

# Component-based hardware/software co-verification for building trustworthy embedded systems <sup>☆</sup>

Fei Xie <sup>a,\*</sup>, Guowu Yang <sup>a</sup>, Xiaoyu Song <sup>b</sup>

<sup>a</sup> Department of Computer Science, Portland State University, Portland, OR 97207, USA

<sup>b</sup> Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97207, USA

Available online 27 September 2006

## Abstract

We present a novel component-based approach to hardware/software co-verification of embedded systems using model checking. Embedded systems are pervasive and often mission-critical, therefore, they must be highly trustworthy. Trustworthy embedded systems require extensive verification. The close interactions between hardware and software of embedded systems demand co-verification. 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.

© 2006 Elsevier Inc. All rights reserved.

**Keywords:** Component-based embedded systems; Component model; Components; Model checking; Compositional reasoning; Hardware/software co-verification; Verification reuse

## 1. Introduction

Embedded systems are pervasive in the infrastructure of our society for diverse tasks such as studying environmental phenomena, instrumenting and managing large-scale systems, and aiding security. An embedded system often consists of a generic processor, mission-specific hardware modules, and software modules that execute on the processor and interact with hardware modules.

Embedded systems are usually strictly constrained in computation, memory, bandwidth, and power. To lower production and deployment costs, embedded systems

are often equipped with slow processor, small memory, rudimentary radio, and limited battery. These constraints require that for a given mission, only the necessary hardware and software modules be loaded into an embedded system. This makes component-based development an appealing and appropriate approach to embedded system development. For instance, the well-known TinyOS (Hill et al., 2000) run-time system for networked sensors, an emerging type of deeply embedded systems, is component-based.

Embedded systems are often mission-critical, deployed in large quantity, and difficult to access after deployment. Therefore, they must be highly trustworthy. Embedded systems often support concurrency intensive operations such as simultaneous monitoring, computation, and communication. However, locks and monitors commonly used to safeguard concurrent operations are often not used in embedded systems due to computational costs. Thus,

<sup>☆</sup> This research was supported by Semiconductor Research Corporation, Contract RID 1356.001.

\* Corresponding author. Tel.: +1 503 725 2403; fax: +1 503 725 3211.  
E-mail addresses: [xie@cs.pdx.edu](mailto:xie@cs.pdx.edu) (F. Xie), [guowu@cs.pdx.edu](mailto:guowu@cs.pdx.edu) (G. Yang), [song@ece.pdx.edu](mailto:song@ece.pdx.edu) (X. Song).

to build trustworthy embedded systems, they must be extensively verified.

Due to strict design constraints of embedded systems, to achieve better performance, hardware and software modules must closely interact and the trade-off between hardware and software must be exploited. This demands hardware/software co-design and, therefore, hardware/software co-verification of embedded systems.

Model checking (Clarke and Emerson, 1981; Quielle and Sifakis, 1982) is a powerful formal verification method which has great potentials in hardware/software co-verification of embedded systems. 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 makes naive application of model checking intractable and requires state space reduction. Co-verification of an embedded system involves its hardware and software, which makes state space reduction more challenging.

Compositional reasoning (Abadi and Lamport, 1995) is a powerful state space reduction algorithm. Using compositional reasoning, 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. Compositional structures of embedded systems may greatly simplify application of compositional reasoning to hardware/software 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 (Jacome and Peixoto, 2001) (i.e., hardware components) and software components (Heineman and Council, 2001; Szyperski, 2002). 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 components, *bridge components*, are introduced, which inter-connect hardware and software components and bridge their semantics gaps.

Our approach to co-verification 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 model-checked while properties of a composite component are checked on its abstractions constructed from verified properties of its sub-components. A system is verified top-down as it is developed through recursive partitions into its components. The partitions reuse components as possible. Verified properties of the reused components are used in constructing the abstractions for verifying properties of the system or higher-level components. Our approach is based on translation-based co-verification (Xie et al.,

2005) where software and hardware modules of a system are translated into a formal model-checkable language, integrated, and model-checked. Translation-based co-verification provides a common formal semantics basis for conducting compositional reasoning across hardware/software boundaries and the basic mechanisms for verifying primitive components and abstractions of systems or composite components.

