Component-Based Hardware/Software Co-Verification [*]

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

Guowu Yang
Dept. of Computer Science
Portland State University
Portland, OR 97207, USA
guowu@cs.pdx.edu

Xiaoyu Song
Dept. of ECE
Portland State University
Portland, OR 97207, USA
song@ece.pdx.edu

Abstract

We present a novel component-based approach to hardware/software co-verification of embedded systems using model checking. Due to their diverse applications and often strict physical constraints, embedded systems are increasingly component-based and include only the necessary components for their missions. In our approach, a component model for embedded systems which unifies the concepts of hardware IPs (i.e., hardware components) and software components is defined. Hardware and software components are verified as they are developed bottom-up. Whole systems are co-verified as they are developed top-down. Interactions of bottom-up and top-down verification are exploited to reduce verification complexity by facilitating compositional reasoning and verification reuse. Case studies on a suite of networked sensors have shown that our approach facilitates major verification reuse and leads to order-of-magnitude reduction on verification complexity.

1 Introduction

Embedded systems are often mission-critical, deployed in large quantity, and difficult to access after deployment. Thus, they must be extensively verified. Due to the strict design constraints of embedded systems, to achieve better performance, hardware and software components must closely interact and hardware/software trade-offs must be exploited. This demands hardware/software co-design and, therefore, hardware/software co-verification of embedded systems.

Model checking [7] is a formal verification method that has great potential in hardware/software co-verification. It provides exhaustive state space coverage for the systems being verified. A stumbling block to scalable application of model checking to co-verification is the intrinsic complexity of model checking. The number of possible states and execution paths in a real-world system can be extremely large, which requires state space reduction. Co-verification of an embedded system involves its hardware and software, which makes state space reduction even more challenging.

Due to their diverse applications and often strict physical constraints, embedded systems are increasingly component-based and include only the necessary components for their missions. Component-based development introduces compositional structures and standard component interfaces into embedded systems and promotes component reuse.

Compositional reasoning [1, 2, 14, 4], as applied in model checking, is a powerful state space reduction algorithm and accomplishes verification of a property on a system by decomposing the system into components, checking the component properties locally, and deriving the system property from the component properties. Compositional structures of embedded systems may greatly simplify application of compositional reasoning to co-verification.

We propose a novel component-based approach to hardware/software co-verification for building trustworthy embedded systems. Embedded systems are structured following a component model that unifies the concepts of hardware IPs [11] (i.e., hardware components) and software components [17]. In this model, verified properties of hardware and software components are associated with the components. Selection of components for reuse is based on their functionalities and also their verified properties. A special type of component, *bridge component*, is introduced, which inter-connects hardware and software components and bridges the hardware/software semantic gaps.

Our approach to co-verification of embedded systems is a synergistic integration of bottom-up component verification and top-down system verification. Hardware and software components are verified as they are developed bottom-up. Properties of a primitive component are directly verified while properties of a composite component are verified on its abstractions constructed from verified properties of its sub-components. A system is verified top-down as it is developed via recursive decompositions into its components. The decompositions reuse components as possible. Verified properties of the reused components are reused in constructing the abstractions for verifying properties of the sys-

[*]This research was supported by Semiconductor Research Corporation, Contract RID 1356.001.

tem or higher-level components. Our approach is based on translation-based co-verification [22] where software and hardware modules of an embedded system are translated into a formal model-checkable language, integrated under the target semantics, and model-checked. Translation-based co-verification provides a common formal semantic basis for conducting compositional reasoning and the basic verification mechanisms for discharging proof obligations.

Our approach has great potential for building trustworthy systems by enabling effective co-verification. Case studies have shown that it achieves major verification reuse and order-of-magnitude reduction on verification complexity.

The rest of this paper is organized as follows. In Section 2, we provide the background of this work. We define the component model unifying hardware IPs and software components in Section 3. In Section 4, we present our component-based approach to co-verification and illustrate it with case studies on a suite of networked sensors. We discuss related work in Section 5 and conclude in Section 6.

2 Background

2.1 Component-based development

A common trend in both hardware and software industries is to develop systems via assembly of components [11, 17]. (In the hardware industry, component-based development is also known as Intellectual Property (IP) based development.) A main advantage of component-based development is reuse of design and development efforts. As verification becomes increasingly important in system development, it is also desired to reuse verification efforts.

2.2 Translation-based co-verification

In [22], we have developed a translation-based approach to co-verification of embedded systems using model checking. Hardware and software modules of an embedded system are automatically translated into the input formal language of a state-of-the-art model checker. The semantics of hardware and software specification languages are simulated by the semantics of the target formal language. We interface the formal models of hardware and software modules by inserting a *bridge module* that 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 certain software variables. The bridge module is specified in a bridge specification language and translated into the formal language. This approach has been implemented for co-verification of software in Executable UML (xUML) [15], an executable dialect of UML, and hardware in Verilog [18]. The implementation integrates two translation-based model checkers: FormalCheck [13] and ObjectCheck [21], both based on the COSPAN model checker [9]. FormalCheck is a commercial tool for hardware verification. ObjectCheck was developed in our previous work for verification of software designs in xUML. xUML has an asynchronous interleaving message-passing semantics: a system consists of object instances interacting via asynchronous message-passing. In a system execution, at any moment only one object instance can progress.

2.2.1 Bridge specification

For translation-based co-verification of an embedded system, a specification of the bridge module is required, which specifies how to interface the software and hardware modules: (1) what software procedure calls or messages are triggered by hardware interrupts; (2) what hardware variables are updated when a procedure call returns or a message is received; (3) what variables in the software modules are mapped to hardware signals; (4) what are the scheduling policies for the software modules, for instance, interrupt priorities. Translation of the bridge specification depends on the software and hardware modules since it refers to semantic entities in both software and hardware modules.

