From System Modeling To Formal Verification

Ajay Chhokra, Sherif Abdelwahed, Abhishek Dubey, Sandeep Neema, Gabor Karsai*
* Institute for Software-Integrated Systems, Vanderbilt University,
Nashville, TN 37235, USA
Email:{chhokraad, sherif, dabhishe, sandeep, gabor}@isis.vanderbilt.edu

***Abstract*—Due to increasing design complexity, modern systems are modeled at a high level of abstraction. SystemC is widely accepted as a system level language for modeling complex embedded systems. Verification of these SystemC designs nullifies the chances of error propagation down to the hardware. Due to lack of formal semantics of SystemC, the verification of such designs is done mostly in an unsystematic manner. This paper provides a new modeling environment that enables the designer to simulate and formally verify the designs by generating SystemC code. The generated SystemC code is automatically translated to timed automata for formal analysis.**

***Keywords*—Generic Modeling Environment, SystemC, UPPAAL, Formal Verification, Cyber Physical Systems**

I. INTRODUCTION

A cyber physical system is a heterogeneous composition of a number of components which have continuous or discrete dynamics. Due to the increasing complexity of the system design, modern systems are modeled at a high level of abstraction. Thus a language that fills the gaps between hardware descriptive languages and software modeling languages is required. Moreover, it should support design space exploration and assure correctness throughout the designing process. SystemC [1] is such a system level language that allows user to model and simulate hardware and software on various abstraction levels. The complexity in the designs has shifted the focus to software modeling languages like UML for providing initial specification of the system at high levels of abstraction on top of SystemC. Generating SystemC code out of these abstract UML diagrams has become common practice. There are few tools [2],[3],[4]which generate SystemC code for a given UML specification.

Safety critical systems, that are found in the automotive industry or in power systems, are a subclass of cyber physical systems where a small glitch may prove to be fatal or lead to huge financial loses. Therefore, functional verification of these designs becomes an important part of the development cycle. Simulation alone cannot ensure the correctness of a design. For non-deterministic or non-terminating systems, covering all possible executions cannot be guaranteed by simulations. Moreover, hardware and software designs are refined to required specification starting from an abstract model. Thus, it becomes very hard to ensure consistency between different stages by reusing simulation results of earlier stages. So, formal verification becomes a necessity to ensure correct behavior.

Automatic verification of the SystemC designs is constrained by the lack of formal semantics of SystemC. There are a number of approaches that address this problem by translating SystemC to a well defined language. For instance, Maraninchi [5] discusses about translating SystemC to PROMELA. But this approach lacks support for primitive channels and does not model SystemC scheduler. The approach described by Habibi and Tahar [6] translates the SystemC designs into an intermediate representation using AsmL [7]. It is an abstract state machine [8] descriptive language that models designs at a high level of abstraction. This makes verification and validation easier but underlying structure of SystemC design is lost. Zhang [9] coins a formalism called waiting state automata. This approach allows verifying SystemC designs up to delta cycles. But this technique also does not model the scheduler and complex interactions between processes. Man [10] discusses a formal language $SystemC^{FL}$ based on process algebras. This language considers only static sensitivity of processes and simple communications amongst them. In [11], the SystemC designs are translated to petri nets based representations. However, this translation leads to huge overheads as extra subnets are required to model the interactions between existing subnets. We are adopting an approach presented in [12],[13] by Herber, which translates the SystemC design to Uppaal timed automata [14]. This technique handles all relevant SystemC language elements, including process execution, interactions between processes, dynamic sensitivity and timing behavior. The mapping produces negligible overhead and compact models.

The contribution of this paper is to provide a modeling environment based on Generic Modeling Environment (GME) [15] that allows the designer to graphically model the system and generates simulation results in the form of vcd waveforms at each step of development cycle. The tool also provides automatic verification of the design against safety and liveness properties by translating the SystemC code to Uppaal. In this paper, we show the applicability of this tool by describing a case study from power systems domain for formally analyzing the behavior of different protection elements.

