The Journal of Systems and Software 83 (2010) 235–252

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Embedded architecture description language

Juncao Li [a], Nicholas T. Pilkington [a], Fei Xie [a,*], Qiang Liu [b]

[a] *Department of Computer Science, Portland State University, Portland, OR 97207, USA*
[b] *School of Software, Tsinghua University, Beijing 100084, PR China*

ARTICLE INFO

Article history:
Received 1 March 2009
Received in revised form 22 July 2009
Accepted 22 September 2009
Available online 30 September 2009

Keywords:
Embedded architecture description language
Components
Architectural patterns
Platform
Verification reuse
Hardware/software co-design
Co-simulation
Co-verification
Co-synthesis

ABSTRACT

In the state-of-the-art hardware/software (HW/SW) co-design of embedded systems, there is a lack of sufficient support for architectural specifications across HW/SW boundaries. Such an architectural specification ought to capture both hardware and software components and their interactions, and facilitate effective design exploitation of HW/SW trade-offs and scalable HW/SW co-verification. In this paper, we present the embedded architecture description language (EADL). EADL is based on a component model for embedded systems that unifies hardware and software components. EADL does not dictate execution and interface semantics of hardware and software components while supporting flexible platform-oriented semantics instantiation. EADL supports concise representation of embedded system architectures and also formulation of architectural patterns of embedded systems. Besides facilitating design reuse, architectural patterns also facilitate verification reuse via association of property templates with these patterns. Effectiveness of EADL has been demonstrated by its successful application in integrating component-based co-design, co-simulation, co-verification, and co-synthesis.

1. Introduction

In today's embedded system design, the boundary between hardware and software has become increasingly blurred: hardware and software closely interact and functionalities often migrate across the boundary. Due to stringent design constraints of embedded systems such as performance, power efficiency, and manufacture costs, hardware and software modules must closely interact and hardware/software (HW/SW) trade-offs must be effectively exploited. This demands HW/SW co-design where system functionalities are allocated across the HW/SW boundaries according to individual applications. HW/SW co-design often results in flexible embedded system architectures (including hardware and software), which are application-specific.

To reduce manufacture and operation costs, it is often required that for a given mission, only necessary hardware and software modules be loaded into an embedded system. This makes component-based development (CBD), developing systems via assembly of components, an appealing and appropriate approach to embedded system development. In both hardware and software industries, CBD (Jacome and Peixoto, 2001; Szyperski et al., 2002) is a common trend. (In hardware industry, CBD is also known as Intellectual Property (IP) based development.) A key objective of CBD, among others, is to reuse design and verification efforts.

Central to CBD of a hardware or software system is the support for architectural specification of this system which captures the components that form the system and their interactions. Further architectural supports include specification of architectural patterns (Shaw and Garlan, 1996) for system composition, which can facilitate both functional reuse and verification reuse. However, in the state-of-the-art HW/SW co-design of embedded systems, there is a lack of sufficient support to architectural specifications across HW/SW boundaries. This is largely due to the major semantic gap between hardware and software components. They are often designed in their native design/implementation languages whose execution and interface semantics differ significantly. For instance, hardware design may follow a synchronous clock-driven signal-passing semantics while software design may follow an asynchronous interleaving message-passing semantics.

In this paper, we present the embedded architecture description language (EADL) whose key features include:

1. EADL is based on a unified component model for embedded systems that unifies hardware and software components and bridges the HW/SW semantic gap.

* Corresponding author. Tel.: +1 503 725 2403; fax: +1 503 725 3211.
E-mail addresses: juncao@cs.pdx.edu (J. Li), nickp@cs.pdx.edu (N.T. Pilkington), xie@cs.pdx.edu (F. Xie), liuqiang@mail.tsinghua.edu.cn (Q. Liu).

2. EADL does not dictate execution and interface semantics of hardware and software components while supporting platform-oriented semantics instantiation.
3. EADL supports concise specification of embedded system architectures and also formulation of architectural patterns of embedded systems.
4. EADL integrates architectural design with assertion-based verification (ABV) (Maliniak, 2002). It supports association of properties (e.g., temporal correctness properties) with components and property templates with architectural patterns, to facilitate HW/SW co-verification using formal methods such as model checking (Clarke and Emerson, 1981; Quielle and Sifakis, 1982).

We have utilized EADL as the common representation for integrating component-based co-design, co-simulation, co-verification, and co-synthesis in the Embedded System Integrated Development Environment (ESIDE). We have instantiated EADL for two networked sensor platforms: one featuring xUML (Mellor and Balcer, 2002), a design-level software specification language and the other based on the TinyOS run-time environment (Hill et al., 2000). Furthermore, we have applied EADL in capturing architectures of networked sensor systems (Hill et al., 2000; Shnayder et al., 2005) based on the TinyOS platform and guiding their HW/SW co-verification. EADL has demonstrated its flexibility in platform-oriented semantics instantiation and effectiveness in capturing architectures and patterns and in simplifying formulation and verification of system and component properties.

The remainder of this paper is organized as follows. In Section 2, we provide the relevant background. In Section 3, we introduce the key language features of EADL. In Section 4, we discuss how EADL is instantiated for an embedded system platform. In Section 5, we present the architecture and functionalities of ESIDE. In Section 6, we discuss how architectural patterns are utilized to assist co-verification. In Section 7, we discuss our experiences with applying EADL as supported by ESIDE. In Section 8, we present related work. In Section 9, we conclude this paper and discuss future work.

2. Background

In this section, we first review a unified component model upon which EADL is developed. We then discuss a unified property specification language for hardware and software, which EADL integrates. A key goal of EADL is to support design for verification: tightly coupling design constructs with their verification counterparts such as their local temporal properties. This is central to improving verification scalability and efficiency. At last, we introduce TinyOS-based sensor systems which serve as a case study.

2.1. Unified component model for embedded systems

In (Xie et al., 2006), a unified component model has been developed for embedded systems that follow an abstract but representative architecture 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, a unified component model as shown in Fig. 2 has been derived, under which an embedded system is assembled from components. There are three types of primitive components: *software components*, *hardware components*, and *bridge components*. Bridge components interact with hardware (or software, respectively) components following hardware (or software) semantics and bridge the semantic gap between hardware and software components by propagating events across the HW/SW semantic boundary. The semantics of bridge components together with the hardware and software semantics abstract the processors, buses, and embedded OS of the targeted embedded system platform. (For more details about the bridge component concept, see Section 4.) Three types of composite components may also be defined: *software components*, *hardware components*, and *hybrid components*. A hybrid component contains both hardware and software sub-components and, therefore, bridge sub-components.

2.1.1. Components

A component C is a triple (E, I, P) where E is the design or implementation of C, I is an interface including the semantic entities for C to interact with its environment and/or for specification of properties of C, and P is a set of temporal properties that are defined on I and have been verified on E. Hardware, software, and bridge components differ in the specification languages for E and I, but share the same specification language for P. The specification languages for E and I are the native design/implementation languages for hardware, software, and bridge components (see Section 4 for examples). Each entry of P is a pair $(p, A(p))$ where p is a temporal assertion 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 includes components that interact with C, and may be different in each system. (See Section 2.2 for the unified property specification language.)

2.1.2. Composition

A composite component, $C = (E, I, P)$, is composed from a set of components, $C_0 = (E_0, I_0, P_0), \ldots, C_{n-1} = (E_{n-1}, I_{n-1}, P_{n-1})$, as follows. E is constructed from $E_0, \ldots, E_{n-1}$ by connecting $E_0, \ldots, E_{n-1}$ through $I_0, \ldots, I_{n-1}$. I may be a hardware interface, a software interface, or a hybrid hardware/software interface depending on what types of components $C_0, \ldots, C_{n-1}$ are. I includes the semantic entities from $I_0, \ldots, I_{n-1}$ that are needed for C to interact with its environment and/or for specification of properties of C. Properties of a composite component are established via verification on abstractions constructed from properties of its sub-components (Xie et al., 2006).

2.2. Unified property specification

Embedded systems control devices and physical or engineered systems that range from hearing aids and automobiles to the electrical power grid and global aviation infrastructure. They are often required to be highly trustworthy. Embedded systems often support concurrency intensive operations such as simultaneous monitoring, computation, and communication. However, locks and

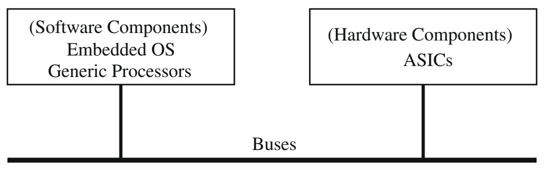


Fig. 1. Abstract architecture.

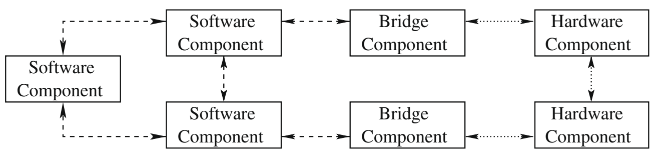


Fig. 2. Unified component model.

monitors commonly used to safeguard concurrent operations are often not used in embedded systems due to computational costs. Construction of highly trustworthy embedded systems requires extensive verification. Power and performance constraints of embedded systems require that hardware and software closely interact and the trade-off between hardware and software be effectively exploited. This requires co-verification.

Model checking (Clarke et al., 1999) is a formal verification method with great potential in HW/SW co-verification of embedded systems. A stumbling block to effective application of model checking to co-verification is the lack of support to unified property specification for hardware, software, and entire embedded systems, i.e., specifying properties of hardware, software, and entire embedded systems in a unified language. Unified property specification is indispensable to verification of system-level properties that span across the HW/SW boundaries. It also facilitates application of compositional reasoning (Clarke et al., 1999) to co-verification by simplifying utilization of properties of hardware and software components as assumptions when verifying other components.

In (Xie and Liu, 2007), we have developed a unified property specification language for HW/SW co-verification of embedded systems. This language, namely xPSL, builds on the IEEE Property Specification Language (PSL) (IEEE, 2005). It extends PSL to support specification of temporal assertions over both hardware and software events. The HW/SW semantic gap is filled by formalizing the semantics of hardware and software events and their temporal correlations based on translation of hardware and software semantics to a common formal semantic basis, in our case, the ω-automata semantics (Kurshan, 1994). While PSL supports both LTL (Pnueli, 1977) and CTL (Clarke and Emerson, 1981) temporal operators, xPSL inherits the linear-time subset of the PSL temporal operators and is, therefore, fully subsumed by ω-automata in expressiveness. xPSL is fully compatible with PSL and readily supports ABV. xPSL facilitates verification reuse: properties of hardware and software components in xPSL can serve as abstractions of the components in system-level verification and can be reused across multiple systems if the components are reused.

