Dependable Model-driven Development of CPS: From Stateflow Simulation to Verified Implementation

YU JIANG, Tsinghua University
HOUBING SONG, Embry-Riddle Aeronautical University
YIXIAO YANG, HAN LIU, and MING GU, Tsinghua University
YONG GUAN, Capital Normal University
JIAGUANG SUN, Tsinghua University
LUI SHA, University of Illinois at Urbana-Champaign

Simulink is widely used for model-driven development (MDD) of cyber-physical systems. Typically, the Simulink-based development starts with Stateflow modeling, followed by simulation, validation, and code generation mapped to physical execution platforms. However, recent trends have raised the demands of rigorous verification on safety-critical applications to prevent intrinsic development faults and improve the system dependability, which is unfortunately challenging. Even though the constructed Stateflow model and the generated code pass the validation of Simulink Design Verifier and Simulink Polyspace, respectively, the system may still fail due to some implicit defects contained in the design model (design defect) and the generated code (implementation defects).

In this article, we bridge the Stateflow-based MDD and a well-defined rigorous verification to reduce development faults. First, we develop a self-contained toolkit to translate a 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 that 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 a 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 offers more opportunities for engineers to achieve high reliability of the target system compared with the traditional act that only relies on Simulink Polyspace. 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 to the development of a typical cyber-physical system-train communication controller based on the IEC standard 61375. Experiments show that more ambiguousness in the standard are detected and confirmed and more development faults and those corresponding errors that would lead to system failure have been removed. Furthermore, the verified implementation has been deployed on real trains.

This article is a significant extension of Jiang et al. (2016c).
This work is supported in part by NSF CNS 13-30077, NSF CNS 13-29886, NSF CNS 15-45002, NSFC 61303014, NSFC 61202010, NSFC 91218302 and Jiangxi Province Major Project 20171ACE50025.
Authors' addresses: Y. Jiang, Y. Yang, H. Liu, M. Gu, and J. Sun, East-Main Building, 11-315, School of Software, Tsinghua University, China; emails: {jy1989, yx1991, lh1990, gming, sjg}@mail.tsinghua.edu.cn; H. Song, Daytona College of Engineering, Daytona Beach campus, FL; email: Houbing.Song@erau.edu; Y. Guan, Main Building, 238, Capital Normal University, Beijing, China; email: guanyong@mail.cnu.edu.cn; L. Sha, Siebel Center for Comp Sci, 201 N. Goodwin Ave. Urbana, IL; email: lrs@cs.uiuc.edu.cn.

2378-962X/2018/08-ART12 $15.00
https://doi.org/10.1145/3078623

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

General Terms: Design, Verification, Measurement

Additional Key Words and Phrases: Model-driven development, simulink stateflow, formal verification, timed automaton, runtime verification

ACM Reference format:
Yu Jiang, Houbing Song, Yixiao Yang, Han Liu, Ming Gu, Yong Guan, Jiaguang Sun, and Lui Sha. 2018. Dependable Model-driven Development of CPS: From Stateflow Simulation to Verified Implementation. *ACM Trans. Cyber-Phys. Syst.* 3, 1, Article 12 (August 2018), 31 pages.
https://doi.org/10.1145/3078623

1 INTRODUCTION

Cyber-physical systems (CPS) have been increasingly adopted (H. Song and Brecher 2016; Ahmed et al. 2013; Mehmood et al. 2017; Jiang et al. 2013b), and Simulink is a widely used tool for model-driven development (MDD) of CPS that provides delicated support for graphical Stateflow modeling, interactive model-level simulation, some basic design validation, along with C, C++, and VHDL code generation and verification (Caspi et al. 2003). In practice, Simulink has been successfully applied across various industry applications such as smart manufacturing control and signal processing systems, where Simulink Design Verifier (MathWorks 2017a) and Simulink Polyspace (MathWorks 2017b) are taking the responsibility of uncovering design defects and implementation defects, respectively. The two defects are the main intrinsic development faults of MDD (Avižienis et al. 2004; Avizienis et al. 2004).

However, for safety-critical applications such as medical devices and avionics, Simulink is still insufficient to ensure dependability. Specifically, the verification capability of Simulink Design Verifier is limited to basic properties. It detects defects 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, guaranteeing the dependability of the whole system remains non-trivial, and implicit design defects and implementation defects wold lead to system failure. Hence, supporting tools with more verification power such as Uppaal (Behrmann et al. 2004) is expected here to check the properties of Stateflow model and more rigorous formal techniques such as runtime verification (Chen and Roşu 2007) should be applied to reduce development faults and ensure the dependability of the automatically implemented CPS.

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 1,366-page user guide informally (MathWorks 2017c). Advanced modeling features such as event stack, event interruption, complex state activating and deactivating mechanism, boundary transition, and transitional action and so on, are non-straightforward to formalize for verification. Although there are many existing works on translation-based verification of the Stateflow model, most of them are efficient and work well covering the most related modeling features within their own domains (Chen et al. 2012). 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 systems.[1]

In this article, we present an approach to address the verification challenge at both the model and implementation levels of Stateflow-based MDD of CPS, thus achieving assured dependability. In terms of model-level verification, we develop a tool STU to translate Stateflow model into timed automata for formal analysis based on a model checking tool called Uppaal (Alur 1999). Timed automata can be used to model and analyze the timing behavior of CPS, and the methods for checking both the safety and liveness properties of timed automata have been well developed and intensively studied in Uppaal. Most frequently used Stateflow modeling features (*composite 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 MathWorks.

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 the Simulink Design Verifier would be found via Uppaal verification. If errors are detected, then the Stateflow model needs to be analyzed and revised with the help of mapping dictionary for translation. As a result, more design defects would be prevented and removed, and 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 that 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 poses 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 could not be described based on the abstract Stateflow model. Hence, more implementation defects wold be prevented and removed. The overall procedure about the proposed extended approach is presented in Figure 1.

The main lessons learned through this work include the following: (1) Even though the Stateflow model and the generated code pass the validation of the Simulink Design Verifier and Simulink Polyspace, respectively, the system may still fail due to some implicit defects contained in the model and generated code, and more comprehensive verification is needed to ensure the dependability; (2) verification is relatively new to MDD engineers, and we need easy to use and low-complexity solutions to convince and facilitate developers to trust and use verification results; and (3) new and improved techniques for explaining the verification results are needed to make verification results more actionable.

The rest of article is organized as follows. Some background 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 presents 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 proposed approach.

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

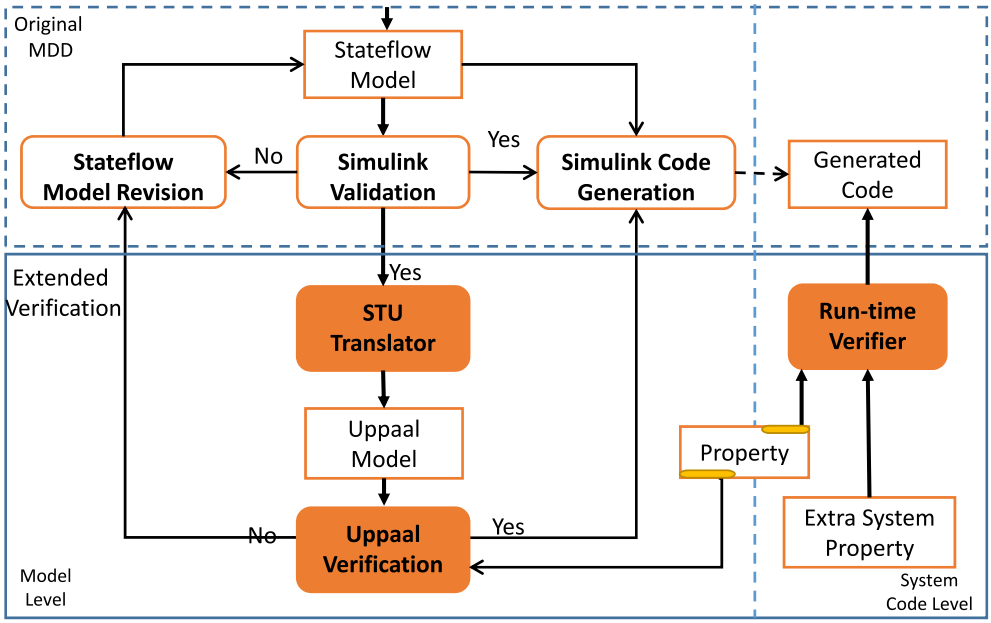

Fig. 1. Integrate STU and runtime verification into Stateflow-based MDD.