The rest of the paper is structured as follows: section II describes the modeling formalism that captures the different aspects of SystemC language. Section III discusses about the generation of SystemC code. Section IV briefly describes the STATE tool [13] and process of converting SystemC code to timed automata. Section V explains the implementation followed by a case study in section VI. Model checking and simulation results for the case study are summarized in section VII. Section VIII documents the conclusion and future work.

II. MODELING PARADIGM

The modeling paradigm is created using generic modeling environment (GME) which is a configurable tool set

Copyright ©2015 ECSI

for creating domain specific modeling and program synthesis environments. GME has all the generic modeling concepts like hierarchy, aspects, constraints, associations, generalization etc. The objects used in the modeling language are Atoms, Models, Connection, References and Folder, for more information see [15]. Fig. 1 captures the complete meta model of this language. The following subsections discuss the core features of SystemC and its implementation in this modeling paradigm.

A. Data Types

Apart form the standard data types which C++ provides, SystemC supports dedicated hardware data types. In our modeling language data types are represented by two atoms *c_types* and *systemc_types*. Both these types inherits from an abstract atom called *keyword* as shown in the Fig. 1.

B. Modules

From the structural aspect, a SystemC design is composed of modules and channels. Modules are the building blocks which represent the computation part of the design. Modules in general, are composed of processes, ports, events, internal channels and variables. These variables can also be instances of other modules. However, in this current implementation we do not consider hierarchical designs and hence channels and objects implying instances of other modules are not visible inside a module. As per Fig. 1, a model labeled as *module* represents SystemC module which encapsulates *ports*, *variables*, *events* and a *behavior* model (GME). Ports and variables are implemented as references to the *keyword* atom. Events are represented with atoms labeled *event*. The *behavior* model is a container for all the processes that provides functionality to a module.

C. Processes

As mentioned earlier, processes are contained inside a module. Processes can be classified into two categories 1) Threads 2) Methods. Method processes are triggered by a list of events statically bound to them. A method runs from beginning to end when triggered and cannot be suspended. On the other hand threads can be paused i.e. their sensitivity can be dynamically altered by using wait function calls. Threads are also of two types one is named as SC_THREAD and other as SC_CTHREAD (clocked). But for this paper we only consider SC_THREAD. As shown in the Fig. 1, model with the name *process*, represents SystemC processes contained inside *behavior* model. Event triggered extended state machine is used as the model of computation to represent the actual behavior of the process. As shown in the Fig. 1, *process* contains an atom and a connection labeled as *state*, *transition* respectively. One *state* object can be connected to another through *transition* connection object. *transition* has a boolean attribute, *Guard* which enables or disables a *transition*. The *state* and *transition* have a common attribute called *Action* which are expressions involving ports, variables, events defined in the module. The attribute *sensitivity_list* associated to a process represents the list of events to which the process is sensitive to. The type of the process is governed by another attribute *Process_type* that has only two values SC_THREAD or SC_METHOD.

D. Channels and Ports

A channel is a special c++ class which implements one or more interfaces. Interfaces are abstract classes which declare a set of methods for accessing a given channel. SystemC supports two types of channels 1) primitive and 2) hierarchal. In this paper we are only modeling one type of primitive channel, signal [1] but this can be easily extended to support other primitive channels. Processes of a module communicate with the help of ports. As shown in Fig. 1, ports and channels are implemented as references with labels *port* and *sc_signal* to *keywords*. *port* has an attribute *type* to identify the nature of the port i.e. in, out or inout.

E. Binding

As shown in the Fig. 1, a GME model *system* is used to show all the bindings between two different instances of modules. Instances of modules are represented by a reference, *sc_module* to *modules*. Apart from *sc_module* references, *system* also consists of *sc_signal* which represents an instance of a primitive channel. In order to connect a ports of the instantiated modules and channels, a connection object labeled as *binding* is used. Each design is contained inside a GME folder to better manage different design files.