In this paper, we specify properties using a set of intuitive temporal templates based on xPSL and ω-automata, as shown in Fig. 3, which have intuitive meanings and also rigorous mappings to property templates written in S/R, the input formal language of the COSPAN model checker (Hardin et al., 1996). (Note that in S/R, both systems and properties are formulated as ω-automata.)

```
Always/Never (f)
After (e) Always/Never (f) [Unless[After] (d)]
After (e) Always/Never (f) [Until[After] (d)]
Always/Never (f) Unless[After] (d)
Always/Never (f) Until[After] (d)

After (e) Eventually (f) [Unless (d)]
Eventually (f) [Unless (d)]
IfRepeatedly (e) Repeatedly/Eventually (f)
IfRepeatedly (e) EventuallyAlways (f)
After (e) EventuallyAlways (f) [Unless (d)]
EventuallyAlways (f)
EventuallyAlways (f) Unless (d)
After (e) Repeatedly (f) [Unless (d)]
Repeatedly (f) [Unless (d)]
IfEventuallyAlways (e) Repeatedly/Eventually (f)
IfEventuallyAlways (e) EventuallyAlways (f)
```

Fig. 3. A list of available property templates.

An example of such templates is After(e)Eventually(d) where the *enabling* condition e and the *discharging* condition d are Boolean propositions declared over semantic entities of hardware or software. The semantic meaning is that after each occurrence of e there eventually follows an occurrence of d. The formal semantics of a property instantiating this template can be precisely defined based on the mappings from the hardware and software semantics to the semantics of S/R and the mapping of this template to a template written in S/R. The property can be automatically translated into S/R based on these mappings.

2.3. TinyOS-based sensor systems

TinyOS-based sensor systems are designed to: (1) run on limited hardware resources known statically, (2) handle events (through hardware interrupts) from the environment, (3) assure reliability for long-lived applications, and (4) satisfy soft real-time requirements. The native programming language of TinyOS is nesC (Gay et al., 2003), a dialect of the C language.

The key design features of TinyOS and nesC are as follows:

- *Component-based specification:* Systems in nesC are built by writing and assembling components. Components have two types: modules (analogous to primitive components) which provide application code implementing one or more component interfaces, and configurations (analogous to composite components) which are compositions of other components.
- *Static*: In nesC, there is no dynamic memory allocation and the function call graph is fully known at compile time. Therefore, all components are static.
- *Concurrency and atomicity:* TinyOS supports two execution priorities, tasks at the lower priority and events at the higher priority. Atomic blocks inside tasks can be defined using the nesC keyword “atomic”.
- *Encapsulation and compilation:* In nesC, components do not completely encapsulate their sub-components. Multiple components are allowed to share a common sub-component. Code in nesC is compiled into C code and all component instances with the same type are compiled into the same copy of C code.
- *HW/SW interfaces:* Hardware platforms of TinyOS are not component-based. TinyOS provides direct function calls such as *inp* and *outp* to interact with hardware while hardware can interrupt software.

3. Key language features of EADL

In this section, we introduce the key language features of EADL. These features, although resembling those of existing architectural description languages, are specially designed for co-design and geared for facilitating co-simulation and co-verification. Component and architecture specifications provide a unified and componentized view of hardware and software while being separated from hardware and software specific semantic details. EADL allows these semantic details be provided by the embedded system platforms. EADL also supports architecture-sensitive property specification, which enables integration of architectural design with ABV: embedding temporal property specification into architectural constructs on various levels.

3.1. Component interfaces

To support architectural specifications, EADL refines the unified component model to accurately capture structures in both component interfaces and component interactions.

3.1.1. Events

EADL employs the event concept to abstract all concrete hardware or software interaction mechanisms: signals, messages, function calls, etc. The event semantics are only precisely defined when EADL is instantiated for a specific embedded system platform (see Section 4). Events in an embedded system can be of different semantics due to the differences between hardware and software semantics. This enables EADL to span across HW/SW boundaries.

3.1.2. Ports

EADL employs the port concept to group events that together realize a certain functionality. Depending on whether a component is providing or utilizing the functionality, the port can be a "provides" or "uses" port in the component interface specification. Each event in a port has an "input" or "output" direction. Whether an event in a port is an input or output to a component also depends on whether the port is provided or used. If a component provides a port, its events conform to the directions as specified in the port; otherwise, its events reverse the directions.

Fig. 4 shows a software sensor component, *SW_Sensor*, in the context of a hybrid sensor component, *HB_Sensor*. The interface of *SW_Sensor* uses three ports *Clock*, *ADC*, and *STQ* and provides one port *SendRcv*. The EADL interface specification of *SW_Sensor* is shown in Fig. 5. The events in these ports are software messages. Fig. 5 also includes the interface specification for a hardware sensor component, *HW_Sensor*. It provides a single port in its interface. The events in this port are hardware signals.

In EADL, ports serve as the basic unit for design and verification reuse. Besides events, a port can also include properties formulated on these events as shown in Fig. 6. These properties are xPSL assertions on the functionality of this port and are categorized into two sets: properties of the port provider (a.k.a. "provides assertions") and properties of the port user (a.k.a. "uses assertions"). The two sets of properties often serve as the assumptions of each other. When a port is reused in a component, depending on whether it is provided or used, the corresponding set of properties are verified on the component. Ports with their properties are put into a library for reuse in defining components, component templates, and architectural patterns.

3.1.3. Composition

In EADL, components are connected on the more abstract port level, instead of the detailed event level. As in Fig. 4, a "connection" links two components through ports of the same type but reversed directions.

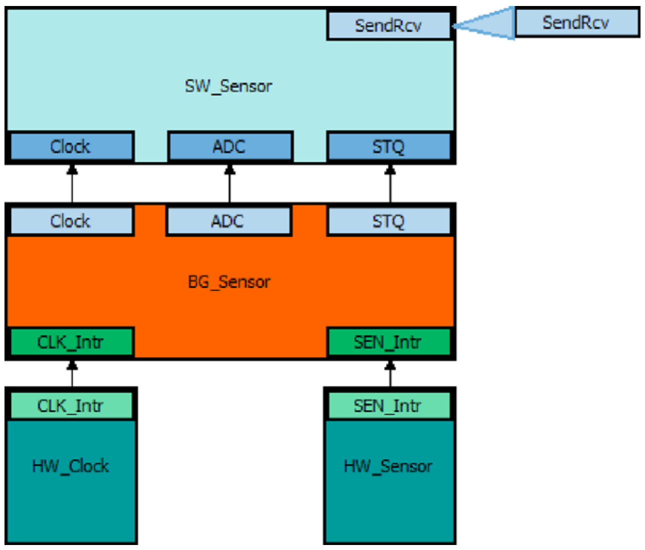


Fig. 4. Software sensor component in context of hybrid sensor component (this screenshot is exported by VisualEADL, a visual modeling toolkit for EADL).

Instead of introducing an explicit concept of connector in EADL, we treat connectors as components that connect other components together. Complex connections among components such as one-to-many connections can be realized by introducing an additional component that have one-to-one connections with all these components while realizing the one-to-many logic in its implementation. Components of an embedded system may follow several different hardware or software interface semantics, which would require many different types of explicit connectors. This would unnecessarily complicate the language definition. Furthermore, treating connectors as components also avoids treating connectors differently in simulation and verification.

3.2. Component-based system architectures

EADL specifies the architecture of a composite component (a system is a composite component) via specifying its configuration, which consists of its sub-components and their connections. The configuration of *HB_Sensor* is shown visually in Fig. 4 and textually in Fig. 5. Besides the sub-components and their connections, port maps are defined between the ports of the composite component and the ports of its sub-components, for instance, *HB_Sensor.SendRcv* is mapped to *SW_Sensor.SendRcv*. For a primitive component, the configuration is replaced by the path to its source file.

3.3. Embedded system architectural patterns

The architecture of a single system or composite component is described by its sub-components and their connections. Common patterns often exist among architectures of such systems or components. EADL provides two mechanisms by which to capture and reuse these commonalities.

3.3.1. Templates

While the architecture of a system or component is captured as a configuration which is based on composition of components, an architectural pattern is captured as a configuration template which is based on composition of component templates. Abstraction of patterns from the component/system architectures is based on abstraction of component templates from components. A component template is a skeleton for components, which captures the parametrized interface shared by these components, the common set of variables of these components, and the templates for properties of these components. The property templates are defined over the parametrized interface and the variable set in the component template. As the component template is instantiated into a component, the property templates are instantiated into component properties.

3.3.2. Patterns

Abstraction of architectural patterns from component or system architectures is based on abstraction of component templates from components. An architectural pattern consists of: (1) a partial description of the interface for a component or system following this pattern, which is made up of ports, (2) a configuration template, from which the configuration of the component or system is instantiated, and (3) property templates specified on the interface and the configuration template, from which properties of the component or system are instantiated. The configuration template consists of concrete components and component templates, and their connections.

We illustrate the architectural pattern concept with a simple but representative pattern of embedded systems, the *SourceToSink* pattern, as shown in Fig. 6. There are two component templates

```
hybrid component HB_Sensor {
  interface {
    provides SendRcv;
    mapping(SendRcv, SW_Sensor.SendRcv); }
  configuration {
    component SW_Sensor; component HW_Clock;
    component HW_Sensor; component BG_Sensor;
    connection(HW_Clock.CLK_Intr, BG_Sensor.CLK_Intr);
    connection(BG_Sensor.Clock, SW_Sensor.Clock);
    connection(BG_Sensor.ADC, SW_Sensor.ADC);
    connection(HW_Sensor.SEN_Intr, BG_Sensor.SEN_Intr);
    connection(BG_Sensor.STQ, SW_Sensor.STQ); } }

software port SendRcv {
  events { output message Send; input message Send_Ack; } }
software port Clock {
  events { output message C_Intr; input message C_Ret; }
  properties { provides assertion CLK_Rpt_Msg
                  p1: Repeatedly_(C_Intr) } }
software port ADC {
  events { input boolean On;
           output message A_Intr; input message A_Ret; } }
software port STQ {
  events { input boolean Empty;
           output message S_Schd; input message S_Ret; } }

hardware port CLK_Intr {
  events { output signal { wire intr_c; }; }
  properties { provides assertion CLK_Rpt_Sig
                  p1: Repeatedly_(intr_c) } }
hardware port SEN_Intr {
  events { output signal { wire intr_s; };
           input signal { wire start_s; }; } }

software component SW_Sensor {
  interface {
    provides SendRecv; uses STQ; uses ADC; uses Clock; }
  configuration { source("SW_Sensor.xuml"); } }

hardware component HW_Clock {
  interface {
    provides CLK_Intr; }
  configuration { source("HW_Clock.v"); } }
hardware component HW_Sensor {
  interface {
    provides SEN_Intr;}
  configuration { source("HW_Sensor.v") ;} }

bridge component BG_Sensor {
  interface {
    provides STQ; provides ADC; provides Clock;
    uses CLK_Intr; uses SEN_Intr; }
  configuration { source("BG_Sensor.bsl"); }
  properties { dependency Clock.CLK_Rpt_Msg: CLK_Intr.CLK_Rpt_Sig } }
```

Fig. 5. EADL specification for hybrid sensor component.

defined, *Source* and *Sink*. Their interfaces are defined through reuse of the port *SendRcv*: *Source* provides the port while *Sink* uses it. The two component templates are connected via this common port. This pattern can be instantiated multiple times in a system, which yields savings in design time and system complexity.