2.2.2 Unified property specification

In co-verification, a unified property specification language for both hardware and software is needed. We have developed such a language for co-verification of software modules in xUML and hardware modules in Verilog. This language is presented in terms of a set of property templates that have intuitive meanings and also rigorous mappings to ω-automata templates written in S/R [9], the input formal language of COSPAN. Our property specification language is linear time, with the expressiveness of ω-automata [12]. (In S/R, both systems and properties are formulated as ω-automata.). The templates define parameterized automata. New templates are formulated as needed by defining their mappings into S/R. A property in this language consists of (1) declarations of propositional predicates over semantic entities in software and hardware modules, and (2) declarations of temporal assertions. A temporal assertion is declared by instantiating a property template: each argument of the template is realized by a propositional expression composed from the declared propositional predicates. (See Section 3 for example properties in this language.)

2.3 Bottom-up verification of software components

In [20], we have developed a bottom-up approach to verification of software components and systems composed from these components. For a primitive component (a component that is built from scratch), its properties are directly model-checked. The properties of a composite component (a system is also a composite component), instead of being directly verified on the component, are verified on its abstractions. The abstraction for checking a property on a composite component is constructed from verified properties of the sub-components. A sub-component property is included in the abstraction if and only if (1) it is related to the property to be checked on the composite component by cone-of-influence analysis [12, 7], (2) its assumptions are enabled, i.e., implied by the properties of other sub-components and the environment assumptions of the composite component, and (3) it is not involved in circular reasoning among the sub-component properties.

How to verify a property on a primitive component or on the abstraction of a composite component depends on the executable representation and the property specification for the components. For instance, for a primitive component specified in xUML and properties specified in the unified property specification language, the component and its properties can be translated into S/R with ObjectCheck and the properties can then be verified on the component with COSPAN. For a composite component, if the properties of the component and its sub-components are all specified in the unified property specification language, a property of the composite component can be verified on its abstraction by translating both the property and the abstraction into S/R.

3 Unified Component Model

To define a unified component model for embedded systems, we start by examining an abstract, but representative architecture of embedded systems as shown in Figure 1. The software components of an embedded system

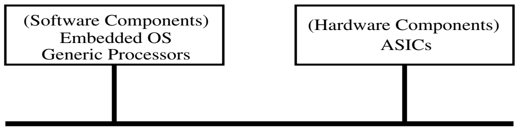

Figure 1. Abstract Architecture

execute on generic processors while the hardware components are implemented as Application Specific Integrated Circuits (ASICs) which connect to the processors via buses. The software and hardware components interact through an embedded OS that also schedules the software components.

From the abstract architecture, we can derive a unified component model as shown in Figure 2. An embedded system is composed of a set of components. There are three types of primitive components: *software components*, *hardware components*, and *bridge components*. Bridge components bridge the hardware/software semantic gap by propagating events across the hardware/software boundary.

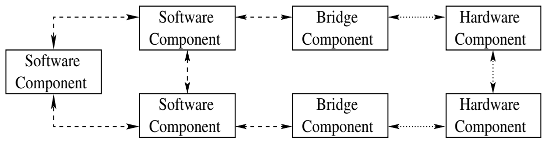

Figure 2. Unified Component Model

Software schedulers are not explicitly represented in this model. Instead, the scheduling constraints are integrated into this model as assumptions of the components. The bridge components and scheduling constraints together abstract the processors, buses, and embedded OS. Three types of composite components may be defined: *software*, *hardware*, and *hybrid*. Sub-components of a composite software (or hardware, respectively) component are all software (or hardware) components. A hybrid component contains both hardware and software components, and therefore, also bridge components. This model basically unifies hardware and software component models.

3.1 Components

A component C is a triple (E, I, P) where E is the executable representation of C, I is the functional interface of C, and P is a set of temporal properties of C that have been verified on E. Hardware components, software components, and bridge components differ in the representations of E and I, but share the same representation of P. Each entry of P is a pair $(p, A(p))$ where p is a temporal property and $A(p)$ is a set of assumptions (i.e., assumed properties) on the environment of C for enabling the verification of p on C. The environment of C is the set of components that interact with C in a composition.

3.1.1 Software components

For a software component, E can be specified in the C programming language or other software languages. To support high-level software design, we adopt the model-driven development [15] and specify software components in an executable design-level language, xUML. I of a software component is a pair, (M, V), where M is a set of input and output messages and V is a set of variables in E that are exported. The component communicates with its environment via asynchronous message-passing. The variables in V are either variables to be mapped to hardware signals or variables to be utilized in scheduling the software component. This interface semantics is determined by the asynchronous interleaving message-passing semantics of xUML.

A software sensor component (denoted by *S-SEN*), which controls a hardware sensor upon clock interrupts, is shown in Figure 3. The dashed box denotes the component boundary. The incoming arrows denote input message types and the outgoing arrows denote output message types. *S-*

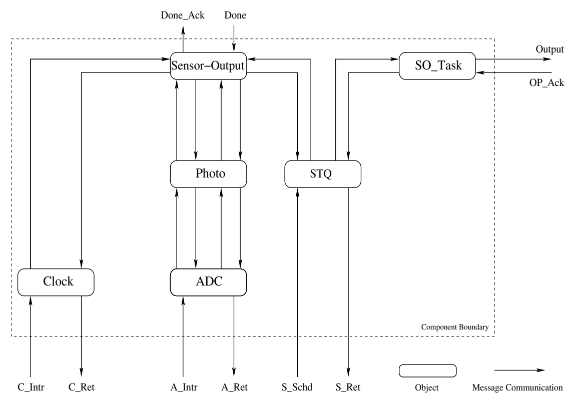

Figure 3. Software sensor component

SEN exports two variables: *ADC.Pending* and *STQ.Empty*. A set of properties that have been verified on *S-SEN* are shown in Figure 4. The properties assert that *S-SEN* repeat-

