Compositional Reasoning for Hardware/Software Co-Verification *

Fei Xie[1], Guowu Yang[1], and Xiaoyu Song[2]

[1] Dept. of Computer Science, Portland State Univ., Portland, OR 97207
{xie, guowu}@cs.pdx.edu
[2] Dept. of Electrical & Computer Engineering, Portland State Univ., Portland, OR 97207
song@ece.pdx.edu

Abstract. In this paper, we present and illustrate an approach to compositional reasoning for hardware/software co-verification of embedded systems. The major challenges in compositional reasoning for co-verification include: (1) the hardware/software semantic gaps, (2) lack of common property specification languages for hardware and software, and (3) lack of compositional reasoning rules that are applicable across the hardware/software boundaries. Our approach addresses these challenges by (1) filling the hardware/software semantic gaps via translation of hardware and software into a common formal language, (2) defining a unified property specification language for hardware, software, and entire systems, and (3) enabling application of existing compositional reasoning rules across the hardware/software boundaries based on translation, developing a new rule for compositional reasoning with components that share sub-components, and extending the applicability of these rules via dependency refinement. Our approach has been applied to co-verification of networked sensors. The case studies have shown that our approach is very effective in enabling application of compositional reasoning to co-verification of non-trivial embedded systems.

1 Introduction

Embedded systems are pervasive in the infrastructure of our society. They are often mission-critical, therefore, must be highly trustworthy. Embedded systems often support concurrency intensive operations such as simultaneous monitoring, computation, and communication. Thus, to build trustworthy embedded systems, they must be extensively verified. Due to strict design constraints of embedded systems, to achieve better performance, hardware and software components must closely interact and the trade-off between hardware and software must be exploited. This demands hardware/software co-design and, therefore, hardware/software co-verification of embedded systems.

Model checking [1, 2] is a powerful formal verification method which has great potential in hardware/software co-verification of embedded systems. It provides exhaustive state space coverages for the systems being verified. However, a stumbling block to scalable application of model checking to co-verification is the intrinsic complexity of model checking. The number of possible states and execution paths in a real-world

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

system can be extremely large, which makes naive application of model checking intractable and requires state space reduction. Compositional reasoning [3–9], as applied in model checking, is a powerful state space reduction algorithm. Using compositional reasoning, model checking of a property on a system is accomplished by decomposing the system into components, model checking the component properties locally on the components, and deriving the system property from the component properties.

Co-verification of an embedded system involves both its hardware and software components, which leads to the following major challenges to compositional reasoning:

1. *Hardware/software semantic gaps.* Hardware usually follows synchronous clock-driven semantics while software semantics are more diversified, e.g., asynchronous interleaving message-passing semantics and event-driven call-return semantics.
2. *Lack of unified property specification languages.* Effective compositional reasoning can benefit greatly from uniform specification of properties of both hardware and software components and, furthermore, properties of entire embedded systems.
3. *Lack of appropriate rules for co-verification.* Existing compositional reasoning rules do not readily address the special needs of co-verification: compositional reasoning involving components of different semantics and components that share sub-components, e.g., an execution scheduler shared by software components.

In this paper, we present and illustrate an approach to compositional reasoning for hardware/software co-verification of embedded systems. This approach addresses the above challenges as follows:

1. The hardware/software semantic gaps are filled via translation of both hardware and software components into a formal language whose semantics serves as the common semantic basis for co-verification and compositional reasoning.
2. A unified property specification language is defined, which supports property specification for hardware components, software components, and furthermore entire embedded systems. This unification of property specification facilitates compositional reasoning across the hardware/software semantic boundaries.
3. A new compositional reasoning rule supports compositional reasoning for components that share sub-components. The new rule and the existing rules are applied across the hardware/software boundaries based on translation. The applicability of these rules is further extended through dependency refinement.

Our approach has been applied to co-verification of networked sensors, an emerging type of embedded systems. Hardware components of sensors are specified in Verilog while software components are specified in C following an asynchronous event-driven call-run semantics of TinyOS [10] or in xUML [11], an executable dialect of UML, following the asynchronous interleaving message-passing semantics. The case studies have shown that our approach enables compositional reasoning of non-trivial embedded systems and achieves order-of-magnitude reduction on verification complexities.