3.4. Architecture-sensitive property specification

In EADL, properties are incorporated in component hierarchies and associated with the relevant components. For a component, its properties can be specified on different semantic levels: port, interface, component, and pattern. Port-level and interface-level properties characterize how the component interacts with its environment. Component-level properties characterize the internal workings of the component, e.g., how its sub-components interact with each other, or the couplings between its internal and environment via its interface, e.g., its state changes in response to outside events. Pattern-level properties or property templates characterize common properties of components of similar architectures. In Fig. 6, *RcvPort.Sender_Handshake* is a port-level property, *Sink_Data_PT* is an interface-level property, and *Data_PT* is a pattern-level property.

Property dependencies can be specified across different semantic levels. For instance, *Sink_Data_PT* depends on *RcvPort.Sender_Handshake*, i.e., *Sink_Data_PT* holds assuming *RcvPort.Sender_Handshake* holds. When an assumption is not visible in scope, this dependency can be specified at a higher semantic level. For instance, in Fig. 5, the dependency of a port-level property *Clock.CLK_Rpt_Msg* on another

```
software port SendRcv {
  events { output message Send; input message Send_Ack; }
  properties {
    provides assertion Sender_Handshake: Receiver_Handshake_S
      p1: After_Never_UnlessAfter_(Send, Send, Send_Ack)
    uses assertion Receiver_Handshake_S: Sender_Handshake
      p1: Never_UnlessAfter_(Send_Ack, Send)
      p2: After_Never_UnlessAfter_(Send_Ack, Send_Ack, Send)
    uses assertion Receiver_Handshake_L: Sender_Handshake
      p1: After_Eventually_(Send, Send_Ack) } }

software template Source {
  interface { provides SendRcv as SendPort; }
  properties {
    assertion Src_Data_PT: SendPort.Receiver_Handshake_S,
                           SendPort.Receiver_Handshake_L
      p1: Repeatedly_(SendPort.Send) } }

software template Sink {
  boolean DataConsumptionFlag;
  interface { uses SendRcv as RcvPort; }
  properties {
    assertion Sink_Data_PT: RcvPort.Sender_Handshake
      p1: IfRepeatedly_Repeatedly_(RcvPort.Send,
                                   DataConsumptionFlag = TRUE)
      p2: IfRepeatedly_Repeatedly_(RcvPort.Send,
                                   DataConsumptionFlag = FALSE) } }

software pattern SourceToSink {
  configuration { template Source; template Sink;
    connection(Source.SendPort, Sink.RcvPort); }
  properties {
    assertion Data_PT
      p1: Repeatedly_(Sink.DataConsumptionFlag = TRUE)
      p2: Repeatedly_(Sink.DataConsumptionFlag = FALSE) } }
```

Fig. 6. An example architectural pattern.

port-level property *CLK_Intr. CLK_Rpt_Sig* is explicitly declared at the component level.

Architecture-sensitive property specification in EADL facilitates compositional reasoning in model checking and complexity reduction in simulation. Placing properties on right semantic levels helps generate succinct proof obligations for compositional reasoning and explicit property dependency helps prevent circular reasoning. When simulating a system, properties on certain ports, interfaces, or components can be enabled or disabled to avoid tracing irrelevant properties and enable quick diagnosis of property violations.

4. Platform-oriented instantiation of EADL

4.1. Embedded system platform

Embedded systems are often domain-specific. An emerging trend in the industry is to supply domain-specific platforms for embedded systems. Such a platform includes processors, buses, and embedded OS for developing embedded systems of a given domain. The platform also provides reusable hardware and software components and common architectural patterns of this domain. A key design goal of EADL is to support architectural specification of embedded systems based on various platforms. To achieve this goal, we design EADL to support platform-oriented instantiation.

To simplify system design, simulation, and verification, our platform concept hides details of processors, buses, and embedded OS via definition of a platform-specific bridge specification language (BSL). The semantics of hardware, software, and bridge components abstract processors, buses, and embedded OS. With this abstraction, a platform for an application domain consists of: (1) software, hardware, and bridge design/implementation languages, (2) compiler support for simulation, verification, and deployment under these languages, and (3) libraries of reusable ports, architectural patterns, and hardware, software, and hybrid components.

4.2. Instantiation of EADL

EADL is designed as an architectural extension for the hardware, software, and bridge design/implementation languages and it gains complete semantics when coupled with these languages. A platform provides the semantics needed for instantiating EADL for an application domain, as shown in Fig. 7. The software, hardware, and bridge semantics determine the semantics of the events in the interfaces of software, hardware, and bridge components specified using EADL. The semantics of the events in turn complete the semantics of xPSL since xPSL provides the temporal operators, but does not dictate the semantics of the events, i.e., the boolean propositions.

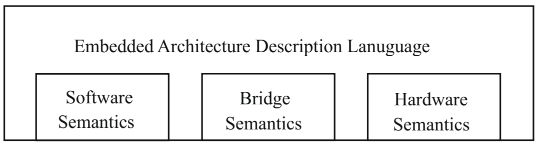


Fig. 7. Instantiation of EADL.

4.2.1. Instantiation on sensor system platform with software in xUML

The components in Fig. 5 and the pattern in Fig. 6 are based on the xUML platform. In order to support high-level design, we adopt the model-driven development (Mellor and Balcer, 2002) for software components, and specify the design E in xUML (Mellor and Balcer, 2002), an executable dialect of UML. The interface I of a software component can include two types of events: a set of input and output messages and a set of exported variables in E. The component communicates with its environment via asynchronous message-passing. The variables in I are mapped to hardware signals and/or utilized in specifying component properties and scheduling constraints. This interface semantics is determined by the asynchronous interleaving message-passing semantics of xUML.

For a hardware component, we specify the design E in Verilog (Palnitkar, 2003). The interface I consists of a set of variables that the hardware component imports from or exports to its environment. The component communicates with its environment synchronously via the variables in I. This interface semantics is determined by the synchronous clock-driven semantics of Verilog.

Bridge components inter-connect hardware and software components. The interface I of a bridge component is a pair (I_H, I_S). I_H is a synchronous shared-variable interface for interactions with hardware components and I_S is an asynchronous message-passing interface for interactions with software components. The interface of the bridge component is determined by the hardware and software components it connects. The design E of a bridge component is formulated in a platform-specific BSL (Xie et al., 2006). This language specifies (1) how hardware signals are mapped to software messages, (2) how software variables are mapped to hardware signals, (3) interrupt priorities, and (4) messages that initiate software tasks. The design of the *BG_Sensor* component in Fig. 5 is specified in this language as shown in Fig. 8.

In EADL, component properties are as integral a part of a component as its design and interfaces. To specify properties of the hardware, software, and bridge components, xPSL also needs to be instantiated on their semantics. This instantiation completes the semantics of the event layer of xPSL which is used to specify system events, i.e., the boolean propositions. The instantiation of xPSL on this platform allows specification of events over software messages, software variables, hardware signals, and hardware registers. For instance, in Fig. 6, the *Sink_Data_PT* assertion of the template *Sink* involves two software events: one over the message *RcvPort.send* and the other over the variable *DataConsumptionFlag*. The property *CLK_Rpt_Sig* on the *CLK_Intr* port in Fig. 5, *Repeatedly (intr_c)*, asserts over a hardware event *CLK_Intr.intr_c* requiring that the clock interrupts repeatedly.

4.2.2. Instantiation on tinyOS sensor system platform

Our second platform was created while we were re-engineering the networked sensor systems included in the TinyOS (Hill et al., 2000) distribution. In this platform, for a software component, the design E is specified in nesC (Gay et al., 2003), a native programming language for sensor software that customizes the C language. The interface I has one type of event: functions. The component communicates with its environment through function calls. This interface semantics is determined by the asynchronous event-driven call-return semantics of TinyOS/nesC. For a hardware component, we adopt the same specification as in the xUML Platform. For a bridge component, its software interface I_S is now function-based. Its specification E is specified in the BSL developed in (Hao et al., 2009), which employs *transactors* to propagate events across the HW/SW boundary. Fig. 9 illustrates the transactor concept. A transaction invoked by a software event will generate a sequence of hardware signals. The transaction invoked from the hardware side is implemented through hardware interrupts. The instantiation of xPSL on this platform allows specification of events over software function calls, software variables, and hardware signals. For instance, a property *After(HW_Timer.INTR) Eventually(SW_Timer.Fire.call)* can be asserted on the first hardware transactor in Fig. 9, which is trivially satisfied according to the semantics of transactors.

Fig. 10 shows the CodeBlue (Shnayder et al., 2005) architectural pattern for medical sensor systems and Fig. 11 shows its EADL specification which includes two concrete components and two component templates. In particular, *T_HB_Sensor* is a hybrid component template which can be instantiated multiple times.

A property template, *CB_P1*, is associated with the CodeBlue pattern. It asserts that after the coordinator receives a query with *SRC* as the sensor to be queried, *SINK* as the requester of the sensor reading, and *T* as the threshold for reporting the sensor reading, the coordinator will eventually report an above-threshold sensor reading to the requester unless the request is canceled. It has an assumption that after the query is received, the hardware sensor reading eventually reaches and stays above the threshold.

5. Embedded system integrated development environment

The architecture of ESIDE is shown in Fig. 12. The key features of ESIDE are derived from the four major stages of HW/SW co-development of embedded systems: co-design, co-simulation, co-verification, and co-synthesis.

System development using ESIDE begins with the selection of a platform, which determines the execution and interface semantics of hardware and software components. The platform also supplies libraries of reusable design constructs such as components and architectural patterns. The HW/SW co-design of an embedded system using ESIDE emphasizes component-based architectures and design-time specification of system and component properties. Placing these concerns at the forefront of the design process lessens the barriers to component-based co-simulation and co-verification. Highly accessible co-simulation and co-verification capabilities tighten the validation feedback loop, allowing for much earlier error detection. Throughout the co-design, the developer periodically validates his or her design through

```
/* Hardware interrupt to software message mappings */
(CLK_Intr.intr_c → Clock.C_Intr)   (SEN_Intr.intr_s → ADC.A_Intr)

/* Software variable to hardware signal mappings */
(ADC.On → SEN_Intr.start_s)

/* Interrupt priorities */
Priorities(CLK_Intr.intr_c, SEN_Intr.intr_s) = {0, 0}

/* Messages for initiating software tasks and their enabling conditions */
SchdSet = {(STQ.S_Schd | (STQ.Empty=False))}
```

Fig. 8. Design of *BG_Sensor Component*.

```
Transactor {
  /* Software transactors */
  void outp(uint_8 val, uint_8 address) {         uint_8 inp(uint_8 address) {
       PSEL = 1;                                       PSEL = 1;
       PADDR = address;                                PADDR = address;
       PWDATA = val;                                   PWRITE = 0;
       PWRITE = 1;                                     @ (posedge PCLK);
       @ (posedge PCLK);                               PENABLE = 1;
       PENABLE = 1;                                    @ (posedge PCLK);
       @ (posedge PCLK);                               PSEL = 0;
       PSEL = 0;                                       PENABLE = 0;
       PENABLE = 0;                                 return PRDATA;
  }                                               }

  /* Hardware transactors */
  HW_Timer.INTR ⇒ SW_Timer.Fire()    1;
  HW_ADC.INTR   ⇒ SW_ADC.DataReady()    1;
}
```

Fig. 9. Transactors and interrupt mappings.