IfRepeatedly (C_Intr) **Repeatedly** (Output);
After (Output) **Never** (Output) **UnlessAfter** (OP_Ack);
Never (Output) **UnlessAfter** (S_Schd);
After (Output) **Never** (Output) **UnlessAfter** (S_Schd);
Never (S_Ret) **UnlessAfter** (OP_Ack);
After (S_Ret) **Never** (S_Ret) **UnlessAfter** (OP_Ack);

After (C_Intr) **Eventually** (C_Ret);
Never (C_Ret) **UnlessAfter** (C_Intr);
After (C_Ret) **Never** (C_Ret) **UnlessAfter** (C_Intr);

After (A_Intr) **Eventually** (A_Ret);
Never (A_Ret) **UnlessAfter** (A_Intr);
After (A_Ret) **Never** (A_Ret) **UnlessAfter** (A_Intr);
After (ADC.Pending) **Never** (ADC.Pending) **UnlessAfter** (A_Ret);

After (S_Schd) **Eventually** (S_Ret);
Never (S_Ret) **UnlessAfter** (S_Schd);
After (S_Ret) **Never** (S_Ret) **UnlessAfter** (S_Schd);
After (STQ.Empty=False) **Never** (STQ.Empty=False) **UnlessAfter** (S_Ret);

Assumptions:
After (Output) **Eventually** (OP_Ack);
Never (OP_Ack) **UnlessAfter** (Output);
After (OP_Ack) **Never** (OP_Ack) **UnlessAfter** (Output);

After (C_Intr) **Never** (C_Intr+A_Intr+S_Schd) **UnlessAfter** (C_Ret);

After (ADC.Pending) **Eventually** (A_Intr);
Never (A_Intr) **UnlessAfter** (ADC.Pending);
After (A_Intr) **Never** (C_Intr+A_Intr+S_Schd) **UnlessAfter** (A_Ret);
After (A_Ret) **Never** (A_Intr) **UnlessAfter** (ADC.Pending);

After (STQ.Empty=False) **Eventually** (S_Schd);
Never (S_Schd) **UnlessAfter** (STQ.Empty=False);
After (S_Schd) **Never** (C_Intr+A_Intr+S_Schd) **UnlessAfter** (S_Ret);
After (S_Ret) **Never** (S_Schd) **UnlessAfter** (STQ.Empty=False);

Figure 4. Properties of software sensor

edly outputs and correctly handles the output handshakes. The assumptions assert that the environment of *S-SEN* correctly responds to the output handshakes, correctly generates clock and sensor interrupts, and correctly schedules the software tasks in *S-SEN*. The property specification is intuitive, for instance, the first statement claims that *S-SEN* outputs repeatedly if it receives clock interrupts repeatedly and the second statement claims that after an output, *S-SEN* will not output unless after an acknowledgment is received.

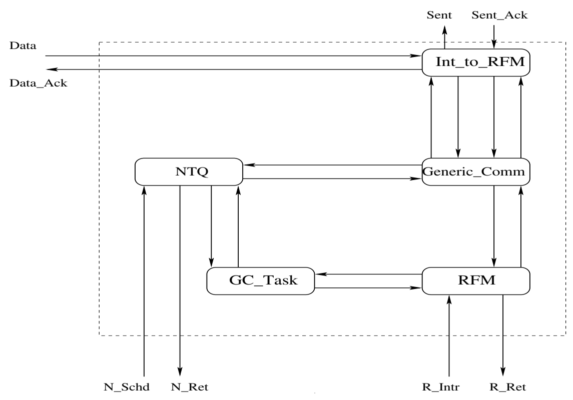

Figure 5. Software network component

Properties:
IfRepeatedly (Data) **Repeatedly** (RFM.Pending);
IfRepeatedly (Data) **Repeatedly** (RFM.Pending=False);

After (Data) **Eventually** (Data_Ack);
Never (Data_Ack) **UnlessAfter** (Data);
After (Data_Ack) **Never** (Data_Ack) **UnlessAfter** (Data);

After (N_Schd) **Eventually** (N_Ret);
Never (N_Ret) **UnlessAfter** (N_Schd);
After (N_Ret) **Never** (N_Ret) **UnlessAfter** (N_Schd);
After (NTQ.Empty=False) **Never** (NTQ.Empty=False) **UnlessAfter** (N_Ret);

After (R_Intr) **Eventually** (R_Ret);
Never (R_Ret) **UnlessAfter** (R_Intr);
After (R_Ret) **Never** (R_Ret) **UnlessAfter** (R_Intr);
After (RFM.Pending) **Never** (RFM.Pending) **UnlessAfter** (R_Ret);

Assumptions:
After (Data) **Never** (Data+N_Schd+R_Intr) **UnlessAfter** (Data_Ack);

After (NTQ.Empty=False) **Eventually** (N_Schd);
Never (N_Schd) **UnlessAfter** (NTQ.Empty=False);
After (N_Schd) **Never** (Data+N_Schd+R_Intr) **UnlessAfter** (N_Ret);
After (N_Ret) **Never** (N_Schd) **UnlessAfter** (NTQ.Empty=False);

After (RFM.Pending) **Eventually** (R_Intr);
Never (R_Intr) **UnlessAfter** (RFM.Pending);
After (R_Intr) **Never** (Data+N_Schd+R_Intr) **UnlessAfter** (R_Ret);
After (R_Ret) **Never** (R_Intr) **UnlessAfter** (RFM.Pending);

Figure 6. Properties of software network

A software network component (denoted by *S-NET*) is shown in Figure 5. It exports two variables: *TQN.Empty* and *RFM.Pending*. The properties that have been verified on *S-NET* are shown in Figure 6. These properties assert that *S-NET* repeatedly sets and clears the *RFM.Pending* variable if it receives data messages repeatedly and it correctly handles the input handshakes. The assumptions assert that the environment of *S-NET* correctly conducts the input handshakes, responds to the value changes of *RFM.pending* with interrupts, and schedules the software tasks in *S-NET*.