The contributions of our approach include the component model for embedded systems that unifies hardware IPs and software components, unified component property specification, and seamless integration of co-verification into component-based development of embedded systems. Our approach has great potentials in building trustworthy embedded 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 our work. We define the component model for embedded systems, which unifies 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

In this section, we first briefly introduce the component-based development of hardware and software. We then discuss our previous work on translation-based co-verification and on bottom-up verification of software components.

### 2.1. Component-based development

In both hardware and software industries, there is a common trend of developing systems via assembly of components (Jacome and Peixoto, 2001; Heineman and Council, 2001; Szyperski, 2002). (In hardware industry, component-based development is known as Intellectual Property (IP) based development.) A main objective of component-based development is to reuse design and development efforts. To achieve this objective, it is required that components capture reusable concepts in an application domain and have standard interfaces that export their functionalities. As verification becomes increasingly important in system development, it is also desired to reuse verification efforts.

### 2.2. Translation-based co-verification

In Xie et al. (2005), 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 hardware/software boundaries, 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.

We reduce co-verification complexity 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 formal models of hardware and software modules, and (3) compositional reasoning across the bridge module.

This translation-based approach has been implemented for co-verification of software designs in Executable UML (xUML) (Mellor and Balcer, 2002) and hardware designs in Verilog (Thomas and Moorby, 1991). The implementation integrates two translation-based model checkers: FormalCheck (Kurshan, 1998) and ObjectCheck (Xie et al., 2002), both of which are based on the COSPAN model checker (Hardin et al., 1996). FormalCheck is a commercial tool for hardware verification. ObjectCheck was developed in our previous work for verification of executable software designs in xUML. xUML has an asynchronous interleaving message-passing semantics. In xUML, a system consists of object instances which interact via asynchronous message-passing. A system execution is an interleaving of state transitions of these object instances, i.e., at any moment only one object instance progresses.

### 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 signals are generated when a procedure call returns or a message is received; (3) what variables in software modules are mapped to hardware signals; (4) what are the scheduling policies for software modules, for instance, interrupt priorities and preemption policies. Translation of the bridge specification depends on the software and hardware modules since it refers to semantic entities in both the software and hardware modules. (See Fig. 9 for an example bridge specification.)

### 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 have rigorous mappings to  $\omega$ -automata templates written in S/R (Hardin et al., 1996), the input formal language of the COSPAN (Hardin et al., 1996) model checker. (In S/R, both systems and properties are formulated as  $\omega$ -automata.) An example of such a template is

After( $e$ ) Eventually( $d$ )

where the *enabling* condition  $e$  and the *discharging* condition  $d$  are propositional logic predicates declared over semantic entities in hardware or software modules. The semantic meaning is that after each occurrence of  $e$  there eventually follows an occurrence of  $d$ . Although similar to the LTL formula  $G(e \rightarrow XF(d))$ , our property does not require a second  $d$  in case that the discharge condition  $d$  is accompanied by a second  $e$ , whereas an initial  $e$  is not discharged by an accompanying  $d$ . This asymmetry meets many requirements of software specification. (On account of this asymmetry, our property cannot be expressed in LTL.)

Our property specification language is linear time, with the expressiveness of  $\omega$ -automata (Kurshan, 1994). The property 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 specified in this language.)

### 2.3. Bottom-up verification of software components

In Xie and Browne (2003), 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 the abstraction if (1) it is related to the property to be checked on the composite component by cone-of-influence analysis (Kurshan, 1994), (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 Object-Check 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 Fig. 1. Under this architecture, the software components of an embedded system execute on generic processors while the hardware components are implemented as Application Specific Integrated Circuits (ASICs). The software components and hardware components interact through an embedded OS that also schedules the execution of the software components.

From this architecture, we can derive a unified component model shown in Fig. 2, which unifies hardware IPs and software components. 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 semantics gap between hardware and software components by propagating events across hardware/software boundaries. Software schedulers are not explicitly represented in this model. Instead, the scheduling constraints are integrated into the component model as assumptions of the components. The bridge components and the scheduling constraints together abstract the embedded OS. Three types of composite components may also be defined: *software components*, *hardware components*, and *hybrid components*. 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, therefore, also bridge components. This model essentially 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 that are defined on  $I$  and 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 interacting 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 design of software components, we adopt the model-driven development (Mellor and Balcer, 2002) and specify software components in a design-level executable 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  $S\text{-}SEN$ ), which controls a hardware sensor upon clock interrupts, is shown in Fig. 3. The dashed box denotes the component boundary. The incoming arrows denote input message types and the outgoing arrows denote output message types.  $S\text{-}SEN$  exports two variables:  $ADC.Pending$  and

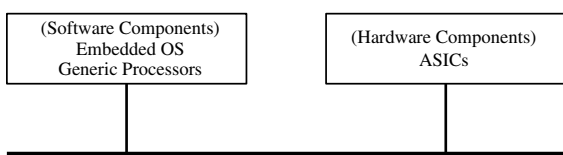


Fig. 1. Abstract Architecture.