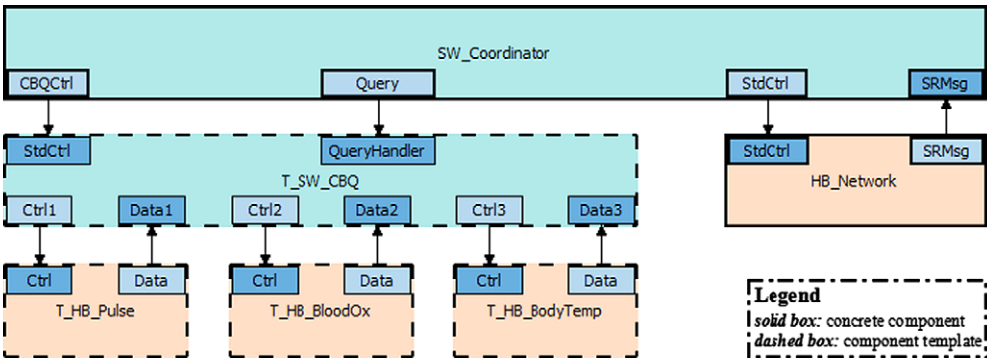


Fig. 10. CodeBlue architectural pattern for medical sensor systems.

```
hybrid pattern P_HB_CodeBlue {
  configuration {
    component HB_Network; component SW_Coordinator;
    template T_HB_Sensor as T_HB_BloodOx;
    template T_HB_Sensor as T_HB_BodyTemp;
    template T_HB_Sensor as T_HB_Pulse;
    template T_SW_CBQ;
    connection(T_SW_CBQ.Ctrl1, T_HB_Pulse.Ctrl);
    connection(T_HB_Pulse.Data, T_SW_CBQ.Data1);
    connection(T_SW_CBQ.Ctrl2, T_HB_BloodOx.Ctrl);
    connection(T_HB_BloodOx.Data, T_SW_CBQ.Data2);
    connection(T_SW_CBQ.Ctrl3, T_HB_BodyTemp.Ctrl);
    connection(T_HB_BodyTemp.Data, T_SW_CBQ.Data3);
    connection(SW_Coordinator.CBQCtrl, T_SW_CBQ.StdCtrl);
    connection(SW_Coordinator.Query, T_SW_CBQ.QueryHandler);
    connection(SW_Coordinator.StdCtrl, HB_Network.StdCtrl);
    connection(HB_Network.SRMsg, SW_Coordinator.SRMsg); }
  properties {
    assertion CBP1: CBA1
      After_Eventually_UnlessAfter_(
        (SW_Coordinator.Query.handleQuery(SRC, SINK, T)),
        (SW_Coordinator.SRMsg.send(SRC, SINK, var > T)),
        (SW_Coordinator.Query.cancelQuery(SRC, SINK)))
    assumption CBA1
      After_EventuallyAlways_(
        (SW_Coordinator.Query.handleQuery(SRC, SINK, T)),
        (T_HB_Sensor[SRC].devSen(var > T))) } }
```

Fig. 11. EADL Spec for CodeBlue pattern.

component-based co-simulation and co-verification. Once the system design is complete and has been validated is it synthesized into the deployable.

In this section, we illustrate the component-based co-development workflow as supported by ESIDE with a case study on the CodeBlue medical sensor system. CodeBlue is intended for

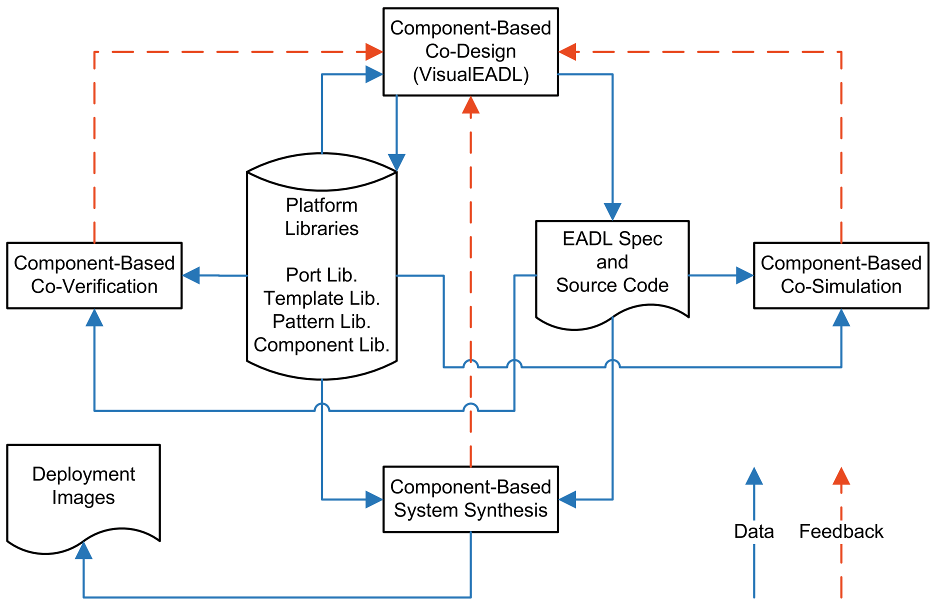

Component-Based Co-Design (VisualEADL)
Platform Libraries
Port Lib.
Template Lib.
Pattern Lib.
Component Lib.
Component-Based Co-Verification
EADL Spec and Source Code
Component-Based Co-Simulation
Deployment Images
Component-Based System Synthesis
Data
Feedback

Fig. 12. ESIDE architecture and features.

monitoring the vital signs of a patient in emergency situations (Shnayder et al., 2005). We have re-engineered the original TinyOS-based implementation of CodeBlue with ESIDE.

5.1. Platform-based development

In ESIDE, embedded systems are developed on top of a particular platform. It is the platform, not EADL, which dictates the operational semantics of the system and its components. Every stage of development is heavily influenced by the developer's platform decision; it determines what hardware and software languages will be used to design primitive components; it describes how primitive components will be verified; it provides simulators for both hardware and software; and it specifies how the embedded system will be synthesized to hardware and software deployables.

The platform also provides a library of common EADL constructs – components, ports, templates and patterns – that can be reused. The content of an ESIDE platform is unlimited in that it may contain any construct expressible by EADL. This includes hardware, software, and bridge aspects as well as both primitive and composite components.

When a new system is to be developed, ESIDE will begin by asking the developer on which platform he or she wants to develop. Once chosen, the platform libraries are available to draw from at any time by means of the platform library view, which is depicted at the bottom of Fig. 13. The platform library is intended to be a dynamic aspect of the platform. During the design phase, a developer may select aspects of his or her project to include in the library for use in future projects. The chosen EADL construct along with any sub-components and dependent files are migrated into the platform library and references to it are updated in the current project. In this way, the developer can contribute to the platform without knowledge of its internal structure.

CodeBlue was developed in the TinyOS networked sensor run-time environment described in Section 2. We began our re-implementation of CodeBlue by developing the TinyOS platform described in Section 4.2.2. TinyOS is a software development environment following the traditional model of embedded system development (see Section 5.7.2). As such, it does not specify a hardware description language, nor does it make explicit the communication channels between hardware and software. In designing our TinyOS platform, we retained nesC as the underlying software language for our TinyOS platform. Verilog is used to develop primitive hardware components, and bridge components are described using a Bridge Specification Language (BSL) (Hao et al., 2009) for this platform.

As previously discussed, TinyOS software components follow their own component model. Because of this, we were able to re-implement the standard TinyOS library in EADL with only minor changes to the original source code (see Section 5.7 for a discussion of some of these differences). TinyOS also supports a number of hardware configurations on which systems can be developed, and provides the hardware abstraction and protection layer for each. We have chosen two of these hardware configurations, Mica and PC, on which to develop our case studies. These hardware configurations are not component-based. We have componentized these configurations using EADL. One direct benefit is that we can now customize the hardware configuration for each system.

5.2. Co-design

At the core of any embedded system development process is the design phase. The traditional design model for embedded systems is a stack. Hardware forms the first layers of the stack, which support the hardware abstractions and protections. All of these must be finalized and become static assumptions before developing software, the top layers of the stack. We believe that this model is inefficient.

ESIDE unifies the component models for hardware and software. This allows both halves of the system to be developed in the same environment. This methodology allows hardware and software be built in a tightly coupled fashion, reducing the code base of both to the minimum required for implementation. Traditional design leads to generic hardware platforms, only parts of which are used in a given application. If hardware and software are co-designed, the excess can be avoided, leading to better resource utilization.

In ESIDE, co-design centers around component-based architectures and formally verifiable properties. Systems are realized by composing simple components, both hardware and software, into ever more complex ones. Component properties are utilized to abstract away implementation details in system verification.

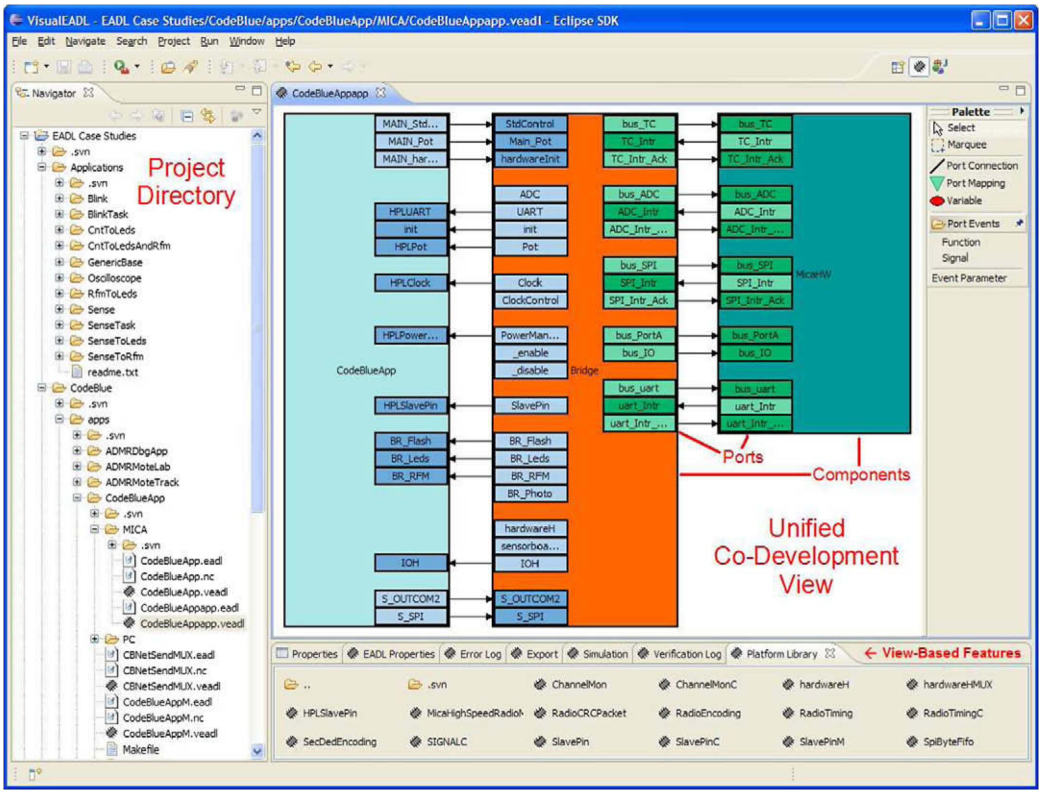

VisualEADL - EADL Case Studies/CodeBlue/apps/CodeBlueApp/MICA/CodeBlueAppapp.veadl - Eclipse SDK
Project Directory
Ports
Components
Unified Co-Development View
View-Based Features
CodeBlueApp
Bridge
MicaHW
Palette

Fig. 13. ESIDE visual modeling interface and platform view.