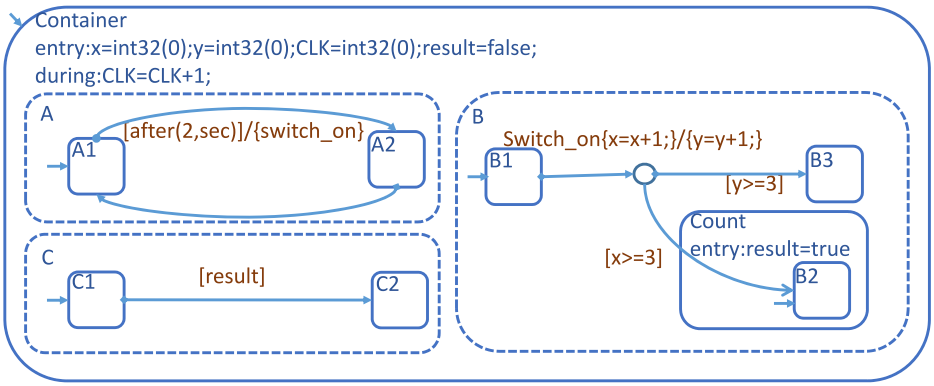

Fig. 2. A Stateflow example for counter task that covers most advanced modeling features.

2 PRELIMINARIES

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

2.1 Simulink Stafeflow

The model in Figure 2 is an example of a Stateflow diagram that covers most advanced modeling features. The outmost composite state *Container* is parallelly decomposed into three sub-composite states, *A*, *B*, and *C*. States *A* and *C* are further serially decomposed into 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 2s, 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. However, the statement $y = y + 1$ is a transitional action that 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.

More specifically, the Stateflow model is an extended hierarchical state machine that 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: This represents the operating mode of the system. The occurrence of an event will trigger the execution of the 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 that are connected in parallel or serially. 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, 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 an explicit or implicit signal that triggers the execution of transition, *condition* is a Boolean expression that allows the transition to be taken with value that is true, the *conditional action* is the operation that is immediately executed when the condition is met, and the *common action* is the operation that will be executed when the condition is met and there is a non-interrupted valid path between the source state and the target state. Each transition also has an implicit priority of execution, determined by the information such as the hierarchy level of the destination state, the position of the transition source, and so on.

Event: There are two types of events 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: This 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

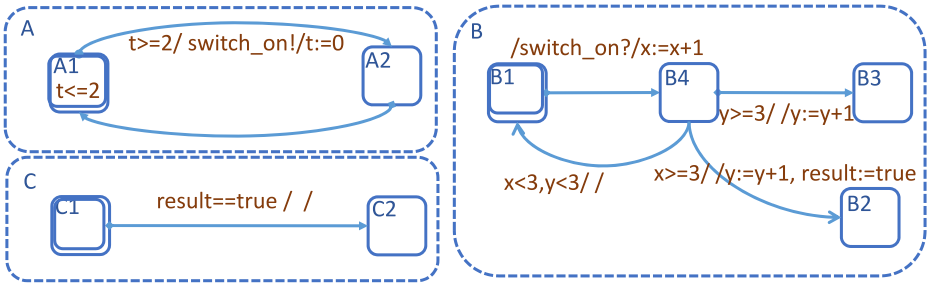

Fig. 3. Manually constructed timed automata for counter.

state is already active and stays in, and *Exit action* is executed when the state changes from active to inactive.

Junction: This 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 represent second, millisecond, and microseconds, respectively.

2.2 Uppaal Timed Automata

The model in Figure 3 is an example of a network of timed automata that 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, then 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 where clock variables evaluate to real numbers, and all clocks progress synchronously. It can be defined as a tuple consisting 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 $(l_1 \xrightarrow{g,\phi,r} l')$, if there is no synchronization over channels (ϕ denotes an "empty" action), or $(l_1 \xrightarrow{g,ch*,r} l')$. 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:

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

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 timed computation tree logic (TCTL). It has been applied successfully ranging from communication protocols to real-time cyber-physical applications.

2.3 Runtime Verification

Runtime verification can be used for many purposes, such as debugging or safety policy monitoring, verification, behavior modification, and so on. 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 Leucker and Schallhart (2009), 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 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.

We use past time linear temporal logic (*ptLTL*) to specify the property that provides temporal operators that refer to past states of an execution trace relative to a current point of reference (Laroussinie et al. 2002). The syntax and semantics are as below.

ptLTL syntax: Let $AP = \{p_1, p_2 \cdots p_i \cdots p_n\}$ be a set of atomic propositions, then *ptLTL* formulae is as follows:

$$\phi, \psi ::= p_i \mid \neg\phi \mid X\phi \mid X^{-1}\phi \mid \phi \wedge \psi \mid \phi S \psi \mid \phi U \psi,$$

where U, S, X, X^{-1} stands for the *"until," "since," "next,"* and *"previous"* temporal operators, respectively. Based on these basic operators and some standard abstractions, temporal operators such as

"eventually," "always," "always in the past," and *"eventually in the past"* can be defined and denoted as F, F^{-1}, G, and G^{-1}, respectively. From definition, we can see that *ptLTL* extends classical logic *LTL* with the past modalities, which is good for complex temporal properties presented in clinical guideline.

ptLTL semantics: *Let ω be an infinite sequence $\omega = \omega_1\omega_2\cdots$*, with a mapping $\eta : \forall i, \omega_i \rightarrow 2^{AP}$ labeling atomic propositions that hold in each position ω_i. With the structure path (ω, η), a non-negative integer i, and *ptLTL* formulae ϕ and ψ, the relation that "ϕ holds at position ω_i" denoted as $\omega, i \models \phi$ can be inductively defined as below:

$$
\begin{aligned}
&\omega, i \models p && \textit{iff } p \in \eta(\omega_i)\\
&\omega, i \models \neg\phi && \textit{iff } \omega, i \not\models \phi\\
&\omega, i \models X\phi && \textit{iff } \omega, i+1 \models \phi\\
&\omega, i \models X^{-1}\phi && \textit{iff } \omega, i-1 \models \phi\\
&\omega, i \models \phi \wedge \psi && \textit{iff } \omega, i \models \phi \textit{ and } \omega, i \models \psi\\
&\omega, i \models \psi S \phi && \textit{iff } \exists m \in [0, i], \omega, m \models \phi,\\
& && \quad \textit{and } \forall n \in [j+1, i]\omega, n \models \psi\\
&\omega, i \models \psi U \phi && \textit{iff } \exists m \in [i, \infty), \omega, m \models \phi,\\
& && \quad \textit{and } \forall n \in [i, m)\omega, n \models \psi,
\end{aligned}
$$

where two *ptLTL* formulae ϕ and ψ are said to be equivalent, when condition "$\omega, i \models \phi$ *iff* $\omega, i \models \psi$" is satisfied for all structure path ω and i.

