Translation-Based Co-Verification

Fei Xie
Dept. of Computer Science
Portland State University
Portland, OR 97207, USA
xie@cs.pdx.edu

Xiaoyu Song, Haera Chung, and Ranajoy Nandi
Dept. of Electrical and Computer Engineering
Portland State University
Portland, OR 97207, USA
{song, hrchung, rnandi}@ece.pdx.edu

Abstract

We propose a translation-based approach to hardware and software co-verification of embedded systems using model checking. Software and hardware designs of an embedded system are translated into the input formal language of a state-of-the-art model checker to enable co-verification. The formal model of the whole system is constructed through integrating the translations of hardware and software designs via a bridge module. The bridge module preserves the semantics of hardware and software. Co-verification complexity is reduced through (1) leveraging reduction algorithms of the target model checkers, (2) applying reduction algorithms in translation via model transformations, and (3) conducting compositional reasoning across the interfaces of the bridge module. Our approach has been implemented to support co-verification of software designs specified in Executable UML and hardware designs specified in Verilog. We have successfully applied this approach to co-verification of networked sensors, an emerging type of embedded systems. The case study has shown that our approach is practical - applicable to embedded systems of real-world scale, and effective - leading to order-of-magnitude reduction on co-verification complexities.

1 Introduction

Embedded systems have become indispensable to the infrastructure of our society, therefore, they must be safe, secure, and reliable. Advanced validation and verification methods are in demand for improving safety, security, and reliability of embedded systems. Model checking [7, 22] is a powerful formal method, which has great potentials in verification of embedded systems. Embedded systems include both hardware and software modules, hence, successful application of model checking to embedded systems requires co-verification of hardware and software, i.e., examining a whole system including its hardware and software modules.

Several major challenges hinder practical application of model checking to co-verification of embedded systems:

- *Applicability of model checkers.* The syntax and semantics of the input languages of model checkers often differ significantly from those of hardware and software specification languages, which prevents straightforward application of the model checkers. However, it is highly desirable to reuse the existing model checkers since it is time and resource consuming to build language-specific model checkers and to reproduce the powerful state space reduction algorithms of the existing model checkers to the same level of efficiency.

- *Variety of hardware/software specification languages.* There exist a large variety of hardware and software specification languages. It is highly desirable that model checking can be included into the system design and validation process without requiring designers to change their favorite system specification languages.

- *Semantics gap between hardware and software.* There exist major semantic gaps between hardware languages such as Verilog and software languages such as Executable UML (xUML) [18]. For instance, Verilog has a clock-driven synchronous semantics while xUML has an asynchronous interleaving message-passing semantics. Such a semantics gap must be properly bridged to facilitate meaningful co-verification.

- *State space explosion problems.* Since both hardware and software modules are included for co-verification, the state space complexity of a whole system can be very large. State space reduction algorithms must work across hardware/software interfaces and explore the structural characteristics of embedded systems.

In this paper, we propose a translation-based approach to co-verification of embedded systems using model checking, which contributes to solution of the above challenges.

In this approach, hardware and software modules of an embedded system are automatically translated into the input language of a state-of-the-art model checker. The hardware and software modules can be specified in various languages and are translated by language-specific translators. The semantics of hardware and software specification languages are simulated by the semantics of the target formal languages. The language-specific translators for hardware (or software, respectively) specification languages share common intermediate representations since they share common features. Our translation-based approach has the following advantages: (1) leveraging state-of-the-art model checkers; (2) easy extension to support emerging hardware and software specification languages; (3) reuse of translator construction efforts via shared intermediate representations.

Upon successful translation of hardware and software modules, two challenges still exist for co-verification:

- Interfacing of the resulting formal models of the hardware and software modules;
- Reduction of the state space complexity of the integrated formal model.

We interface the formal models of hardware and software modules through inserting a *bridge module* that is specified in the target formal language and bridges the gap between the hardware and software semantics. The bridge module interacts with the hardware and software modules following the hardware and software semantics, respectively. It propagates events across the hardware/software interface, for instance, generating software messages or invoking procedures upon hardware interrupts and producing hardware signals upon value changes in software variables. We reduce co-verification complexities by (1) leveraging state space reduction algorithms of the target model checkers, (2) applying reduction algorithms in translation and preserving validity of the reductions when interfacing the formal models of hardware and software modules, and (3) conducting compositional reasoning [1] across the bridge module.

The rest of this paper is organized as follows. Section 2 provides the background of our work. Section 3 presents the details of our translation-based approach to co-verification. Section 4 illustrates our approach with a case study on networked sensors. Section 5 discusses the related work. Section 6 concludes and presents the future work.

2 Background

In development of our translation-based approach to co-verification, we select COSPAN [10] as the target model checking engine. However, our approach does not depend on COSPAN and can be readily re-targeted to other model checkers such as SMV [17]. Our work integrates two translation-based model checkers, FormalCheck [14] and ObjectCheck [26], both of which are based on COSPAN. FormalCheck is commercially available for hardware verification while ObjectCheck was developed in our previous work for verification of executable software designs.

