Unified Property Specification for Hardware/Software Co-Verification *

Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207, USA
xie@cs.pdx.edu

Huaiyu Liu
Corporate Technology Group
Intel Corporation
Hillsboro, OR 97124, USA
huaiyu.liu@intel.com

Abstract

Hardware/software co-verification is becoming an indispensable tool for building highly trustworthy embedded systems. A stumbling block to effective co-verification using model checking is the lack of support to unified property specification for hardware, software, and entire embedded systems. In this paper, we develop xPSL, a unified property specification language for co-verification. xPSL extends the IEEE Property Specification Language (PSL) to support specification of temporal assertions over both hardware and software events. The semantics of hardware and software events and their temporal correlations are formalized based on translation of both hardware and software semantics to a common formal semantic basis. xPSL has been applied in co-verification to specify properties of hardware and software components, and furthermore entire embedded systems. Case studies have shown that xPSL is very effective in enabling co-verification of system-level properties and facilitating compositional reasoning.

1 Introduction

Embedded systems are often required to be highly trustworthy. Building such systems requires extensive verification. Power and performance constraints of embedded systems require that their hardware and software closely interact and the hardware/software (HW/SW) trade-off be effectively exploited. This demands HW/SW co-design and, thus, HW/SW co-verification of embedded systems.

Model checking [3] 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 [3] to co-verification by simplifying utilization of properties of hardware and software components as assumptions when verifying other components.

A major challenge to unified property specification is the significant HW/SW semantic gap. Hardware usually follows synchronous clock-driven semantics while software semantics are more diversified, e.g., asynchronous interleaving message-passing semantics and asynchronous event-driven call-return semantics. A unified property specification language must be able to capture hardware and software events in various semantics and their temporal correlations. Other desirable characteristics of such a language include: (1) expressive, i.e., able to express a wide range of temporal properties, (2) easy to be adopted by the industry, in particular, applicable in assertion-based verification (ABV) [10], and (3) amenable to verification reuse.

In this paper, we develop 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). 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 both hardware and software semantics to a common formal semantic basis. xPSL is fully compatible with PSL and, therefore, 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.

We have applied xPSL in different approaches to co-verification. Case studies on networked sensors have shown xPSL is very effective in supporting unified property specification, enabling co-verification of system-level properties, and facilitating compositional reasoning for co-verification.

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

The reminder of this paper is organized as follows. We provide the background of this work in Section 2. In Section 3, we define xPSL through extending PSL. We formalize the xPSL semantics based on semantics translation in Section 4. In Section 5, we present application of xPSL in different approaches to co-verification. We discuss related work in Section 6 and conclude this paper in Section 7.

2 Background

In this section, we first introduce application of ABV and PSL in the hardware domain. We then present a hardware semantics that PSL already supports, two representative software semantics that we will extend PSL to support, and a formal semantics based on which we are going to unify property specification for hardware and software.

2.1 Assertion-Based Verification (ABV) and IEEE Property Specification Language (PSL)

ABV was introduced to enable integration of validation and verification with design of hardware. ABV requires component developers to specify temporal correctness properties of components as they are developed. Such properties can be used in testing, formal verification, and run-time checking. Component properties are often specified in standard property specification languages such as PSL, which facilitates reuse of component properties.

PSL is the de facto standard property specification language for ABV of hardware. PSL has a nice layered structure, which consists of four semantic layers: boolean layer, temporal layer, verification layer, and modeling layer. The boolean layer supports specification of event monitors, essentially propositional boolean expressions, over behaviors of hardware systems. The temporal layer defines temporal assertions, essentially LTL [14] or CTL [2] formulae, based on the event monitors. The verification layer governs how the assertions are used in verification, i.e., as properties or assumptions, and defines their assume-guarantee relationships. The modeling layer supports environment modeling.