3 RELATED WORK

In the past decades, a variety of computation models and the corresponding toolkits has been proposed to facilitate the design of cyber-physical system, among which the two most widely used are the SCADE suite based on safety state machine (Berry 2007) and the Simulink toolkit based on Stateflow (Caspi et al. 2003). 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 the SCADE observer itself. The Boolean outputs are the proof objectives for DV that then automatically produces counter-examples. However, while the 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 (Qian et al. 2015; Basold et al. 2014). 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.

Similarly to a SCADE suite, 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 the Simulink Design Verifier in acquiring correctness of Stateflow model, which can be classified into two categories, simulation-based techniques, and verification-based techniques. The simulation-based technique is adopted widely, while the main challenge is to solve the coverage of simulation patterns. Many researchers have developed test generation tools for Simulink designs, including Reactis (Sims and

DuVarney 2007), T-VEC (Blackburn and Busser 1996), Beacon Tester (Tester 2017), and AutoMOTgen (Gadkari et al. 2008), and so on. 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 the Simulink Stateflow model (Alur et al. 2008).

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 (Bouajjani et al. 1997), Lustre (Halbwachs et al. 2002), SMV (Mcmillan 1993), PAT (Sun et al. 2009), hoare logic, and SAL (Wernli et al. 2007), which can be verified through the corresponding supporting tools. Most of them perform 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 the PAT-based (Chen et al. 2012) verification technique, they covered most of the advanced features of Stateflow, with limited support of the event interrupt dispatch mechanism and time operation support. There is also some nice work translating Uppaal timed automata to Simulink Stateflow for simulation and code generation (Pajic et al. 2012, 2014). 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, and so on, 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 have 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 that has been limitedly supported before. Uppaal is chosen, because timed automata can be used to model and analyze the 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 the 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 a 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 Jiang et al. (2015), Jiang et al. (2013a), Jiang et al. (2015), Jiang et al. (2016b), and Yang et al. (2016). For example, in Jiang et al. (2015), the author proposes a heterogeneous modeling language to model both data-oriented and control-oriented behavior of train controller. In Jiang et al. (2013a), 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.

4 EXTENDED DEPENDABLE 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.

4.1 Formal Verification of Stateflow

In 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 noted, the key differences between Stateflow and timed automata are as follows:

(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 a hierarchy structure that is combined with a 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.

4.1.1 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 a 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, then the event will be deleted by the extra *daemon automata*, which is responsible for deleting the 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, then 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().*

```
void DispatchEvent(Event E){
   PushEvent(E.Event, E.Dest, -1, E.AutomatonType, true);
}
```

Listing 2. Dispatch an event.

```
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 3. Push an event.

```
void PopEvent(Event Stack){
   Top--;
}
```

Listing 4. Pop an event.

```
Event StackTopEvent(){
   return Stack[Top];
}
```

Listing 5. Return an event.

```
bool EventSentToMe(){
   bool r1 = false;
   bool r2 = false;
   Event E = StackTopEvent();
   r1 = (E.Dest == StateId);
   r2 = (E.AutomatonType == Type);
   return r1 && r2;
}
```

Listing 6. Estimate the event sent to itself or not.

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 that 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, then *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, then 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 7. Delete an invalid event.

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

Listing 8. Generate a system event.

Based on the structured virtual stack, we translate Stateflow into Uppaal timed automata automatically. As noted, 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.

4.1.2 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, then s^{lf} should be activated first, and this is especially true for cross-boundary transitions. 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 push 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 Algorithm 1 in detail, as 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, then the deactivation event for sub-state with the lowest priority is pushed into stack through *DispatchEvent()* function first, and then the parallel sub-states with higher priorities are handled sequentially. If the current state is refined in serially connected sub-states, then the deactivation event for current active sub-state is pushed into the stack directly. The second

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
}
```

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, then it will dispatch an event to the corresponding *action automata*. If the current state is also a sub-state, then 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, as presented in Algorithm 2.

ALGORITHM 2: Composite State activation logic

```
Void StateDeactivationLogic(int state_s^f)
{
  DispatchActivationToParent(state_s^f);
  HandleActivation(state_s^f);
}

void DispatchActivationToParent(int state_s^f)
{
ParStates ← Parstate(s^f);
if (PaStates is deactive) then
    DispatchEvent(Event ActivationEvent);
end if
}
void HandleActivation(int state_s^f)
{
if (s^f has attached entry action) then
    Event ActivationEvent.AutomatonType = action;
    DispatchEvent(Event ActivationEvent).
end if
if (s^f is a sub-state) then
    int UpperLevelDest = AutomatonID(Parstate(s^f));
    Event ActivationEvent.Dest = UpperLevelDest;
    DispatchEvent(Event ActivationEvent);
end if
}
```

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 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.

4.1.3 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:

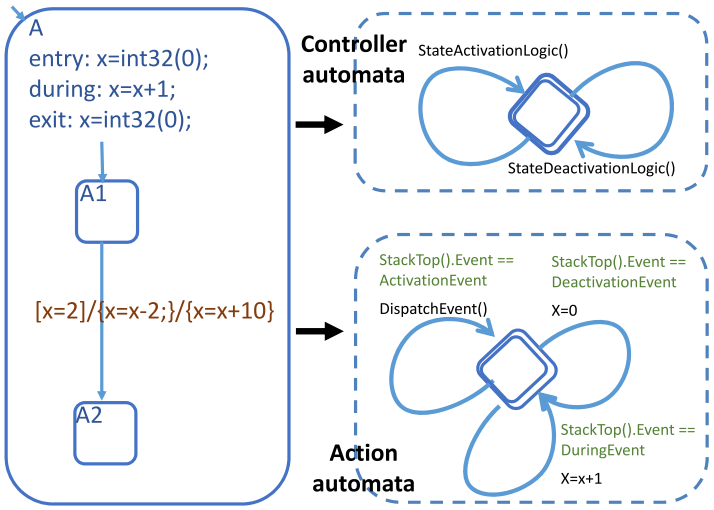

Fig. 4. The controller and action automata for a composite state transformation, capturing activation, and deactivation.

(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, then 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

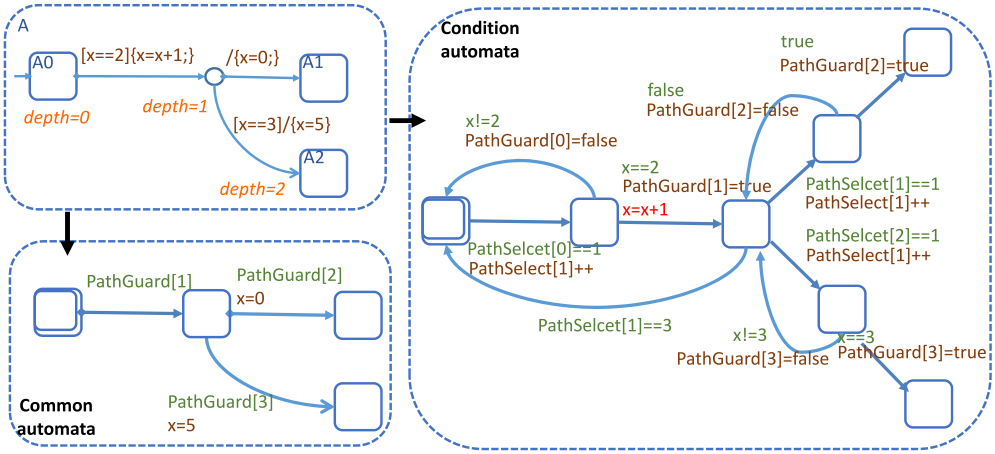

Fig. 5. The common and condition automata for a composite state transformation, capturing internal transition.

