

Embedded Architecture Description Language

Juncao Li, Nicholas T. Pilkington, and Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207, USA
{juncao, nickp, xie}@cs.pdx.edu

Qiang Liu
School of Software
Tsinghua University
Beijing, 100084, P. R. China
liuqiang@mail.tsinghua.edu.cn

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). It 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 system synthesis.

1 Introduction

Due to stringent design constraints of embedded systems such as performance, power efficiency, and manufacture costs, their hardware and software modules must closely interact and hardware/software (HW/SW) trade-offs must be effectively exploited. This demands HW/SW co-design.

To reduce manufacture and operation costs, it often requires 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 [6, 15] is a common trend. (In hardware industry, CBD is also known

as Intellectual Property (IP) based development.) The main objective of CBD is to reuse design and verification efforts.

A key 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 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 semantics gap between hardware and software components. They are often designed in their native design/implementation languages whose execution and interface semantics differ significantly.

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

- EADL is based on a unified component model for embedded systems that unifies hardware and software components and bridges the HW/SW semantics gaps.
- EADL does not dictate execution and interface semantics of hardware and software components while supporting platform-oriented semantics instantiation.
- EADL supports concise specification of embedded system architectures and also supports formulation of architectural patterns of embedded systems.
- EADL integrates architectural design with assertion-based verification (ABV). It supports association of properties (e.g., temporal correctness properties) with components and property templates with architectural patterns, to facilitate embedded system verification.

We have utilized EADL as the common representation for integrating component-based co-design, co-simulation, co-verification, and system synthesis in the Embedded Software Integrated Development Environment (ESIDE). We have instantiated EADL for two networked sensor platforms with different software languages: xUML [10] (a design-level language) and nesC [3] (an implementation-level language). Furthermore, we have applied EADL in capturing

architectures of networked sensor systems [5, 14] and guiding their HW/SW co-verification. EADL has demonstrated its flexibility in platform-oriented instantiation and its effectiveness in capturing architectures and patterns and in simplifying formulation of system and component properties.

The remainder of this paper is organized as follows. In Section 2, we review a unified component model upon which EADL is developed. 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 application of EADL in ESIDE. In Section 6, we discuss our experiences using EADL. In Section 7, we present the related work. In Section 8, we conclude this paper and discuss the future work.

2 Background: A Unified Component Model

In [18], a unified component model has been developed for embedded systems that follow an abstract but representative architecture as shown in Figure 1. Under this architec-

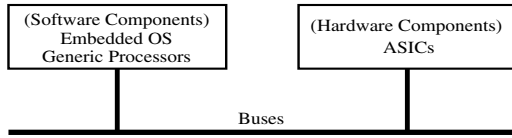


Figure 1. Abstract Architecture

ture, 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 Figure 2 has been derived, under which an embed-

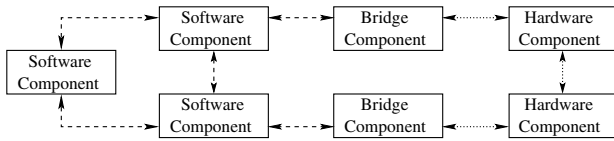


Figure 2. Unified Component Model

ded 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.

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 language for E and I , but share the same specification language for P . 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 in a system, and may be different in each system.

Composition. A composite component, $C = (E, I, P)$, is composed from a set of components, $C_0 = (E_0, I_0, P_0), \dots, C_{n-1} = (E_{n-1}, I_{n-1}, P_{n-1})$, as follows. E is constructed from E_0, \dots, E_{n-1} by connecting E_0, \dots, E_{n-1} through I_0, \dots, I_{n-1} . I may be a hardware interface, a software interface, or a hybrid hardware/software interface depending on what types of components C_0, \dots, C_{n-1} are. I includes the semantic entities from I_0, \dots, 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 [18].

3 Key Language Features of EADL

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.

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.

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.