An example assertion specified in PSL is shown in Figure 1. In this assertion, *H-SEN.start_s* is a boolean expression defined using the boolean layer of PSL. It refers to the *start_s* signal in the hardware module, *H-SEN*, and is true when this signal is set. The keywords, *always*, *next!*, and *eventually!*, are temporal operators from the temporal layer of PSL and are corresponding to temporal operators of LTL. The keyword, *assert*, is a verification directive indicating that this assertion is a property to be checked on the hardware module. The property asserts that every time after the *start_s* signal is set, the *intr_s* signal will eventually be set.

assert always (H-SEN.start_s → (**next!** (**eventually!** H-SEN.intr_s)));

Figure 1. Example Hardware Property in PSL

2.2 Hardware, Software, and Formal Semantics

Synchronous Clock-Driven Semantics of Verilog. In the IEEE standard, the semantics of the Verilog hardware description language is defined informally by means of a discrete-event simulator. We adopt the semantics of a Verilog subset that can be formalized via translation to the S/R language [4], the formal input language of the COSPAN model checker [4]. This translation has been implemented in FormalCheck [9]. Abstractly, a Verilog model consists of a set of inter-connecting modules. The sequential portion of a module consists of flip-flops that keep the states of the module. The outputs of a flip-flop can be updated based on its inputs at the positive edge or negative edge of the system clock. The outputs of combinational circuits are updated based on their inputs instantly if zero delay is assumed.

Asynchronous Event-Driven Call-Return Semantics of TinyOS C Subset. TinyOS [5] is a component-based runtime environment for networked sensors. A system developed in the TinyOS C subset consists of a scheduler and a hierarchy of components. A component has four interrelated parts: a fixed-size data frame, a set of command handlers, a set of signal handlers, and a bundle of tasks. Command handlers, signal handlers, and tasks execute in the context of the data frame and operate on its state, and are implemented as functions that are invoked following the call-return semantics. Higher level components issue commands to lower level components and lower level components send signals to the higher level components. The lowest level components abstract hardware. A hardware interrupt triggers a fountain of processing that goes upward via signals and can bend downward via commands. Commands and signals are intended to perform a small, fixed amount of work. Tasks perform the primary work and are atomic with respect to other tasks though they can be preempted by interrupts. Tasks can call lower level commands, send higher level signals, and post other tasks within a component. Tasks allow concurrency since they execute asynchronously with respect to interrupt handling.

Asynchronous Interleaving Message-Passing Semantics of Executable UML. Executable UML (xUML) [13] is an executable dialect of UML supporting model-driven development of embedded software. xUML features an asynchronous interleaving message-passing (AIM) semantics. A system consists of a set of interacting object instances. The behavior of each object instance is specified by an extended Moore state model in which each state may be associated with a state action. A state action is a program

segment that executes upon entry to the state. Object instances interact through asynchronous message-passing. In a system execution, at any given moment only one object instance can progress by executing a state transition or a state action in its extended Moore state model. The execution of a state transition or a state action is run-to-completion.

ω-Automaton Semantics of S/R. In S/R, a system P is composed of synchronously interacting processes, conceptually ω-automata [8]. A process consists of state variables, selection variables, inputs, state transition rules, and selection rules. Selection variables define the outputs of this process. This process inputs a subset of all the selection variables of other processes. State transition rules update state variables as functions of the current state, selection variables, and inputs. Selection rules assign values to selection variables as functions of state variables. Such a function is non-deterministic if several values are possible for a selection variable in a state. The "selection/resolution" execution model of S/R is synchronous clock-driven, under which a system of processes behaves in a 2-phase procedure every logical clock cycle: *[1: Selection Phase]* Every process "selects" a value possible in its current state for each of its selection variables. The values of the selection variables of all the processes form the global selection of the system. *[2: Resolution Phase]* Every process "resolves" the current global selection simultaneously by updating its state variables according to its state transition rules. In S/R, a property T to be checked on a system P is also modeled by an ω-automaton. COSPAN performs the verification by checking the ω-automata language containment, $\mathcal{L}(P) \subseteq \mathcal{L}(T)$.

3 xPSL: Extending PSL for Co-Verification

There are two major tasks in extending PSL for HW/SW co-verification: (1) developing language constructs for formulating software events and their temporal correlations with hardware events; (2) formalizing the semantics of these constructs based on translation of hardware and software design/implementation languages into a common semantic basis. We introduce the xPSL language extensions to PSL in this section and formalize these extensions in next section.