3.1.2 Hardware components

For a hardware component, E can be specified in Verilog or other hardware specification languages. In our study, we assume that E is specified in Verilog. I consists of a set of variables that the hardware component imports from or exports to its environment. The hardware component communicates with its environment through the exported or imported variables in I. This interface semantics is determined by the synchronous clock-driven semantics of Verilog.

The interfaces of three hardware components, *clock, sensor, and network* (denoted by *H-CLK*, *H-SEN*, and *H-NET*, respectively) are shown in Figure 7. The incoming arrows

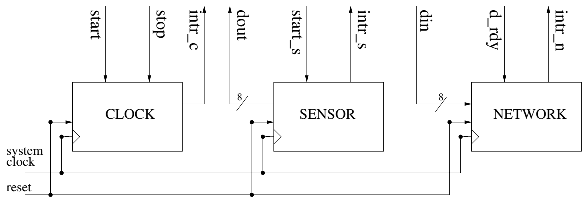

Figure 7. Basic hardware components

denote imported variables and the outgoing arrows denote exported variables. A set of verified properties of the three components are shown in Figure 8. The properties of *H-CLK* assert that *H-CLK* generates interrupts repeatedly. The properties of *H-SEN* assert that after *H-SEN* is started, it will generate an interrupt eventually and it will not generate the interrupt unless after it is started. The properties of *H-NET* assert that (1) after *H-NET* receives data, it will eventually generate a transmission complete interrupt and it will not generate the interrupt unless after it is started and (2) if *H-NET* receives data repeatedly, it transmits repeatedly.

Properties of hardware clock component:
Repeatedly (intr_c);

Properties of hardware sensor component:
After (start_s) **Eventually** (intr_s);
Never (intr_s) **UnlessAfter** (start_s);
After (intr_s) **Never** (intr_s) **UnlessAfter** (start_s);

Properties of hardware network component:
After (d_rdy) **Eventually** (intr_n);
Never (intr_n) **UnlessAfter** (d_rdy);
After (intr_n) **Never** (intr_n) **UnlessAfter** (d_rdy);
IfRepeatedly (d_rdy) **Repeatedly** (flag);
IfRepeatedly (d_rdy=False) **Repeatedly** (flag=False);

Figure 8. Properties of hardware components

3.1.3 Bridge components

Bridge components inter-connect hardware and software components. They extend the concept of bridge module (defined in [22] and briefly introduced in Section 2.2.1) by allowing multiple bridge components in a system. This enables more flexible composition of hardware and software components and creation of hybrid composite components containing both hardware and software sub-components. The interface of a bridge component is a pair (I_H, I_S). I_H is a synchronous shared-variable interface for interactions with hardware components and I_S is an asynchronous message-passing interface for interactions with software components. The interface of the bridge component is determined by the hardware and software components that it connects. E of a bridge component is specified in the bridge specification language discussed in Section 2.2.1.

We illustrate the concept of bridge component by defining a bridge component that inter-connects *S-SEN*, *H-CLK*, and *H-SEN*. The bridge component is shown in Figure 9. The interface of the bridge component is derived from the interfaces of *S-SEN*, *H-CLK*, and *H-SEN* by including the same messages and variables but reversing their input/output directions. The executable specification of the bridge component defines: (1) how hardware signals are mapped to software messages, for instance, the hardware clock interrupt, *intr_c*, is mapped to the *C_Intr* message of the software clock; (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, both interrupts are of the same priority; (4) messages that initiate software tasks, for instance, the *Schedule* message of the *STQ* object, and the conditions under which the tasks are ready to be scheduled.

Bridge interface:
I_H = {input_var = {H-CLK.intr_c, H-SEN.intr_s}
        output_var = {H-SEN.start}}

I_S = {output_msg = {S-SEN.C_Intr, S-SEN.A_Intr, S-SEN.S_Schd}
        input_msg = {S-SEN.C_Ret, S-SEN.A_Ret, S-SEN.S_Ret}
        var = {S-SEN.ADC.On, S-SEN.STQ.Empty}}

Bridge executable representation:
/*Hardware interrupt to software message mapping*/
(H-CLK.intr_c, S-SEN.C_Intr)
(H-SEN.intr_s, S-SEN.A_Intr)

/*Software variable to hardware signal mapping*/
(S-SEN.ADC.On, H-SEN.start)

/*Interrupt priority*/
Priorities(H-CLK.intr_c, H-SEN.intr_s) = {0, 0}

/*Messages for initiating software tasks*/
SchdSet = {(S-SEN.STQ.S_Schd | (S-SEN.STQ.Empty=False))}

Figure 9. A bridge component example

The bridge components abstract the hardware/software interfaces and also abstract part of the embedded OS by specifying the interrupt priorities, and the software tasks that need to be scheduled to execute and their enabling conditions. Software schedulers are not explicitly represented in this component model. Instead, scheduling policies are specified as assumptions of the software components. The embedded OS determines the scheduling polices.

