ESIDE: An Integrated Development Environment for Component-Based Embedded Systems

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

Abstract

In this paper we present ESIDE, an integrated development environment for component-based embedded systems. It leverages component-based software engineering principles to facilitate efficient, scalable, and robust hardware/software co-design, co-simulation, co-verification, and their seamless integration. We first describe the architecture and features of ESIDE. We then discuss several design decisions that we faced in developing ESIDE and the trade-offs in making these decisions. To provide perspective, we report our experiences in re-engineering TinyOS-based networked sensor systems into complete component-based designs that cover both hardware and software.

1 Introduction

From mobile phones to automobiles to household appliances, embedded systems are a ubiquitous computing substrate for today's technologies. As the complexity of these systems increases, so does the need for a scalable development methodology; one that allows simultaneous co-development of both hardware and software.

Due to stringent design constraints on embedded systems such as performance, power efficiency, and manufacturing 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. Co-simulation has been an effective approach to jointly evaluate hardware and software design. It can detect bugs and reveal run-time inefficiencies under the simulation coverage. To ensure dependability, certain critical properties need to be proved, thus formal co-verification, which exhaustively searches the system state spaces, is necessary.

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 appropriate and scalable approach to embedded system development.

Currently, co-simulation and co-verification are rarely considered until a system is largely designed, and are not tightly integrated with co-design. This makes co-simulation and co-verification hard and sometimes even infeasible. For instance, complex couplings between components can dramatically increase the complexity of co-verification; however, these couplings are often needed for performance improvement and power reduction. The competing demands of verification and optimization must be considered starting at co-design, while co-simulation and co-verification must provide informative and timely feedback to co-design.

A key to integrating co-simulation and co-verification with co-design is support from an integrated development environment. We have developed an Embedded System Integrated Development Environment (ESIDE), which is based on the Embedded Architecture Description Language (EADL) [9]. The key features of ESIDE are as follows:

- Platform-based development: The developer can select from different embedded system platforms. A platform defines the operational semantics of the hardware and software, and provides compiler, simulator, and verifier supports and reusable design libraries.
- Visual architectural modeling of components/systems, component templates, and architecture patterns.
- Reuse library management: Components, component templates, and architectural patterns are properly grouped and managed based on the platform.
- Integration of architectural design and assertion-based verification: Assertions are introduced in co-design and validated in co-simulation and co-verification.
- Integrated co-simulation and co-verification: Co-simulation and co-verification operate on the same component-based architectural model from co-design, and no additional manual setup effort is needed.

Using ESIDE, we have assembled several embedded system platforms: the Mica platform [2] and the PC simulation platform [7] for networked sensor systems, and the Microsoft Invisible Computing platform [12]. Systems designed based on both Mica and PC platforms target execution on TinyOS [7], a component-based run-time environment for networked sensors. For TinyOS-based systems, (1) hardware resources are limited, known statically, and not component-based; (2) applications are built from a suite of reusable system components coupled with application-specific code; (3) the software/hardware interfaces are blurred and not component-based; (4) all software components are unfolded into C code during the compilation, which may introduce problems for simulation and verification due to the loss of component-based architectures.

We have re-engineered 12 TinyOS-based sensor systems using ESIDE. While re-engineering the software components and re-designing the hardware into components, we made several design modifications towards better integration of co-design, co-simulation and co-verification. For example, in the lowest-level TinyOS components, we replaced the function calls directly interacting with the hardware with software interfaces connecting to bridge components [5]. Bridge components capture HW/SW interactions, hide implementation details that are not component-based, and support efficient transaction-level simulation.

The reminder of this paper is organized as follows. In Section 2, we provide the background on EADL and TinyOS. In Section 3, we introduce the overall architecture and key features of ESIDE. In Section 4, we discuss the design decisions of ESIDE and the trade-offs leading to these decisions. In Section 5, we summarize our experiences in re-engineering TinyOS-based sensor systems using ESIDE. In Section 6, we present the related work. Finally in Section 7, we conclude this paper and discuss future work.

2 Background

2.1 Embedded Architecture Description Language

EADL supports component-based architecture specification on three levels: (1) component interfaces built upon events and ports, (2) component-based system architectures, and (3) architectural patterns based on component templates.