Figure 3 shows the interface of a software sensor component, *SW_Sensor*, which uses three ports *CLK*, *ADC*, and *STQ* and provides one port *SendRcv*. The EADL interface

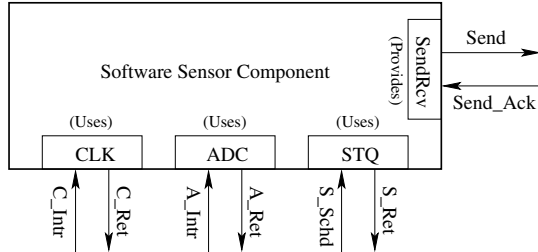


Figure 3. Interface of Software Sensor

specification of *SW_Sensor* is shown in Figure 4. The events in these ports are software messages. Figure 4 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. (Space limitation precludes showing the component implementations.)

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 Figure 5. They are specified in xPSL [17], a property specification language that extends the IEEE Property Specification Language (PSL) to specify temporal properties of both hardware and software, and entire embedded systems. xPSL utilizes a common set of temporal operators while allowing them to operate on both hardware and software events. These properties are correctness 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 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.

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

3.2 Component-Based System Architectures

EADL specifies the architecture of a composite component (a system is a composite component) through specifying its configuration which consists of all its sub-components, and the connections between them. The configuration of

```

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

software port SendRcv {
  events { output Message Send; input Message SendAck; }
}
software port CLK {
  events { output Message C_Intr; input Message C_Ret; }
}
software port ADC { provides boolean On;
  events { output Message A_Intr; input Message A_Ret; }
}
software port STQ { provides boolean Empty;
  events { output Message S_Schd; input Message S_Ret; }
}

hardware port CLK_Intr { events {output Signal intr_c; }
}
hardware port SEN_Intr { events {output Signal intr_s;
  input Signal start_s; }
}

software component SW_Sensor {
  interface { provides SendRcv; uses CLK, ADC, STQ; }
  configuration { source ("source.path"); }
}

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

bridge component BG_Sensor {
  interface { provides CLK, ADC, STQ; uses CLK_Intr, SEN_Intr; }
  configuration { source ("BG_Sensor.bg"); }
}

/*BG_Sensor.bg*/
/* Hardware interrupt to software message mappings */
(CLK_Intr.intr_c → CLK.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))}

```

Figure 4. EADL Spec for Sensor Hybrid

a hybrid sensor is shown in Figure 4. Besides the sub-components and their connections, port maps are also provided to indicate the correspondence between the ports of the composite component and the ports of its sub-components. Such a map can only be one-to-one. For a primitive component, the configuration is replaced by the path of its source file. For instance, the source code of the bridge component, *BG_Sensor*, is shown in Figure 4, which is specified in a platform-specific bridge specification language (BSL) [18]. (Further discussion about bridge components and the BSL can be found in Section 4.)

3.3 Embedded System Architectural Patterns

There often exist common patterns among architectures of systems or components. While the architecture of a system or composite component is captured as a configuration which consists of components and their connections, an ar-

```

software port SendRcv {
  events { output Message Send; input Message Send_Ack; }
  properties {
    provides assertion Sender_Handshake : Receiver_Handshake_S
      After (Send) Never (Send) UnlessAfter (Send_Ack);
    uses assertion Receiver_Handshake_S : Sender_Handshake
      Never (Send_Ack) UnlessAfter (Send);
    After (Send_Ack) Never (Send_Ack) UnlessAfter (Send);
    uses assertion Receiver_Handshake_L : Sender_Handshake
      After (Send) Eventually (Send_Ack); }
}

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

software template Sink {
  boolean DataConsumptionFlag;
  interface { uses SendRcv as RcvPort; }
  properties {
    assertion Sink_Data_PT : Sender_Handshake
      IfRepeatedly (RcvPort.Send) Repeatedly (DataConsumptionFlag);
    IfRepeatedly (RcvPort.Send) Repeatedly (!DataConsumptionFlag); }
}

software Pattern SourceToSink {
  configuration {
    template Source; template Sink;
    connection (Source.SendPort, Sink.RcvPort); }
  properties {
    assertion Data_PT
      Repeatedly (Sink.DataConsumptionFlag);
    Repeatedly (!Sink.DataConsumptionFlag); }
}

```

Figure 5. An Example Architectural Pattern

chitectural pattern is captured as a configuration template which consists of both concrete components and component templates as well as their connections. Abstraction of patterns from component or system architectures is based on abstraction of component templates from components. A component template is a skeleton that captures the common interface, variables, and properties shared by a class of similar components. The properties are defined over the interface and the variable set of the component template.