2.1 COSPAN model checker

The COSPAN model checker implements the automata-theoretic approach [13] to model checking. In this approach, a system is modeled by an automaton P and a property to be checked is modeled by an automaton T. The verification consists of checking whether the language of P is contained in the language of T, $\mathcal{L}(P) \subset \mathcal{L}(T)$, known as the language containment test. Typically, P is not monolithic, but is represented as a synchronous parallel composition $P = P_1 \otimes \ldots \otimes P_k$ of component processes all modeled as automata. COSPAN checks language containment by either an explicit state space enumeration algorithm or a BDD-based symbolic search algorithm.

COSPAN inputs the S/R [10] automaton language. In S/R, a system is composed of synchronously interacting processes (or automata). A process consists of state variables, selection variables, inputs, state transition rules, and selection rules. Selection variables define the outputs of the process. Each process inputs a subset of all the selection variables of other processes. State transition rules update state variables and are functions of the current state, selection variables, and inputs. Selection rules assign values to selection variables as functions of state variables. Such a function is non-deterministic if several values are possible for a selection variable in a state. The "selection/resolution" execution model of S/R is clock-driven, synchronous, and parallel, under which a system of processes behaves in a two-phase procedure every logical clock cycle:

- [1: Selection Phase] Every process "selects" a value possible in its current state for each of its selection variables. The values of the selection variables of all the processes form the global selection of the system.
- [2: Resolution Phase] Every process "resolves" the current global selection simultaneously by updating its state variables upon enabled state transition rules.

COSPAN implements a suite of powerful state space reduction algorithms including localization reduction and user-defined homomorphic reduction [13]. COSPAN applies these state space reduction algorithms by transforming a given S/R model into a semantically equivalent one with a reduced state space, with respect to a property or a set of properties. Other state space reduction algorithms, such as predicate abstraction [9], partial order reduction [21] and assume-guarantee style [1] of compositional reasoning, have been implemented as transformations to an S/R model before the model is checked by COSPAN.

2.2 FormalCheck

FormalCheck is a toolkit from Cadence Design Systems, which supports model checking of hardware system designs specified in Verilog or VHDL. FormalCheck is essentially translation-based and its verification engine is COSPAN. FormalCheck provides a property specification interface which allows designers to specify properties to be checked on a hardware design by instantiating a set of property templates with semantic entities from the hardware designs. Given a hardware design specified in Verilog or VHDL and a property to be verified on the design, both the design and the property are automatically translated into S/R. The semantics of Verilog and VHDL are simulated with the semantics of S/R. The resulting S/R property is checked on the S/R model by COSPAN. Upon detection of a violation of the property, a waveform that captures a system execution that violates the property is presented to the designers.

Verification complexity reduction in FormalCheck relies on both the automatic reduction algorithms of COSPAN such as localization reduction and the user-guided reduction algorithms such as homomorphic reduction and compositional reasoning. FormalCheck provides user interfaces that support specification of reduction directives, for instance, how to decompose a system design and its properties.

2.3 ObjectCheck

ObjectCheck provides comprehensive support to model checking of executable software system designs in xUML and is based on automatic translation. The architecture of ObjectCheck is shown in Figure 1. ObjectCheck supports

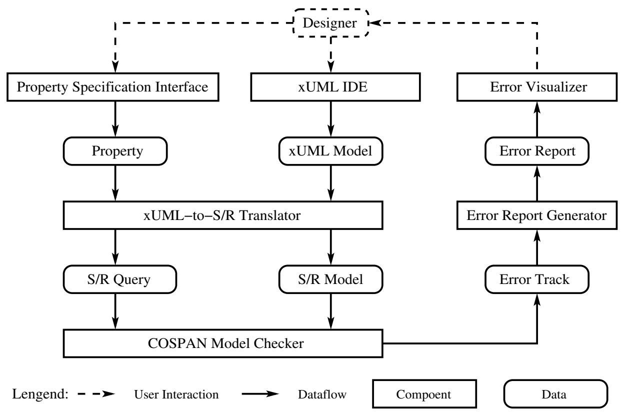

Figure 1. ObjectCheck architecture

xUML level property specification, xUML-to-S/R translation, error report generation, and error visualization.

xUML features an asynchronous interleaving message-passing semantics. In xUML, a system consists of a set of object instances, which communicate via asynchronous message-passing. The behavior of each object instance is specified by an extended Moore state model in which each state may be associated with a state action. A state action is a program segment that executes upon entry to the state. In a system execution, at any given moment only one object instance progresses by executing a state transition or a state action in its extended Moore state model.

The asynchronous interleaving execution of an xUML system is simulated by the synchronous parallel execution of its corresponding S/R system. Each object instance in the xUML system is mapped to an automaton in the S/R system. An additional automaton, *scheduler*, is introduced to enforce the interleaving execution of the automata corresponding to xUML object instances. The asynchronous message-passing of xUML is simulated by synchronous variable-sharing of S/R via modeling the message queue of an object instance as a separate S/R automaton.

In ObjectCheck, state space reduction algorithms are applied in two phases: in translation and in verification. There are reduction algorithms such as partial order reduction, which are very effective for asynchronous semantics, but not supported in COSPAN. These algorithms may be applied to a software design in its translation, for instance, a static version [16] of partial order reduction, is applied in the xUML-to-S/R translation. The reduced S/R model is further reduced by the reduction algorithms of COSPAN.