III. SystemC Code Generation

Each folder in the modeling paradigm represents a design that contains two types of files. One, a header file corresponding to each module used in the design. The header file gives the definition of a module in terms of ports, variables and processes. Second, is the source file which contains the main function. It includes the declaration of signals, instances of modules and bindings between ports of different instances of modules followed by trace objects. The subsection below briefly describes the steps taken to translate the GME model into c++ code.

A. Generation of Module Definitions

- For each *port* reference in the module, a port is declared in the header file by getting the value of attribute *Port_type*, source and the label of the *port* reference.
- The variables declared inside a module can be of two types, one being accessible to all processes and other being local to a specific process. For each *variable* reference, a variable is declared whose type and name are defined by the source and the label associated to the reference object. As behavior of the processes is described using state machines, a variable is required to store the state of the automata. We use an enumerated type variable whose elements are the labels of different states in the automata.
- Since modules are c++ classes so we need to define a constructor which initializes the data member and also describes the static sensitivity of the processes. Two different Macros are used for defining a constructor depending on the number of input arguments of the constructor. If there is no input argument except the name/label of the module then we use SC_MODULE macro otherwise

[1]Since signal channel is considered, we restrict our discussion only to specialized ports (sc_in, sc_out and sc_inout) only.

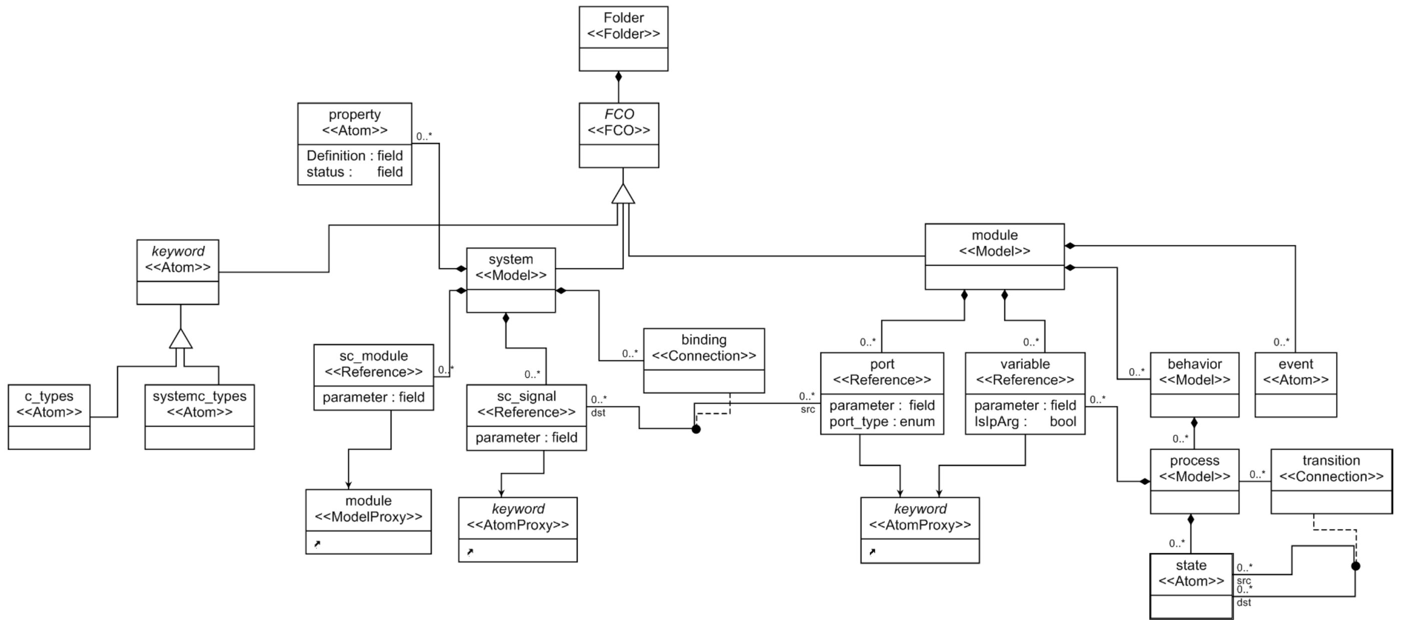

Fig. 1. Meta model

SC_HAS_A_PROCESS macro serves the purpose. Every process has an attribute called the sensitivity_list which lists the ports that can trigger a process. The list may contain more than one ports.

B. Source File Generation

- For each *sc_signal* reference in the *system* model, a sc_signal is declared as *sc_signal <source of reference> reference label ;*
- For each *sc_module* reference in the *system* model, a module is instantiated. The type of the module is the source of reference and the attribute *parameter* provides the list of input arguments that are needed.
- Every *binding* connection is translated to a port binding statement. The source link of the connection provides the specific port of the module and the destination link provides the channel to which the port is bound to.
- In the end a pointer to a sc_trace is created. This pointer is responsible for storing all the traces. By default all the signals will be automatically recorded.