Component Interfaces. EADL employs the event concept to abstract all concrete hardware and 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. EADL utilizes the port concept to group events that together realize a certain functionality. A portion of the `StdControl` port is given in Figure 1. As this port was developed on the TinyOS platform the software events assume *function* semantics.

```
software port StdControl {
  events {
    input function { result_t init(); };
    input function { result_t start(); };
    input function { result_t stop(); };
  }

  properties {
    provides assertion P1:  U1
      s1:  After_Eventually_(init.call=TRUE,
             init.ret=TRUE)
      s2:  After_Eventually_(start.call=TRUE,
             start.ret=TRUE)
      s3:  After_Eventually_(stop.call=TRUE,
             stop.ret=TRUE)
    uses assertion U1
      s1:  Never_UnlessAfter_(start.call=TRUE,
             init.call=TRUE)
      s2:  Never_UnlessAfter_(stop.call=TRUE,
             init.call=TRUE)}
}
```

Figure 1. Port Example

Depending on whether a component is providing or utilizing the functionality, a port can be a "provided" or "used" 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 directions. When composed, components are connected on the more abstract port level, instead of the detailed event level.

Temporal property specification is an integral part of EADL. In the above example, two groups of properties, `P1` and `U1`, are specified. During verification, `P1` will be checked on every component providing `StdControl`, asserting that the functions will terminate; `U1` will be checked on every component using `StdControl`, asserting that functions `start` and `stop` should never be called before function `init` is called. The "provides" and "uses" properties can serve as assumptions for each other. For example, `P1` uses `U1` as an assumption.

Component-Based System Architectures. EADL specifies the architecture of a composite component (note that a system is a composite component) by specifying its configuration, which consists of all its sub-components and their connections. Figure 2 illustrates the `TimerC` component from the TinyOS platform. Besides the sub-components and their connections, port mappings are also provided to indicate a correspondence between the ports of the composite component and those of its sub-components, for example, the mapping between the `Timer` port of the `TimerC` component and the `Timer` port of its sub-component, `TimerM`. Such a mapping must be one-to-one and between ports of the same type. For a primitive component, the configuration is replaced by the path of its source file.

```
software component TimerC {
  interface {
    provides StdControl, Timer_p as Timer;
    uses PowerManagement, Clock, IOH;

    mapping(Timer, TimerM.Timer);
    mapping(StdControl, TimerM.StdControl);
    mapping(PowerManagement,
              TimerM.PowerManagement);
    mapping(Clock, TimerM.Clock);
    mapping(IOH, TimerM.IOH);}

  configuration {
    component TimerM;
    component NoLeds;
    connection(NoLeds.Leds, TimerM.Leds);}

  properties {
    assertion ClockP1
      s1:  After_Eventually_
            (StdControl.init.call=TRUE,
              Clock.setRate.call=TRUE)}
}
```

Figure 2. Component Example

Properties can be specified directly on components (e.g., those in Figure 2). These are combined with properties derived from the component's ports and the pattern(s) it implements to form the complete set of component properties.

Embedded System Architectural Patterns. There often exist common patterns among architectures of systems or components. An EADL architectural pattern consists of three parts: (1) a partial description of the interface for a composite component or system following this pattern, which consists of ports, (2) a configuration template, which is composed of components and component templates and from which the configuration of the composite component or system is instantiated, and (3) property templates specified on the interface and the configuration template, from which the component or system properties are instantiated.

Space limitation prohibits an in-depth description of patterns and other advanced features of EADL (refer to [9]).

2.2 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 [4], 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 EADL primitive components) which provide application code implementing one or more component interfaces, and configurations (analogous to EADL 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 Architecture and Features of ESIDE

The architecture of ESIDE is shown in Figure 3. 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.

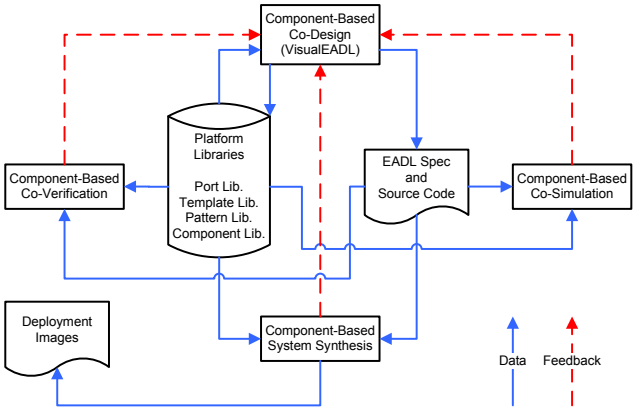

Figure 3. ESIDE Architecture and Features

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 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 monitoring the vital signs of a patient in emergency situations [14]. We have re-engineered the original TinyOS-based implementation of CodeBlue with ESIDE.