Systems are designed visually in ESIDE, as shown in Fig. 13. The center area of the environment displays the configuration of an embedded system. Components serve as the basic building blocks, and come in three varieties: software, hardware, and bridge. Components are connected through ports, shown in Fig. 13 as small rectangles embedded in each sub-component. An arrow connecting two ports of the same type indicates a channel for inter-component communication. This communication always flows from the provided port to the used port.

The design process starts with primitive components, which have no EADL sub-structure definition. In its place, primitive components contain an implementation of their interface in the appropriate platform native languages. Properties may be specified on primitive components, and those properties must be satisfied by the native language implementation in order for the system to pass verification.

Primitive components are then composed to create more complex ones. There are three types of composition: composite software, hardware, and hybrid components. The former two may contain any number of software and hardware components, respectively. The latter may contain components of any type. A composite component may also contain other composite components. Fig. 13 shows a hybrid composite component (also a system) containing a composite software component, a primitive bridge component, and a composite hardware component.

Several levels into the software side of CodeBlue is the ADMR component, which handles multicast routing. Fig. 14 shows the configuration of ADMR. Both primitive and composite software sub-components, which are indistinguishable at this level of abstraction, are composed to implement the functionalities of ADMR. The ADMR interface is the collection of ports not contained in any sub-component. They are realized by the ports of the sub-components as indicted by the mapping triangles that connect them. Each port of the component's overall interface must be implemented by exactly one sub-component. It is not allowed that two sub-components "share" the implementation of a single port (see discussion in Section 5.7.3).

5.3. Co-simulation

The ways in which co-simulator configurations differ vary from platform to platform. However, given a single platform there is much commonality in how the co-simulator is configured for different systems based on that platform. Fig. 15 illustrates the co-simulation environment setup flow for the Mica platform. To set up the co-simulation environment, the BSL compiler retrieves the hardware platform components in Verilog (e.g., the processor and the bus) and the software platform components in nesC (e.g., the embedded OS). In our study, we employ ModelSim (Mentor Graphics, 2009) and Giano (Forin et al., 2006) as the foundations for our system co-simulator. ModelSim is a hardware simulator that is capable of simulating hardware designs written in Hardware Description Languages (HDLs) such as Verilog, VHDL, and SystemC. Giano is a full-system real-time simulator. It incorporates simulation of processors, I/O sub-systems, and peripherals of a system. ModelSim can be attached to Giano and be responsible for simulation of reconfigurable FPGAs. The communication between hardware and software components is done through the Programming Language Interface (PLI) between Giano and ModelSim. The PLI is masked by and configured via the bridge components.

The bridge components can be simulated on two different levels: Register Transfer Level (RTL) and Transaction Level (TL). For the RTL simulation, the processor, the bus, and the OS are included in the system co-simulation and they are configured according to the bridge components. For the TL simulation, the platform components such as the processor, the bus, and the OS are excluded to accelerate the simulation speed. The hardware and software

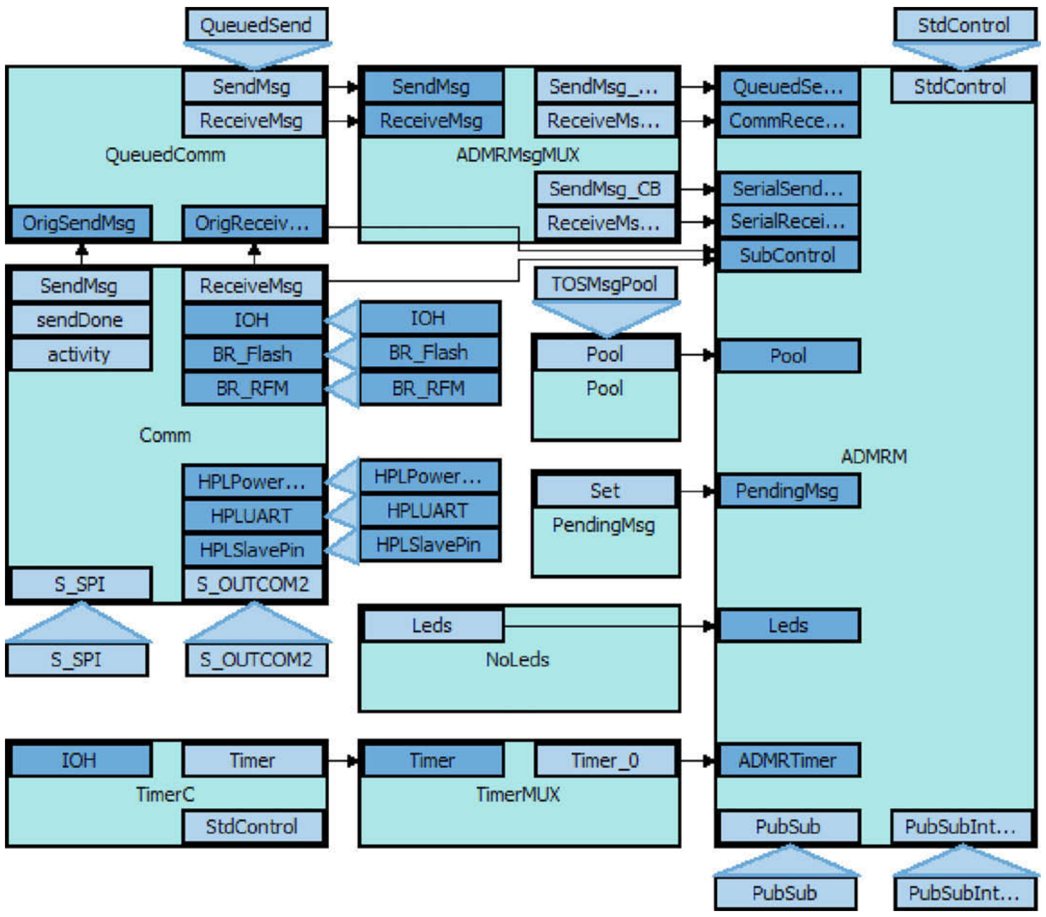


Fig. 14. Configuration of ADMR component.

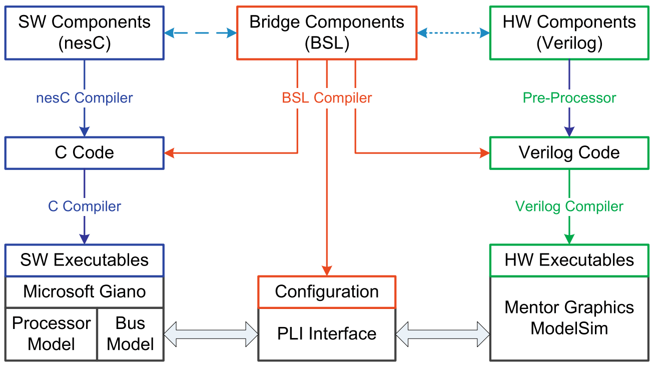


Fig. 15. Co-simulation environment setup.

components are connected directly by the transactors, which convert between software events and hardware signals.

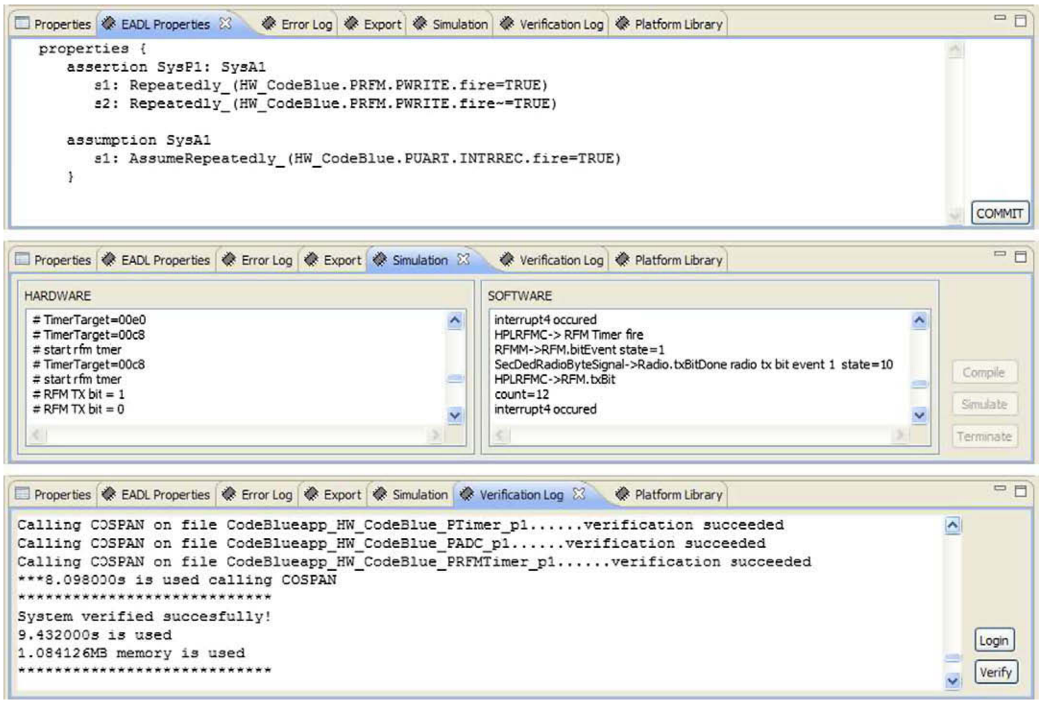
The co-simulation feature is integrated in the ESIDE interface. A button-click directs ESIDE to compile the project for simulation and invoke the appropriate simulators. The developer can examine system behavior by way of the simulation view in the center of Fig. 16. (The three separate views in Fig. 16 can be selected from the view-based feature tabs in Fig. 13.) When simulating an embedded system, the developer is allowed to watch events propagate from component to component, leveraging the visual model from co-design. State information for each component is also available for inspection at all times.

5.4. Co-verification

As components are developed, either for a specific system or for inclusion in a platform, their properties are specified in a separate view (see the top portion of Fig. 16). The integration of property specification into the design phase encourages the developer to think about behavior in terms of properties that can be formally verified. This shifts the dependency for validation away from simulation alone and toward the inclusion of verification. Once properties have been specified, a button-click submits the component in focus to the appropriate verification engines.

Primitive components are verified by directly model-checking their source code using corresponding verification engines. However, once a primitive component has been verified, its behavior can be abstracted by its verified properties. Using the properties as abstractions, inspection of primitive source code can often be avoided when verifying higher-level components. Properties of a composite component are checked on abstractions constructed from verified properties of its sub-components. A system is verified top-down as it is developed via recursive decompositions into its components. Verified properties of the reused components are reused in constructing the abstractions for verifying properties of the system or higher-level components.