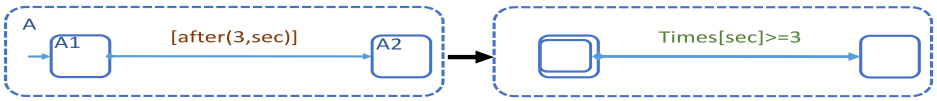

Fig. 6. Time transformation integrated in the condition automata.

node and 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.

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.

4.1.4 Timer Transformation Rule. Within the Stateflow model, time operation is based on event and is usually used as a time-related condition on transition. As described, it is characterized as *[TmOp(Num, Event)]*. We count the appearance times *(Num)* of event *(Event)*. The value is increased by one and stored using an int array *Times[ ]*, when an event is dispatched. An example is presented in Figure 6, and 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*,

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

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 14,590 lines of java code with two supporting libraries (JDOM

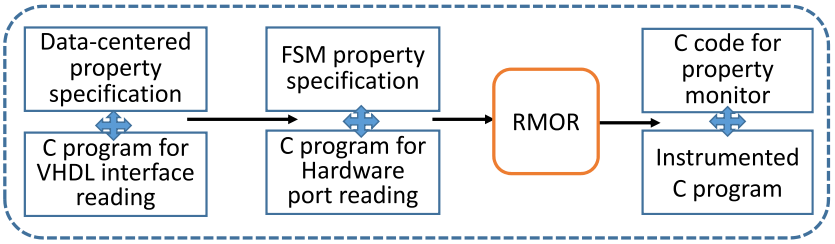

Fig. 7. Hardware runtime verification customisation on Software.

used for read and write XML file and Antlr is used for abstract syntax tree construction and update). The parser extracts Stateflow model from Simulink project file into memory. The translator transfers the 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 the Stateflow model based on Uppaal and can be downloaded from Yu (2017).

4.2 Runtime Verification of System

The key technical ingredient in runtime verification is to specify dynamic runtime environment-related properties that could not be easily described based on the abstract Stateflow model and choose proper runtime monitoring tools according to adopted programming language. Over the past decade, tremendous effort has been invested in developing program runtime verification systems (Chen and Roşu 2005). Most of these works can be regarded as an extension of AspectJ (Kiczales et al. 2001). Some exceptions are ARACHNE (Douence et al. 2006) and RMOR (Havelund 2008). 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 (Pellizzoni et al. 2008).

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.

Based on the complexity reduction idea presented in our previous work (Sha 2001) 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 Jiang et al. (2016a), we have designed data-centered domain description language *DRTV* and translated the data-centered model to property monitors, based on which the main customization is presented in Figure 7. 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.

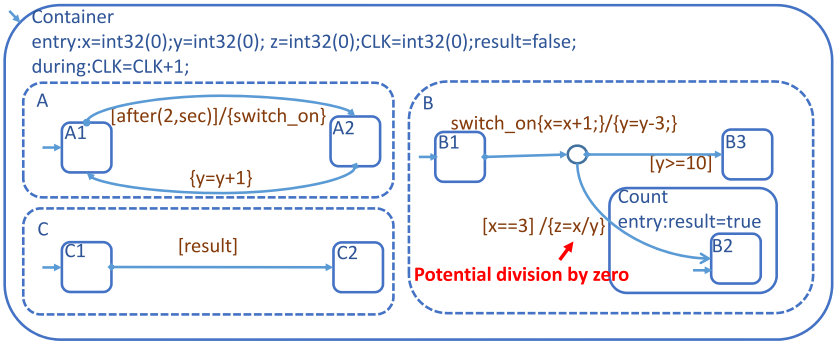

Fig. 8. Manual model for validation testing.

To initiate the runtime verification, we analyze the safety and liveness properties used in previous Uppaal verification to infer part of the specifications here. Furthermore, for system from real-world industrial domains, the cyber components often operate on physical execution platforms in dynamic environments. The assumptions and uncertainties about the physical-related operation environment are difficult to address in model-level verification but will also lead to critical failures. For example, the EU Ariane 5 rocket explosion during launch was traced back to the integration of a reused Ariane 4 software module, which relies on the fact that the speed variable is physically impossible to overflow in "Ariane 4."[2] They should be captured more actually. We can use the *ptLTL* formula of data-centered description language *DRTV* or FSM property specification of RMOR to facilitate the enforcement of such properties on system running in dynamic environment.

5 EXPERIMENT RESULTS

To evaluate the proposed approach in terms of development faults detection and prevention, we apply it to some artificial Stateflow-based design examples and a real train controller design case. We compared the number of design defects and implementation defects detected by the original Stateflow-based MDD and the extended approach.[3] For the artificial examples, we inject 100 development faults, which are 50 division by zero design defects and 50 deadlock design defects into the Stateflow models, and the Stateflow Design Verifier only detects 32 defects with 8 false positives, while Uppaal verification detects 100 defects with no false positives. For the real train controller design example, six implicit defects in the Stateflow model that cannot be detected in Design Verifier are detected in Uppaal verification based on the translated timed automata, and five implementation defects that cannot be supported in Polyspace can be specified and detected in a runtime verification monitor. Furthermore, real platform-based simulation shows that if those defects are not revised during the development procedure and remained in the system deployment, then the system will fail.

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 8, 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,

[2] http://www.around.com/ariane.html.

[3] The presented Steteflow models, translated timed automata, and properties specifications could be downloaded in website (Yu 2017).

Table 1. Property List

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

and the model passes the verification. But according to manual analysis, the value of y would be zero after 6s. Design Verifier failed to detect this implicit but general bug contained in the model.

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

In Table 1, "E <>" is a temporal keyword that means "eventually," "Process_Chart_Container_B.SSID49" is an automata state name corresponding to the Stateflow junction node, "Chart_x == 3" is an 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. Furthermore, we construct another 100 Stateflow models with division by zero defects or deadlock defects, Stateflow Design Verifier only detects 32 defects with 8 false positives, while Uppaal verification detects 100 defects with no false positives.

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 the 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, then 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 temporal properties are also supported, and the overall procedure is similar to Figure 7.

Real-time Train Controller: We apply the proposed approach to a real industrial application of Stateflow-based MDD of a train communication control system. According to IEC Standard 61375 (Schifers and Hans 2000), the control system consists of many multifunction vehicle bus (MVB) controllers that interconnect devices within a vehicle. The 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, the 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. Detailed functions of the MVBC are mainly based on the real-time protocol (RTP), which defines the rules (master-slave communication principle, data frame format, and timing requirements, etc.) for *Process Data* and *Message Data* transmission. The MVBC communication function model is an abstract representation of these typical behavior rules of MVBCs, which are well defined in IEC-61375-1.

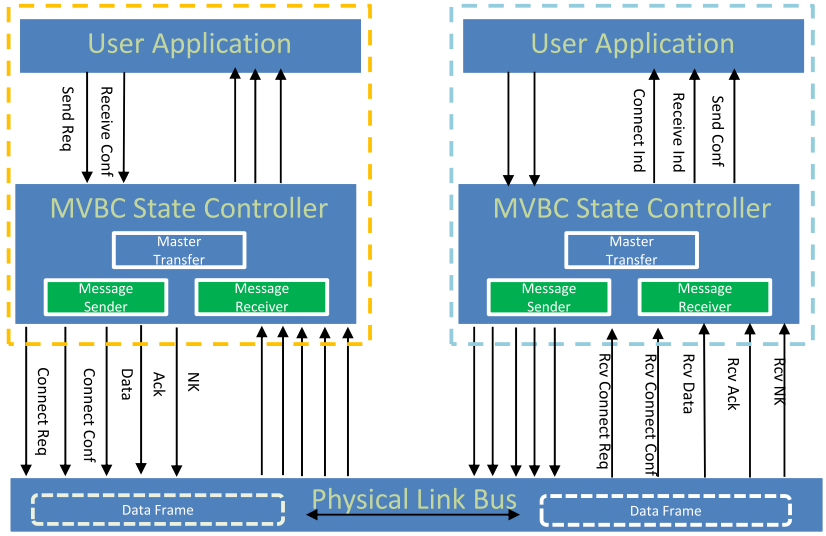

Fig. 9. System architecture model of the MVBC, where communication primitives are on the arrow.