3 Translation-Based Co-Verification

We first introduce an abstract architecture for embedded systems. We then present a framework for translation-based co-verification of embedded systems that conform to this architecture. (The framework is not limited to embedded systems of this architecture and can be readily extended to other types of embedded systems.) After that, we discuss how to interface hardware and software modules as we integrate FormalCheck and ObjectCheck for co-verification following the framework. Finally, we explore how to scale co-verification by reducing state space complexities.

3.1 Abstract architecture of embedded systems

A common architecture for embedded systems is shown in Figure 2(a). Under this architecture, a system consists of

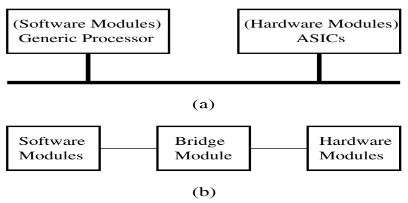

Figure 2. Architecture of embedded systems

a generic processor and multiple application-specific integrated circuits (ASICs). The processor and ASICs are connected by buses. Software modules of an embedded system execute on the generic processor while hardware modules of the system are implemented and executed by ASICs.

In our co-verification study, we are interested in correctness of the software and hardware modules. Therefore, we assume that the generic processor is correct and the ASICs faithfully implement the designs of the hardware modules. Based on these assumptions, we can then define an abstract architecture for embedded systems as shown in Figure 2(b). In this architecture, software and hardware modules interact through a bridge module which converts between software and hardware semantics: propagating events and data, such as hardware interrupts and updates to hardware and software variables, across the semantic boundaries and providing scheduling decisions for execution of software modules.

3.2 Translation-based co-verification framework

Our framework for translation-based co-verification is shown in Figure 3, which features a staged translation of the hardware language, software language, and bridge specification language. For instance, the hardware language is first

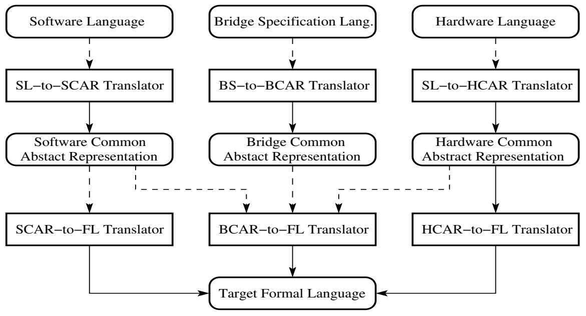

Figure 3. A framework for co-verification

translated into a hardware common abstract representation (HCAR) and the HCAR is then translated to the target formal language. Similarly, we have the concepts of software common abstract representation (SCAR) and bridge common abstract representation (BCAR) for translation of the software language and the bridge specification language.

The common abstract representations (CARs) are intermediate representations for translation, however, they are not intermediate languages with fixed semantics. Each CAR has a common semantic core that contains common semantic constructs of multiple software or hardware specification languages, for instance, xUML and SDL [12], Verilog and VHDL, etc. For translation of each language, a CAR profile is defined, which consists of the semantic constructs from the core and also language-specific semantic constructs. Such an intermediate representation is easy to extend and customize, facilitates reuse of translation efforts, and offers flexibility in supporting multiple languages. (For details on translation and reuse through CARs, see [27].)

Besides software and hardware modules, a bridge specification is required for co-verification, which provides the information for interfacing software and hardware modules:

- What software procedure calls or messages are triggered by hardware interrupts.
- What hardware variables are updated when a procedure call returns or a message is received.
- What variables in software modules are mapped to variables in hardware modules.
- What is the scheduling policy of software modules, for instance, interrupt priorities and preemption policies.

Translation of the bridge specification depends on software and hardware modules since the bridge specification refers to semantic entities in both software and hardware modules.

3.3 Interfacing hardware and software modules

How to interface hardware and software modules depends on the hardware semantics and software semantics, is based on the semantics of the target formal language, and is guided by the bridge specification. Correctly interfacing hardware and software modules requires: (1) inserting a bridge module that properly propagates events across hardware and software semantic boundaries; (2) correctly implementing a scheduling policy for the execution of entities in the software modules based on interactions with the hardware modules, for instance, hardware interrupts.

In this section, we discuss how to interface the formal models of hardware and software modules in integration of FormalCheck and ObjectCheck for co-verification. ObjectCheck supports translation of xUML and SDL to S/R via a SCAR that facilitates translation reuse of common semantic constructs such as message sending and receiving, and control flow structures [27]. An early research version of FormalCheck supports translation of Verilog and VHDL to S/R via a HCAR. (The authors have no knowledge about the internals of the commercial version of FormalCheck.) Hereafter, we assume that software modules are specified in xUML while hardware modules are specified in Verilog.

3.3.1 Dual interfaces of bridge module

The bridge module between hardware and software modules exhibits dual behaviors. It interacts with the hardware modules following the clock-driven synchronous semantics and with the software modules following the asynchronous interleaving message-passing semantics.

As software modules are designed, they are often validated in a closed system that consists of the software modules and a stub hardware module. The stub hardware module is specified in xUML and simulates the expected behaviors of hardware modules. Similarly, hardware modules are often validated with a stub software module specified in Verilog. The dual interfaces of the bridge module can be obtained by inheriting the interfaces of the S/R process translated from the stub hardware module and of the S/R process translated from the stub software module.