As shown in Fig. 17, ESIDE supports an automatic query-verification-feedback loop when a system or component is verified. Verification starts with a property query to ESIDE. If the property has already been verified, ESIDE will trivially return the memorized result; otherwise, properties of a primitive component are directly model-checked by the primitive verification engine. Properties of a composite component are processed by the composite



```
properties {
   assertion SysP1: SysA1
      s1: Repeatedly_(HW_CodeBlue.PRFM.PWRITE.fire=TRUE)
      s2: Repeatedly_(HW_CodeBlue.PRFM.PWRITE.fire~=TRUE)

   assumption SysA1
      s1: AssumeRepeatedly_(HW_CodeBlue.PUART.INTRREC.fire=TRUE)
   }
```

```
# TimerTarget=00e0
# TimerTarget=00c8
# start rfm tmer
# TimerTarget=00c8
# start rfm tmer
# RFM TX bit = 1
# RFM TX bit = 0
```

```
interrupt4 occured
HPLRFMC-> RFM Timer fire
RFMM->RFM.bitEvent state=1
SecDedRadioByteSignal->Radio.txBitDone radio tx bit event 1  state=10
HPLRFMC->RFM.txBit
count=12
interrupt4 occured
```

```
Calling COSPAN on file CodeBlueapp_HW_CodeBlue_PTimer_p1......verification succeeded
Calling COSPAN on file CodeBlueapp_HW_CodeBlue_PADC_p1......verification succeeded
Calling COSPAN on file CodeBlueapp_HW_CodeBlue_PRFMTimer_p1......verification succeeded
***8.098000s is used calling COSPAN
****************************
System verified succesfully!
9.432000s is used
1.084126MB memory is used
****************************
```

Fig. 16. Property specification, simulation, and verification views.

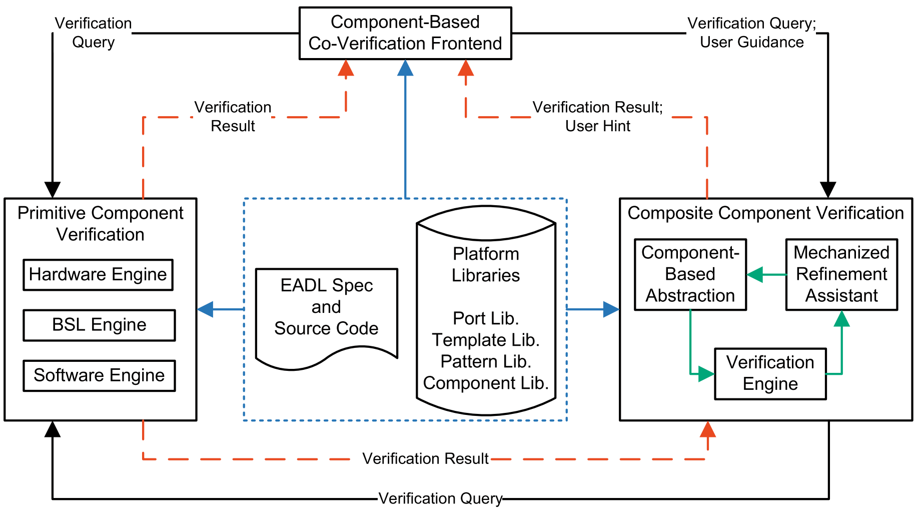

Component-Based Co-Verification Frontend
Verification Query
Verification Query; User Guidance
Verification Result
Verification Result; User Hint
Primitive Component Verification
Hardware Engine
BSL Engine
Software Engine
EADL Spec and Source Code
Platform Libraries
Port Lib.
Template Lib.
Pattern Lib.
Component Lib.
Composite Component Verification
Component-Based Abstraction
Mechanized Refinement Assistant
Verification Engine
Verification Result
Verification Query

Fig. 17. Co-verification tool support.

verification engine, which may need to query the appropriate verification engines for the sub-components with new properties if the previously verified properties of the sub-components are not sufficient abstractions. When the composite verification engine fails to detect new properties to improve the abstractions of the sub-components, it will give feedback to the developer, often in the form of an error trace, and ask users for refinement hints. Such hints are usually new properties to improve the sub-component abstractions.

Verification is nominally a predicate operation. Either a component's properties hold on the given implementation or they do not. However, in the case where a property does not hold, further information about the way in which the implementation fails to meet specifications is highly desirable. In order to provide this kind of information, our verification process includes the generation of an error trace for properties that do not hold. As with simulation, ESIDE not only displays this information textually, but also allows the developer to review it visually, following the error trace through the system using a mechanism similar to visual co-simulation. The result of these combined features is a development environment that not only unifies hardware and software models but also unifies the co-design, co-simulation and co-verification of those systems, leading to a highly available, efficient, and powerful embedded system development environment.

5.5. Unified playback of error traces

As discussed in their respective sections, ESIDE's co-simulation and co-verification features support the capabilities of playing back the error traces generated in simulation and verification. Although their purposes differ, there are significant similarities in their playbacks. ESIDE leverages these similarities to unify the playback of error traces.

Different formal verification methods and tools often lead to differences in the error traces that result from verification failures. The TinyOS platform uses the COSPAN verification engine (Hardin et al., 1996), which models all aspects of a system with variables. For example, every function in software is modeled with two implicit variables, *call* and *ret*, to represent the call and return events of that function.

There is similar variety in the data collected during simulation. Some simulators may choose to record value changes for declared variables, whereas others may only record memory reads and writes. Some simulators record time in terms of seconds, while others use clock cycles.

Fig. 18 illustrates how various error traces are unified for playback in ESIDE. A common format for traces that VisualEADL can use to visually "play back" verification or simulation traces has been designed. A trace of this format contains information about the variables and component interface activities. The choice of simulation and verification technologies is specific to each platform. As such, the platform is also responsible for providing data transformation logic that converts platform-specific error traces into the common format, which can then be executed by the playback engine.

5.6. Co-synthesis

The final stage in embedded system development is co-synthesis: generation of a hardware and software executables of the designed system. Each platform contains compilers to translate EADL structural information into its native languages. Primitive component implementations are combined with constructs generated from composite components into a complete native language implementation of the hardware and software of the designed system. Native language tools can then be used to, for instance, burn FPGAs and compile software to realize the physical system.

When a system is synthesized to the platform native code, the structure-expressing overhead of EADL is greatly reduced or entirely removed. When synthesizing the remodeled CodeBlue system to nesC code, we compared the resulting code against the original source. For verification of accuracy, we also compared the applications' behaviors in TOSSIM (Hill et al., 2000), the TinyOS simulation engine. Both exhibited the same behavior. Next, we compared the resulting code base against the original. In the new version, there are 52 components and 43 ports. In the original, there were 43 components and 30 ports. The eight extra components are all for overcoming various signal routing problems inherent in the paradigm shift (see Section 5.7.3). Most extra ports were introduced while making the hardware interfaces explicit.

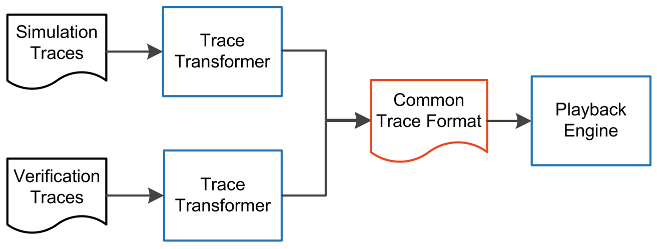


Fig. 18. Trace unification for playback.

5.7. ESIDE design decisions

5.7.1. Encapsulation and compilation

In defining the semantics of component composition, TinyOS does not treat each instance of a component as a separate software entity. Instead, all components are implemented in static space and shared between those components that utilize them. While this decision deviates from the component-based development principle of encapsulation, it does have practical benefits. Having only a single instance of each component resident in memory reduces the code footprint of a TinyOS-based sensor system.

In ESIDE, we chose not to deviate from the encapsulation principle. We believe the benefits to such adherence outweigh the costs. Firstly, strict encapsulation allows for more efficient component-based co-verification. Secondly, systems developed in fuller adherence to component-based methodologies tend to be more intuitive. Finally, those aspects that truly warrant static space implementations may still be placed in static space via the platform languages' native mechanisms. In co-synthesis, EADL allows the developer to instruct synthesis of several instances of a component into a static copy.

ESIDE provides a warning in the situation that a sub-component is shared by multiple components, since the proved properties of the sub-component may require additional assumptions to hold. For example, the IntToRfm component has a property asserting all transmission requests will be acknowledged, assuming no consecutive transmission requests without an acknowledgement. Two different components A and B that contain IntToRfm can satisfy the assumption, respectively, but when A and B are used in the same project, the assumption no longer holds. Since there is only one copy of IntToRfm, A and B can both send data and wait for acknowledgement at the same time. This is possible due to the concurrent nature of TinyOS.

5.7.2. Hardware/software boundary

The traditional model of embedded system development is stack-based. TinyOS components communicate with hardware at the primitive level via special library function calls. This works well for software development on predetermined hardware. However, in co-design, communication between hardware and software is horizontal, not vertical, and it cannot be handled on the primitive level. At least one bridge component is required for the communication. For these reasons, a significant number of ports had to be introduced in our models for the now non-primitive HW/SW communication.

Furthermore, faithful re-modeling of TinyOS-based systems in ESIDE leads to a monolithic design with a single system-level software component, a single hardware component, and a bridge component connecting them. Not only does this insufficiently leverage the co-development model, but it also necessitates the mapping of interfaces from primitive software components to system level and back down to primitive hardware components. An example of this monolithic design is shown in Fig. 13. We believe these issues will be resolved by relaxing the faithfulness requirement and fully adopting the co-development methodology. By re-engineering the TinyOS systems from the ground up, adhering to the co-development model, the distance between hardware and software will be minimized, and the unnatural mappings of our current implementations will not manifest.

5.7.3. Port mapping semantics

Whether port mapping is one-to-one or one-to-many has a significant impact on its semantic interpretation. In the case of the former, a port mapping indicates equivalence of identity. One way to think of this is that the mapped port is only an alias for the port to which it is mapped. In the latter case, the mapped port is not equivalent to any one of the ports to which it is mapped, but represents a functionality comprised by their composition. The problem is then to define the semantics of that composition. Does each provide a partial implementation of the whole? Does each provide a full interface, from which one is chosen by means of some meta-mapping attribute? Are all interfaces called sequentially (or simultaneously) and their results combined to form a single return value? These questions are answered when the semantics of the component model is specified.

In TinyOS, one-to-many mappings are allowed. When a function is called in an interface that is mapped to multiple sub-interfaces, each of those interfaces is called in turn, and the results are combined in one of two methods. The language defines default combining semantics for many built-in types. For instance, multiple return values of the `result_t` type are combined by the logical and operator, so that `SUCCESS` is returned only in the case where each function returned `SUCCESS`. If, however, the data type is not built-in or the language does not provide default combining logic, the developer may introduce his or her own. Although one-to-many mappings are powerful, we chose to support one-to-one mapping in ESIDE. The simplicity and elegance of one-to-one mappings removes much of the confusion that is inherent to one-to-many mappings as to the intended semantics. We were able to faithfully model TinyOS systems with the introduction of multiplexing components that explicitly realize the one-to-many mappings.