Related Work. There has been much research on compositional reasoning [9]. Particularly relevant is assume-guarantee compositional reasoning, which was introduced by Chandy and Misra [3] and Jones [4] for analyzing safety properties. Abadi and Lamport [5], Alur and Henzinger [6], and McMillan [7] extended assume-guarantee compositional reasoning to liveness properties. However, these extensions are incomplete,

i.e., there exist properties of systems which are true but not provable under these extensions [12]. Amla, Emerson, Namjoshi, and Trefler [8] proposed a sound and complete compositional reasoning rule for both safety and liveness properties. Our approach builds on the previous work on compositional reasoning and enables application of compositional reasoning across the hardware/software boundaries. This is based on translating hardware and software into the same formal model-checkable language.

The rest of this paper is organized as follows. In Section 2, we provide the background of this work. We discuss how translation fills the hardware/software semantic gaps in Section 3. In Section 4, we define a unified property specification language. We present compositional reasoning for co-verification in Section 5. In Section 6, we illustrate our approach with case studies on networked sensors. We conclude in Section 7.

2 Background

2.1 A Formal Semantics: ω-Automaton Semantics

We adopt the L-process model of ω-automaton semantics. Details of this model can be found in [13]. Only the concepts essential for understanding this paper are given below.

Definition 1. *For an L-process, ω, its language, $\mathcal{L}(\omega)$, is the set of all infinite sequences accepted by ω.*

Definition 2. *For an L-process, ω, $\mathcal{L}_*(\omega)$ denotes the set of all finite prefixes of $\mathcal{L}(\omega)$.*

Definition 3. *For L-processes, $\omega_1, \ldots, \omega_n$, their synchronous parallel composition, $\omega = \omega_1 \otimes \ldots \otimes \omega_n$, is also an L-process and $\mathcal{L}(\omega) = \cap\mathcal{L}(\omega_i)$.*

Definition 4. *For L-processes, $\omega_1, \ldots, \omega_n$, their Cartesian sum, $\omega = \omega_1 \oplus \ldots \oplus \omega_n$, is also an L-process and $\mathcal{L}(\omega) = \cup\mathcal{L}(\omega_i)$.*

For a language $\mathcal{L}$ of infinite sequences over a set of variables, V, the safety closure [14] of $\mathcal{L}$, denoted by $cl(\mathcal{L})$, is defined as the set of infinite sequences over V where $x \in cl(\mathcal{L})$ iff for each finite prefix y of x, there exists an infinite sequence z, $y : z \in \mathcal{L}$. ($y : z$ denotes the concatenation of y and z where y and z are sequences over V.) In [13], $cl(\mathcal{L})$ is termed as the smallest limit prefix-closed language containing $\mathcal{L}$.

Definition 5. *The safety closure $CL(\omega)$ of an L-process ω is an L-process whose language is the safety closure of the language of ω, $\mathcal{L}(CL(\omega)) = cl(\mathcal{L}(\omega))$.*

$CL(\omega)$ can be derived from ω by changing the fairness condition of ω to true.

Definition 6. *For a set S of finite sequences over a set of variables V, the limit of S, denoted by $lim(S)$, is the set of infinite sequences whose finite prefixes are all in S.*

Notations. Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$, $\mathcal{L}_1 \Rightarrow \mathcal{L}_2$ denotes $\mathcal{L}_1 \subseteq \mathcal{L}_2$, and $\mathcal{L}_1 \equiv \mathcal{L}_2$ denotes $\mathcal{L}_1 \subseteq \mathcal{L}_2$ and $\mathcal{L}_2 \subseteq \mathcal{L}_1$.

Lemma 1. $cl(\mathcal{L}(\omega)) \equiv lim\ \mathcal{L}_*(\omega)$

Proof of Lemma 1: Follows from the definitions of cl and lim. □

Under the ω-automaton semantics model checking is reduced to checking L-process language containment. Suppose a system is modeled by the composition $\omega_1 \otimes \ldots \otimes \omega_n$ of L-processes, $\omega_1, \ldots, \omega_n$, and a property to be checked on the system is modeled by an L-processes, ω. The property holds on the system if and only if the language of $\omega_1 \otimes \ldots \otimes \omega_n$ is contained by the language of ω, $\mathcal{L}(\omega_1 \otimes \ldots \otimes \omega_n) \subseteq \mathcal{L}(\omega)$.