3.3.2 Conversion of software and hardware semantics

The dual interfaces of the bridge module are connected internally to implement the conversions from hardware interrupts to software messages and the updates to hardware variables upon consumption of software messages. Suppose that a hardware interrupt i is mapped to a software message m. The S/R process, which simulates the Verilog process that generates i, outputs i through a selection variable. In the same S/R clock cycle, the bridge module records the interrupt. When the interrupt is processed, the bridge module passes an instance of m to the S/R process that simulates the message queue of the receiver xUML object instance. Suppose that a software message m' is to be propagated to the hardware modules. The bridge module includes a set of selection variables. Upon consumption of m', the bridge module updates the selection variables to reflect m' and in the same clock cycle, the S/R processes simulating the hardware modules can act on the value changes in the selection variables. The mapping between a hardware variable and a software variable is realized by unifying the two variables and adding proper references to the unified variable. Where the unified variable is kept is determined by whether hardware or software modules are responsible for updating it.

3.3.3 Extension to software scheduler

In embedded systems, the execution order of software entities (for instance, xUML object instances) depends not only on the interactions among software entities, but also on the interactions between software and hardware modules. For instance, an object instance is ready to execute when (1) it enters a state and is ready to execute the state action or (2) it is ready to consume a message and a message is available in its message queue. The message may have been generated by an object instance or in response to a hardware interrupt. We assume that state actions are run-to-completion, i.e., interrupts are not processed when a state action is executed.

Other factors may also affect scheduling of software entities. There are often multiple hardware interrupts and they may have different priorities. If interrupt preemption is allowed, there may be multiple concurrent interrupt-initiated threads of execution in the whole system. However, at any given time, only one such thread is active. There may also be software-initiated threads of execution in the system. The system may schedule tasks to execute asynchronously. Interrupts may or may not be processed when a task is running and tasks may interleave in different granularity.

Scheduling policies are integrated into the translation as templates. We have implemented a template for generating bridge modules that support a simple customizable scheduling policy: (1) Interrupts are processed according to designer-specified priorities in the bridge specification. (2) Priorities of interrupt processing and software-initiated tasks are designer-specified. (3) Interrupts are not processed when a software-initiated task is executing. (Work on the templates for other scheduling policies are in progress.)

To implement the above policy, we extend the bridge module by adding an interrupt vector and a software task vector. The vectors keep track of the unprocessed interrupts and the software tasks to be scheduled. They are updated according to the inputs from the hardware modules and the software modules. When a task or an interrupt processing is done, a scheduling decision is made according to the vectors and designer-specified priorities. If an interrupt is processed, a message corresponding to the interrupt is sent to the S/R process that simulates the xUML object instance responsible for processing the selected interrupt. If a task is selected to execute, the task is initiated by sending a message to the S/R process that simulates the xUML object instance responsible for the task. When either an interrupt processing or a task is done, the thread of control returns back to the bridge module through a return message. The extended bridge module replaces the original scheduler in the S/R model translated from the software modules.

3.4 A unified property specification language

Co-verification demands a unified language for specifying properties of software modules, hardware modules, and whole systems. Although software modules and hardware modules have different semantics, their translations to the semantics of the target formal language provides a common basis for specifying properties of whole systems.

For co-verification of embedded systems with software modules in xUML and hardware modules in Verilog, we define a unified property specification language, which is linear-time and with the expressiveness of ω-automata [13]. This language consists of a set of property templates that have intuitive meanings and also rigorous mappings to property templates formulated in S/R[1]. The templates define parameterized automata. A property in this language consists of (1) declarations of propositional logic predicates over semantic entities of software modules in xUML and of hardware modules in Verilog, and (2) declarations of temporal predicates. A temporal predicate is declared by instantiating a property template: each argument of the template is replaced by a propositional expression composed from the declared propositional predicates. (See Section 4 for example properties in this language.)

[1]In S/R, both systems and properties are formulated as ω-automata.

3.5 State space reduction for co-verification

Effective state space reduction is crucial to scalability of co-verification. Co-verification examines the integrated formal model of an embedded system including its software modules, hardware modules, and bridge module, which is more complex than any of these modules. In integrating FormalCheck and ObjectCheck for co-verification, we leverage the built-in reduction algorithms and search algorithms of COSPAN such as localization reduction and symbolic model checking. We also apply static partial order reduction [16] in translation of software modules and conduct compositional reasoning [1] across the bridge module.

3.5.1 Symbolic model checking

Among the built-in algorithms of COSPAN, the symbolic search algorithm based on BDDs is of particular interest. Symbolic model checking has been effective in hardware verification but less effective in verification of concurrent software. It is important to understand how effective symbolic model checking is in co-verification. Integration of FormalCheck and ObjectCheck enables application of the symbolic search algorithm in co-verification of an embedded system since both software and hardware modules of the system are translated into S/R. (See Section 4 for case studies of symbolic model checking in co-verification.)

3.5.2 Static partial order reduction

Partial order reduction (POR) [21] is readily applicable to asynchronous interleaving semantics. POR takes advantage of the fact that when components of a system are not tightly coupled, different execution orders of actions or transitions of different components may result in the same global state. Under certain conditions [21], intuitively, when the interim states are not relevant to the property being checked, model checkers need only to explore one of the possible execution orders. This effectively reduces the complexity of model checking. The asynchronous interleaving semantics of xUML suggests application of POR. POR is applied to an xUML model through static partial order reduction (SPOR) [16], a static analysis procedure that transforms the model prior to its translation into S/R. SPOR restricts the transition structure of the model with respect to a property. For different properties, an xUML model may be translated to different S/R models if SPOR is applied.