IV. SystemC To Uppaal Translation

The important prerequisite to verify a design expressed in any language is the formal semantics of that language. As stated earlier, SystemC in not a well defined language, i.e. it lacks formal semantics. In this section, we briefly explain the STATE tool which maps the informally defined SystemC code into well defined semantics of Uppaal timed automata. STATE tool takes as input a SystemC design and generates a corresponding Uppaal model. This whole translation takes place in two steps:

1) A SystemC design is parsed by Karlsruhe Parser [16] to generate an Abstract Syntax Tree (AST) in XML.
2) This file serves as an input to the STATE tool to generate Uppaal model in another XML file. Within the STATE tool, translation from AST to Uppaal model takes place in two phases:
   - In the first phase, Uppaal model is created from the AST by the transformation engine.
   - In the second phase, the model obtained in the first phase is optimized for reduction in number of templates and variables etc.

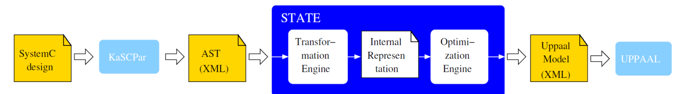

Fig. 2. Tool chain form SystemC to Uppaal [13]

The implementation of STATE tool consists of three packages: 1) *model* 2) *engine* 3) *optimization*

The package *model* consists of classes for the internal representation of both the SystemC and the timed automata model. The package *engine* contains classes that implement the transformation rules from SystemC to Uppaal. The package *optimization* contains several optimizations to make the generated Uppaal model smaller and easier to read. Fig. 2 shows the block diagram of the tool-chain.

Since SystemC is an extension of c++, it inherits the full semantic scale of c++. In order to translate it to a more restricted/ less expressive Uppaal modeling language, certain constraints are imposed on SystemC designs that are listed in [13].

V. Setup

Simulation and verification can be computationally expensive and time consuming for complex designs. Setting up nodes dedicated for these two tasks, aids in the developing process. The modeling environment is installed in a physical node and

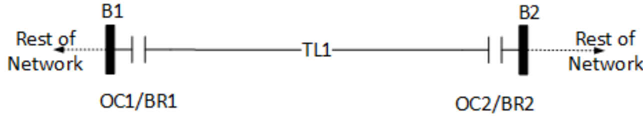

Fig. 3. A segment of power transmission system

two virtual nodes are reserved for simulating and verifying the design. The configurations are summarized in the Table I. The GME tool generates set of header files, source file, simulation and verification scripts and sends these files over SCP to simulation and verification machines. The simulation and verification results are sent back to the physical node in the form of vcd waveforms and text file labeled as log.txt. The text file contains the status of all the properties and a counter trace if any of the property fails to satisfy.