Definition 7. *Given two L-processes ω_1 and ω_2, ω_1 implements ω_2 (denoted by $\omega_1 \models \omega_2$) if $\mathcal{L}(\omega_1) \subseteq \mathcal{L}(\omega_2)$.*

2.2 S/R Language: A Realization of ω-Automaton Semantics

The S/R language is the input formal language of the COSPAN model checker [15]. In S/R, a system P is composed of synchronously interacting processes, conceptually ω-automata. A process consists of state variables, selection variables, inputs, state transition rules, and selection rules. Selection variables define the outputs of the process. Each 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 to be checked is also modeled by an ω-automaton T. COSPAN performs the verification by checking the language containment, $\mathcal{L}(P) \subseteq \mathcal{L}(T)$, using either an explicit state space enumeration algorithm or a symbolic (BDD-based or SAT-based) search algorithm.

2.3 A Hardware 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. The translation has been implemented in FormalCheck [16]. Abstractly, a Verilog model consists of a number of 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 the negative edge of the system clock. The outputs of combinational circuits are updated based on their inputs instantly if zero delay is assumed.

2.4 Two Software Semantics

Asynchronous Event-Driven Call-Return Semantics of TinyOS TinyOS [10] is an operating system for networked sensors. It is component-based and is readily extensible and configurable via developing new components and including only the necessary

components in the system configuration for a given mission. A complete TinyOS system configuration consists of a scheduler and a graph of components. A component has four interrelated parts: a set of command handlers, a set of event handlers, a fixed-size data frame, and a bundle of tasks. Command handlers, event handlers, and tasks execute in the context of the frame and operate on its state and are implemented as functions which are invoked following the call-return semantics. Higher level components issue commands to lower level components and lower level components signal events to the higher level components. The lowest level of components abstracts physical hardware.

Event handlers are invoked to deal with hardware events, either directly or indirectly. The lowest level components have handlers connected directly to hardware interrupts. An event handler can deposit information into its frame, post tasks, signal higher level events or call lower level commands. A hardware event triggers a fountain of processing that goes upward through events and can bend downward through commands. In order to avoid cycles in the command/event chain, commands cannot signal events. Commands and events are intended to perform a small, fixed amount of work.

Tasks perform the primary work. They are atomic with respect to other tasks, though they can be preempted by events. Tasks can call lower level commands, signal higher level events, and post other tasks within a component. The semantics of tasks make it possible to allocate a single stack that is assigned to the currently executing task. Tasks allow concurrency since they execute asynchronously with respect to events. However, tasks must never block or spin wait or they will prevent progress in other components. While events and commands approximate light-weight instantaneous computations, task bundles provide a way to incorporate arbitrary computations into the event-driven model. The task scheduler is FIFO, utilizing a bounded size scheduling queue.

Asynchronous Interleaving Message-Passing (AIM) Semantics of Executable UML Executable UML (xUML) [11] is an executable dialect of UML supporting model-driven development of embedded software. System models in xUML can be simulated with execution simulators and can also be automatically compiled into C/C++. xUML features an asynchronous interleaving message-passing semantics. Under this 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 communicate with each other 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.

3 Translations of Hardware and Software

For practical reasons, hardware and software components of an embedded system are often specified in various languages with different semantics, for instance, the ones given in Section 2. However, to formally verify correctness properties of the entire system, a common formal semantic basis is needed, upon which events in hardware and software components can be precisely defined and, furthermore, related to one another.

This enables meaningful specification and reasoning of system-level properties which often span across the hardware and software boundaries.

Leveraging the formal semantic basis to fill the hardware/software semantic gaps requires translations of the hardware and software languages to the formal language. The translations formalize the hardware and software semantics by simulating them with the formal semantics. (Restrictions are applied to the software semantics to ensure software components be finite-state.) The translations enable reuse of model checkers and compositional reasoning rules that have been developed for the formal semantics.

The translation from Verilog to S/R has been implemented in FormalCheck [16], which simulates the synchronous clock-driven semantics of Verilog with the selection/resolution semantics of S/R. The xUML-to-S/R translation has been implemented in ObjectCheck [17], which simulates the AIM semantics with the selection/resolution semantics of S/R. In this section, we briefly discuss the translation from TinyOS to S/R.

3.1 Translation from TinyOS to S/R

The TinyOS-to-S/R translation simulates the asynchronous event-driven call-return semantics of TinyOS with the selection/resolution semantics of S/R and is currently being implemented. Each component in a TinyOS system is mapped to multiple automata in the resulting S/R system: the fixed-size data frame is modeled by an automaton which keeps the state of the data frame and each event handler, command handler, or task is also modeled as an automaton which updates the data frame by interacting with the data frame automaton. An additional automaton, *scheduler*, is introduced in the S/R system and it determines which event handler, command handler, or task should be executed. The *scheduler* exports a selection variable, *choice*, imported by the automata corresponding to event handlers, command handlers, and tasks. At any given moment, the *scheduler* selects an automaton corresponding to an event handler, command handler, or task by setting *choice* to a particular value. Only the chosen automaton executes a state transition corresponding to the execution of a C language statement in the event handler, command handler, or task. Other automata follow a self-loop transition back to their current states.