The effectiveness of SPOR is pertinent to how much concurrency is allowed in software modules. There is often no concurrent execution as the software modules respond to hardware interrupts. The concurrency in software modules mainly lies in the asynchronous execution of software-initiated tasks. Therefore, the amount of concurrency allowed depends on the scheduling policy that is implemented in the bridge module. For instance, for the simple scheduling policy given in Section 3.3.3, the concurrent execution of software-initiated tasks is allowed, however, a task is essentially run-to-completion. Therefore, SPOR explores the possible interleavings of the tasks. If we implement a scheduling policy of finer granularity, for instance, allowing interleaved execution of portions of the tasks, there are more possible interleavings for SPOR to explore. The scheduling policies determine the interleavings allowed in the system execution. Work is in progress to extend the SPOR algorithm to consider scheduling policies.

3.5.3 Compositional reasoning across bridge module

Using compositional reasoning [1], model checking of a property on a system is accomplished by decomposing the system into components, checking component properties locally on the components, and deriving the system property from the component properties.

Under our abstract architecture of embedded systems, software and hardware modules interact through the bridge module, thus simplifying compositional reasoning. It is intuitive to partition an embedded system at the interfaces of the bridge module with the software and hardware modules since software messages and hardware signals are clearly identified in these interfaces and the conversion between messages and signals is explicitly defined in the bridge specification. Compositional reasoning can also be recursively applied to the hardware and software of the system.

The dual behavior of the bridge component simplifies formulation of the properties of software modules (or hardware modules, respectively). Although a system-level property may involve events in both hardware and software modules, the properties of the software (or hardware, respectively) modules and their assumptions on the bridge module are solely based on the asynchronous message-passing semantics (or the clock-driven synchronous semantics).

Meaningful compositional reasoning for co-verification is enabled by unifying the asynchronous semantics of software and the clock-driven semantics of hardware based on their translations to the target formal semantics. In [25], we have developed a translation-based approach to compositional reasoning, which allows component properties to be formulated in their original semantics, verified in the target semantics, and reasoned in the original semantics. This approach is based on a rigorous mapping from the original semantics to the target semantics and reuses compositional reasoning rules been established in the target semantics. Correctness of this approach has been proved in [25].

To enable compositional reasoning for co-verification, we extend translation-based compositional reasoning to support verification of properties of the bridge module. The bridge module has properties and assumptions in both hardware and software semantics. These properties and assumptions are translated into S/R. The assumptions together with the S/R translation of the bridge module form a closed system on which the properties can be verified. Creation of the closed system is greatly simplified by the fact that in S/R, system models, properties, and assumptions are all formulated as ω-automata. If the S/R properties are successfully verified on the closed system, we can conclude that the properties of the bridge module are verified.

There may exist circular dependencies among properties of different modules. Translation-based compositional reasoning reuses compositional reasoning rules in the S/R semantics that prevent circular reasoning, such as [3].

4 Co-Verification of Networked Sensors

This section illustrates our approach to translation-based co-verification with its application to networked sensors. Networked sensors are becoming increasingly important for diverse tasks such as tools to study environmental phenomena, to instrument and manage large-scale systems (e.g., manufacturing), to aid security, and so on. A sensor is a battery-powered embedded system whose hardware consists of a processor and a number of plug-in hardware modules such as sensing and transmitting devices.

4.1 Berkeley Motes and TinyOS

The most well-known networked sensors are the Motes from Berkeley and their run-time system is TinyOS [11]. Due to stringent resource constraints of the Motes, TinyOS features a component-based architecture, which allows designers to select and load only the necessary software components for an application onto the Motes. A TinyOS application is developed via implementing application-specific components and composing them with reused TinyOS components into a run-time TinyOS instance which can then be loaded onto Motes for execution. TinyOS and its run-time instances are closely coupled with the Motes hardware. Networked sensors are often manufactured in large quantity and are not retrievable after deployment. They are also required to support concurrent sensing, transmission, and routing operations. Therefore, sensors must be highly reliable. This demands co-verification of the whole sensors.

A TinyOS application is composed of a set of components. Components communicate through events and commands. Hardware interrupts trigger events. Events propagate upward in the component hierarchy and the propagations may bend downward to invoke commands of lower-level components. To avoid loops, commands cannot generate events. Interrupt processing may generate tasks that execute asynchronously. TinyOS applications are originally developed in the C programming language. Events and commands are implemented as procedure invocations.

4.2 Hardware, software, and bridge specification

We apply translation-based co-verification to a TinyOS application and the Motes hardware which together transmit readings of the hardware sensor on the physical network.

4.2.1 Executable design of TinyOS in xUML

To raise the abstraction level for development of TinyOS applications, we have re-engineered TinyOS and its applications following the Model-Driven Architecture [20]. TinyOS applications are developed through designing their xUML models and automatically compiling the xUML models into C based on pre-defined software architectures. TinyOS components are modeled as xUML object instances or as compositions of object instances. The events and commands in TinyOS applications are explicitly represented as messages among the object instances.