3.1.4 Hybrid components

Hybrid components package hardware and software components into reusable units since hardware and software components are often closely related and reused together, e.g. a device and its driver. A hybrid component may have only a software interface if its hardware can be completely encapsulated or it may have a hybrid hardware/software interface similar to the interface of a bridge component. (Examples of hybrid components are given in Section 4.

3.2 Composition

A composite component, $C = (E,\ I,\ P)$, is composed from a set of simpler components, $C_0 = (E_0,\ I_0,\ P_0), \ldots, C_{n-1} = (E_{n-1},\ I_{n-1}, P_{n-1})$, as follows. E is constructed from $E_0, \ldots, E_{n-1}$ by connecting $E_0, \ldots, E_{n-1}$ through $I_0, \ldots, I_{n-1}$. I may be a hardware interface, a software interface, or a hybrid hardware/software interface depending what types of components $C_0, \ldots, C_{n-1}$ are. Essentially, I includes the semantic entities from $I_0, \ldots, I_{n-1}$ that are needed for C to interact with its environment or for specification of scheduling constraints of C. We discuss how to establish properties of a composite component from properties of its sub-components in Section 4.

4 Component-Based Co-Verification

In this section, we present our approach to component-based co-verification of embedded systems and illustrate this approach with its application to a family of networked sensors. Our approach seamlessly integrates verification into the component-based development lifecycle of embedded systems and is a synergistic integration of bottom-up component verification and top-down system verification.

The component-based development lifecycle for a family of embedded systems consists of three major activities, *basic component development*, *system development*, and *new component development*. Basic component development takes place when the family is created. As the family evolves, system development and new component development are repeated as needed and often interleave.

4.1 Bottom-up verification of basic components

When an embedded system family is created, its primitive hardware and software components are identified by domain analysis and developed from scratch. These primitive components can be further composed bottom-up to develop basic composite components of the family.

For verification of basic components, we extend the bottom-up approach developed in [20]. Properties of the components are formulated according to domain analysis. A primitive hardware (or software, respectively) component is verified using FormalCheck (or ObjectCheck) through translation of the component and its properties into S/R. Properties of a composite component are verified by checking the properties on abstractions of the composite component. The verification is again through translation into S/R.

4.1.1 Verification of primitive components

A domain analysis on the family of networked sensors based on UC Berkeley motes [10] identifies a set of primitive components of the family. The set includes three hardware components: *H-CLK*, *H-SEN*, and *H-NET*, and two software components: *S-SEN* and *S-NET*, as defined in Section 3. We have verified the properties of *S-SEN* in Figure 4 and the properties of *S-NET* in Figure 6 with ObjectCheck and we have also verified the properties of *H-CLK*, *H-SEN*, and *H-NET* in Figure 8 with FormalCheck. The time usages and memory usages for these verification runs are shown in Table 1. The properties of these components are verified

Components	Time (Seconds)	Memory (MBytes)
S-SEN	18.66	8.49
S-NET	18.06	9.11
H-CLK	0.21	3.38
H-SEN	0.22	3.39
H-NET	0.22	3.39

Table 1. Time and memory usages for verifying the properties of primitive components

under their corresponding environment assumptions.

4.1.2 Verification of a basic sensor system

After the basic components are developed, the natural next step is to develop a basic sensor system from these components so that these components can be evaluated in a system context. Note that a system is also a composite component. Figure 10 shows how the basic components are composed bottom-up into a basic system. *H-CLK* generates periodic

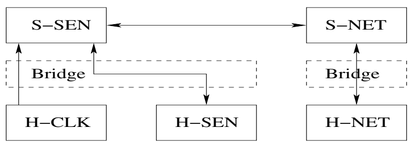

Figure 10. A basic sensor system

interrupts to *S-SEN*. Upon a clock interrupt, *S-SEN* starts *H-SEN*. When *H-SEN* finishes sensing, it interrupts *S-SEN* to pass sensor readings to *S-SEN*. *S-SEN* sends sensor readings to *S-NET*. If *H-NET* is free, *S-NET* delivers a data packet to *H-NET*. After the packet is transmitted, *H-NET* interrupts *S-NET* to report the transmission. The hardware and software components are connected via two bridge components.

Formulating the properties of the bridge components and their assumptions is straightforward. Properties (or assumptions, respectively) of the software and hardware components that are formulated on the interactions with the bridge components are essentially assumptions (or properties) of the bridge components. For instance, the second group of properties (or assumptions) of *S-SEN* in Figure 4, formulated on the clock interrupts generated by the bridge component between *S-SEN* and hardware and *S-SEN*'s responses to these interrupts, are assumptions (or properties) of the bridge component. Additional properties of the bridge components are formulated on the mappings between hardware and software events. The properties of the two bridge components are verified using 3.76 seconds and 6.03 megabytes and 0.66 seconds and 4.07 megabytes, respectively.

A system-level property P_1 to be verified on the basic sensor system is shown in Figure 11. P_1 asserts that the basic sensor system transmits on the network repeatedly. Repeated setting and clearing of a flag in *H_NET* indicates repeated transmission. To verify P_1, we construct an abstraction of the basic sensor system as follows:

```
Repeated (H-NET.flag); Repeated (H-NET.flag=False);
```

Figure 11. Repeated transmission property

1. A system in S/R is constructed to abstract the sensor system. For each hardware or software component, a S/R process (conceptually, an ω-automaton) is introduced, which simulates the interface of the component. Within the constraint of the interface, the S/R process behaves non-deterministically. Essentially, we translate the interface of the component into S/R. All these S/R processes are composed together as how their corresponding components are composed in Figure 10.

2. Starting from P_1, a cone-of-influence analysis is conducted on the verified properties of the hardware and software components based on the component interfaces and the component composition graph. All component properties related to P_1 by the analysis are included in the abstraction. They are used to constrain the S/R system: A property of a hardware or software component is translated to a S/R process and composed with the S/R process abstracting the component.