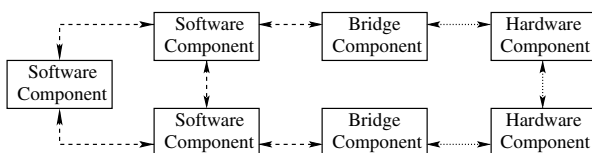


Fig. 2. Unified Component Model.

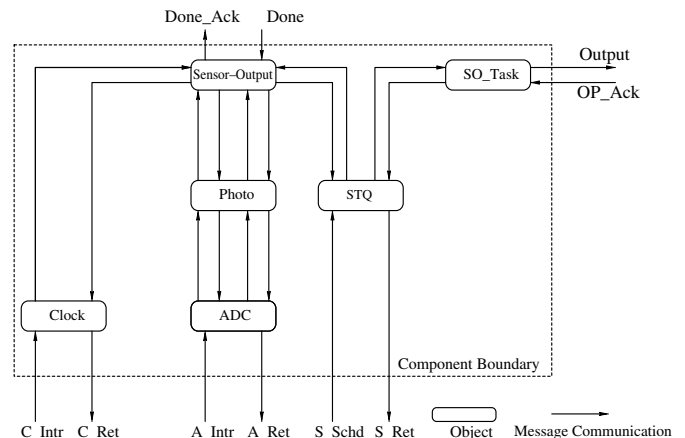


Fig. 3. Software sensor component.



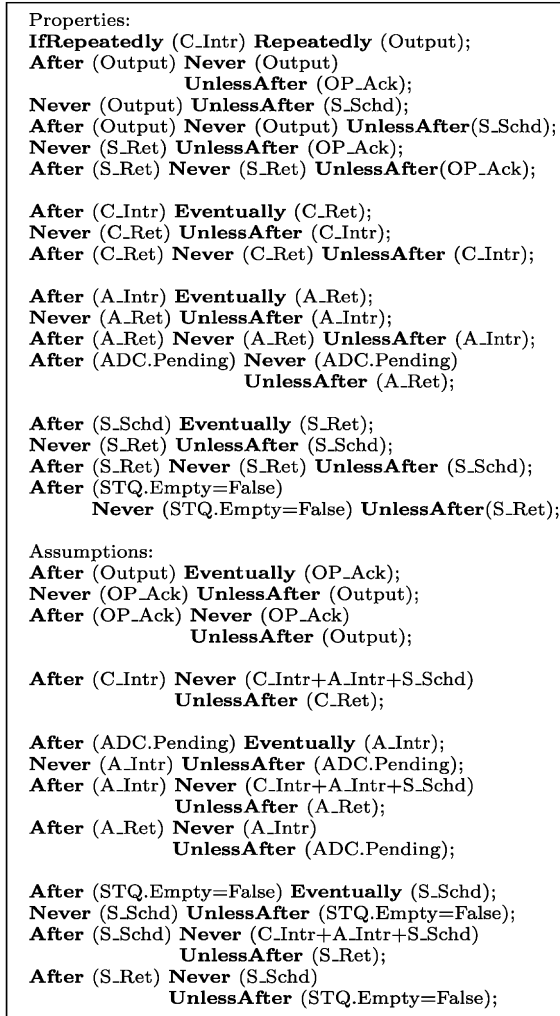


Fig. 4. Properties of software sensor.

*STQ.Empty*. A set of properties that have been verified on *S-SEN* are shown in Fig. 4. The properties assert that *S-SEN* repeatedly 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.

A software network component (denoted by *S-NET*) is shown in Fig. 5. It exports two variables: *NTQ.Empty* and *RFM.Pending*. The verified properties of *S-NET* are shown in Fig. 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*.

