Component-Based Hardware/Software Co-Verification for Building Trustworthy Embedded Systems*

Fei Xie[a†], Guowu Yang[a], and Xiaoyu Song[b]

[a]Dept. of Computer Science, Portland State University, Portland, OR, 97207, USA

[b]Dept. of Electrical and Computer Engineering, Portland State University, Portland, OR 97207, USA

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.

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 [14] 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, 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 [8,22] is a powerful formal verification method which has great potentials in hardware/software co-verification of embedded

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

†Corresponding Author: Tel.: +1 503 725-2403; Fax: +1 503 725-3211.

Email addr.: xie@cs.pdx.edu (F. Xie), guowu@cs.pdx.edu (G. Yang), and song@ece.pdx.edu (X. Song)

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 [1] 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 [15] (i.e., hardware components) and software components [13,23]. 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 [29] 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 [15,13,23]. (In hardware industry, component-based development is known as Intellectual Property (IP) based development.) A main objective of component-based develop-

ment 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 [29], 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) [20] and hardware designs in Verilog [24]. The implementation integrates two translation-based model checkers: FormalCheck [18] and ObjectCheck [28], both of which are based on the COSPAN model checker [12]. 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 Figure 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 ω-automata templates written in S/R [12], the input formal language of the COSPAN [12] model checker. (In S/R, both systems and properties are formulated as ω-automata.) An example of such a template is

$$\texttt{After}(e)\ \texttt{Eventually}(d)$$

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 ω-automata [17]. 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 [27], 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 [17], (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(a). Under

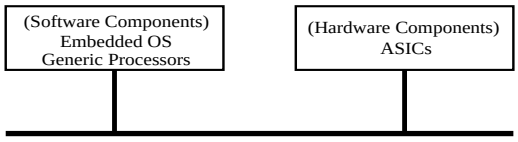

Figure 1. Abstract Architecture

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 Figure 2, which unifies hardware IPs and software components. An

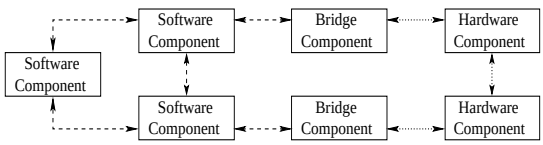

Figure 2. Unified Component Model

embedded system is composed of a set of compo-

nents. 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 [20] 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-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-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* repeatedly outputs and correctly handles the output handshakes. The assumptions assert that the environment of *S-SEN* correctly responses 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.

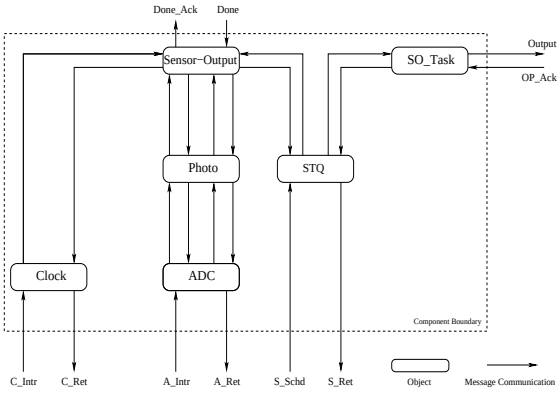

Figure 3. Software sensor component

A software network component (denoted by *S-NET*) is shown in Figure 5. It exports two variables: *NTQ.Empty* and *RFM.Pending*. The verified properties of *S-NET* are shown in Figure 6.

```
Properties:
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

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, responses 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

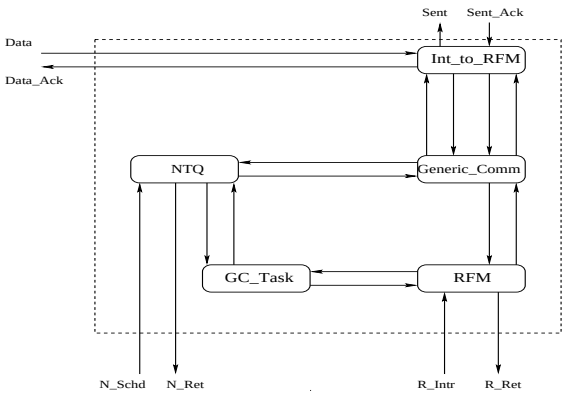

Figure 5. Software network component