Note that to include a component property into the abstraction, two conditions must be validated: (1) the assumptions of the property are implied by the properties of other components, which can be validated by a simple model checking run; (2) the property is not involved in circular reasoning among component properties. Circular reasoning can be avoided using the following methods (but not limited to them): (1) avoid using an assumption that creates a dependency cycle; (2) use temporal induction proposed by McMillan [14]; (3) use the compositional reasoning rule proposed by Amla, et al. [4].

The abstraction includes the properties of *S-SEN* in Figure 4, the properties of *S-NET* in Figure 6, the properties of the hardware components in Figure 8, and the properties of the bridge components. *S-SEN* and *S-NET* satisfy the handshake-related assumptions of each other. The other assumptions of *S-SEN* and *S-NET* are satisfied by the properties of the hardware components via the event mappings of the bridge components. The properties of the hardware, software, and bridge components shown in Figure 12 imply P_1. (Note that *S-SEN.Output* is mapped to *S-NET.Data*.)

```
Repeatedly (H-CLK.intr_c);
IfRepeatedly (H-CLK.intr_c) Repeatedly (S-SEN.C_Intr);
IfRepeatedly (S-SEN.C_Intr) Repeatedly (S-SEN.Output);
IfRepeatedly (S-NET.Data) Repeatedly (S-NET.RFM.Pending);
IfRepeatedly (S-NET.Data) Repeatedly (S-NET.RFM.Pending=False);
IfRepeatedly (S-NET.RFM.Pending) Repeatedly (H-NET.d_rdy);
IfRepeatedly (S-NET.RFM.Pending=False) Repeatedly(H-NET.d_rdy=False);
IfRepeatedly (H-NET.d_rdy) Repeatedly (H-NET.flag);
IfRepeatedly (H-NET.d_rdy=False) Repeatedly (H-NET.flag=False);
```

Figure 12. Comp. properties that imply P_1

The implication is established by model checking P_1 on the abstraction, which takes 0.1 seconds and 3.40 megabytes.

The abstraction is conservative. If the property holds on the abstraction, it also holds on the system; otherwise, the abstraction can be refined by verifying additional component properties and including them in the abstraction. If the property does not hold on the system, error trace analysis and abstraction refinement are likely to uncover the cause. (See below for an example of bug detection through abstraction refinement.) Verification of additional properties are rarely needed for widely reused components.

This approach to abstraction construction extends the approach in [20] and constructs abstractions of embedded systems composed of hardware, software, and bridge components. It is enabled by the unified component model and the unified property specification. An abstraction of a composite component that is not a complete system is constructed the same way except that an additional S/R process is added to create a closed S/R system. This S/R process is constrained by the environment assumptions of the component.

The second property P_2 to be verified on the basic system is shown in Figure 13. P_2 asserts that there are no consecutive 1's in the transmission sequence numbers. We construct an abstraction for verifying P_2. However, no component properties are included since no component properties related to P_2 have been verified.

```
Never ((S-NET.RFM.Prev = 1) AND (S-NET.RFM.Buf = 1)
                          AND (S-NET.RFM.Status = Transmitting));
```

Figure 13. No consecutive 1's property

This abstraction needs to be refined. The component properties needed for verifying P_2 are introduced based on domain knowledge. An abstraction is constructed from the component properties assuming they hold. If P_2 is successfully verified on the abstraction, the component properties are then verified. The following properties are introduced for *S-SEN*: there are no consecutive 1's in the sequence numbers of the outputs of *S-SEN* and *S-SEN* will not output a new sensor reading unless after it receives transmission acknowledgment for the previous reading. (For conciseness, the formal property specifications are not shown.)

Verification of the new property on *S-SEN* detects a bug in *S-SEN*: *S-SEN* may output a new sensor reading to *S-NET* although *S-NET* has not acknowledged the transmission of the last sensor reading. The bug is fixed. The property is successfully verified on the corrected *S-SEN*. (For conciseness, the properties of other components are not shown.) After all new component properties are successfully verified, we can conclude that P_2 holds on the basic system.

4.2 Top-down system verification

New systems in the embedded system family are developed top-down. Given its functional requirements, a system is decomposed into its hardware and software components. The decomposition is guided by domain knowledge and considers existing components. The interface of each component is defined and its properties are specified. If there is an existing component matching the interface and properties, the component can be reused. If there is no matching component, the component is either developed from scratch as a primitive component or further decomposed.

Verification is integrated in the top-down system development. As a composite component is decomposed into its sub-components, the sub-component properties are formulated. The properties of the composite component are verified on its abstractions constructed from the sub-component properties assuming the sub-component properties hold. If the properties of the composite components are successfully verified on the abstraction, the top-down system development proceeds; otherwise, the decomposition or the sub-component properties are revised. For a reusable sub-component, if the required properties have been verified on the sub-component, nothing needs to be done; otherwise, the properties are verified on the component top-down. For a new primitive component, its properties are verified by directly model checking its executable representation. For a new composite component, its properties are verified as it is further decomposed top-down. If the properties of a component cannot be verified, the component design or the previous decompositions are revised.

4.2.1 Verification of multi-sensor system

We illustrate top-down system verification by verifying a multi-sensor system. The functional requirement of this system is that it should properly control multiple hardware sensors, e.g., temperature and humidity sensors. The system can be decomposed into its hardware and software components as shown in Figure 14. It can be observed that the multi-sensor system reuses the existing components with a new bridge component that connects *S-SEN*, *H-CLK*, and the two hardware sensors. Upon a clock interrupt, *S-SEN* starts both hardware sensors. Upon completion of sensing, each sensor may interrupt and pass data to *S-SEN*.