We extend the boolean layer of PSL to support monitoring of software events, identify necessary restrictions to the temporal layer when it is used to specify assertions over software events, and keep the other layers the same. This provides backward compatibility with PSL and a unified view of hardware, software, and system-level properties.

3.1 Defining xPSL Profiles for Software Languages

PSL currently provides profiles (a.k.a. flavors) for standard hardware design languages such as Verilog and VHDL. Such a profile basically defines how to specify event monitors, i.e., propositional boolean expressions, over hardware systems designed in a specific hardware design language. The PSL profile definition has a rough structure as shown in Figure 2 with some unnecessary complications omitted. (For more details, see [7].) Using such a profile, hardware

```
HDL_or_PSL_Expression ::= HDL_Expression | PSL_Expression | . . .

PSL_Expression ::= Boolean → Boolean | Boolean ↔ Boolean

HDL_Expression ::= HDL_EXPR

Flavor Macro HDL_EXPR =  Veilog: Verilog_Expression
                       | VHDL: VHDL_Expression
                       | . . .
```

Figure 2. PSL Profile for HDLs

events are essentially formulated as boolean propositions on variables in hardware designs. The types of variables differ for different hardware design languages, e.g., the basic types in Verilog include boolean, bit, bit vectors, integer, etc. The operators pertain to these types of variables are also imported from the hardware design languages.

We extend the boolean layer of PSL through defining the profiles for software design/implementation languages. Due to the higher abstraction levels of the software design/implementation languages, software events are formulated not only on the variables, but also on the control points and the communication mechanisms such as messages or function calls. The general approach to defining an xPSL profile for a software design/implementation language is to identify the variable types, the control point types, and the communication mechanisms and provide language constructs for defining propositional boolean expressions over the above constructs. As examples, we introduce the xPSL profiles for xUML and for the TinyOS C subset below.

xPSL Profile for xUML. In the xPSL profile for xUML, primitive boolean expressions include message expressions and control point expressions in addition to expressions over variables, as shown in Figure 3. The message expres-

```
Primitive_xUML_Expression ::=  Variable_Expression
                             | Message_Expression
                             | ControlPoint_Expression

Message_Expression ::= Message_Name [Predicate List over Parameters]

ControlPoint_Expression ::=  Object_Instance.Status = State_Name
                           | Object_Instance.Status = ControlPoint_Label
```

Figure 3. PSL Profile for xUML

sions are declared over occurrences of message instances. Since a message instance can have parameters, a message expression can be declared over these parameters by including a list of predicates over these parameters and this expression is true only when a message instance satisfying all

the predicates is generated. The control point expressions are declared over the control points in an xUML design. The basic semantic units in xUML are object instances. In the state model of an object instance, a control point can refer to a state or a point in a state action. A state action is a piece of sequential program. Figure 4 shows a segment of

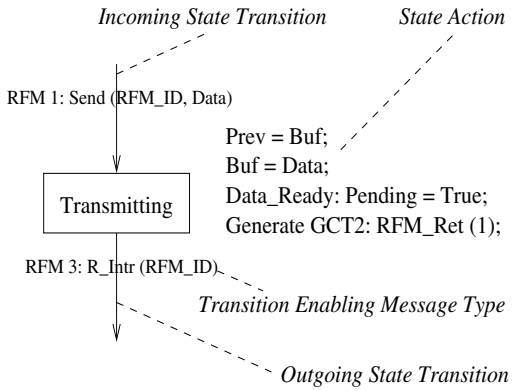

Figure 4. Control Points in xUML

a state model, which contains a single state, *Transmitting*. The state model moves into this state upon receiving the *RFM1: send* message and the state action stores the data to be sent and sets a pending flag. A control point in the state action is tagged by a label if this point is of interest in property specification. In this case, *Data_Ready* is a control point label. Object instances can be composed into primitive components which can be further composed into composite components. The control points in an object instance are indexed following the object/component hierarchy.