Based on the above abstractions, an architectural pattern consists of three parts: (1) a partial description of the interface for a component or system following this pattern, which consists 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.

We illustrate the architectural pattern concept with a simple but representative pattern of embedded systems, the *SourceToSink* pattern, as shown in Figure 5. There are two component templates 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 in system complexity and size.

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.

We first give our definition of the platform concept. To simplify system design and verification, the platform concept will hide details of processors, buses, and embedded OS through definition of a platform-specific BSL. The semantics of hardware, software, and bridge components together abstract processors, buses, and embedded OS. With this abstraction, an embedded system platform for an application domain consists of: (1) software, hardware, and bridge design/implementation languages, (2) compiler supports to these languages for simulation, verification, and deployment, 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 Figure 6. The software, hardware, and bridge semantics

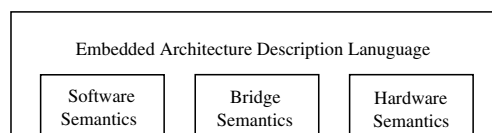


Figure 6. Instantiation of EADL

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 the property specification language, xPSL, since xPSL provides the temporal operators, but does not dictate the semantics of the atom propositions, basically, the events.

Next, we present an EADL instantiation on a sensor system platform. The components and pattern in Figure 4 and Figure 5 are based on this platform. For a software component, to support high-level design, we adopt the model-driven development [10] and specify the design *E* of software components in xUML [10], 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. 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 [18]. 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 Figure 4 is specified in this language.

5 EADL in Embedded System Development

5.1 Embedded System Integrated Development Environment

Figure 7 illustrates an Embedded System Integrated Development Environment (ESIDE) that we have been developing. In ESIDE, EADL serves as the vehicle for in-

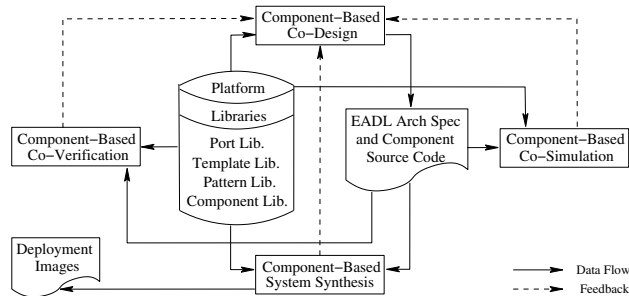


Figure 7. Embedded System IDE (ESIDE)

tegrating component-based co-design, co-simulation, co-verification, and system synthesis. Embedded system development using ESIDE starts with selecting a platform for

embedded systems. This platform is utilized to instantiate EADL and also provides the platform-specific libraries of ports, component templates, patterns, and reusable components. Component-based co-design generates the architectural specifications of the system being designed and its components. Source code of these components, developed anew or reused, is associated with their architectural specifications. The architectural specifications are utilized in both component-based co-simulation and co-verification together with the platform-specific libraries. There are feedbacks from co-simulation and co-verification to co-design for reporting design errors that are detected. The validated and verified designs are compiled into deployment images for both hardware and software by the component-based system synthesis. There are also feedbacks from system synthesis to co-design for reporting deployment issues.

5.2 Component-Based Co-Design

VisualEADL Toolkit. To support architectural design using EADL, we have developed the VisualEADL toolkit, implemented as an Eclipse plug-in. VisualEADL serves as the frontend of ESIDE and integrates the co-simulation, co-verification, and system synthesis toolkits. VisualEADL starts by requesting the designer to select an embedded system platform. VisualEADL then populates its modeling environment with ports, component templates, architectural patterns, and reusable hardware, software, and hybrid components out of the platform-specific libraries.

Component-Based Co-Design Lifecycle. The component-based co-design lifecycle, as supported by ESIDE, consists of the following three major phases.

Platform Creation. The lifecycle starts with creation of an embedded system platform. A platform can be created by an embedded system platform vendor or in house by system designers. VisualEADL provides an interface for creating platforms and populating the platform-specific libraries. In this phase, the hardware platform, the operating system, and the hardware and software design/programming languages are selected. The BSL is defined by abstracting the platform components and interfacing the hardware and software semantics. Primitive hardware and software components are identified from previous ad-hoc systems built on this platform and from architectural patterns of the application domain or are developed from scratch. These components are designed with VisualEADL and coded in their respective design/programming languages. These primitive components can be further composed bottom-up to develop reusable composite components of this platform.