The xUML model of the TinyOS application is shown in Figure 4. (Due to space limitation, only the object collaboration diagram is shown.) The application is essentially partitioned into two high-level modules: Sensor and Network. The Sensor module consists of the six object instances to the left and handles hardware clock and sensor interrupts. The Network module consists of the five object instances to the right and transmits data on the network. Sensor readings are passed from the Sensor module to the Network module via asynchronous tasks.

4.2.2 Verilog design of Motes

The hardware design of the Motes is not readily available in Verilog. As part of our effort to enable co-design and co-verification of TinyOS applications, we model the Motes hardware platform in Verilog. The Verilog design is shown in Figure 5. Three hardware modules, *clock*, *sensor*, and

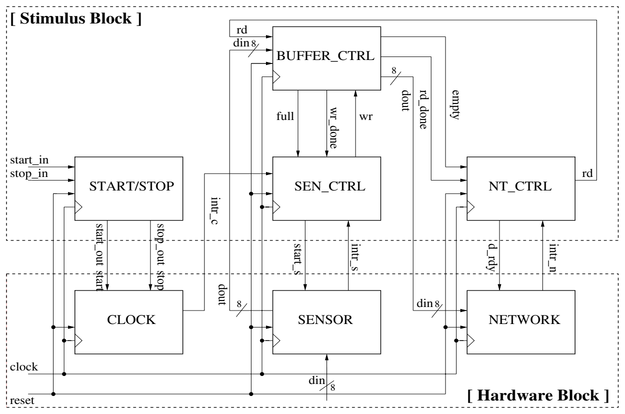

Figure 5. Verilog design of motes

network, are modeled. The clock interrupts periodically. The sensor generates sensor readings at an adjustable delay

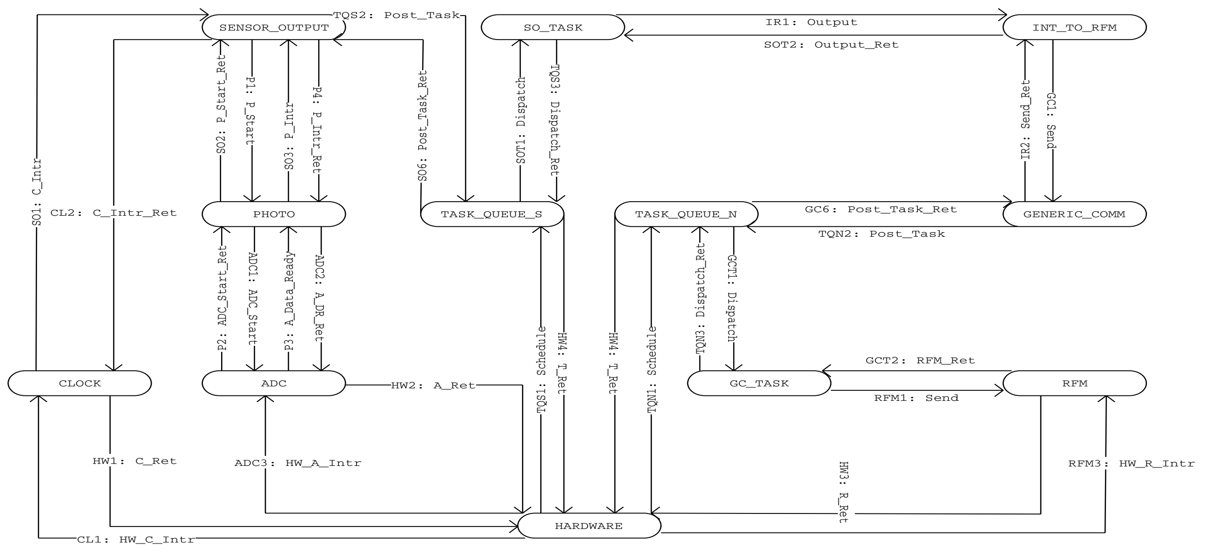

Figure 4. xUML model of a TinyOS instance (object collaboration diagram)

upon request. The network transmits data following a simple MAC protocol. A stimulus block is modeled in Verilog, which facilitates validation of the hardware design.

4.2.3 Bridge specification

To enable co-verification, a bridge specification is needed to formulate how to integrate the hardware design of the Motes with the TinyOS application. The core segments of the bridge specification is shown in Figure 6. It defines: (1) how hardware signals are mapped to software messages, for instance, the hardware clock interrupt, *intr_c*, is mapped to the *HW_C_Intr* message of the software clock object; (2) how software variables are mapped to hardware signals, for instance, the *On* variable of the *ADC* object is mapped to the *start* signal of the hardware sensor; (3) the interrupt priorities, for instance, all interrupts are of the same priority; (4) messages that initiate software tasks, for instance, the *Schedule* message of the *TQS* object, and the conditions under which the tasks are ready for being scheduled; (5) the preemption policies, for instance, no preemption is allowed.

```
/* Hardware interrupt to software message mapping */
(HW.CLOCK.intr_c, SW.CLOCK.CL1: HW_C_Intr)
(HW.SENSOR.intr_s, SW.ADC.ADC3: HW_A_Intr)
(HW.NETWORK.intr_n, SW.RFM.RFM3: HW_R_Intr)

/* Software variable to hardware signal mapping */
(SW.ADC.On, HW.SENSOR.start)
(SW.RFM.Pending, HW.NETWORK.d_rdy)

/* Interrupt priority */
Priority(HW.CLOCK.intr_c) = 0
Priority(HW.SENSOR.intr_s) = 0
Priority(HW.NETWORK.intr_n) = 0

/* Messages for initiating software tasks */
SchedSet = {(TQS1:Schedule | !Process_TQS.EMPTY),
            (TQN1:Schedule | !Process_TQN.EMPTY)}

/* Preemption policies */
Interrupt_Task_Preemption_Enabled = NO
Interrupt_Interrupt_Preemption_Enabled = NO
```

Figure 6. Bridge specification

4.3 Translation and interfacing of HW and SW models

The xUML design of the TinyOS application and the Verilog design of the Motes hardware are translated into S/R by ObjectCheck and FormalCheck, respectively. The resulting S/R models are not shown due to space limitations.