As presented in Figure 9, the abstract system architecture model contains the `User Application`, `MVBC State Controller`, and `Physical Link Bus`. The `MVBC State Controller` mainly contains three components: `Master Transfer`, `Message Sender`, and `Message Receiver`. The components `Message Sender` and `Message Receiver` are responsible for transmitting the message from a producer to a consumer and provide the flow control and error recovery from end to end through a sliding window. The transmission of message is divided into three phases with related communication primitives: connection establishment (*Connect Req, Connect Conf, etc.*), acknowledged data transmission (*Data, Ack, Rcv Data, etc.*), and disconnection (*DisConnect Req, DisConnect Conf, etc.*). The component `Master Transfer` selects a master MVBC from one of the several MVBCs with bus administrator ability at the end of a macro period. These three components embody the core function of an MVBC. The information of the MVBC, i.e., device address and master frame sequence, is initialized during deployment. Detailed description about the abstracted model can be referred to IEC-61375-1.

Traditionally, the most widely used MVB controller, D113, was 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 cooperated to develop their MVB controller based on our proposed approach, and the result controller is named TiMVB. First, we build a Stateflow model strictly according to the description of IEC Standard 61375. The overall structure of the model is presented in Figure 9, and we got 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 Fig. 105 of IEC 61375 are modeled as Stateflow model, the main logic are presented in Figure 10. 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 0.3s and verify some properties described in Table 3. Those properties are derived from real potential hazards of system failure, and details are described in Appendix A. For example, in the MVB master and slave rotation process, there may be inconsistency such that two masters appear at the same time. In the communication process,

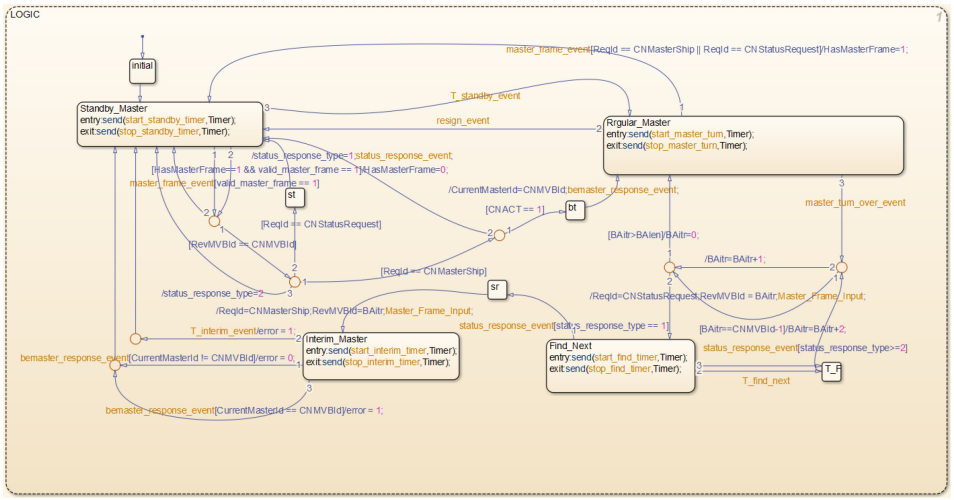

Fig. 10. Model for the master transfer logic, this model is for mvb current standby mvb controller, and the parallel stateflow model for current master mvb controller is the same but initialized in "Regular_Master" state. The state "Fine_Next" is used to pass the control from current master to standby master.

Table 2. Property List

Property	Formula	Time (seconds)
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

there may be inconsistencies such that the frame sequences are out of order or not satisfied with time requirements.

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 properties 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, then 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 the 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 Fig. 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 Yu (2017). 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 (Jiang et al. 2013a, 2015) 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 have already been 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. Details about the code generation procedure are described in Appendix B. Before synthesizing those codes in the FPGA and ARM processor directly, we encode some lightweight runtime monitors into the generated code first. As described in Section 4.2, we use a 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 4μs, 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 42.7μs. 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 11 and 12.

Unfortunately, the runtime monitor reports an error because of TimeoutReply event. The time is 6.4μs, which is greater than 4μs. We solve the problem by changing the time-consuming GPIO operation of notifying the arrival of the master frame to the direct hardware interrupt and change the