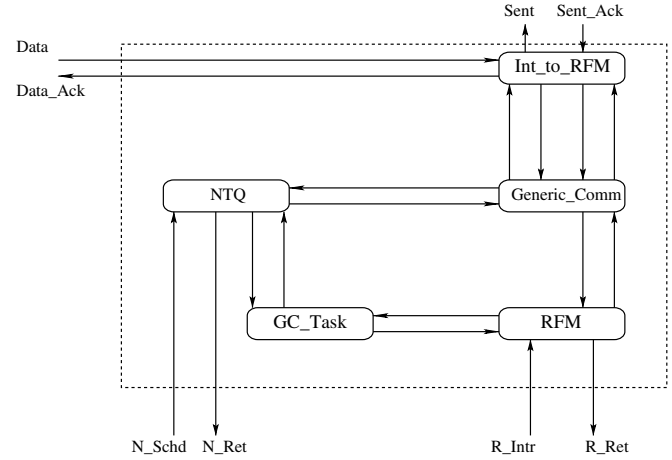


Fig. 5. Software network component.

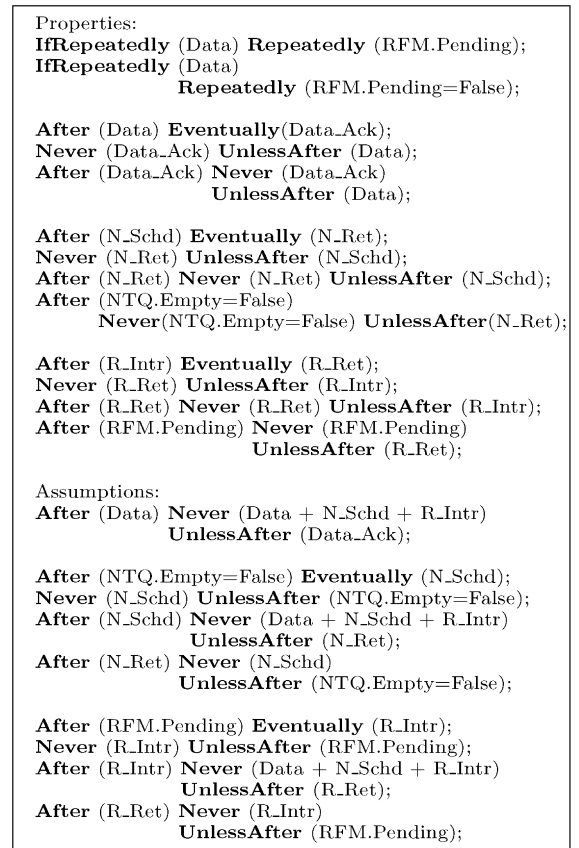


Fig. 6. Properties of software network.

### 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 signals that the hardware component imports from or exports to its environment. A hardware component communicates with its environment through the exported or imported signals in *I*. This interface semantics

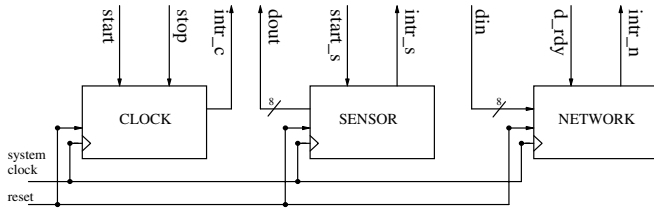


Fig. 7. Basic hardware components.

```

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);

```

Fig. 8. Properties of hardware components.

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 Fig. 7. The incoming arrows denote imported signals and the outgoing arrows denote exported signals. A set of verified properties of the three components are shown in Fig. 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.

### 3.1.3. Bridge components

Bridge components inter-connect hardware and software components. They extend the concept of bridge module (introduced in Xie et al. (2005) and briefly discussed 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 composite components include 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 interacting with hardware components and  $I_S$  is an asynchronous message-passing interface for interacting 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.

```

Bridge interface:
 $I_H = \{input\_signals = \{H-CLK.intr\_c, H-SEN.intr\_s\}$ 
 $output\_signals = \{H-SEN.start\}\}$ 

```

```

 $I_S = \{output\_msgs = \{S-SEN.C.Intr, S-SEN.A.Intr,$ 
 $S-SEN.S.Schd\}$ 
 $input\_msgs = \{S-SEN.C.Ret, S-SEN.A.Ret,$ 
 $S-SEN.S.Ret\}$ 
 $vars = \{S-SEN.ADC.On, S-SEN.STQ.Empty\}\}$ 

```

```

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))}

```

Fig. 9. A bridge component example.