6. Pattern-guided co-verification

A major challenge in component-based co-verification is the property formulation problem: (1) what are the system properties to verify, (2) what are the component properties needed for verifying the system properties, and (3) what are the environment assumptions necessary for establishing these properties. The problem may significantly hinder effectiveness of component-based co-verification. The increasing adoption of ABV alleviates this problem since designers are required to formulate the component properties as a component is designed. However, this problem persists since, in essence, it is due to lack of knowledge about possible environments of components, and it also plagues ABV although on a lesser extent. In addition, ABV requires major manual efforts in property formulation. Therefore, it is highly desired for heuristics that can reduce the property and assumption formulation efforts for embedded systems and composite components which follow commonly used architectures.

Integration of co-design and co-verification enables more effective verification. EADL provides the language support for capturing architectures and architectural patterns of embedded systems and also supports association of properties with components and properties templates with architectural patterns. Our approach utilizes EADL to address the property formulation challenge in the following ways: (1) pattern-guided property formulation, (2) pattern-guided property decomposition, and (3) pattern-guided circular reasoning prevention.

6.1. Pattern-guided property formulation

Patterns can guide property formulation for both systems and reusable components. A difficulty in ABV is to identify the potential environments for a reusable component and how it interacts with the environments. Patterns essentially abstract the potential environments for reusable components. If a component is designed to be reused under a given pattern, the pattern often determines the properties that ought to hold on the component and their

```
software port StdCtrl { events {input: start, stop;}}

software port QueryHandler {
  events {input: handleQuery(int src, sink, threshold), cancelQuery(int src, sink);
            output: dataReady(int var); }
}

software port Data { events {input: getData(); output: dataReady(int var);} }

software template T_SW_CBQ {
  int QrySink[NumofSen], QryThreshold[NumofSen];
  interface {
    provides StdCtrl as StdCtrl;
    provides QueryHandler as QueryHandler;
    uses multi Sen[NumofSen] { StdCtrl as Ctrl; Data as Data }
  }

  properties{
    assertion CBQ_P1 After(QueryHandler.handleQuery(SRC, SINK, T))
                     Repeatedly(Sen[SRC].Data.getData)
                     UnlessAfter(QueryHandler.cancelQuery(SRC, SINK));
    assertion CBQ_P2 After(QueryHandler.handleQuery(SRC, SINK, T))
                     EventuallyAlways ((QrySink[SRC]=SINK)
                                       *(QryThreshold[SRC]=T))
                     UnlessAfter(QueryHandler.cancelQuery(SRC, SINK));
    assertion CBQ_P3 After((Sen[SRC].Data.dataReady(var>QryThreshold[SRC]))
                           *(QrySink[SRC]=SINK))
                     Eventually(QueryHandler.dataReady(SRC, SINK,
                                                (var>QryThreshold[SRC])))
                     UnlessAfter(QueryHandler.cancelQuery(SRC, SINK));
  }
}
```

Fig. 19. EADL Spec for CBQ template.

appropriate environment assumptions. An example is the *T_SW_CBQ* component template under the CodeBlue pattern which dictates the interactions between *CBQ* and other components and suggests the properties in Fig. 19. (Port properties in this template are omitted for simplicity.) As the number of sensors that CBQ manages varies depending on the individual system, the number of corresponding ports varies accordingly. Keyword multi is used to support this feature. By using multi to group the ports, it indicates this is a group of ports that can be instantiated multiple times according to the number of sensors in a specific system.

Another difficulty in property formulation is how to derive appropriate behavior rules of the system from the system requirements, i.e. what are the system properties to verify. Patterns are utilized as the vehicle to address this problem during the system design process. Given the system requirements, the architecture patterns used to structure the system are selected. These patterns suggest what properties to verify on the system. In EADL, this is supported by pattern-level property templates which are specified on the system-level (or composite component level) interfaces and variables. Property *CB_P1* in Fig. 11 is such an example.

6.2. Pattern-guided property decomposition

In component-based development of embedded systems, the decomposition process is top-down, which recursively decomposes a system into its components and their inter-connecting relations, until reaching the primitive components or the reusable components. The decomposition process often depends on knowledge and experience of system architects to determine the decompositions, e.g., hardware and software partitions. Architectural patterns facilitate this process by capturing reusable knowledge about system architectures.

The pattern-guided decomposition process starts with pattern selection. In this step, architectural patterns are manually selected according to the specification of system/component requirement and interface. If a pattern is successfully selected from the pattern library, the system/component is decomposed into concrete components and component templates following the pattern. The decomposition stops at a concrete component which is reused directly. A component template, which is only a component skeleton with ports and property templates, is utilized to select a component matching the template specification. This decomposition stops if the component can be reused; otherwise, there are two ways to design the component: as a primitive component by which the decomposition stops with direct implementation or as a composite component by which the decomposition continues recursively. If there is no appropriate pattern that matches the system/component specification, the decomposition is conducted manually.

To enable efficient component-based co-verification, we integrate verification into the top-down decomposition process, in particular, integrating property decomposition with system decomposition, which is a major advantage of our approach. As an architectural pattern is selected to guide a decomposition, its associated property decomposition strategy is also utilized to decompose the pattern-level properties into the component properties. As a component template is utilized to select a component, both the interface/port templates and the property templates are used to guide the component selection. A basic approach to specification of decomposition strategies is to associate appropriate property templates with the component templates in a pattern and define the dependency links from a pattern-level property templates to the component property templates. Dependencies among component properties are captured as assumptions of these properties. For instance, for the pattern-level property of the CodeBlue pattern, we can define a decomposition strategy as shown in Fig. 20. (Port properties involved in this strategy are omitted for simplicity.) This strategy must be instantiated for each individual system since the properties of the *CBQ* component depends on how many sensors are included.

This straightforward approach to strategy specification is cumbersome when a lot of property templates need to be specified. And as discussed above, for certain pattern, the number of components that are involved in the pattern are only known when the pattern is instantiated in a system context. Therefore, more convenient ways to specify decomposition strategies is needed. Language supports for specifying decomposition strategies as simple programs are also provided. These programs generate component properties according to the parameters used in instantiating architectural patterns.

6.3. Pattern-guided circular reasoning avoidance

There may exist assume-guarantee dependency cycles among component properties, which can potentially lead to circular reasoning. Property dependency cycles may be accidentally intro-

```
// Pattern-level property
  assertion CB_P1: CB_A1
    After(SW_Coordinator.Query.handleQuery(SRC, SINK, T))
    Eventually(SW_Coordinator.SRMsg.send(SRC, SINK, var>T))
    UnlessAfter(SW_Coordinator.Query.cancelQuery(SRC, SINK));
  assumption CB_A1
    After(SW_Coordinator.Query.handleQuery(SRC, SINK, T))
    EventuallyAlways(T_HB_Sensor[SRC].devSenVar>T);

// Properties of Coordinator
  assertion CRD_P1
    After (SW_Coordinator.Query.dataReady(SRC, SINK, var>T))
    Eventually(SW_Coordinator.SRMsg.send(SRC, SINK, var>T));
  assertion CRD_P2
    Never (SW_Coordinator.Query.handleQuery(SRC, SINK, T)*
             SW_Coordinator.Query.cancelQuery(SRC, SINK));

// Properties of CBQ as shown in Figure 19

// Properties of Sensors
  assertion SEN_P1 After(GSI.getData) Eventually(GSI.dataReady(var=drvSenVar));
  assertion SEN_P2 IfEventuallyAlways(devSenVar>T) EventuallyAlways(drvSenVar>T);

// Decomposition strategy
CB_P1 → CRD_P1, CRD_P2, CBQ_P1, CBQ_P2, CBQ_P3, SEN_P1, SEN_P2;
```

Fig. 20. An example decomposition strategy.

duced by incorrect formulation of component properties or intentionally introduced to reflect the nature of component interactions and simplify property specification. The first type of cycles can be eliminated by cycle detection. For the second type of cycles, additional verification work may be needed to show that such cycles will not cause circular reasoning. Solution of this problem can be made more efficient through architectural patterns since once a pattern can be shown free of circular reasoning, the instantiations of the pattern are free of circular reasoning. In essence, the circular reasoning detection is conducted only once for the pattern and is reused when the pattern is reused.

7. Evaluation

7.1. Effectiveness

We have evaluated ESIDE by remodeling 12 TinyOS-based sensor systems. Except for the design changes in Section 5.7, we attempted to preserve the original structure of software components. We re-designed the hardware to be component-based and provided bridge components accordingly. Because of the TinyOS design features discussed in Section 5.7.2, each re-modeled system contains only one bridge component at the top hierarchical level, thus containing no hybrid components below the system level.

Table 1 shows the statistics of the remodeled systems compared to the original systems. The additional components and ports are used to imitate the TinyOS designs such as one-to-many mapping and shared sub-components and to componentize the hardware. We modeled two TinyOS platform libraries: the Mica library and the PC library (used for TOSSIM (Hill et al., 2000) simulation). All remodeled systems are compiled back into TinyOS code and simulated by TOSSIM to ensure that our remodeling was faithful.

We have conducted component-based co-simulation and co-verification during the re-engineering. Table 2 shows the simulation and verification statistics on selected systems. The co-simulation is driven by test vectors that induce one iteration of the data-producing-consuming loop of these systems. It can be observed that the co-simulation can be sped up three orders of magnitude by simulating the bridge components on the TL level rather than the RTL level (Hao et al., 2009). The co-verification step verifies a system-level property that ensures repeated data-producing-consuming in these systems. The time and memory usages are listed for verification of this property on abstractions constructed from properties of the first-level components (Li et al., 2008). We expect further benefits in co-simulation and co-verification if the systems are designed from scratch following the co-development model, instead of re-engineered.

The verification statistics of the sensor system family based on the Mica Platform is shown in Table 3. The first part of the table illustrates the scale of this sensor system family in terms of numbers of systems, components, ports, component templates, and patterns. The second part illustrates the amount of architectural reuse in component-based co-design. The third part illustrates the amount of reuse in component-based co-verification. Two categories of system properties have been verified. Since the basic functionality of the sensor systems is data generation and consumption, the first categories of properties includes a property of each system that asserts that the system repeatedly consumes data. The data consumption event is different in each system. The second category includes properties that ensure that systems using buffers not overflow these buffers. The component properties listed in Table 3 include both the port-level and component-level properties. Each property may include multiple assertions which together capture one behavioral aspect of a component.

Co-verification was accomplished on all the systems, many of which fail a straightforward application of model checking to entire systems due to state space explosions. Examples of such explosions can be found in (Xie et al., 2007). It can be observed from Table 3 that for the sensor system family, property reuse reduces the number of component properties that need to be verified by about 92%. Since the two categories of system properties verified reflects the most generic system functionalities, most of the component properties can be generated from property templates, therefore, manual efforts in property formulation are significantly reduced. For more system-specific properties, the manual efforts may increase as fewer property templates can be reused.

A buffer overflow vulnerability in several systems is detected through verification of the buffer overflow property. This vulnerability is associated with the *SourceToSink* pattern, under which the sink acknowledges the reception of a piece of data from the source. However, the sink does not notify the source whether the data has been consumed. The buffer overflow vulnerability may manifest if the source generates data faster than the sink can consume. This led us to introduce a new pattern where the source waits for the sink consumption notification before sending the next data.