We have created a platform for networked sensor systems: (1) software components specified in nesC [3], a native programming language for sensor software; (2) hardware components specified in Verilog; (3) bridge compo-

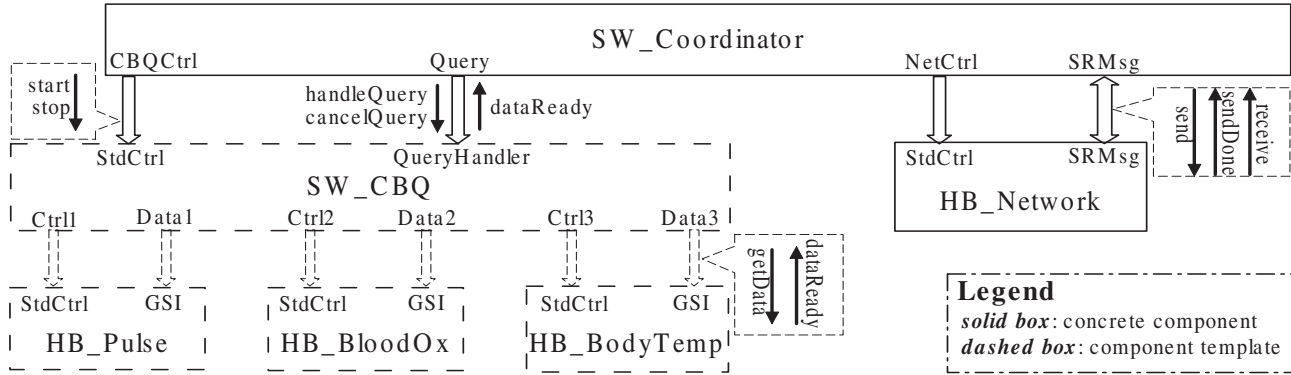


Figure 8. CodeBlue Architectural Pattern for Medical Sensor Systems

nents specified in a BSL that is similar to the language used in Figure 4 while interactions with software components are conducted through function calls instead of message-passing. The platform-specific libraries are populated as we used ESIDE to re-engineer the networked sensor systems included in the TinyOS [5] distribution. The libraries are further expanded as we re-engineered the sensor systems in the CodeBlue [14] distribution.

Figure 8 shows the CodeBlue architectural pattern for medical sensor systems and Figure 9 shows the EADL specification of this pattern which includes two concrete components and two component templates. In particular,

```

hybrid pattern P_HB_CodeBlue {
  configuration {
    component SW_Coordinator, HB_Network;
    template T_SW_CBQ;
    multi template T_HB_Sensor[NumofSen];
    connection (HB_Network.StdCtrl, SW_Coordinator.NetCtrl);
    connection (HB_Network.SRMsg, SW_Coordinator.SRMsg);
    connection (T_SW_CBQ.StdCtrl, SW_Coordinator.CBQCtrl);
    connection (T_SW_CBQ.QueryHandler, SW_Coordinator.Query);
    connection (T_HB_Sensor.StdCtrl, T_SW_CBQ.Ctrl);
    connection (T_HB_Sensor.GSI, T_SW_CBQ.Data);
  }

  properties {
    assertion CBP1 : CBA1
      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 CBA1
      After(SW_Coordinator.Query.handleQuery(SRC, SINK, T))
      EventuallyAlways(T_HB_Sensor[SRC].devSenVar>T);
  }
}

```

Figure 9. EADL Spec for CodeBlue Pattern

T_HB_Sensor is a hybrid component template which can be instantiated multiple times in a system instance.

System Development. New systems on the platform are developed top-down. Given its functional requirements, a system is decomposed into its hardware, software, and bridge components. The interface of each component is defined and its properties are specified. The decomposition considers components from the platform-specific libraries. If there

is an existing component matching the interface and properties, the component can be reused. If there is no matching component, the component is either developed from scratch as a primitive component or further decomposed.