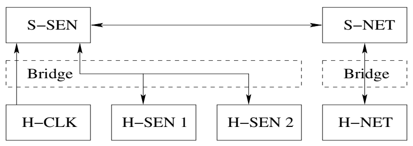

Figure 14. Multi-sensor system

We verify P_1 on the multi-sensor system. (For simplicity, hereafter, we only verify P_1 on sensor systems.) All components of the system, except the new bridge component, are reusable and their properties have been verified. Properties of the bridge component (not shown for conciseness) are formulated the same way as those of the bridge components in the basic system. They are verified using 10.24 seconds and 6.05 megabytes. The abstraction of the multi-sensor system for verifying P_1 is constructed from the component properties. P_1 was successfully verified on the abstraction using 0.1 seconds and 3.40 megabytes.

4.2.2 Verification of encryption-enabled sensor system

Development of new sensor systems may introduce new components. For instance, to develop a security enhanced sensor network, it is desired that some sensors in a sensor network be able to encrypt the sensor readings before transmitting the readings. Based on the requirement of such a sensor system, the system can be decomposed into its components as shown in Figure 15. A hardware encoder, *H-*

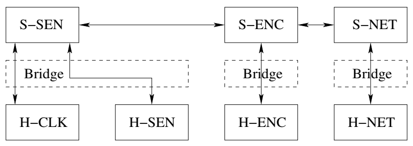

Figure 15. Encryption-enabled sensor

ENC and its software controller, *S-ENC* are introduced. In the system execution, *S-SEN* passes sensor readings to *S-ENC* which invokes *H-ENC* to encrypt the sensor readings.

The interface of *S-ENC* is defined as follows: input message types = {*Raw*, *Encoded_Ack*, *E_Intr*}, output message types = {*Raw_Ack*, *Encoded*, *E_Ret*}, and externally visible variables = {*ENC.Pending*}. The properties of *S-ENC* for verifying P_1 on the whole system are shown in Figure 16. The properties assert that *S-ENC* outputs encoded data repeatedly if it inputs raw data repeatedly and it properly handles the input and output handshakes. The assumptions assert that the environment correctly handles the handshakes with *S-ENC* and generates interrupts to *S-ENC* in response to its encoding requests. The interface of *H-ENC* and the properties of *H-ENC* for verifying P_1 are also formulated (not shown for conciseness). The properties of *S-ENC* are verified on its executable using 0.24 seconds and 3.57 megabytes while verification of *H-ENC* takes 0.22 seconds and 3.39 megabytes. A new bridge component con-

IfRepeatedly (Raw) **Repeatedly** (Encoded);

After (Raw) **Eventually** (Raw_Ack);
Never (Raw_Ack) **UnlessAfter** (Raw);
After (Raw_Ack) **Never** (Raw_Ack) **UnlessAfter** (Raw);

After (Encoded) **Never** (Encoded) **UnlessAfter** (Encoded_Ack);
Never (Encoded) **UnlessAfter** (E_Int);
After (Encoded) **Never** (Encoded) **UnlessAfter**(E_Intr);
Never (E_Ret) **UnlessAfter** (Encoded_Ack);
After (E_Ret) **Never** (E_Ret) **UnlessAfter**(Encoded_Ack);

After (E_Intr) **Eventually** (E_Ret);
Never (E_Ret) **UnlessAfter** (E_Intr);
After (E_Ret) **Never** (E_Ret) **UnlessAfter** (E_Intr);
After (ENC.Pending) **Never** (ENC.Pending) **UnlessAfter** (E_Ret);

Assumptions:
After (Raw) **Never** (Raw+E_Intr) **UnlessAfter** (Raw_Ack);

After (Encoded) **Eventually** (Encoded_Ack);
Never (Encoded_Ack) **UnlessAfter** (Encoded);
After (Encoded_Ack) **Never** (Encoded_Ack) **UnlessAfter** (Encoded);

After (ENC.Pending) **Eventually** (E_Intr);
Never (E_Intr) **UnlessAfter** (ENC.Pending);
After (E_Intr) **Never** (Raw+E_Intr) **UnlessAfter** (E_Ret);
After (E_Ret) **Never** (E_Intr) **UnlessAfter** (ENC.Pending);

Figure 16. Properties of software encoder

necting *S-ENC* and *H-ENC* is introduced. Its properties are verified using 0.18 seconds and 3.56 megabytes. The abstraction for verifying P_1 on the encryption-enabled sensor system is constructed from the properties of its components. P_1 is successfully verified on the abstraction using 0.13 seconds and 3.56 megabytes.

4.3 Integrated bottom-up and top-down verification of new components

Verification of new components exploits the interaction of bottom-up and top-down verification. New components may be introduced and verified in top-down development of new systems, such as *S-ENC* and *H-ENC*, and they may also be introduced and verified through bottom-up component development due to technology advances, such as new sensing and communication modules.