TABLE I. SPECIFICATIONS

Node	Nature	CPU Cores	Memory	OS
GME Host	Physical	8	16 GB	Windows 7
Verification	Virtual	4	8 GB	Ubuntu 14.10
Simulation	Virtual	4	8 GB	Ubuntu 14.10

VI. CASE STUDY

Electric power systems include a complex network of cyber-physical components. The physical components of a power system can be broadly classified as Generators, Loads, Lines, Transformers, Buses, and Protection devices. The dynamics of these components are governed by the laws of physics. The cyber components are related to the Protection devices as well as the supervisory control and data acquisition (SCADA) systems. These electric power systems are almost always operational and are subjected to dynamic, operational and environmental constraints. The stress and strain introduced due to changing loads and operational requirements contribute towards the degradation and eventual failure of physical components are thereby affecting the nominal operation of these systems. The protection devices are designed to detect any abnormal operation, and isolate the faulty components, thereby arresting the failure propagation and protecting the healthy components. However, mis-operation and failures in these protection devices, can lead to cascading effects that can bring down the power-supply for a large network i.e. blackouts [17]. Therefore, it is very important to formally analyze the behavior of these devices.

In order to analyze the behavior of the cyber components, the hybrid dynamics of the circuit has been converted to a discrete system. Consider a section of the transmission line as shown in the Fig. 3. Transmission line TL1 is connected to buses B1 and B2 through three phase breakers. These breakers BR1 and BR2 are controlled by over current relays OC1 and OC2. The individual modules are briefly explained as follows:

A. TwoBusSystem

This module models the different faults associated to the transmission line TL1 and their effects. Whenever a fault is present in the transmission line, the fault current increases and the load current decreases. The fault currents are represented

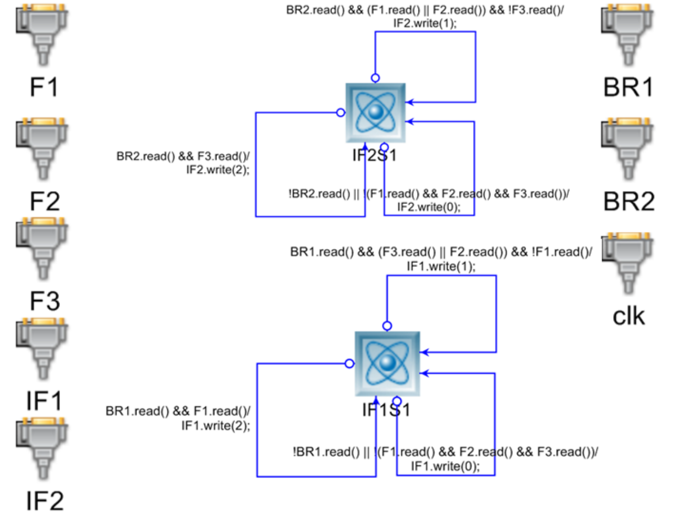

Fig. 4. TwoBusSystem model

with signals sigIF1 and sigIF2. These signals are discrete in nature and can take only three values: 0 (absent), 1 (low) and 2 (high). The value of these signals is updated by two processes in the module through ports IF1 and IF2. Both the processes are sensitive to a high frequency clock signal at port clk. The ports IF1(IF2) is connected to port I of the over current relay OC1(OC2). The value of fault current is influenced by two quantities: 1) location of fault and status of breakers. If the fault is close to the left end of the transmission line (close to OC1) then fault current sigIF1 will be high and sigIF2 will be low. So in order to cover all the cases, transmission line is divided into three [2] segments. The location of fault is represented by the signals at ports F1, F2 and F3 (corresponding to three segments). The status of breakers is represented by the signals at ports BR1 and BR2. Fig. 4 shows the ports and behavioral model of TwoBusSystem module.