The translation of the bridge specification depends on the translations of the hardware and software modules. As shown in Figure 4 and Figure 5, software modules are modeled with a stub hardware module and the hardware modules are modeled with a stimulus block. The stub hardware module (or the stimulus block, respectively) are translated with the software modules (or the hardware modules). Their S/R translations provide the dual interfaces of the bridge module. The logic of the bridge module is generated from the bridge specification based on the dual interfaces. It bridges software and hardware semantics, for instance, converting between software messages and hardware signals, and implements the scheduling policy. The pseudo code of sample segments of the logic is shown in Figure 7. The clock interrupt is captured by a flag, which is updated upon the hardware interrupt and the software message corresponding to the return of the interrupt handler. Interrupt handlers and software tasks are scheduled randomly at the same priority. Scheduling decisions determines what messages are sent to object instances in the software modules.

```
stvar clk_intr_flag : (0, 1)
asgn clk_intr_flag → 1 ? (INTR_CLOCK_COUNTER.intr_c = 1) |
                     0 ? (InputMSG = Process_HARDWARE_HW1) |
                     clk_intr_flag
. . .
selvar choice : (0..5)
asgn choice := { 0 ? (clk_intr_flag = 1),
                 . . .
                 3 ? (!Process_TQS.EMPTY),
                 . . .
                 }
. . .
selvar OutputSignal : (...)
asgn OutputSignal := CL1 : (choice == 0)
                     TQS1 : (choice == 3)
                     . . .
```

Figure 7. Pseudo code of bridge module

4.4 Property verification and state space reduction

We evaluated our translation-based co-verification approach in verifying two system-level properties on the whole system. We also analyzed effectiveness of the reduction algorithms: symbolic model checking (SMC), static partial order reduction (SPOR), and compositional reasoning.

The two system-level properties are formulated in Figure 8 using our unified property specification language. Property P_1 checks whether the whole system can repeatedly transmit sensor readings on the physical network. This property monitors a flag in the hardware network module. Repeated sets and clears of the flag indicate repeated radio transmissions. (This property does not require strict alternation of sets and clears.) Property P_2 checks whether the whole system can get in a state where the status of the RFM object is *Transmitting*, the previous sensor reading sequence number is 1, and the current sequence number is also 1.

```
Property P1: Repeated transmission on physical network
   Repeatedly(HW.Network.Flag == 1)
   Repeatedly(HW.Network.Flag == 0)

Property P2: No consecutive 1's as transmission sequence numbers
   Never ((SW.RFM.Prev == 1) AND (SW.RFM.Buf == 1)
         AND (SW.RFM.Status == Transmitting))
```

Figure 8. Two system-level properties

P_1 was successfully verified on the S/R model of the system. Verification of P_2 uncovered a bug in the TinyOS instance: A hand-shake protocol controlling a buffer in the *GENERIC_COMM* object is not correctly implemented. If the hardware sensor runs faster than the hardware network, a reading may overwrite the previous reading and be transmitted twice. This bug has been detected when the software modules are checked separately. The same result from co-verification of the whole system further confirms this bug.

The time and memory usages for model checking of the two properties are shown in Table 1. (All verification runs are conducted on a Linux system with dual CPUs at 1.2 GHZ and 2 GB physical memory.) We observe that SMC

Props	SMC	SPOR	Time (Seconds)	Memory (MBytes)
P_1	Off	Off	-	out of memory
P_1	On	Off	7902	391.63
P_1	Off	On	-	out of memory
P_1	On	On	8273	477.54
P_2	Off	Off	11.7	0.92
P_2	On	Off	81.7	20.8
P_2	Off	On	11.6	0.92
P_2	On	On	94.9	18.2

Table 1. Verification complexity comparison

significantly reduces the time and memory usages for verifying P_1 while SPOR does not.[2] There are only two tasks that can execute concurrently in this system, therefore, there is insufficient concurrency to be explored. We believe that SPOR will achieve better reduction on systems with many concurrent tasks, which is a claim that we will validate in future research. In some cases, the time or memory usages are higher when using SPOR with SMC than just using SMC since SPOR may affect the ordering of BDD nodes. SMC uses more time and memory than explicit state enumeration in verifying P_2 due to the overhead of creating all the BDD nodes. These observations indicate that it is important to include all the reduction algorithms in co-verification.