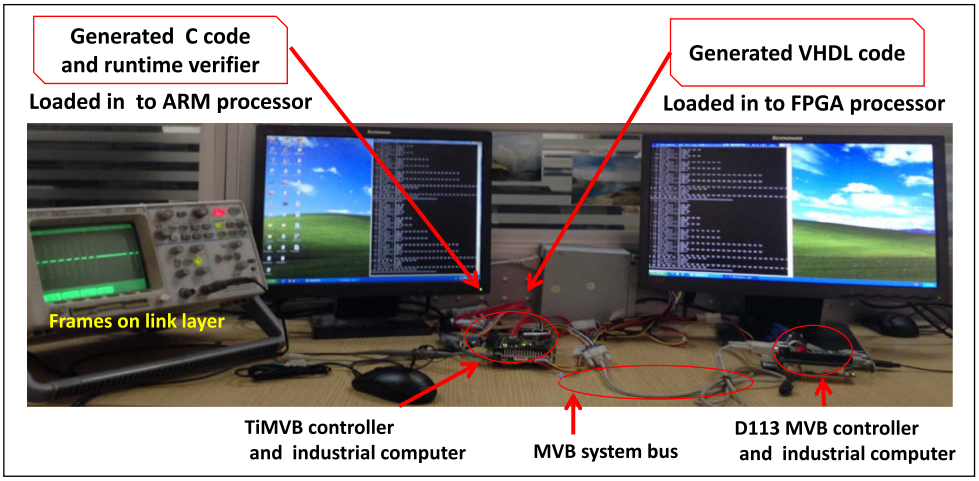

Fig. 11. Real system platform simulation between D113 controller and TiMVB controller. The left is the implemented TiMVB, and the right is D113.

```
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 9. Runtime Monitoring for Time Interval Between Master and Slave Frames.

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 3.4μs for the revised one, as in Figure 12. If we do not revise this implementation defects and this implementation fault is passed through the deployment stage, after 1,000 micro cycles, then the accumulated time deviations would lead to crash of the data communication service. In summary, six implicit defects in Stateflow model that cannot be detected in Design Verifier are detected in Uppaal verification based on the translated timed automata, and five implementation defects that cannot be supported in Polyspace can be specified and detected in runtime verification monitor. The implemented TiMVB based on the proposed approach is now deployed in real trains of China and Argentina.

Scalability Analysis: With the artificial examples and the real system design, the main procedures of our approach have been demonstrated including the model transformation, the model verification, and code level runtime monitor. During the model transformation, the two points of importance are the parser and translator, both with the complexity O(N), where N is the sum number of state, event, and junction node contained in the Stateflow model. During the model verification, the complexity depends on the verification algorithm applied to the timed automata. In Uppaal, the overall time complexity for model checking a property formula ϕ is linear in $|\phi|$ and polynomial in $|N'|$. The size of $|\phi|$ equals the number of logical connectives and temporal

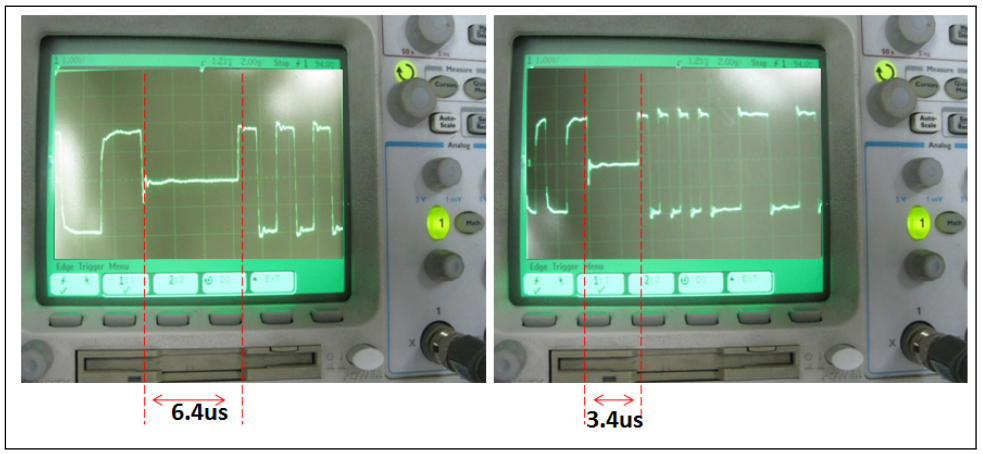

Fig. 12. We use oscilloscope to test the result that we get from the monitor. The left is the system for the GPIO operation that is 6.4μs, and the right for the hardware interrupt and higher access priority one which is 3.4μs.

operators in the formulas plus the sum of the sizes of the temporal operators, and $|N'|$ equals the number of states contained in the timed automata. For the lightweight code level runtime monitor, the verification complexity is linear to the size *ptLTL* formula $|\phi'|$, and RMOR will generate the corresponding monitor code automatically, and the generation complexity is also linear. Based on the complexity analysis, we can draw the conclusion that the proposed approach can be applied to the analysis of complicated Stateflow model and design of real systems, detect and prevent more development faults with more rigorous verification, and improve the dependability of the cyber-physical systems.

6 LESSONS WE LEARNED

(a) System may still fail even when they pass the validation of Design Verifier and Polyspace, and more comprehensive verification is needed to ensure the dependability: During the procedure of Stateflow-based MDD, even when the Stateflow model and the generated code pass the validation of Simulink Design Verifier and Polyspace, respectively, some complex system products may still fail because of some implicit bugs contained in the model and the generated code. Detection of those implicit bugs related with hardware code and temporal properties are not well supported in current version of Design Verifier and Polyspace, and the false-positive rate of deadlock and division by 0 is very high in Design Verifier. During the validation of MVB Stateflow model, five of nine reported deadlocks are not real deadlocks. Enhanced verification is needed.

Another interesting point within this lesson is that even if the model and generated code pass the proposed verification in the lab and processing place, the final system product may still fail. This is often in industrial systems. One important reason is that the unexpected dynamic runtime environment of industrial applications effects the hardware platform or the behavior of code. For example, during the engineering development practice of the MVB system, the initial version of TiMVB works well in the lab and processing lab. When deployed on a railway in Beijing for a final no-load test, it fails at night and early morning but works well in daytime. We found that the system failure is because, during the winter of Beijing, the temperature of early morning and night is low, and it affects the synthesized hardware, making the delays longer and influencing the function correctness of the generated hardware code. We solve the problem by change the commercial chip

to industrial chip, which is more immune to temperature effects. Those kind of easy but easy to ignore properties and assumptions related with environment and physical platform need to be captured during the procedure of MDD.

(b) Easy to use solutions are needed to facilitate developers to use formal verification: Verification is relatively new to MDD engineers, and we need easy and low-complexity solutions to convince and facilitate developers to believe and use verification results. Currently, it is not reasonable for developers to pay much extra efforts and to be an expert for complex verification techniques, we need to reduce their work by automatic translation of model and low complex customization. For example, during the customization of the runtime verifier, we plan to use separate software and hardware monitor directly, but the developers from CNR thought that heterogenous monitors will bring challenges for them to handle and increase the risk brought by new techniques. Hence, we reduce the efforts by a data-centered monitor for both hardware and software. Also, the developers suggest that it would be better for us to provide some templates to translate the property described for the Design Verifier to the property described for Uppaal and RMOR in our future work, which will increase their willingness and facilitate their practice of our approach in their future developments.

(c) Actionable verification result is needed to convince developers to believe and use verification results: New and improved techniques for explaining the verification results are needed to make verification results more actionable. The problem of a failing verified property is to figure out which parts of the model and code were responsible for its failure. Although Uppaal will produce some counter example and the trace leading to the violation, the developer might still have to retrace some of the steps in the original Stateflow model before being able to identify the cause. We have provided mapping dictionary to help retrace. Automatical trace back tools from counter example of Uppaal timed automata and RMOR to Stateflow model are requested to help developers to debug more easily later.

7 CONCLUSION AND DISCUSSION

In this article, we present a novel 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 verifying more safety and liveness properties. With customizing runtime monitors to generate 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 physical execution environment in consideration.

Discussion and Ongoing Work: Right now, our approach covers all semantic examples in Stateflow user guide (MathWorks 2017c) except for examples with encoded Matlab function. We plan to capture the translation of function in 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 investigated. Besides, because execution semantics of Stateflow is described in informal natural languages based on examples, it is impossible to formally prove the equivalence and correctness of the transformation. We acquire correctness by carefully comparing simulation results of the translated model, including the value and state sequence step by step, in the same way as in 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 techniques 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 the 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.