The top-down decomposition reuses architectural patterns as possible. We illustrate the role of architectural patterns with the decomposition of a multi-sensor system. The functional requirement of this system is that it should properly operate multiple hardware sensors, e.g., temperature and pulse sensors, and transmit their readings. The system can be decomposed into its hardware and software components as shown in Figure 10. The decomposition fol-

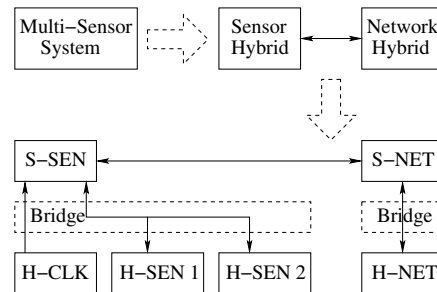


Figure 10. Pattern-Guided Decomposition

lows the architectural patterns of the networked sensor domain. The top-level pattern is the *SourceToSink* pattern in Figure 5, following which the system is decomposed into two hybrid components: sensor hybrid and network hybrid. The two components are then further decomposed, which may follow additional architectural patterns.

Platform Extension. New components may be introduced in top-down development of new systems and they may also be introduced through bottom-up component development due to technology advances, such as new sensing and communication modules for the networked sensor application domain. The new components can be further composed with existing components or among themselves to construct larger composite components bottom-up.

5.3 Component-Based Co-Simulation

Figure 11 illustrates the architecture of a co-simulator for component-based embedded systems. The keys of this ar-

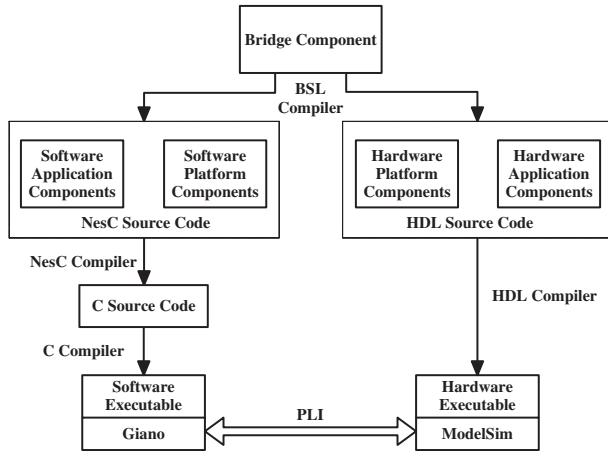


Figure 11. Co-Simulation Architecture

chitecture are the compilers for hardware, software, and bridge components. The BSL compiler generates the hardware platform components such as the bus and the interrupt queue in Verilog and the software platform components such as the interrupt manager and the scheduler in nesC. Furthermore, the BSL compiler configures the platform components of an embedded system such as the processor. It is also responsible for establishing mappings between hardware signals and software functions/variables by generating hardware code in Verilog and software code in nesC. The HW/SW event mappings are also used to configure the interface between the software simulator, Giano [1], and the hardware simulator, ModelSim [11], which is used for HW/SW synchronization. The software (or hardware, respectively) components including those generated by the BSL compiler are compiled into executables by the software (or hardware) compiler.

5.4 Component-Based Co-Verification

In component-based HW/SW co-verification [18], hardware and software components are verified as they are developed bottom-up. Properties of a primitive component are directly model-checked and properties of a composite component are checked on its abstractions constructed from verified properties of its sub-components. A system is verified top-down as it is developed 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.

Architecture-Based Property Formulation and Reuse.

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 for establishing these properties. We utilize component templates and architectural patterns, and their associated property templates to address this challenge [8]. The basic approach is to use the property templates to guide formulation of properties of components and systems as they are instantiated from the component templates and architectural patterns. We also support definition of property decomposition strategies for an architectural pattern which specify how to decompose pattern-level property templates into properties of the concrete components and property templates of the component templates. EADL integrates architectural design with ABV: as components and systems are architected, their properties are formulated. Property formulation and reuse take place on three levels: port, component template, and architectural pattern.

Examples of Architecture-Based Reuse.