A sample software property specified using this profile is shown in Figure 5. The boolean expression is declared over

```
assert never ((S-NET.RFM.Prev = 1)
              and (S-NET.RFM.Buf = 1)
              and (S-NET.RFM.Status = Transmitting));
```

Figure 5. No Consecutive 1's Property

the *RFM* object instance in the *S-NET* component. *Prev* and *Buf* are two variables in *RFM*, storing the previous and current transmission sequence numbers. *(S-NET.RFM.Status = Transmitting)* indicates that *RFM* is in its *Transmitting* state. This is a safety property, which asserts that when *RFM* is transmitting, its previous and current sequence numbers should not both be 1. The property can be made more precise by changing (S-NET.RFM.Status = Transmitting) to (S-NET.RFM.Status = Data_Ready) which refers to the control point label defined above in the state action in Figure 4.

xPSL Profile for TinyOS C Subset. In the TinyOS C subset, the basic semantic unit is a component. The signal handlers, command handlers, and tasks in a component are all implemented as functions. The components can be composed into composite components hierarchically. In the xPSL profile for the TinyOS C subset, primitive boolean expressions include function expressions and control point expressions in addition to expressions over variables, as shown in Figure 6. Function expressions have two subtypes:

```
Primitive_TOS_C_Expression ::=  Variable_Expression
                              | Function_Expression
                              | ControlPoint_Expression

Function_Expression ::= Function_Name.entry [Predicate List over Parameters]
                        Function_Name.exit [Predicate over Return Value]

ControlPoint_Expression ::=  Function_Name.Status = ControlPoint_Name
```

Figure 6. PSL Profile for TinyOS C Subset

expressions declared over function calls and expressions declared over function returns. These expressions can be declared over parameters of function calls and return values of function returns. Control point expressions are declared over the control points in signal handlers, command handlers, or tasks. Such a point is tagged by a label and indexed following the component hierarchy.

3.2 Restrictions to PSL Temporal Layer

We design the temporal layer of xPSL to be the linear-time subset of the temporal layer of PSL (which covers LTL and the linear-time subset of CTL[1]) for the reasons below: (1) Linear-time properties are more intuitive to specify and more amenable to compositional reasoning than branching-time properties [15]; (2) Properties expressible in ω-automata [8], the common formal semantic basis for co-verification that we select (See Section 4), is linear-time.

For software semantics in which operations are not time-constrained, e.g., the semantics of TinyOS and xUML, we further restrict the temporal layer of PSL by prohibiting definition of clocked expressions and sequential expressions over software events, since these expressions are defined related to a system clock. For software semantics in which operations are time-constrained, if these time constraints can be related to the system clock, the above restriction can be relaxed: clocked or time-constrained assertions over software events can be defined. We are currently developing an xPSL profile for a timed semantics of xUML in which delays in term of discrete time intervals can be associated with the state transitions and state actions in the state models.

4 Semantics Unification via Translation

Each xPSL profile provides the syntax for specifying hardware (or software, respectively) events and their temporal correlations for hardware (or software) in a single design/implementation language. Its semantics relies on the

[1] xPSL assertions that belong to the linear-time subset of CTL are essentially assertions that are expressible in both LTL and CTL.

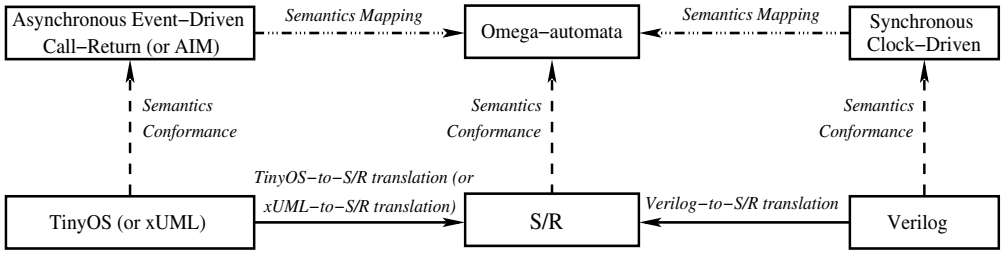

Figure 7. Model Translations Realize Semantic Mappings for Co-Verification

execution model of the design/implementation language, although the same set of temporal operators are used across all profiles. In hardware/software co-design, both hardware and software design/implementation languages are used. To co-verify hardware and software, multiple xPSL profiles need to be used to capture hardware and software events and their temporal correlations. However, these events are specified using different PSL profiles with different semantics. Therefore, to reason about properties specified in xPSL, a common formal semantic basis is needed, upon which hardware and software events can be formally defined and related to one another and temporal operators can be given a single semantics that is applicable in all hardware and software xPSL profiles. This enables meaningful specification and co-verification of system-level properties.