```
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

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 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 denote imported

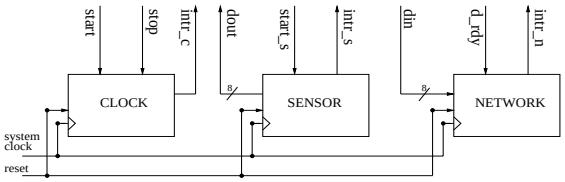

Figure 7. Basic hardware components

signals and the outgoing arrows denote exported signals. 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 (introduced in [29] 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 subcomponents. 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.

We illustrate the concept of bridge component by defining a bridge component that interconnects *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 signals but reversing their input/output directions. The executable specifi-

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))}

Figure 9. A bridge component example

cation 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 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, $C_0 = (E_0,\ I_0,\ P_0)$, ..., $C_{n-1} = (E_{n-1},\ I_{n-1}, P_{n-1})$, as follows. E is constructed from E_0, ..., E_{n-1} by connecting E_0, ..., E_{n-1} through I_0, ..., 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, ..., C_{n-1} are. Essentially, I includes the semantic entities from I_0, ..., 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 [27]. 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 [14] 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*, *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.

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

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

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. Figure 10 shows how the basic components are composed bottom-up into a basic system. *H-CLK* generates

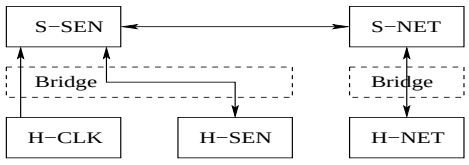

Figure 10. A basic sensor system

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 Figure 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 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

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

Figure 11. Repeated transmission property

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:

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 Figure 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 [19]; or (3) use the compositional reasoning rule proposed by Amla, et al [4].

The abstraction constructed 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. 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 Figure 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 seconds and 3.40 megabytes.

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 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 [27] 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.

```
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 second property P_2 to be verified on the basic system is shown in Figure 13. P_2 asserts

```
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

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. 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 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 interrupts and passes data to *S-SEN*.

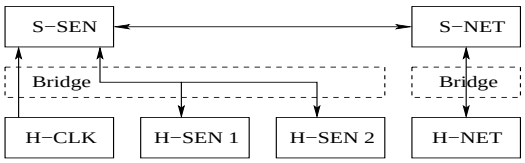

Figure 14. Multi-sensor system

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 partitioned into its components as shown in Figure 15. A hard-

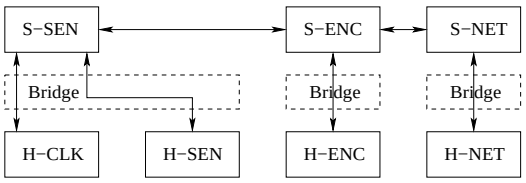

Figure 15. Encryption-enabled sensor system

ware 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 Figure 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 seconds and 3.57 megabytes while verification of *H-ENC* takes 0.22 seconds and 3.39 megabytes. A new bridge component connecting *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.

```
Properties:
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

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-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 fol-

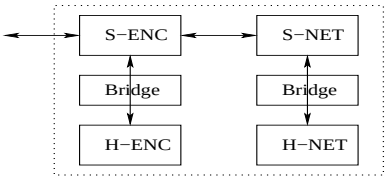

Figure 17. Encryption-enabled network comp.

lowing 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

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

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

Table 2
Time and memory usage comparison

	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

the component-based approach, *TB* denotes the translation-based approach, and "-" 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 [15, 13,23]. A fundamental problem in component-based development is how to derive the proper-

ties 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 [9,26]. 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 [11], 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 Seamless [21] and academic projects such as Ptolemy [6]. 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 [3], LOTOS [25], Co-design Finite State Machines (CFSMs) [5], and petri-net based languages such as PRES [10]. 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 [1,2,19,4] 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 [16] and of component-based software systems [7,27]. 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 knowledges 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. We also thank Haera Chung and Ranajoy Nandi for their help.

REFERENCES

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