B. Directional Timed Over Current Relay

Fig. 5 shows the highly abstracted model of directional timed over current relay. The over current relay is a simple device which samples the continuous current and compares the computed phasers to a threshold in order to detect the presence of a fault. If the fault is detected a trip signal is sent to the breaker.

The timed over current relay has two thresholds TH1 and TH2 (TH1>TH2), if current crosses TH1 a trip command to sent immediately and if the current exceeds only the lower threshold, TH2 then over current relay waits for some time before sending the trip signal. As mentioned in previous section, sigIF1/sigIF2 that represents fault current is connected to port I of relay. The behavioral model contains a process OverCurrentP1 that reads the port I at fixed rate and decides to issue a trip command or not. The relay OC1(OC2) controls the breaker BR1(BR2) through signals TripDR1(TripDR2) from port BR. The process OverCurrentP1 has following states:

[2] first segment: sigIF1>sigIF2; second segment: sigIF1=sigIF2; third segment= sigIF1<sigIF2

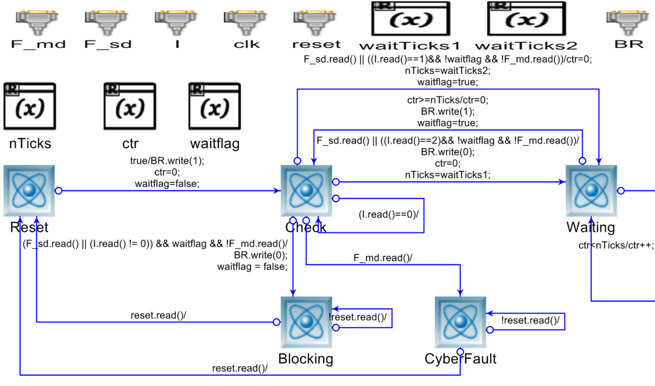

Fig. 5. Directional timed over current relay model

- **Reset**: This is the initial state of the system. This state implies the relay has been manually reset by the operator. The state machine will enter this state when there is an active high signal at reset port.
- **Check**: In this state, machine checks the value of the signal at port I. If the value of I is 2, state machine sets the value at port BR to be 0 and enters the Wait state. If the value is 1, it shifts to Wait state but does not issue any command to breaker.
- **Wait**: This state implies the presence of fault in the transmission line. The state machine moves back to Check state in order to check the status of fault after a certain period of time identified by the variable nTicks.
- **Blocking**: This state implies the fault has not been cleared in the interval defined by the variable nTicks and operator should manually reset the relay by sending an active high signal at port reset.
- **CyberFault**: This state implies a cyber-fault (false negative) has occurred. This fault can be induced in the relay by issuing a high signal at port F_md. This fault disables the relay to detect any fault by forcing to shift from Check state to CyberFault state. A manual reset signal from the operator is required to move out of this state. There is one more failure mode present in the relays which gets activated by high signal at port F_sd. This fault forces the relay to incorrectly send a command to breaker even when there is no fault in the system (false positive).

C. Three Phase Breaker

Fig. 6 shows a simplified model of a breaker. It has 4 input ports and 1 output port labeled as *clk, F_SO, F_SC, cmd,* and *BR_Status*. The model includes one process which is sensitive to changes in port clk. F_SO and F_SC ports are used to induce faults in the breaker. The breaker has two states: **Open** and **Close**. It reacts to the commands sent by the relay at cmd port and changes its state accordingly. An active high (low) signal on the cmd port instructs the breaker to close the circuit (open). However, presence of fault in the breaker does not let the breaker to act normally. An active high signal at the port F_SO port forces the breaker to remain stuck in open state and will ignore all the commands from the relay. Similarly, active high signal at F_SC forces the breaker to remain in the close state.