APPENDICES

A REQUIREMENTS FORMALIZATION

The MVBC requirements are mainly derived from the descriptions of the MVBC conformance testing standard IEC-61375-2, accompanied with some hazard analysis in the real use of MVBC. For example, there are two requirements described in natural language as below:

(1) ***Message Transmission:*** During the acknowledged data transfer stage, when some packets are lost in the physical link layer, they should be retransmitted.
(2) ***Master Transfer:*** During the master transfer procedure, there is one and only one master MVBC contained in the train communication network.

The first requirement is related to the correctness of the acknowledged data transfer stage, where the producer sends the individual data packet of the message to the consumer. The consumer side may bundle acknowledgments. For example, the consumer may acknowledge several packets at the same time by acknowledging the packet with the highest sequence number only. When packets fail to be acknowledged, the producer shall retransmit them. Besides, the consumer may indicate to the producer that it receives an out of sequence packet. In this case, the consumer shall send a *Negative Acknowledgment* packet, indicating from which packet on it requires retransmission. For the second requirement, it is originated from the fact that the mastership can be shared by two or more MVBCs with administrator ability, which each exercises mastership for the duration of a turn. Also in case of failure of a master MVBC, mastership should be transferred to another MVBC. But it is not allowed that two MVBCs act as the master in the same time.

In the original development process, the requirements are tested through simulation as defined in the standard IEC-61375-2. Although the general methodology and procedures are well specified to test that the implemented MVBC is confirmed to the function described in IEC-61375-1, the result is highly dependent on the input patterns of the test cases, where the coverage of some extreme conditions may be ignored and hard to be enumerated due the limited number of input patterns. While in enhenced dependability development approach, we try to ensure that the MVBC implementation satisfies the requirements with formal verification, which is more rigorously than simulation-based testing.

A.1 Formalization of Model Verifiable Requirement

These requirements that are related to general functions of control logic and independent of platform are categorized as model verifiable requirements. We formalize the requirements as timed computation tree logic formulas defined on the formal model, and verify them on the translated formal timed automata. Let us take the two requirements described in the natural language above as an example. The safety hazards of the first requirement happen during the data acknowledgement process and the data retransmission process. The safety hazard of the second requirement happens during the MVBC master rotation process.

Table 3. Property List

Property	Formula	Stage
P1	A[ ](RECEIVER.SEND_NK→(NK==true))	Transmission
P2	A[ ]((NK==true)→(NK_number==next_send)	Transmission
P3	A[ ](RECEIVER.SEND_AK → (AK==true))	Transmission
P4	A[ ]((AK==true)→(AK_number==next_send))	Transmission
P5	A[ ] Process_Chart_OneMVB1(2)_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master and Process_Chart_OneMVB2(1)_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	Transfer
P6	A[ ] not (Process_Chart_OneMVB1_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master) and Process_Chart_OneMVB2_LOGIC .Chart_OneMVB2_LOGIC_Rrgular_Master	Transfer
P7	A[ ] not (Process_Chart_OneMVB1_LOGIC .Chart_OneMVB1_LOGIC_Standby_Master) and Process_Chart_OneMVB2_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	Transfer

For the retransmission process, we formalize the requirement as two properties based on the timed automata. When the SENDER automaton receives the rcv_NKi? signal, it will decide whether the sequence i lies in the legal interval or not. The decision logic is implemented on the guard of transition as (expected < NK_number ≤ send_not_yet). The value of this decision expression is assigned to a variable NK. At the same time, when the RECEIVER automaton sends a legal send_NKi! signal and switches to the SEND_NK state, the SENDER automata should decide the sequence number i to be true with Boolean evaluation ($NK == true$) and be able to deliver the retransmission request. This property is formalized as P1 in Table 1. Another property for the retransmission process is about rolling back the window of the SENDER automata in the retransmission process. After receiving the rcv_NKi? signal, the SENDER automaton will roll back the sending sliding window and retransmit the previous data packets from the sequence number i, Which means that the next_send data packet of the SENDER module must equal to the value of the NK_number. This property is formalized as P2 in Table 1.

In the same way, we can formalize two properties for the data acknowledgment process of the first requirement and three properties for the MVBC master rotation process of the second requirement. All these requirements and their formalization are presented in Table 3. For example, the property P6 means that in the MVB master and slave rotation process, there may be inconsistency such that two masters appear at the same time.

A.2 Formalization of Implementation Verifiable Requirement

These requirements related with dynamic runtime situation and uncertain environment are categorized as implementation verifiable requirements. For these safety requirements, they are not easy to be defined in the abstract timed automata level, and we use runtime verification technique and formalize the requirements based on the detail implementation code to verify the correctness.

Let us look at the two requirements below, which are described in the natural language in the original standard. These two requirements are not easy to be captured in model level, because it is not easy to model dynamic transmission delay of data on MVB bus and dynamic processing delay of hardware platform, even with a preliminary channel model and clock variable in Uppaal timed automata.

(1) ***Message Transmission(P8):*** The suggested time constraint on a slave MVBC between the finish of a master frame receiving and the start of a slave frame responding should be less than 4us.
(2) ***Message Transmission(P9):*** The suggested time constraint on a master MVBC between the finish of a master frame sending and the start of a response slave frame receiving should be less than 42.7us.

We formalize these two requirements as the runtime verification property presented in the Listing 1. We define some events based on the variables of the generated C code of Times, which are configured to I/O pins of the real hardware platform and will be continuously loaded by accompanied C functions. Then, the property and the accompanied C functions are transformed and input to RMOR to get the instrumented code, which can be made as an integral part of the target generated system, verifying and guiding its execution within the dynamic environment.