7.2. Usability

Three student groups from a graduate software engineering class have successfully applied our tool in their course projects.

Table 1
Remodeling statistics.

# of systems remodeled	Orig	12 New
# of components in Mica platform library	43	52
# of components in PC platform library	46	51
# of ports/interfaces in Mica platform library	30	43
# of ports/interfaces in PC platform library	29	40
# of components specific for each system	38	41
# of ports/interfaces specific for each system	8	10
# of hardware components developed	N/A	26
# of bridge components developed	N/A	2

Table 2
Simulation and verification statistics.

System	Co-simulation		Co-verification	
	RTL (s)	TL (s)	Time (s)	Memory (MB)
SenseTask	1.887	0.004	22.34	1.644
SenseToLeds	1.587	0.003	27.02	3.765
SenseToRfm	6.837	0.008	52.61	3.806
CodeBlue	7.456	0.011	34.66	5.792

Table 3
Sensor system family statistics.

# of systems	12
# of components	119
# of ports	72
# of component templates	9
# of architectural patterns	4
Times of component reuses	405
Times of port reuses	2670
Times of component template reuses	38
Times of pattern reuse	12
# of system properties verified	16
# of component properties verified	341
# of comp. properties to be verified if no reuse	4493
Times of property template reuses (from ports, component templates and patterns)	172

None of the students has background in formal verification. The time taken to train each group with the design methodology and the verification engine was less than 4 hours. One student group successfully developed and verified a family of TinyOS networked sensor systems with more than 50 components. Another group designed a smart home system with more than 30 components. This system was designed based on existing systems such as modems, cellular phones, and smart home controllers, each of which is treated as a component with rigorously defined interfaces. The students verified their design successfully using the composite component verification engine. In this study, we observed that a large obstacle for students is to learn how to assert properties using the property templates provided.

In our own experience, we found that the ability to explore our systems visually and to modify our models through "drag-and-drop" without worrying about the underlying EADL syntax led to a significant reduction in the tedium of system development. We also believe that our visual modeling methods will lead to increased productivity.

8. Related work

There has been much research on component-based hardware and software development (Jacome and Peixoto, 2001; Heineman and Councill, 2001; Szyperski et al., 2002). A fundamental problem in component-based development is how to establish 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, 2009; Wallnau et al., 2003). The vision of PACC is that software components have certified properties (e.g., performance) and the behavior of systems assembled from components is predictable. 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. Our project shares the PACC vision while extending this vision by (1) defining a component-based architectural description language (ADL) for embedded systems that unifies hardware and software components and (2) formally establishing properties of an embedded system from properties of its hardware and software components.

Much research has been done on developing hardware, software, and embedded systems ADLs (see Tomiyama et al., 1999; Medvidovic and Taylor, 2000) for their comprehensive surveys). Among those ADLs, the most closely related are SAE AADL (Society of Automotive Engineers (SAE), 2004), Metropolis (Sangiovanni-Vincentelli, 2007), and Ptolemy (Lee, 2003). The AADL is an industry standard designed for the specification, analysis, and automated integration of real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems. Metropolis features a flexible and formal semantics based upon the tagged signal model [14] that allows it to represent a wide variety of models of computation. Furthermore, it supports platform-based design, behavior-architecture mapping, and orthogonalization of concerns at the levels of communication-computation-coordination, architecture-function-mapping, and behavior-performance. Ptolemy focuses on component-based heterogeneous modeling. It uses tokens as the underlying communication mechanism. Directors regulate how actors in the design fire and how tokens are used to communicate between them. Ptolemy uses hierarchical composition to handle heterogeneity. Each level in a hierarchy has a director that organizes the firing of the actors at that level. EADL differentiates from the above ADLs in that it does not require any particular execution semantics and can be instantiated on any hardware and software execution semantics to enable component-based co-design, co-simulation, co-verification, and co-synthesis based on these semantics.

Design patterns (Gamma et al., 1994) are concerned with reuse of programming structures at the algorithmic or data structure level while architectural patterns (Perry and Wolf, 1992; Shaw and Garlan, 1996; Buschmann et al., 1996) are concerned with reusable structural patterns of software system with respect to their components. Architectural patterns have been applied in software design, validation, documentation, etc. Our research utilizes architectural patterns to facilitate formulation and verification of properties of embedded systems and their components. The EADL representation of architectural patterns is partially motivated by that of ACME (Garlan et al., 1997). In ACME, architectural patterns are specified as first-class language constructs. EADL representation of architectural patterns specially targets HW/SW co-design, co-simulation, co-verification, and co-synthesis of embedded system families.

9. Conclusions and future work

In this paper, we have presented EADL, an architecture description language for embedded systems. EADL captures both hardware and software components and their interactions, and gains its flexibility from its support to platform-oriented instantiation. It has demonstrated its effectiveness in serving as the vehicle for integrating component-based co-design, co-simulation, co-verification, and co-synthesis in ESIDE. For next steps, we will explore how EADL can be utilized to facilitate analysis of system and component properties other than temporal correctness properties.

Acknowledgments

This research received financial support from Semiconductor Research Corporation (Contract #: 1356.001), National Science Foundation of the United States (Grant #: 0720546), Chinese National Basic Research and Development 973 Program (Grant #: 2004CB719400), and Chinese National High Technology 863 Program (Grant #: 2006AA01Z155, 2007AA01Z122, and 2007AA04Z135). We thank Kecheng Hao for his help on co-simulation.

References

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-oriented Software Architecture: A System of Patterns. John Wiley & Sons Inc..
Clarke, E.M., Emerson, E.A., 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In: Logic of Programs Workshop.
Clarke, E.M., Grumberg, O., Peled, D., 1999. Model Checking. MIT Press.
CMU/SEI, 2009. Pacc (predictable assembly from certifiable components) <http://www.sei.cmu.edu/pacc>.
Crnkovic, I., 2005. Component-based software engineering for embedded systems. In: ICSE, 2005.
Forin, A., Neekzad, B., Lynch, N.L., 2006. Giano: the two-headed system simulator. Tech. Rep. MSR-TR-2006-130, Microsoft Research.
Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Object-Oriented Software. Addison-Wesley.
Garlan, D., Monroe, R.T., Wile, D., 1997. Acme: an architecture description interchange language. In: CASCON.
Gay, D., Levis, P., Behren, R., Welsh, M., B.E., D. Culler, 2003. The nesC language: a holistic approach to networked embedded systems. In: PLDI.
Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D., 2003. The nesC language: a holistic approach to networked embedded systems. In: PLDI.
Hao, K., Xie, F., 2009. Componentizing hardware/software interface design. In: DATE.
Hardin, R.H., Har'El, Z., Kurshan, R.P., 1996. COSPAN. In: CAV.
Heineman, G.T., Councill, W.T. (Eds.), 2001. Component-based Software Engineering: Putting the Pieces Together. Addison-Wesley.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J., 2000. System architecture directions for networked sensors. In: ASPLOS.
IEEE, 2005. IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2005). IEEE.
Jacome, M.F., Peixoto, H.P., 2001. A survey of digital design reuse. IEEE Design and Test of Computers 18 (3).
Kurshan, R.P., 1994. Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press.
Lee, E.A., 2003. Overview of the ptolemy project. Tech. Rep. UCB/ERL M03/25, UC Berkeley.
Li, J., Sun, X., Xie, F., Song, X., 2008. Component-based abstraction and refinement. In: Proc. of ICSR.
Maliniak, D., 2002. Assertion-based verification smooths the road to IP reuse. Electronic Design.
Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng. 26 (1).
Mellor, S.J., Balcer, M.J., 2002. Executable UML: A Foundation for Model Driven Architecture. Addison Wesley.
Mentor Graphics, 2009. ModelSim <http://www.mentor.com>.
Palnitkar, S., 2003. Verilog HDL, second ed. Prentice-Hall, US.
Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. SIGSOFT SEN 17 (2).
Pnueli, A., 1977. The temporal logic of programs. In: Proc. of 18th IEEE Symposium on Foundations of Computer Science.
Quielle, J.P., Sifakis, J., 1982. Specification and verification of concurrent systems in CESAR. In: Symposium on Programming.
Sangiovanni-Vincentelli, A.L., 2007. Quo vadis sld: Reasoning about trends and challenges of system-level design. Proceedings of the IEEE 95 (3).
Shaw, M., Garlan, D., 1996. Software Architecture: Perspective on an Emerging Discipline. Prentice Hall.
Shnayder, V., Chen, B.R., Lorincz, K., Fulford-Jones, T.R.F., Welsh, M., 2005. Sensor networks for medical care. Tech. Rep., Harvard University.
Society of Automotive Engineers (SAE), The SAE AADL Language Standard (AS-5506), SAE, 2004.
Szyperski, C., Gruntz, D., Murer, S., 2002. Component Software – Beyond Object-oriented Programming. Addison Wesley.
Tomiyama, H., Halambi, A., Grun, P., Dutt, N., Nicolau, A., 1999. Architecture description languages for system-on-chip design. In: APCHDL.
Wallnau, K.C., 2003. A technology for predictable assembly from certifiable components. Tech. Rep., CMU/SEI-2003-TR-009.
Xie, F., Liu, H., 2007. Unified property specification for hardware/software co-verification. In: COMPSAC.
Xie, F., Yang, G., Song, X., 2006. Component-based hardware/software co-verification. In: MEMOCODE.
Xie, F., Yang, G., Song, X., 2007. Component-based hardware/software co-verification for building trustworthy embedded systems. Journal of Systems and Software, 80 (5).

Juncao Li received the B.S. and M.S. degrees in Automatic Measurement and Control from Harbin Institute of Technology, Harbin, China in 2004 and 2006, respectively. He joined the Department of Computer Science at Portland State University in 2006 as a Ph.D. student. He is now working toward his Ph.D. degree in computer science.

His current research interests include formal hardware/software interface specification and verification, component-based co-design and co-verification of embedded systems, and static program analysis.

Nicholas T. Pilkington received the B.S. degree and the M.S degree in Computer Science from Portland State University in 2007 and 2009, respectively. He is currently employed as a systems engineer at Intel.

Fei Xie received the B.E. degree in computer science and engineering from Northwestern Polytechnical University, Xi¢an, Shaanxi, China in 1995. He received the M.E. degree in computer science and technology from Tsinghua University, Beijing, China in 1998. He received the Ph.D. degree in computer science from the University of Texas at Austin, Austin, Texas, U.S.A. in 2004. He is currently an associate professor in the Department of Computer Science, Portland State University, Portland, Oregon, U.S.A.

His research interests include embedded systems, software engineering, and formal methods. He is particularly interested in development of formal method based techniques and tools for building safe, secure, and reliable software and embedded systems.

Qiang Liu received the B.S. degree and the M.S. degree in Computer Science from Tsinghua University in 1985 and 1988, respectively. She is currently an associate professor in School of Software at Tsinghua University and teaches courses on software engineering and software project management. Her research interests include software engineering, workflow technology, Web technology and e-learning. She has served as Program Committee member for the International Conference on Computer Supported Cooperative Work in Design 2007–2009. She is a member of IEEE Computer Society.