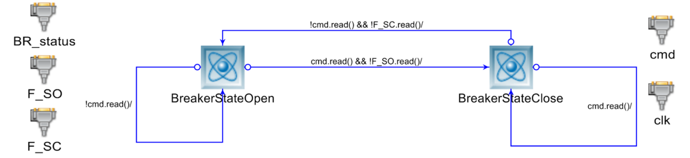

Fig. 6. Three Phase Breaker model

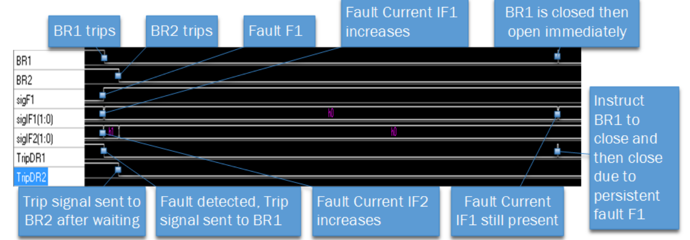

Fig. 7. Case 1: Persistent Fault F1

D. Fault Generator

This module induces faults in above mentioned modules. It has only one input port clk and 10 output ports. It contains two processes P1 and P2 both sensitive to signal at port clk. Process P1 triggers physical fault in the TwoBusSystem module and P2 activates cyber faults in relays and breakers modules.

Using these modules and the topology shown in the Fig. 3, we'll simulate the behavior of the relays and breakers for two different fault scenarios followed by verification of model of relays against liveness and safety properties.

VII. Results

A. Simulation Results

1) **Persistent Physical fault F1:** A fault F1 is introduced in TL1. This fault forces SampleTwoBusCircuit module to update sigIF1 and sigIF2 to 2, 1 respectively. Due to this fault, OverCurrentRelay1 instructs Breaker1 to open instantaneously and shifts to wait state for some time. After some time, relay again checks the status of the fault by sending a close command. As the fault is permanent, it instructs the breaker to open again and moves to blocking state. The steps taken by the relay OverCurrentRelay2 are different as the value of signal at port I is 1. The relay after detecting the fault moves to the wait state without sending any trip signal to the Breaker2. After some time, the relay checks the status of the fault. As the fault is still present it moves to blocking state and instructs the breaker to open. Fig. 7 shows the status of breakers and commands sent by the relays.
2) **Persistent Cyber Fault and Physical Fault F1:** In addition to F1, a cyber fault F_md is induced in the relay OverCurrentRelay2. Because of this fault the relay fails to detect any physical faults. As a result, fault F1 is not detected. OverCurrentRelay1 behaves normally while the other relay does not detect anything at all as shown in the Fig. 8.

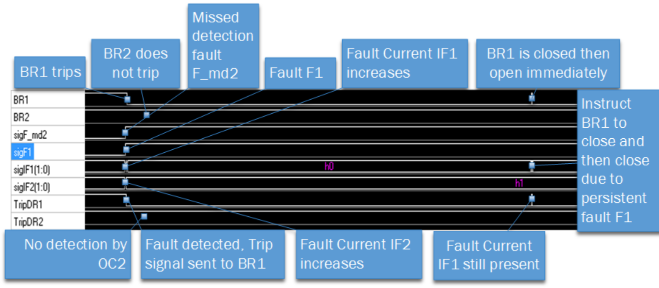

Fig. 8. Case 2: Persistent fault F1 and missed detection fault F_md2

TABLE II. Computational effort of the transformation

Parse time (ms)	240
Translation time (ms)	65
Composition time (ms)	65
Optimization time (ms)	752

B. Model Checking Results

Table II shows the time taken during translating SystemC code to Uppaal timed automata templates and the table III summarizes the number of templates, variables, channels and clocks are created in the translated Uppaal model.

TABLE III. Uppaal model

Templates	27
Binary Channels	132
Broadcast Channels	69
Integer Variables	263
Boolean Variables	11
Clock variables	2

The system under test is verified against following liveness and safety properties.