The new components can be further composed with existing components or among themselves to construct larger composite components bottom-up. For instance, *S-ENC*, *S-NET*, *H-ENC*, and *H-NET* can be composed into an encryption-enabled network component, *E-NET*, as shown in Figure 17. *S-NET* and *H-NET* have been verified bottom-

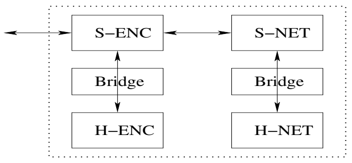

Figure 17. Encryption-enabled network

up as the family was created. *S-ENC* and *H-ENC* has been verified in top-down verification of the encryption-enabled sensor system. Based on their properties, *E-NET* is verified bottom-up. The interface of *E-NET* includes the following messages: *Raw* and *Raw_Ack* for interaction with other components and *E_Intr*, *N_Schd*, *R_Intr*, *E_Ret*, *N_Ret*, and *R_Ret* for specification of scheduling constraints. The properties of *E-NET* are shown in Figure 18. The properties

Properties:
IfRepeatedly (Raw) **Repeatedly** (HNET.flag);
IfRepeatedly (Raw) **Repeatedly** (HNET.flag=False);

After (Raw) **Eventually**(Raw_Ack);
Never (Raw_Ack) **UnlessAfter** (Raw);
After (Raw_Ack) **Never**(Raw_Ack) **UnlessAfter**(Raw);

Assumptions:
After (Raw) **Never** (Raw+E_Intr+N_Schd+R_Intr) **UnlessAfter** (Raw_Ack);
After (E_Intr) **Never** (Raw+E_Intr+N_Schd+R_Intr) **UnlessAfter** (E_Ret);
After (N_Schd) **Never** (Raw+E_Intr+N_Schd+R_Intr) **UnlessAfter** (N_Ret);
After (R_Intr) **Never** (Raw+E_Intr+N_Schd+R_Intr) **UnlessAfter** (R_Ret);

Figure 18. Properties of *E-NET*

assert that *E-NET* repeatedly transmits if there are inputs repeatedly and that it properly handles input handshakes. The assumptions assert that the environment correctly handles the handshakes with *E-NET* and respects the scheduling constraints of *E-NET*. The properties are successfully verified on an abstraction of *E-NET*, constructed from the verified properties of *S-NET*, *S-ENC*, *H-NET*, *H-ENC*, and the two bridge components. The verification takes 0.13 seconds and 3.55 megabyte. *E-NET* and its properties can then be reused in building new sensor systems.

4.4 Evaluation

Our approach to component-based co-verification is evaluated through comparing the time and memory usages for verifying P_1 on the three sensor systems: the basic system, the multi-sensor system, and the encryption-enabled sensor system using this approach with the time and memory usages for verifying these three systems using the basic translation-based approach in [22]. The comparison is shown in Table 2. ("-" denotes running out of memory, *CB*

	Usages	Basic	Multi	Encrypting
TB	Time (Sec)	31272.8	-	-
TB	Mem. (MB)	1660.62	-	-
CB	Time (Sec)	41.89	10.34	0.77
CB	Mem. (MB)	9.11	6.05	3.57

Table 2. Time and memory usage comparison

denotes the component-based approach, and *TB* denotes the translation-based approach.) All verification runs were conducted on a SUN workstation with dual CPUs at 1 GHZ and 2 GB physical memory. The time (or memory, respectively) usage for verifying a system using the component-based approach is the sum (or max) of the time (or memory) usages for verifying the new components and the abstraction. It can be observed that the component-based approach has order-of-magnitude reduction on the verification time and memory usages for verifying the basic sensor system. The reductions on the multi-sensor system and the encryption-enabled sensor system are more significant since the translation-based approach runs out of memory on both systems while the component-based approach achieves major reuse of verification efforts and only requires the verification of the new hardware, software, and bridge components and the abstractions of the two systems. The component-based approach requires the extra cost of abstraction construction and the manual effort of formulating component properties which, we believe, can be greatly reduced by domain knowledge and are compensated by being able to verify systems that cannot be verified, otherwise.

5 Related Work

Co-verification of embedded systems falls into two major categories: co-simulation and formal co-verification. Our approach belongs to the latter. Hardware/software co-simulation of embedded systems is supported by industrial tools such as Mentor Graphics Seamless [16] and academic projects such as Ptolemy [6]. Co-simulation does not provide exhaustive state space coverage. Various formal languages have been proposed for specifying embedded systems, such as Hybrid Automata [3], LOTOS [19], Co-design Finite State Machines (CFSMs) [5], and petri-net based languages such as PRES [8]. Hybrid automata and CFSMs have been directly model-checked while LOTOS and PRES have been verified via translation to directly model-checkable languages. Our approach differs by supporting specification of hardware or software components in their native languages and exploiting compositional structures of embedded systems for co-verification.

There has been much research [1, 2, 14, 4] on compositional reasoning of hardware and software systems. Our approach builds on the previous work and differs by applying compositional reasoning across hardware/software boundaries. It is enabled by the component model which unifies hardware IPs and software components and by translation-based co-verification [22] which provides a common formal semantic basis for conducting compositional reasoning.

6 Conclusions and Future Work

We have presented a component-based approach to hardware/software co-verification of embedded systems using model checking. This approach has great potential for building trustworthy embedded systems. It achieves major verification reuse and order-of-magnitude reduction on co-verification complexity, therefore, enabling co-verification of more complex embedded systems. As the next step, we plan to further automate our approach in system decomposition and property formulation, by leveraging domain knowledge such as composition patterns of embedded systems.

Acknowledgment

We gratefully acknowledge the contributions and help from James C. Browne, Robert P. Kurshan, and Vladimir Levin.

References

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

[2] R. Alur and T. Henzinger. Reactive modules. *FMSD*, 15(1), 1999.

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

[4] 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.

[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*.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

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

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

[10] 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.

[11] M. F. Jacome and H. P. Peixoto. A survey of digital design reuse. *IEEE Design and Test of Computers*, 18(3), 2001.

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

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

[14] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Cadence Design Systems Technical Reports*, 1999.

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

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

[17] C. Szyperski and et al. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 2002.

[18] D. E. Thomas and P. R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, 1991.

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

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

[21] 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.

[22] F. Xie, X. Song, H. Chung, and R. Nandi. Translation-based co-verification. In *Proc. of MEMOCODE*, 2005.