```
DataCenter Monitor TimeConstraints()
{
    P8 event TimeoutReply =
      ((T_Master_Receive-T_Slav_Send)<4)
    P9 event TimeoutResponse =
      ((T_Master_Send-T_Slav_Receive)<42.7)
    event Trigger =
       TimeoutReply || TimeoutResponse;
    state safe{
     When Trigger -> error;
    }
}
```

Listing 10. Runtime Property Definition for the Time Interval Between the Master and Slave Frames During the Message Transmission Process.

In this way, we can formally verify not only the requirements related with general function of control logic but also the requirements related to the dynamic physical part. Note that, the Uppaal model verification and RMOR runtime verification cooperate together to acquire a higher dependability confidence for safety critical systems. More specifically, we formalize 92 critical model verifiable requirements and 29 critical implementation verifiable requirements. The main criteria to distinguish between the two types of requirements is to find out whether the timed automata model has sufficient information for describing the requirement in TLTL format or the integrated code has sufficient information for describing the requirement in RMOR property format. The timed automata usually rely on the different level of abstraction(state and variables) and is not detailed enough to specify some requirements such as time delay restrictions on the physical bus. Which means that when all elements in the TLTL formula of the text-based requirements are described in the timed automata, it is the first type of requirement, otherwise, it is the second type of requirement. Besides, there are also several requirements that cannot be formalized such as the requirement of the industrial grade type of the MVBC hardware chip. These requirements need to be manually checked or formalized with additional information from additional sensors.

B CODE SYNTHESIS AND INTEGRATION

For the code synthesis, automatical code generation tools can be applied to reduce the hard work efforts of manual implementation, which is also more human error prone. For example, the engineers from the industrial sources (the Duagon company, the China CR corporation) report that their MVBC is developed by directly writing underlying C or VHDL code manually, where there are still some bugs such as dead logic. Besides, the automatical code generation of Simulink facilitate the traceability between the model and implementation, which results in better documentations and easier maintains.

Before applying the code generation algorithm, we need to do some isolations on the model. The isolation is needed because we construct and initialize the stateflow for two or more MVBCs for comprehensive verification, and now need the timed automata of a single MVBC for code synthesis. The single MVBC should receive and send packets onto the physical bus for communication with other MVBCs, and we use the event of Simulink Stateflow to simulate the communication for the packets of sending and receiving events.

One way for isolation is to build a general environment model, which is ready to receive any output event from the isolated MVBC and send input event to the isolated MVBC. Then, we can generate execution code for both MVBC and the general environment, and manually separate the generated code. Another way for isolation is to do some reverse engineering, where the declartion denoting the packets of sending and receiving events are reversed to the general variable. For example, the event declaration [rcv_connect_req] can be replaced by a declaration of Boolean variable rcv_connect_req. At the same time, an evaluation expression rcv_connect_req == true should be added to the guard segment, and an assignment expression rcv_connect_req := true should be added to the action segment. We use the second way, because it can be automatically accomplished by parsing and updating the XML file of the Stateflow model, and the second isolation way is more closed to the real operation scenario where the sending and receiving packets from the physical bus is asynchronous. Besides, because the generated code is tightly coupled, the manually separation code of the first way is more error prone.

After that, we also need to add some glue code, which is mainly used for two functionalities, the interface between the software and hardware platform, and timing implementation of the generated code on the hardware platform. For interface, we just need to initialize some configure mapping files, mapping the variable of software to the GPIO of the hardware platform. Accompanied type conversion functions may be needed. For clocks, let sc be a global system clock. For each timer x in the stateflow, let x_{reset} be an integer variable holding the system time of the last clock reset. The value of the clock is then $(sc - x_{reset})$, and a reset can be performed as $x_{reset} := sc$.

Finally, based on the generated code and the handwriting glue code, we can formalize the implementation verifiable requirements as presented in Section A. We input the formalized properties and the integrated code to RMOR to generate the runtime verifier, and the system integration is instrumented with the verifier for the runtime verification. The integrated verifier keeps verifying the requirements on the running executable system. To improve the dependability confidence, we can also formalize some model verifiable requirement into verifier, with the cost of increasing the storage overhead of the system.

REFERENCES

Syed Hassan Ahmed, Gwanghyeon Kim, and Dongkyun Kim. 2013. Cyber physical system: Architecture, applications and research challenges. In *Wireless Days*. 1–5.

Rajeev Alur. 1999. Timed automata. In *Proceedings of the International Conference on Computer Aided Verification*. 8–22.

Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. 2008. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *Proceedings of the ACM & IEEE International Conference on Embedded Software (EMSOFT'08)*. 89–98.

Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. 2004. Dependability and its threats: A taxonomy. In *Building the Information Society*. Springer, 91–120.

Algirdas Avizienis, J.-C. Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Depend. Sec. Comput.* 1, 1 (2004), 11–33.

Henning Basold, Henning Gnther, Michaela Huhn, and Stefan Milius. 2014. An open alternative for SMT-based verification of scade models. In *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*. 124–139.

Gerd Behrmann, Alexandre David, and Kim G. Larsen. 2004. A tutorial on uppaal. In *Formal Methods for the Design of Real-time Systems*. Springer, 200–236.

Grard Berry. 2007. Synchronous design and verification of critical embedded systems using SCADE and Esterel. In *Proceedings of the International Conference on Formal Methods for Industrial Critical Systems*. 2–2.

M. R. Blackburn and R. D. Busser. 1996. T-VEC: A tool for developing critical systems. In *Proceedings of the 11th Conference on Computer Assurance, Systems Integrity, Software Safety, and Process Security (Compass'96)*. 237–249.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *Proceedings of the International Conference on Concurrency Theory*. 135–150.

Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. 2003. From simulink to SCADE/lustre to TTA: A layered approach for distributed embedded applications. 153–162.

Chunqing Chen, Jun Sun, Yang Liu, Jin Song Dong, and Manchun Zheng. 2012. Formal modeling and validation of Stateflow diagrams. *Int. J. Softw. Tools Technol. Trans.* 14, 6 (2012), 653–671.

Feng Chen and Grigore Roşu. 2005. Java-MOP: A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 546–550.

Feng Chen and Grigore Roşu. 2007. Mop: An efficient and generic runtime verification framework. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 569–588.

Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. 2006. An expressive aspect language for system applications with Arachne. In *Transactions on Aspect-Oriented Software Development I*. Springer, 174–213.

Ambar A. Gadkari, Anand Yeolekar, J. Suresh, S. Ramesh, Swarup Mohalik, and K. C. Shashidhar. 2008. AutoMOTGen: Automatic model oriented test generator for embedded control systems. In *Proceedings of the International Conference on Computer Aided Verification*. 204–208.

S. Jeschke H. Song, D. Rawat and C. Brecher. 2016. *Cyber-Physical Systems: Foundations, Principles and Applications*.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 2002. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (2002), 1305–1320.

Klaus Havelund. 2008. Runtime verification of C programs. In *Testing of Software and Communicating Systems*. Springer, 7–22.

Yu Jiang, Han Liu, Hui Kong, Rui Wang, Mohammad Hosseini, Jiaguang Sun, and Lui Sha. 2016a. Use runtime verification to improve the quality of medical care practice. In *Proceedings of the International Conference on Software Engineering Companion*. 112–121.

Yu Jiang, Han Liu, Houbing Song, Hui Kong, Ming Gu, Jiaguang Sun, and Lui Sha. 2016b. Safety-assured formal model-driven design of the multifunction vehicle bus controller. In *Proceedings of the 21st International Symposium Formal Methods (FM 2016)*. Springer, 757–763.

Yu Jiang, Yixiao Yang, Han Liu, Hui Kong, Ming Gu, Jiaguang Sun, and Lui Sha. 2016c. From stateflow simulation to verified implementation: A verification approach and a real-time train controller design. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16)*. IEEE, 1–11.

Yu Jiang, Hehua Zhang, Zonghui Li, Yangdong Deng, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2015. Design and optimization of multiclocked embedded systems using formal techniques. *IEEE Trans. Industr. Electr.* 62, 2 (2015), 1270–1278.

Yu Jiang, Hehua Zhang, Xiaoyu Song, William N. N. Hung, Ming Gu, and Jiaguang Sun. 2013a. Verification and implementation of the protocol standard in train control system. In *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC'13)*. IEEE, 549–558.

Yu Jiang, Hehua Zhang, Xiaoyu Song, Xun Jiao, William N. N. Hung, Ming Gu, and Jiaguang Sun. 2013b. Bayesian-network-based reliability analysis of PLC systems. *IEEE Trans. Industr. Electron.* 60, 11 (2013), 5325–5336.

Yu Jiang, Hehua Zhang, Huafeng Zhang, Han Liu, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2015. Design of mixed synchronous/asynchronous systems with multiple clocks. *IEEE Trans. Parallel Distrib. Syst.* 26, 8 (2015), 2220–2232.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *Proceedings of the Conference on Object-Oriented Programming (ECOOP'01)*. Springer, 327–354.

François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. 2002. Temporal logic with forgettable past. In *Proceedings of the Symposium on Logic in Computer Science*. IEEE Computer Society, 383–383.

Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J Logic Algebr. Program.* 78, 5 (2009), 293–303.

The MathWorks. 2017a. SimulinkDesignVerifier. Retrieved from http://www.mathworks.com.
The MathWorks. 2017b. SimulinkPolySpace. Retrieved from http://www.mathworks.com.
The MathWorks. 2017c. Stateflow user guide. Retrieved from http://www.mathworks.com.
Kenneth L. Mcmillan. 1993. *The SMV System*. Springer US. 161–165.
Amjad Mehmood, Syed Hassan Ahmed, and Mahasweta Sarkar. 2017. *Cyber-Physical Systems in Vehicular Communications*.
Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. 2012. From verification to implementation: A model translation tool and a pacemaker case study. In *Proceedings of the 2012 IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*. IEEE, 173–184.
Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. 2014. Safety-critical medical device development using the UPP2SF model translation tool. *ACM Trans. Embed. Comput. Syst.* 13, 4s (2014), 127.
Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. 2008. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the Real-Time Systems Symposium 2008*. IEEE, 481–491.
J. Qian, J. Liu, X. Chen, and J. Sun. 2015. Modeling and verification of zone controller: The SCADE experience in china's railway systems. In *Proceedings of the 2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS'15)*. 48–54. DOI: http://dx.doi.org/10.1109/COUFLESS.2015.15
C. Schifers and G. Hans. 2000. IEC 61375-1 and UIC 556-international standards for train communication. In *Proceedings of the IEEE 51st Vehicular Technology Conference Proceedings (VTC'00)*, Vol. 2. 1581–1585. DOI: http://dx.doi.org/10.1109/VETECS.2000.851393
Lui Sha. 2001. Using simplicity to control complexity. *Softw. IEEE* 18, 4 (2001), 20–28.
Steve Sims and Daniel C. DuVarney. 2007. Experience report: The reactis validation tool. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 137–140.
Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *Proceedings of the International Conference on Computer Aided Verification*. 709–714.
Tester. 2017. Applied dynamics international. Retrieved from http://www.adi.com.
Heini Wernli, Marcus Paulat, Martin Hagen, and Christoph Frei. 2007. SALA novel quality measure for the verification of quantitative precipitation forecasts. *Month. Weather Rev.* 136, 11 (2007), 4470–4487.
Yixiao Yang, Yu Jiang, Ming Gu, and Jiaguang Sun. 2016. Verifying simulink stateflow model: Timed automata approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 852–857.
Yu. 2017. Uiuc. Retrieved from https://sites.google.com/site/jiangyu198964/home.

Received January 2017; revised March 2017; accepted March 2017