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 Fig. 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 signals 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.

The bridge components not only abstract the hardware/software interfaces, but also abstract part of the embedded OS by providing necessary information about what are the software tasks that need to be scheduled to execute and their enabling conditions. Software schedulers are not explicitly specified in this component model. Instead, scheduling policies are specified as assumptions of the software components. The embedded OS determines the scheduling policies.

### 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,  $C_0 = (E_0, I_0, P_0), \dots, C_{n-1} = (E_{n-1}, I_{n-1}, P_{n-1})$ , as follows.  $E$  is constructed from  $E_0, \dots, E_{n-1}$  by connecting  $E_0, \dots, E_{n-1}$  through  $I_0, \dots, I_{n-1}$ .  $I$  may be a hardware interface, a software interface, or a hybrid hardware/software interface depending what types of components  $C_0, \dots, C_{n-1}$  are. Essentially,  $I$  includes the semantic entities from  $I_0, \dots, I_{n-1}$  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 illustrates this approach with its application to a suite 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 an embedded system family 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 Xie and Browne (2003). 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 notes (Hill et al., 2000) 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*, which have been defined in Section 3. We have verified *H-CLK*,

Table 1

Time and memory usages for verifying the properties of primitive components

Components	Time (s)	Memory (megabyte)
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

*H-SEN*, and *H-NET* with FormalCheck and we have also verified *S-SEN* and *S-NET* with ObjectCheck. The time and memory usages for these verification runs are shown in Table 1. The properties of the components are verified under their corresponding environment assumptions.

#### 4.1.2. Verification of a basic sensor system

After the primitive components of the sensor system family 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. Fig. 10 shows how the basic components are composed bottom-up into a basic system. *H-CLK* generates periodical 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, respectively) of *S-SEN* in Fig. 4, which are formulated on the clock interrupts generated by the bridge component between *S-SEN* and hardware and their responses from *S-SEN*, are assumptions (or properties) of the bridge component. The properties of the two bridge components are verified using 3.76 s and 6.03 megabyte and 0.66 s and 4.07 megabyte, respectively.

A system-level property  $P_1$  to be verified on the basic sensor system is shown in Fig. 11.  $P_1$  asserts that the basic sensor system transmits on the network repeatedly.

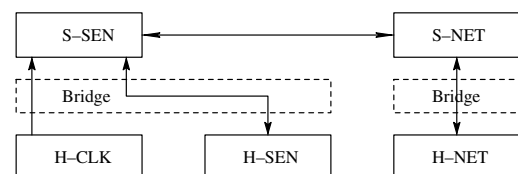


Fig. 10. A basic sensor system.

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

Fig. 11. Repeated transmission property.

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:

1. A system in the S/R language is constructed to abstract the sensor system. For each hardware or software component, a S/R process is introduced. The S/R process 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 through the simulated interfaces as how the components are composed in Fig. 10.
2. Starting from  $P_1$ , a cone-of-influence analysis is conducted on 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.

The constrained S/R system is the abstraction. Note that to include a related 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 via a simple model checking run; (2) the property does not involve 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 (1999); or (3) use the compositional reasoning rule proposed by Amla et al. (2001).

The abstraction constructed includes the properties of *S-SEN* in Fig. 4, the properties of *S-NET* in Fig. 6, the properties of the hardware components in Fig. 8, and the properties of the bridge components. The assumptions of *S-SEN* and *S-NET* are satisfied by the properties of the hardware components through the conversion of the bridge components. *S-SEN* and *S-NET* satisfy the handshake-related properties of each other. The properties of the hardware, software, and bridge components shown in Fig. 12 imply  $P_1$ . (Note that *S-SEN.Output* is mapped to *S-NET.Data*.) The implication relationship is established by model checking  $P_1$  on the abstraction, which takes 0.1 s and 3.40 megabyte.

The abstraction is conservative. If the property holds on the abstraction, it holds on the system; otherwise, the abstraction can be refined by verifying additional component properties and including them into the abstraction. If the property does not hold on the system, error trace

```
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);
```

Fig. 12. Comp. properties that imply  $P_1$ .

analysis and abstraction refinement are likely to uncover the cause. (See below for an example of bug detection.) Verification of additional properties are rarely needed for widely reused components.