We select the ω-automaton semantics as the common semantic basis since it has been successfully applied in hardware verification [9], software verification [16], and co-verification [17], and has the nice feature of representing systems and properties uniformly as ω-automata [4]. However, our approach does not depend on a particular formal semantics and other formal semantics may also be used.

4.1 Translation of Hardware and Software semantics

Leveraging the formal semantic basis to formalize and correlate hardware and software events requires translations of hardware and software languages to the formal language. The translations formalize the hardware and software semantics by simulating them with the formal semantics.

As shown in Figure 7, the Verilog-to-S/R translation and the TinyOS-to-S/R (or xUML-to-S/R, respectively) translation realize the semantic mappings from the synchronous clock-driven semantics and the asynchronous event-driven call-return semantics (or the AIM semantics) to the ω-automaton semantics and, therefore, enable co-verification and compositional reasoning of systems with hardware in Verilog and software in the TinyOS C subset (or in xUML). The Verilog-to-S/R translation has been implemented in FormalCheck [9]. The xUML-to-S/R translation has been implemented in ObjectCheck [16]. The translation from TinyOS instances into S/R has been developed in [19]. The Verilog-to-S/R translation is conceptually simple since both Verilog and S/R have a synchronous semantics. A key in this translation is simulation of the Verilog clock cycle with the S/R clock cycle. Since a flip-flop in Verilog can be updated at either the positive edge or the negative edge of the system clock, a Verilog clock cycle is mapped to two consecutive S/R clock cycles. A challenge to the xUML-to-S/R translation and the TinyOS-to-S/R translation is simulation of the asynchronous semantics of xUML and TinyOS with the synchronous semantics of S/R. Our solution to this challenge is to introduce an automaton in the resulting S/R model of an xUML system (or a TinyOS instance), which interacts with the automata simulating object instances (or components) and schedules their executions.

4.2 Translation of xPSL Properties to ω-automaton

Translation of xPSL properties to ω-automata involves translation of (1) event monitors defined using the boolean layer of xPSL and (2) temporal assertions defined using the temporal layer of xPSL. Translation of event monitors utilizes translations from hardware and software to ω-automata to relate automata translated from the properties to automata translated from the hardware and software.

The temporal operators of xPSL are essentially the temporal operators of LTL and CTL. All assertions that use LTL operators are translatable since ω-automata subsume LTL in expressiveness. We support translation of the assertions which belong to the linear-time subset of CTL indirectly, in that we require these assertions be rewritten in their equivalent LTL forms before being translated.

There has been a lot of research on translating LTL formulae to automata and such translation incurs exponential blow-up in the worst case [3]. Our xPSL to ω-automata translation takes a pattern-based approach. We identify a set of commonly used property patterns written in xPSL and their corresponding patterns in ω-automata. In translation, we apply pattern-matching to replace an xPSL pattern with its corresponding ω-automaton pattern. With this approach, we essentially restrict to a simple subset of xPSL to avoid exponential blow-ups in property translation. To further simplify our implementation, we reuse property patterns in ω-automata already been developed, namely, the patterns defined in the FormalCheck Query Language [9].

5 Application of xPSL in Co-Verification

We have applied xPSL in HW/SW co-verification of embedded systems following two different but related approaches: translation-based co-verification (TBCV) [17] and component-based co-verification (CBCV) [18]. TBCV is the foundation of co-verification and CBCV optimizes co-verification through exploiting component-based architectures. In this section, we re-examine the case studies presented in [17] and [18] while focusing on how xPSL facilitates TBCV and CBCV.