3.1 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 Figure 4. 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 a platform that reflected this environment, which we refer to as the TinyOS platform. TinyOS is a software development environment following the traditional model of embedded system development (see Section 4.2). 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) [5] 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 4 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 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.

3.2 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, resulting in 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.

Systems are designed visually in ESIDE, as shown in Figure 4. 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 Figure 4 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.

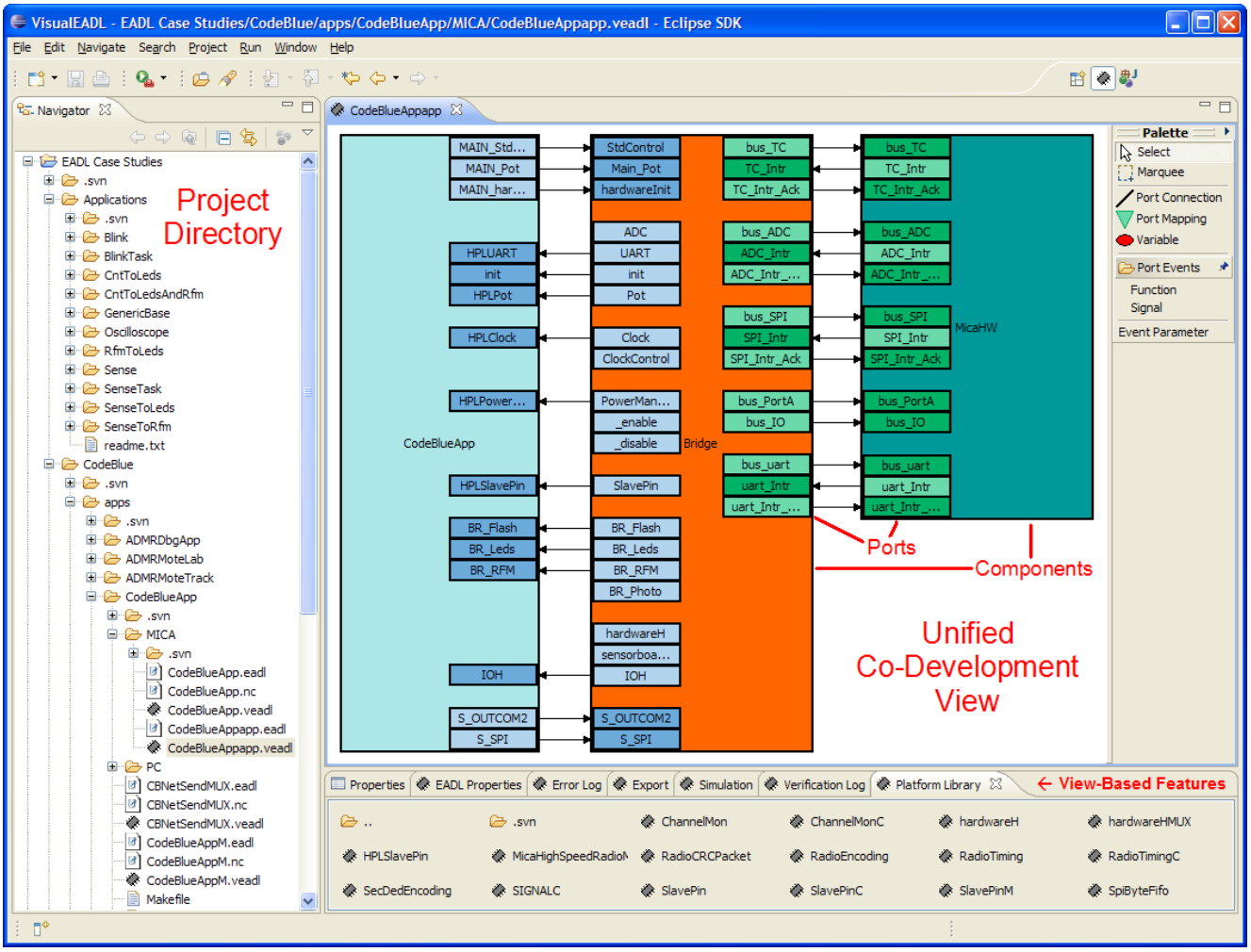

Figure 4. ESIDE Visual Modeling Interface and Platform View

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. Figure 4 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. Figure 5 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 discussions in Section 4.3.)