This approach to abstraction construction extends the approach in Xie and Browne (2003) and constructs abstractions of embedded systems composed of hardware, software, and bridge components. It is enabled by the unified component model and the unified component 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 composite component.

The second property  $P_2$  to be verified on the basic system is shown in Fig. 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.

This abstraction need 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.) The verification of the new property of *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.

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

Fig. 13. No consecutive 1's property.



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 partitioned into hardware and software components. The partition is guided by domain knowledge and considers the available components. The interface of each component is defined and its properties are specified. If there is a component available that matches the interface and the 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 partitioned.

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 has been verified on the sub-component, nothing need 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 partitioned 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, for instance, a temperature sensor and a humidity sensor. The sensor system can be partitioned into hardware and software components as shown in Fig. 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 interrupts and passes data to *S-SEN*.

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 s and 6.05 megabyte. 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 s and 3.40 megabyte.

##### 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 partitioned into its components as shown in Fig. 15. A hardware encoder, *H-ENC* and its software controller, *S-ENC* are introduced. In 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 Fig. 16. The properties assert that *S-ENC* outputs encoded data repeatedly if it inputs raw data repeatedly and it correctly 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 s and 3.57 megabyte while verification of *H-ENC* takes 0.22 s and 3.39 megabyte. A new bridge component connecting *S-ENC* and *H-ENC* is introduced. Its properties are verified using 0.18 s and 3.56 megabyte. The abstraction for verifying  $P_1$  on the encryption-enabled sensor system is constructed from the properties of its components.  $P_1$  is

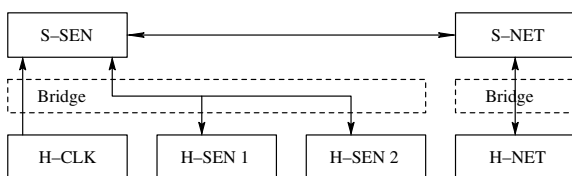


Fig. 14. Multi-sensor system.

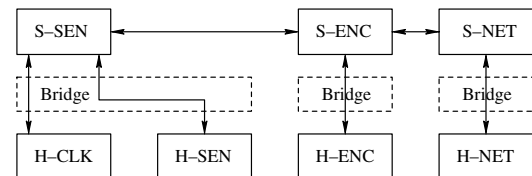


Fig. 15. Encryption-enabled sensor system.

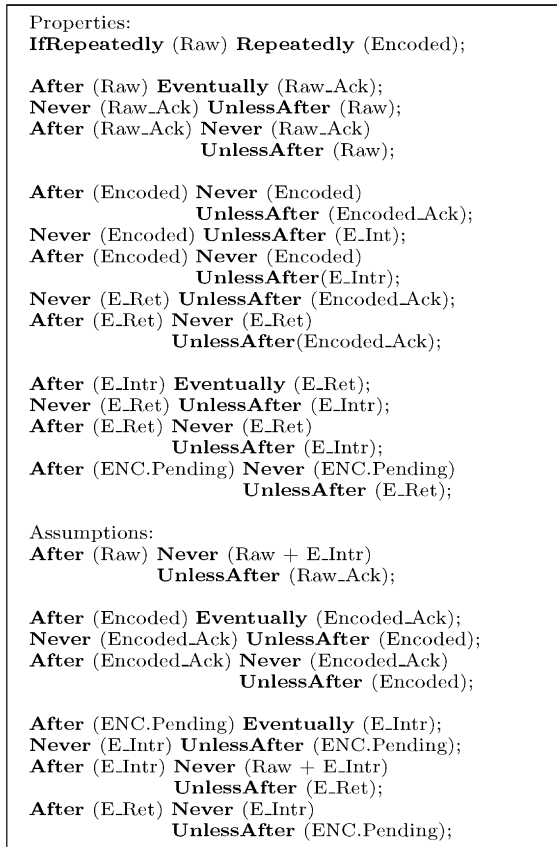


Fig. 16. Properties of software encoder.

successfully verified on the abstraction using 0.13 s and 3.56 megabyte.

#### 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 Fig. 17. *S-NET* and *H-NET* have been verified bottom-up as basic components. *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

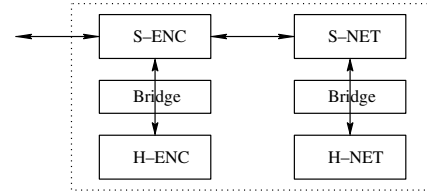


Fig. 17. Encryption-enabled network comp.