5.1 xPSL in Translation-Based Co-Verification

In TBCV, hardware and software modules of an embedded system are automatically translated into the input formal language of a state-of-the-art model checker. We interface the formal models of hardware and software modules by inserting a *bridge module* that bridges the gap between the hardware and software semantics. The bridge module interacts with the hardware and software modules following the hardware and software semantics, respectively. It propagates events across the hardware/software interface, for instance, generating software messages or invoking procedures upon hardware interrupts and producing hardware signals upon value changes in certain software variables. The bridge module is specified in a domain-specific bridge design language and translated into the formal language. This approach has been realized for embedded systems with hardware in Verilog and software in xUML based on the semantics translations discussed in Section 4.

xPSL facilities TBCV in that xPSL enables specification of system-level properties across the hardware and software boundary. It provides profiles for various hardware and software design/implementation languages while semantically relating the profiles upon a common formal semantic basis.

Figure 8 shows a sensor system composed from a set of hardware modules: clock, sensor and network (denoted

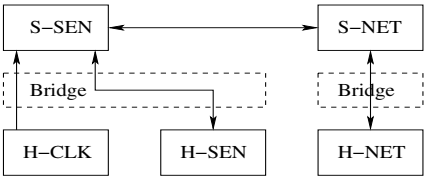

Figure 8. An Example Sensor System

by *H-CLK*, *H-SEN*, and *H-NET*), software modules: sensor and network (denoted by *S-SEN* and *S-NET*), and bridge modules connecting the hardware and software modules. *H-CLK* periodically interrupts *S-SEN*. Upon a clock interrupt, *S-SEN* starts *H-SEN*. When *H-SEN* finishes sensing, it interrupts *S-SEN* to pass sensor readings to *S-SEN*. *S-SEN* sends sensor readings to *S-NET*. If *H-NET* is free, *S-NET* delivers a data packet to *H-NET*. After the packet is transmitted, *H-NET* interrupts *S-NET* to report the transmission.

A system level property P_1 in xPSL, which is to be verified on the sensor system, is shown in Figure 9. This prop-

```
assert (always (S-NET.RFM_Pending=false)) ||
       ((S-NET.RFM_Pending=false) Until (H-SEN.intr_s));
```

Figure 9. System Property 1

erty asserts that there will not be data ready to send, which is indicated by *S-NET* keeping its *RFM_Pending* variable as false, unless the sensor has produced a piece of data, which is indicated by *H-SEN* setting its *intr_s* signal. This property involves a hardware event and a software event. We check this property with COSPAN on the S/R model of the system generated using TBCV. The property is successfully verified, using 12247.5 seconds and 1175.45 megabytes.

5.2 xPSL in Component-Based Co-Verification

A stumbling block to scalability of TBCV is the intrinsic complexity of model checking. The state space of a whole embedded systems with hardware and software can be extremely large and make TBCV intractable. CBCV is developed to address this problem through exploiting component-based architectures of embedded systems. In CBCV, embedded systems follow the component model shown in Figure 10, which unifies hardware and software component models. In this model, an embedded system

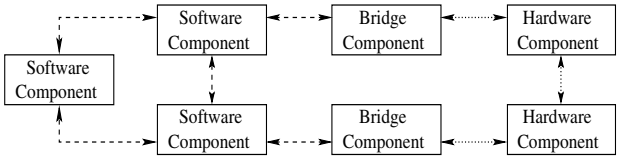

Figure 10. Unified Component Model

is composed of a set of components. There are three types of primitive components: *software, hardware, and bridge components*. Bridge components bridge the hardware/software semantic gap by propagating events across the hardware/software boundary. Composite components are composed from primitive components hierarchically.

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. Although hardware, software, and bridge components differ in their representations of E and I, their properties are all specified in xPSL. In this study, hardware components are designed in Verilog, software components are designed in xUML, and bridge components are designed in a domain-specific bridge design language [17].

A major challenge in component-based co-verification is how to obtain component properties. This challenge can be largely addressed by integrating component-based co-verification with assertion-based verification which requires designers to specify properties of their designs and are increasingly adopted by the industry. xPSL enables this integration since it is backward compatible with PSL, thus readily supporting assertion-based verification.

We employ xPSL to specify temporal assertions serving as component properties and assumptions. Each entry of P is a pair $(p, A(p))$ where p is a temporal assertion in xPSL and $A(p)$ is a set of assertions in xPSL that serve as the assumptions on the environment of C for enabling the verification of p on C. xPSL provides a unified way to specify properties of primitive hardware, software, and bridge components and properties of systems and composite components: using the same set of temporal operators while allowing monitoring of both hardware and software events.