3.3 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. Figure 6 illustrates the co-simulation environment setup

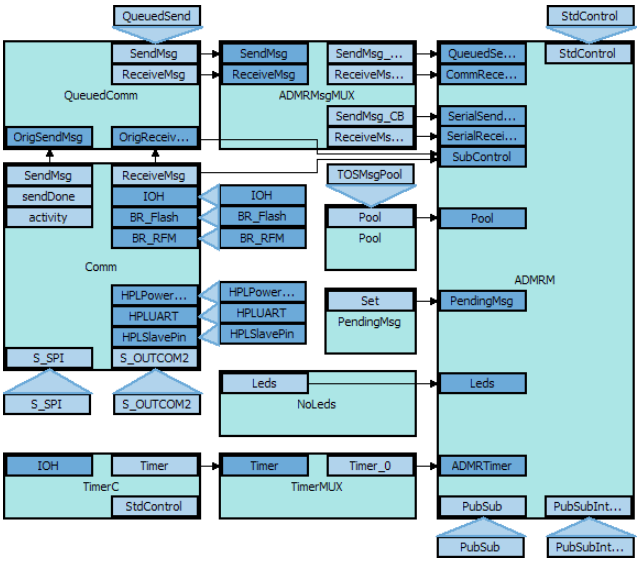

Figure 5. Configuration of ADMR Component

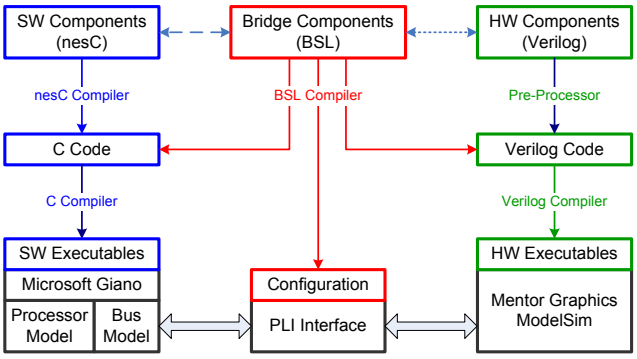

Figure 6. Co-Simulation Environment Setup

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 [11] and Giano [3] 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 and the bus 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 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 Figure 7. (The three separate views in Figure 7 can be selected from the view-based feature tabs in Figure 4.) 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.

3.4 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 Figure 7). 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 Figure 8, 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 verification