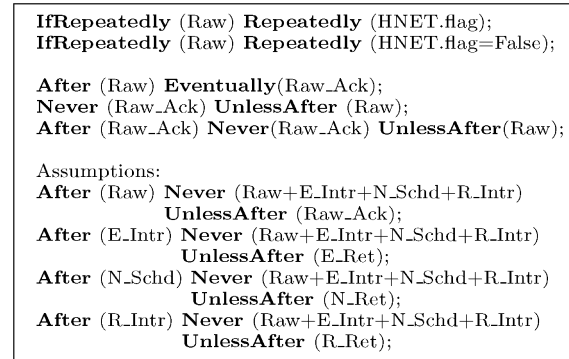
Fig. 18. Properties of *E-NET*.

Fig. 18. The properties 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 s and 3.55 megabyte. *E-NET* and its properties can then be reused in building new sensor systems.

#### 4.4. Evaluation

We evaluate our approach to component-based co-verification by 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 the three systems using the basic translation-based approach discussed in 2.2. The comparison is shown in Table 2. (*CB* denotes the component-based approach, *TB* denotes the translation-based approach, and

Table 2  
Time and memory usage comparison

	Usages	Basic	Multi	Encrypting
TB	Time (s)	31272.8	–	–
TB	Mem. (megabyte)	1660.62	–	–
CB	Time (s)	41.89	10.34	0.77
CB	Mem. (megabyte)	9.11	6.05	3.57

“-” denotes running out of memory.) All verification runs are conducted on a SUN workstation with dual CPUs at 1 GHZ and 2 GB physical memory. The time (or memory, respectively) usage of verifying a system using the component-based approach is the sum (or max) of the time (or memory) usages of 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 to verify 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

There has been much research on component-based hardware and software development (Jacome and Peixoto, 2001; Heineman and Councill, 2001; Szyperski, 2002). A fundamental problem in component-based development is how to derive the properties of compositions from the properties of components, including correctness properties, performance properties, real-time properties, etc. A well-known project targeting this problem in component-based software development is the PACC initiative from CMU/SEI: Predictable Assembly from Certifiable Components (CMU/SEI; Wallnau, 2003). The vision of PACC is that software components have certified properties (for example, performance) and the behavior of systems assembled from components is predictable. Our project shares the same vision as PACC while extending this vision by (1) defining a component model for embedded systems that unifies hardware and software components and (2) formally establishing the properties of an embedded system from the properties of its hardware and software components.

There has also been research on component-based software engineering for embedded systems such as (Crnkovic, 2005), focusing on embedded software. Due to the close interactions between hardware and software of embedded systems, there is a desire to reason about hardware and software components under a unified component model.

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* and academic projects such as *Ptolemy* (Berkeley). Co-simulation does not provide exhaustive state space coverage and may be insufficient for building highly trustworthy embedded systems.

Various formal languages have been proposed for specifying embedded systems, e.g., Hybrid Automata (Alur et al., 1996), LOTOS (van Eijk et al., 1989), Co-design Finite State Machines (CFSMs) (Balarin et al., 1996), and petri-net based languages such as PRES (Cortes et al., 2000). Hybrid automata and CFSMs have been directly model-checked. 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.

Formal co-verification with model checking provides exhaustive state space coverage while may suffer from state space explosion. There has been much research (Abadi and Lamport, 1995; Alur and Henzinger, 1999; McMillan, 1999; Amla et al., 2001) on compositional reasoning in model checking of hardware systems or software systems. Our approach builds on the previous work on compositional reasoning. It differs from the previous work in that it applies compositional reasoning across the hardware/software boundary. This is enabled by the component model for embedded systems which unifies hardware IPs and software components and the unified component property specification. This is also enabled by translation-based co-verification which provides a common formal semantics basis for compositional reasoning and provides the basic mechanisms for verifying primitive hardware or software components and abstractions of systems or composite components.

There has also been research on formal verification of IP-based hardware systems (Karlsson et al., 2004) and of component-based software systems (Chaki et al., 2003; Xie and Browne, 2003). Our work differs by co-verifying hardware and software components of embedded systems.

