terms of state and transition numbers. This is mainly caused by complex event stack of Stateflow, and hieratical, crossover and interruptible execution logic. We plan to optimize our translating strategy to get more compassed timed automata and add some position information to make the translated timed automata well displayed in Uppaal. Automatical trace back tools from the counterexample of Uppaal timed automata to Stateflow model will be researched. Besides, 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. Furthermore, we have also checked with engineers from MathWorks to validate our translation.

As for runtime verification, we make use of existing tools and previous data-centered runtime verification technique, and customize them onto the automatically generated codes of the validated model directly. Currently, runtime verified properties need to be written manually, which is sometimes time-consuming because properties are related with underlying codes. We plan to study the relationship and mapping rules between Stateflow model and generated codes, and try to automatically generate and infer the specification of runtime verification properties, from those verified at the model level.

ACKNOWLEDGEMENT

This work is supported in part by NSF CNS 13-30077, NSF CNS 13-29886, NSF CNS 15-45002, NSFC 61303014, NSFC 61202010, and NSFC 91218302.

REFERENCES

[1] P. Caspi and etc., "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," in *ACM Sigplan Notices*, vol. 38, no. 7. ACM, 2003, pp. 153–162.
[2] SimulinkDesignVerifier. http://www.mathworks.com.
[3] SimulinkPolySpace. http://www.mathworks.com.
[4] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236.
[5] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *ACM SIGPLAN Notices*, vol. 42, no. 10. ACM, 2007, pp. 569–588.
[6] I. The MathWorks, "Stateflow user guide," www.mathworks.com.
[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*, vol. 14, no. 6, pp. 653–671, 2012.
[8] R. Alur, "Timed automata," in *Computer Aided Verification*. Springer, 1999, pp. 8–22.
[9] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.
[10] S. Sims and D. C. DuVarney, "Experience report: the reactis validation tool," in *ACM SIGPLAN Notices*, vol. 42, no. 9. ACM, 2007, pp. 137–140.
[11] 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*. IEEE, 1996, pp. 237–249.
[12] B. Tester, "Applied dynamics international," www.adi.com/.
[13] 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*. Springer, 2008, pp. 204–208.
[14] 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*. ACM, 2008, pp. 89–98.
[15] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR'97: Concurrency Theory*. Springer, 1997, pp. 135–150.
[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.
[17] K. L. McMillan, "The smv system," in *Symbolic Model Checking*. Springer, 1993, pp. 61–85.
[18] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "Pat: Towards flexible verification under fairness," in *Computer Aided Verification*. Springer, 2009, pp. 709–714.
[19] H. Wernli, M. Paulat, M. Hagen, and C. Frei, "Sal-a novel quality measure for the verification of quantitative precipitation forecasts," *Monthly Weather Review*, vol. 136, no. 11, pp. 4470–4487, 2008.
[20] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam, "From verification to implementation: A model translation tool and a pacemaker case study," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012, pp. 173–184.
[21] 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)*, vol. 13, no. 4s, p. 127, 2014.
[22] J. Yu, in *Uiuc*. https://sites.google.com/site/jiangyu198964/home.
[23] F. Chen and G. Roşu, "Java-mop: A monitoring oriented programming environment for java," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 546–550.
[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP 2001Object-Oriented Programming*. Springer, 2001, pp. 327–354.
[25] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt, "An expressive aspect language for system applications with arachne," in *Transactions on Aspect-Oriented Software Development I*. Springer, 2006, pp. 174–213.
[26] K. Havelund, *Runtime verification of C programs*. Springer, 2008.
[27] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable cots-based real-time embedded systems," in *Real-Time Systems Symposium, 2008*. IEEE, pp. 481–491.
[28] L. Sha, "Using simplicity to control complexity," *IEEE Software*, no. 4, pp. 20–28, 2001.
[29] Y. Jiang and L. Sha, "Use Runtime Verification to Improve the Quality of Medical Care Practice," in *38th International Conference on Software Engineering, 2016*.
[30] I. Commission *et al.*, "Iec 61375," *Train Communication Network*.
[31] Y. Jiang, H. Zhang, X. Song, W. N. Hung, M. Gu, and J. Sun, "Verification and implementation of the protocol standard in train control system," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 549–558.
[32] Y. Jiang, H. Zhang, H. Liu, X. Song, M. Gu, and J. Sun, "Design of mixed synchronous/asynchronous systems with multiple clocks," 2014.
[33] G. Berry, "Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel," in *Formal Methods for Industrial Critical Systems, 2007*.
[34] J. Qian *et al.*, "Modeling and Verification of Zone Controller: the SCADE Experience in China?s railway systems," in *?2015 IEEE/ACM International Workshop on Complex Faults and Failures in Large Software Systems*.
[35] H. Basold*et al.*, "An Open Alternative for SMT-Based Verification of SCADE Models," in *?2014 Formal Methods for Industrial Critical Systems*.