1) $A[]!deadlock$
   - This property ensures there is no deadlock in the system.
2) $((sigF1\$val||sigF2\$val1||sigF3\$val)\&\&(!sigF_md1\$val)) --> TripDR1\$val$
   - This property ensures if the fault is detected by the relay OC1 it must issue a trip signal after some time.
3) $((sigF1\$val||sigF2\$val1||sigF3\$val)\&\&(!sigF_md2\$val)) --> TripDR2\$val$
   - This property ensures if the fault is detected by the relay OC2 it must issue a trip signal after some time.

The table IV lists the results of Uppaal model checker along with the time taken, and memory consumed during each property verification.

TABLE IV. Computational effort of verification

Property	*Virtual memory peaks*	*Verification time*	*Verdict*
1	2,582,440KB	1225.429s	satisfied
2	2,938,772KB	1213.541s	satisfied
3	2,938,772KB	1210.39s	satisfied

VIII. Conclusion

In this paper, we presented a modeling environment that aids the designer by graphically modeling both the structure and behavior of different components of a design. The tool allows the verification and simulation of each step of the design flow. In future, we'll add more features of SystemC language to the modeling paradigm like hierarchal channels and transaction level models. We'll increase the scope of this tool by allowing code generation for hardware descriptive languages like VHDL or SystemVerilog.

Acknowledgment

This work is funded in part by the National Science Foundation under the award number CNS-1329803. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

References

[1] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.

[2] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh, "Yaml: a tool for hardware design visualization and capture," in *Proceedings of the 13th international symposium on System synthesis*. IEEE Computer Society, 2000, pp. 9–14.

[3] C. Xi, L. JianHua, Z. ZuCheng, and S. YaoHui, "Modeling systemc design in uml and automatic code generation," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. ACM, 2005, pp. 932–935.

[4] B. A. Correa, J. F. Eusse, D. Múnera, S. Sepúlveda, J. F. Vélez, and J. E. Aedo, "Uml2sc: A tool for developing complex electronic systems using uml and systemc," *Revista Facultad de Ingeniería Universidad de Antioquia*, no. 48, pp. 165–173, 2009.

[5] F. Maraninchi, "A systemc/tlm semantics in promela and its possible applications," *month*, 2007.

[6] A. Habibi and S. Tahar, "An approach for the verification of systemc designs using asml," in *Automated Technology for Verification and Analysis*. Springer, 2005, pp. 69–83.

[7] Y. Gurevich, B. Rossman, and W. Schulte, "Semantic essence of asml," *Theoretical Computer Science*, vol. 343, no. 3, pp. 370–412, 2005.

[8] E. Börger and R. F. Stärk, *Abstract State Machines: A Method for High-level System Design and Analysis; with 19 Tables*. Springer Science & Business Media, 2003.

[9] Y. Zhang, F. Védrine, and B. Monsuez, "Systemc waiting-state automata," in *First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007)*, 2007, p. 56.

[10] K. L. Man, "An overview of systemc fl," in *Research in Microelectronics and Electronics, 2005 PhD*, vol. 1. IEEE, 2005, pp. 145–148.

[11] D. Karlsson, P. Eles, and Z. Peng, "Formal verification of systemc designs using a petri-net based representation," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. European Design and Automation Association, 2006, pp. 1228–1233.

[12] P. Herber, J. Fellmuth, and S. Glesner, "Model checking systemc designs using timed automata," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. ACM, 2008, pp. 131–136.

[13] P. Herber, *A Framework for Automated HW/SW Co-Verification of SystemC Designs using Timed Automata*. Logos Verlag Berlin GmbH, 2010.

[14] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236.

[15] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary*, vol. 17, 2001.

[16] F. Karlsruhe, "Kascpar-karlsruhe systemc parser suite, 2012."

[17] K. Yamashita, J. Li, P. Zhang, and C.-C. Liu, "Analysis and control of major blackout events," in *Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES*. IEEE, 2009, pp. 1–4.