## 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 in building highly 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. Its effectiveness roots in seamless integration of verification into the component-based development lifecycle of embedded systems and exploitation of their compositional structures. 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.

## Acknowledgements

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

Levin. We also thank Haera Chung and Ranajoy Nandi for their help.

## References

- Abadi, Martin, Lamport, Leslie, 1995. Conjoining specifications. *TOPLAS* 17 (3), 507–534.
- Alur, Rajeev, Henzinger, Thomas, 1999. Reactive modules. *FMSD* 15 (1), 7–48.
- Alur, Rajeev, Henzinger, Thomas A., Ho, P.H., 1996. Automatic symbolic verification of embedded systems. *IEEE TSE* 22 (3), 181–201.
- Amla, Nina, Emerson, Allen. E., Namjoshi, Kedar S., Trefler, Richard. 2001. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In: *Proceedings of TACAS*.
- Balarin, F., Hsieh, H., Jurecska, A., Lavagno, L., Sangiovanni-Vincentelli, A., 1996. Formal verification of embedded systems based on CFSM networks. In: *Proceedings of DAC*.
- Berkeley. Ptolemy project. Available from: <<http://ptolemy.eecs.berkeley.edu/index.htm>>.
- Chaki, Sagar, Clarke, Edmund, Groce, Alex, Jha, Somesh, Veith, Helmut. 2003. Modular verification of software components in C. In: *ICSE*.
- Clarke, Edmund M, Allen Emerson, E., 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In: *Proceedings of Logic of Programs Workshop*.
- CMU/SEI. The PACC (Predictable Assembly from Certifiable Components) initiative. Available from: <<http://www.sei.cmu.edu/pacc>>.
- Cortes, Luis Alejandro, Eles, Petru, Peng, Zebo, 2000. Formal coverification of embedded systems using model checking. In: *Proceedings of EUROMICRO*.
- Crnkovic, Ivica, 2005. Component-based software engineering for embedded systems. In: *ICSE*.
- Hardin, Ronald H., Har'El, Zvi, Kurshan, Robert P., 1996. COSPAN. In: *Proceedings of CAV*.
- Heineman, George T., Councill, William T. (Eds.), 2001. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, MA.
- Hill, Jason, Szewczyk, Robert, Woo, Alec, Hollar, Seth, Culler, David E., Pister, Kristofer S.J., 2000. System architecture directions for networked sensors. In: *Proceedings of ASPLOS*.
- Jacome, Margarida F., Peixoto, Helvio P., 2001. A survey of digital design reuse. *IEEE Design and Test of Computers* 18 (3), 98–107.
- Karlsson, Daniel, Dles, Petru, Peng, Zebo, 2004. A formal verification methodology for IP-based designs. In: *Proceedings of DSD*.
- Kurshan, Robert P., 1994. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ.
- Kurshan, Robert P., 1998. *FormalCheck User Manual*. Cadence.
- McMillan, Ken L., 1999. *A methodology for hardware verification using compositional model checking*. Cadence Design Systems Technical reports.
- Mellor, Stephen J., Balcer, Marc J., 2002. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, Reading, MA.
- Mentor Graphics. Seamless. Available from: <<http://www.mentor.com>>.
- Quielle, Jean Pierre, Sifakis, Joseph, 1982. Specification and verification of concurrent systems in CESAR. In: *Proceedings of Symposium on Programming*.
- Szyperski, Clemens, 2002. *Component Software – Beyond Object-Oriented Programming*. Addison Wesley, Reading, MA.
- Thomas, Donald E., Moorby, Philip R., 1991. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Dordrecht.
- van Eijk, P.H.J., Vissers, C.A., Diaz, M. (Eds.), 1989. *The Formal Description Technique LOTOS*. Elsevier, Amsterdam.
- Wallnau, Kurt C., 2003. A technology for predictable assembly from certifiable components. Technical report, CMU/SEI-2003-TR-009.
- Xie, Fei, Browne, James C., 2003. Verified systems by composition from verified components. In: *Proceedings of ESEC/FSE*.
- Xie, Fei, Levin, Vladimir, Browne, James C., 2002. Objectcheck: a model checking tool for executable object-oriented software system designs. In: *Proceedings of FASE*.
- Xie, Fei, Song, Xiaoyu, Chung, Haera, Nandi, Ranajoy, 2005. Translation-based co-verification. In: *Proceedings of MEMOCODE*.