Under the component-based approach to co-verification, 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. The decompositions reuse components as possible. Verified properties of the reused components are reused in constructing the abstractions for verifying properties of the system or higher-level components.

Given a property $(p, A(p))$ to be verified on a component C that is composed from C_0, ..., C_{n-1}, an abstraction of C for verifying $(p, A(p))$ is constructed as follows:

1. Construct a system of non-deterministic ω-automata ω_0, ..., ω_{n-1} each of which corresponds to C_i, $0 \leq i < n$ and simulates the interface of C_i. The ω-automata are composed through the interfaces that they simulate, as how C_0, ..., C_{n-1} are composed. A non-deterministic ω-automaton ω_e that simulates the environment of C is also added to close the system.

2. Constrain ω_i, $0 \leq i < n$, by composing ω_i with the ω-automata translated from the properties of C_i that are related to $(p, A(p))$ by cone-of-influence [3, 8] analysis and are *enabled*. A property of C_i is enabled iff its assumptions are implied by the enabled properties of other sub-components and/or the assumptions in $A(p)$. And constrain ω_e by composing ω_e with the ω-automata translated from the assumptions in $A(p)$.

This abstraction construction is enabled by unified property specification using xPSL and translation-based unification of various xPSL profiles. There may exist circular dependencies among the sub-component properties. The abstraction cannot include the sub-component properties unless circular reasoning can be prevented. Circular reasoning can be avoided using the following methods (but not limited to them): (1) avoid using an assumption that creates a dependency cycle; (2) use temporal induction proposed by McMillan [12]; or (3) use the compositional reasoning rule proposed by Amla, et al. [1] or the rule proposed by Xie, et al. [19]. These rules are defined in the ω-automaton semantics and reused in hardware/software co-verification based on semantics unification via translation.

The component-based approach has been applied to co-verification of networked sensor systems. Figure 11 shows a system-level property P_2 in xPSL to be verified on the sensor system in Figure 8 using this approach. P_2 asserts

```
assert (always (eventually! (H-CLK.intr_c))) →
            (always (eventually! (H-NET.flag)));
assert (always (eventually! (H-CLK.intr_c))) →
            (always (eventually! (not H-NET.flag)));
```

Figure 11. System Property 2

that the sensor system transmits on the network repeatedly if it receives clock interrupts repeatedly. Repeated setting and clearing of a flag in *H_NET* indicates repeated transmission. To verify P_2, we construct an abstraction of the sensor system using the algorithm above.

The abstraction includes the properties of the hardware, software, and bridge components that are related to P_2 and enabled. *S-SEN* and *S-NET* satisfy the handshake-related assumptions of each other. The other assumptions of *S-SEN* and *S-NET* are satisfied by the properties of the hardware components via the event mappings of the bridge components. Figure 12 shows the properties of the hardware, soft-

```
Properties of Bridge between H-CLK and S-SEN
    (always (eventually! (H-CLK.intr_c))) →
            (always (eventually! (S-SEN.C_Intr)));
Properties of S-SEN
    (always (eventually! (S-SEN.C_Intr))) →
            (always (eventually! (S-SEN.Output)));
Properties of S-NET
    (always (eventually! (S-NET.Data))) →
            (always (eventually! (S-NET.RFM.Pending)));
    (always (eventually! (S-NET.Data))) →
            (always (eventually! (not S-NET.RFM.Pending)));
Properties of Bridge between S-NET and H-NET
    (always (eventually! (S-NET.RFM.Pending))) →
            (always (eventually! (H-NET.d_rdy)));
    (always (eventually! (not S-NET.RFM.Pending))) →
            (always (eventually! (not H-NET.d_rdy)));
Properties of H-NET
    (always (eventually! (H-NET.d_rdy))) →
            (always (eventually! (H-NET.flag)));
Property of Bridge
    (always (eventually! (not H-NET.d_rdy))) →
            (always (eventually! (not H-NET.flag)));
```

Figure 12. Comp. Properties Implying P_2

ware, and bridge components that imply P_2. (Note that *S-*

SEN.Output is mapped to *S-NET.Data*.) The implication is established by model checking P_2 on the abstraction, which takes 0.1 seconds and 3.40 megabytes. Therefore, P_2 holds.