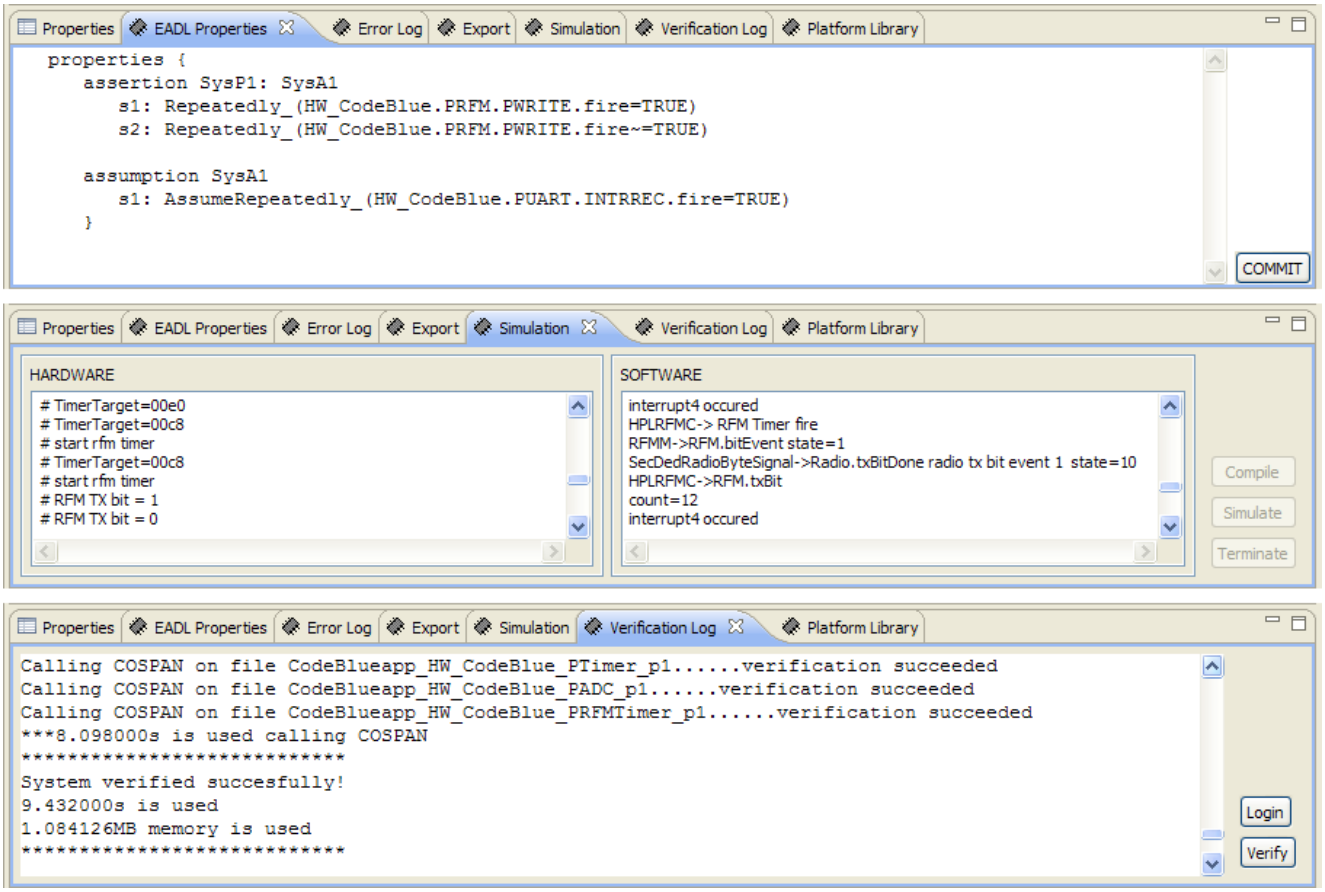

Figure 7. Property Specification, Simulation, and Verification Views

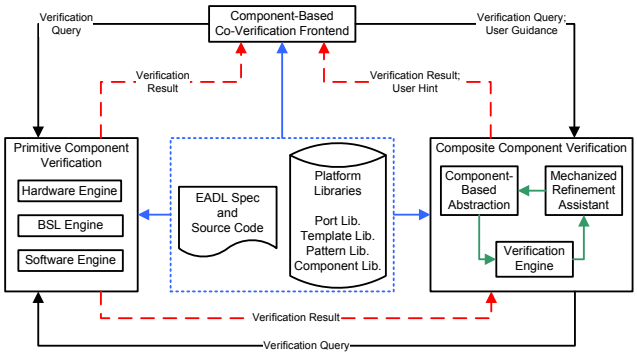

Figure 8. Co-Verification Tool Support

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 abstractions of the sub-components.

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.

3.5 Unified Playback of Error Traces

As discussed in their respective sections, the co-simulation and co-verification features of ESIDE 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 [6], 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.

Figure 9 illustrates how various error traces are unified for playback in ESIDE. A common format for traces that

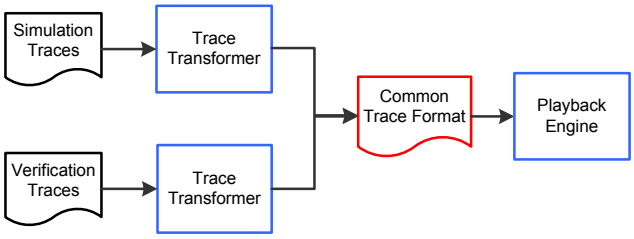

Figure 9. Trace Unification for Playback

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. This feature is currently under development and will be integrated into future versions of ESIDE.