We applied compositional reasoning by partitioning the whole system at the interface between the software modules and the bridge module. We decompose P_1 into two properties, P_{11} on the software and P_{12} on the hardware with the bridge module, which are not shown due to space limitation. P_{11} is verified on the software assuming that P_{12} holds on the hardware using 3194 seconds and 228.08 megabytes. P_{12} is verified on the hardware assuming that P_{11} holds on the software using 246 seconds and 9.35 megabytes. (SMC are on and SPOR are off.) The inter-dependency of P_{11} and P_{12} is validated to rule out circular reasoning using the compositional reasoning rule from [3]. P_1 is derived from P_{11} and P_{12}. In [24], we have further partitioned P_{11} into two properties on the two software modules, Sensor and Network (see Section 4.2.1), respectively. Verification of the two properties takes 101 seconds and 33.67 megabytes and 18 seconds and 6.82 megabytes, respectively. It can be observed that compositional reasoning achieves order-of-magnitude reduction on co-verification complexity.

5 Related Work

Hardware and software co-verification of embedded systems commonly follows two approaches: co-simulation and formal co-verification. Our work is in the formal category.

[2]The original SPOR algorithm is applied since the scheduling policy that is defined by customizing the policy template in Sec 3.3.3 with the bridge specification requires no change to the SPOR algorithm.

Co-simulation of hardware and software of embedded systems is widely supported by industrial tools such as Mentor Graphics Seamless [19] and academic research projects such as Ptolemy [6]. Recently co-simulation has been extended to support multithreading-based models [4], where the whole implementation of the system and its properties appear as a set of communicating threads.

Theorem proving and model checking have both been applied to formal co-verification. Due to space limitation, we focus on the application of model checking. Various formal specification languages have been proposed for embedded systems, such as Hybrid Automata [2], LOTOS [23], Co-design Finite State Machines (CFSMs) [5], and petrinet based languages such as PRES [8]. Hybrid automata and CFSMs have been directly model-checked while LOTOS and PRES have been model-checked through translation to directly model-checkable languages such as SMV.

Our approach supports specification of hardware and software in their native languages and applies model checking via translation. It generalizes the work [15] by Kurshan, et al. on verifying hardware in software context where the software context of the hardware is modeled in SDL and translated into S/R and integrated with the S/R translation of the hardware. Our approach defines a general architecture for translation-based co-verification, which supports easy extension for new hardware/software languages, facilitates translation reuse, and supports compositional reasoning.

6 Conclusions and Future Work

In this paper, we have presented a translation-based approach to co-verification. This approach has great potentials in improving safety, security, and reliability of embedded systems by giving system designers a powerful toolkit for model checking a whole embedded system including its hardware and software. The case study on network sensors has shown that this approach enables co-verification of embedded systems of real-world scale and achieves order-of-magnitude reduction on co-verification complexities.

Timing verification is important to co-verification of embedded systems. COSPAN has strong support for timing verification. We plan to extend our toolkit to support timing co-verification. Templates for more advanced software module scheduling policies will be developed. The SPOR algorithm will be extended to consider the various scheduling policies. We also plan to re-target our toolkit to SMV by reusing translator modules following our general framework and study trade-offs in using COSPAN and SMV.

Acknowledgment

We gratefully acknowledge the contributions of James C. Browne, Robert P. Kurshan, and Vladimir Levin to this research. We also thank Douglas V. Hall for his support.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *TOPLAS*, 17(3), 1995.

[2] R. Alur, T. A. Henzinger, and P. H. Ho. Automatic symbolic verification of embedded systems. *IEEE TSE 22(3)*, 1996.

[3] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In *Proc. of TACAS*, 2001.

[4] M. Azizi, E. M. Aboulhamid, , and S. Tahar. Properties coverification for HW/SW systems. In *Proc. of ECS*, 1999.

[5] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal verification of embedded systems based on cfsm networks. In *Proc. of DAC*, 1996.

[6] Berkeley. Ptolemy project. In *http://ptolemy.eecs.berkeley.edu/index.htm*, 2005.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of Logic of Programs Workshop*, 1981.

[8] L. A. Cortes, P. Eles, and Z. Peng. Formal coverification of embedded systems using model checking. In *Proc. of EUROMICRO*, 2000.

[9] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of CAV*, 1997.

[10] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. of CAV*, 1996.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, 2000.

[12] ITU. *ITU-T Recommendation Z.100 (03/93) - Specification and Description Language (SDL)*. ITU, 1993.

[13] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[14] R. P. Kurshan. *FormalCheck User Manual*. Cadence, 1998.

[15] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Verifying hardware in its software context. In *Proc. of ICCAD*, 1997.

[16] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Proc. of TACAS*, 1998.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[18] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.

[19] Mentor Graphics. Seamless. In *http://www.mentor.com*.

[20] OMG. *http://www.omg.org/mda*. OMG, 2005.

[21] D. Peled. Ten years of partial order reduction. 1998.

[22] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of SOP*, 1982.

[23] P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The formal description technique LOTOS*. Elsevier, 1989.

[24] F. Xie and J. C. Browne. Verified systems by composition from verified components. In *Proc. of ESEC/FSE*, 2003.

[25] F. Xie, J. C. Browne, and R. P. Kurshan. Translation-based compositional reasoning for software systems. In *Proc. of FME*, 2003.

[26] F. Xie, V. Levin, and J. C. Browne. ObjectCheck: A model checking tool for executable object-oriented software system designs. In *Proc. of FASE*, 2002.

[27] F. Xie, V. Levin, R. P. Kurshan, and J. C. Browne. Translating software designs for model checking. In *Proc. of FASE*, 2004.