The abstraction is conservative. If the property holds on the abstraction, it also holds on the system; otherwise, the abstraction can be refined by verifying additional component properties and including them in the abstraction. If the property does not hold on the system, error trace analysis and abstraction refinement are likely to uncover the cause. The abstraction refinement process for verifying the property in Figure 5 detects a bug in *S-SEN*: *S-SEN* may output a new sensor reading to *S-NET* although *S-NET* has not acknowledged the transmission of the last sensor reading.

It can be observed that specifying properties of hardware or software components only involves one xPSL profile due to the separation of bridge components while specifying properties of the bridge components involves multiple profiles of xPSL since they interact with both hardware and software components. This makes it easy for a designer to determine when multiple xPSL profiles are needed: for system-level properties or properties of bridge components.

The time and memory usages for model checking the component properties are shown in Table 1. However,

Components	Time (Seconds)	Memory (MBytes)
S-SEN	18.66	8.49
S-NET	18.06	9.11
BRDG	86.05	15.83
H-CLK	0.21	3.38
H-SEN	0.22	3.38
H-NET	0.22	3.38

Table 1. Time and Memory Usages for Model Checking the Component Properties

if P_2 is verified with TBCV, 50800 seconds and 730.54 megabytes are needed. It can be observed that CBCV leads to order-of-magnitude reduction on the verification time and memory usages. This reduction is essentially rooted in compositional reasoning. However, leveraging such reduction is enabled and facilitated by assertion-based verification using xPSL over both hardware and software components.

6 Related Work

Traditionally in model checking, properties are specified in formal logics, such as LTL [14] for SPIN [6], CTL [2] for SMV [11], and ω-automata for COSPAN [4]. The system events, i.e., boolean propositional expressions, which are used in these properties are specified in the formal modeling language of the model checkers, for instance, Promela [6], SMV [11], and S/R [4]. Lack of standard and easy-to-use property specification languages has been one of the major impediments to widespread adoption of formal verification techniques such as model checking. PSL [7] addresses this problem in the context of hardware verification.

7 Conclusions

In this paper, we have presented xPSL, a unified property specification for hardware, software, and entire embedded systems, which extends the IEEE PSL into hardware/software co-verification. The keys to this extension include language supports for monitoring both hardware and software events and translation-based unification of hardware and software semantics. Case studies have shown that xPSL is very effective in enabling co-verification of system-level properties and facilitating compositional reasoning.

References

[1] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In *Proc. of TACAS*, 2001.

[2] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of Logic of Programs Workshop*, 1981.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

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

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, 2000.

[6] G. J. Holzmann. The model checker SPIN. *TSE 23(5)*, 1997.

[7] IEEE. *IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2005)*. IEEE, 2005.

[8] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[9] R. P. Kurshan. *FormalCheck User Manual*. Cadence, 1998.

[10] D. Maliniak. Assertion-based verification smooths the road to IP reuse. *Electronic Design*, September 2002.

[11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[12] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Cadence Design Systems Technical Reports*, 1999.

[13] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley, 2002.

[14] A. Pnueli. The temporal logic of programs. In *Proc. of IEEE Symposium on Foundations of Computer Science*, 1977.

[15] M. Y. Vardi. Branching vs. linear time: Final showdown. In *Proc. of TACAS*, 2001.

[16] F. Xie, V. Levin, and J. C. Browne. Objectcheck: A model checking tool for executable object-oriented software system designs. In *Proc. of FASE*, 2002.

[17] F. Xie, X. Song, H. Chung, and R. Nandi. Translation-based co-verification. In *Proc. of MEMOCODE*, 2005.

[18] F. Xie, G. Yang, and X. Song. Component-based hardware/software co-verification. In *Proc. of MEMOCODE*, 2006.

[19] F. Xie, G. Yang, and X. Song. Compositional reasoning for hardware/software co-verification. In *Proc. of ATVA*, 2006.