There is a property template *Data_PT* associated with the *SourceToSink* pattern, which can be used to generate a property to be verified on a system or a composite component following this pattern, for instance, the multi-sensor system in Figure 10. The property template asserts that there is repeated data consumption at the sink. There are also property templates *Src_Data_PT* and *Sink_Data_PT* associated with the *Source* and *Sink* templates, which can be used to generate component properties needed for verification of the pattern-level property. Since the component templates include the port *SendRcv*, they also inherit the port properties. What properties to inherit depends on whether the port is provided or used. If *HB_Sensor* in Figure 4 is considered as an instantiation of the *Source* template, then the property template of *Source* can be instantiated and verified on *HB_Sensor*.

A property template, *CBPI*, is associated with the Code-Blue 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.

5.5 Component-Based System Synthesis

The component-based system synthesizer employs a similar architecture as the component-based co-simulator. The key difference is that the compilers in the co-simulator compile the components into executable for simulation while the compilers in the system synthesizer compile the components into hardware (or software, respectively) images that are used to program FPGAs and configure hardware components (or loaded by the operating systems for execution).

6 Experiences with EADL

Using EADL as supported by ESIDE, we have successfully re-engineered the sensor systems from the TinyOS distribution and further the CodeBlue distribution: (1) created a sensor system platform using ESIDE, (2) restructured the systems following the unified component model, (3) specified all the components in EADL and included them in the platform libraries, (4) co-simulated and co-verified all the systems against a common set of system properties capturing minimal system correctness, for instance, no buffer overflow (safety) and repeated transmission (liveness), and (5) compared the systems synthesized from the component-based designs in EADL with the original systems. The component-based co-verification was accomplished on all the systems, many of which failed a straightforward application of model checking to entire systems due to state space explosion. Several buffer overflow vulnerabilities in the sensor systems were detected. A similar re-engineering effort for the systems based on the Microsoft Invisible Computing platform [4] is currently ongoing.

7 Related Work

There have been much research on both hardware, software, and embedded systems architecture description languages (ADLs) (see [16, 9] for their comprehensive surveys). Among those ADLs, the most closely related are SAE AADL [12], Metropolis [13], and Ptolemy [7]. The AADL is an industry standard designed for the specification, analysis, and automated integration of real-time performance-critical distributed computer systems. Metropolis features a flexible and formal semantics based upon the tagged signal model. Ptolemy focuses on component-based heterogeneous modeling. It uses tokens as the underlying communication mechanism. 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 system synthesis based on these semantics. The EADL representation of architectural patterns is partially motivated by ACME [2]. In ACME, architectural patterns are explicit semantic entities.

8 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 system 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.

9 Acknowledgment

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).

References

- [1] A. Forin, B. Neekzad, and N. L. Lynch. Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research, 2006.
- [2] D. Garlan, R. T. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON*, 1997.
- [3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI*, 2003.
- [4] J. Helander and A. Forin. Mmlite: a highly componentized system architecture. In *8th ACM SIGOPS European Workshop*, 1998.
- [5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *ASPLOS*, 2000.
- [6] M. F. Jacome and H. P. Peixoto. A survey of digital design reuse. *IEEE Design and Test of Computers*, 18(3), 2001.
- [7] E. A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, UC Berkeley, 2003.
- [8] J. Li, F. Xie, and H. Liu. Guiding component-based hardware/software co-verification with patterns. In *EUROMICRO SEAA*, 2007.
- [9] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1), 2000.
- [10] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.
- [11] Mentor Graphics. ModelSim. <http://www.mentor.com>.
- [12] S. of Automotive Engineers (SAE). *The SAE AADL Language Standard (AS-5506)*. SAE, 2004.
- [13] A. L. Sangiovanni-Vincentelli. Quo vadis sld: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3), 2007.
- [14] V. Shnayder, B. R. Chen, K. Lorincz, T. R. F. Fulford-Jones, and M. Welsh. Sensor networks for medical care. Technical report, Harvard University, 2005.
- [15] C. Szyperski and et al. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 2002.
- [16] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system-on-chip design. In *APCHDL*, 1999.
- [17] F. Xie and H. Liu. Unified property specification for hardware/software co-verification. In *COMPSAC*, 2007.
- [18] F. Xie, G. Yang, and X. Song. Component-based hardware/software co-verification. In *MEMOCODE*, 2006.