3.6 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 [7], 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 53 components and 49 ports. In the original, there were 48 components and 30 ports. The five extra components are all for overcoming various signal routing problems inherent in the paradigm shift (see Section 4.3). Most extra ports were introduced while making the hardware interfaces explicit.

4 Methodology and Design Decisions

4.1 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 developing 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 that are 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 by 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.

4.2 Hardware/Software Boundary

The traditional model of embedded system development is stack-based. TinyOS components communicate with hard-

ware at the primitive level through special library function calls. This works well for software development on predetermined hardware. However, in the co-development model, 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, 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 figure 4. 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.

4.3 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.

5 Evaluation

Effectiveness. We have evaluated ESIDE by remodeling 12 TinyOS-based sensor systems. Except for the design changes in Section 4, 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 4.2, 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, which is used for TOSSIM [7] simulation. All the remodeled systems are compiled back into TinyOS code and simulated by TOSSIM to ensure that our remodeling was faithful.

# of systems remodeled		12
	Orig	New
# of components in Mica platform library	55	64
# of components in PC platform library	58	63
# of ports/interfaces in Mica platform library	30	49
# of ports/interfaces in PC platform library	29	46
# of components specific for each system	38	41
# of ports/interfaces specific for each system	8	10
# of hardware components developed	N/A	12
# of bridge components developed	N/A	2

Table 1. Remodeling Statistics

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

System	Co-Simulation		Co-Verification	
	RTL (Sec)	TL (Sec)	Time (Sec)	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 2. Simulation and Verification Statistics

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 [5]. 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 the system-level property on abstractions constructed from properties of the first-level components [10]. 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.

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

6 Related Work

There have been many integrated development environments for embedded systems, such as Metropolis [13], Milan [1], and Ptolemy [8]. Metropolis supports platform-based design, behavior-architecture mapping, and orthogonalization of concerns at the levels of communication-computation-coordination, architecture-function-mapping, and behavior-performance. It features a flexible and formal semantics based upon the tagged signal model that allows it to represent a wide variety of models of computation. Milan employs a model-based solution for co-design and co-simulation. Different simulators can be integrated once different simulation models are interpreted into the common model supported in Milan. Ptolemy focuses on component-based heterogeneous modeling. It uses tokens as the underlying communication mechanism and uses hierarchical composition to handle heterogeneity. ESIDE, however, does not require a specific computation model or semantics, and it supports instantiation of EADL on any hardware and software execution semantics to enable component-based co-design, co-simulation, co-verification, and co-synthesis.

7 Conclusions and Future Work

In this paper, we have presented ESIDE, an integrated development environment for component-based embedded systems. Case studies on re-engineering networked sensor systems have demonstrated that component-based development using ESIDE facilitates effective co-simulation and co-verification while preserving the efficiencies needed by embedded systems. Next, we will extend ESIDE to animate feedbacks from co-simulation and co-verification on the visual design models and support analysis of non-functional properties such as performance and power consumptions.

8 Acknowledgement

This research received financial support from Semiconductor Research Corporation (Contract #: 1356.001) and National Science Foundation (Grant #: 0720546).

References

[1] A. Bakshi, V. Prasanna, and A. Ledeczi. Milan: A model based integrated simulation framework for design of embedded systems. In *LCTES*, 2001.

[2] Crossbow. Mica. http://www.xbow.com.

[3] A. Forin, B. Neekzad, and N. Lynch. Giano: The two-headed system simulator. Technical Report MSR-TR-2006-130, Microsoft Research, 2006.

[4] D. Gay, P. Levis, R. Behren, M. Welsh, B. E., and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, 2003.

[5] K. Hao and F. Xie. Componentizing hardware/software interface design. In *DATE*, 2009.

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

[7] 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.

[8] E. A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, UC Berkeley, 2003.

[9] J. Li, N. T. Pilkington, F. Xie, and Q. Liu. Embedded architecture description language. In *Proc. of COMPSAC*, 2008.

[10] J. Li, X. Sun, F. Xie, and X. Song. Component-based abstraction and refinement. In *Proc. of ICSR*, 2008.

[11] Mentor Graphics. ModelSim. http://www.mentor.com.

[12] Microsoft Research. Microsoft invisible computing. http://research.microsoft.com/invisible.

[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.