Event handlers, command handlers, and tasks are implemented as C functions in TinyOS. The call-return semantics is simulated with the semantics of S/R as follows. The caller exports a Boolean selection variable which is set to true when the call is made. The callee imports this variable and responds to the call if the variable is set to true. Parameters of the call are passed via additional selection variables. The callee exports a selection variable which indicates the call return and is imported by the caller. The return value of the call is passed via additional selection variables of the callee.

In TinyOS, tasks are atomic with respect to other tasks, but can be preempted by events. We assume that a task can be preempted in between the execution of two consecutive C language statements. The preemption is implemented through the *scheduler* adjusting the value of the *choice* variable. In between the execution of two consecutive C language statements in a task, the *scheduler* checks for hardware interrupts. If there exists an interrupt, the *choice* is set to the automaton simulating the event handler of the interrupt. The *choice* is set back to the task when the interrupt handling is done.

4 Unified Property Specification Language

Co-verification examines both hardware and software components, and entire embedded systems. It is highly desirable to have a unified property specification language for both hardware and software components, and entire systems. We have developed such a language based on ω-automata, which extends the hardware property specification language of FormalCheck [16]. This unified language is presented in terms of a set of property templates shown in Figure 1, which have intuitive meanings and also rigorous

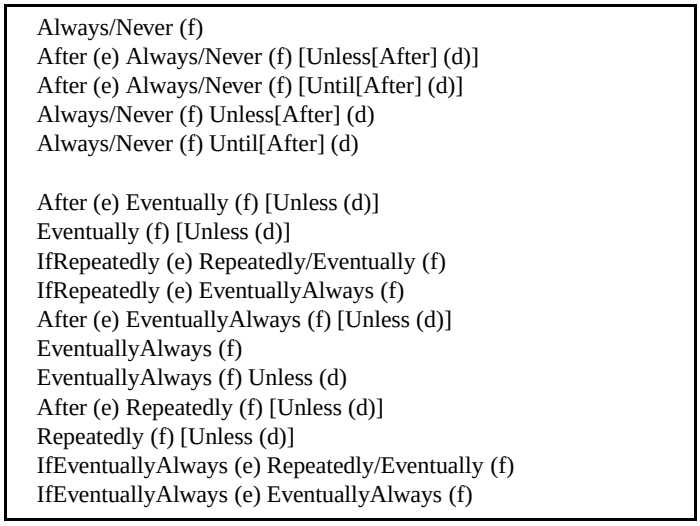

Fig. 1. A list of available property templates

mappings to property templates written in S/R. (Note that in S/R, both systems and properties are formulated as ω-automata.) An example of such templates is

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

where the *enabling* condition e and the *discharging* condition d are Boolean propositions declared over semantic entities of hardware or software. The semantic meaning is that after each occurrence of e there eventually follows an occurrence of d. Although similar to the LTL formula $G(e \rightarrow XF(d))$, our property does not require a second d in case the discharge condition d is accompanied by a second e, whereas an initial e is not discharged by an accompanying d. This asymmetry meets many requirements of software specification. (On account of this asymmetry, our property cannot be expressed in LTL.) The formal semantics of a property instantiating this template can be precisely defined based on the mappings from the hardware and software semantics to the semantics of S/R and the mapping of this template to a template written in S/R. The property can be automatically translated into S/R based on these mappings.

Our property specification language is linear-time, with the expressiveness of ω-automata [13]. The templates define parameterized automata. The language is readily extensible: new templates can be formulated as needed. A property in this language

consists of (1) declarations of Boolean propositions over software or hardware semantic entities, and (2) declarations of temporal assertions. A temporal assertion is declared through instantiating a property template: each argument of the template is realized by a Boolean expression composed from the declared Boolean propositions.

5 Compositional Reasoning for Co-Verification

5.1 Previous Work: Translation-Based Compositional Reasoning for Software

In [18], we developed translation-based compositional reasoning (TBCR), an approach to application of compositional reasoning in model checking software systems based on translation. If a translation can be shown to preserve the validity of properties (e.g., for the xUML-to-S/R translation, we established that the translation is linear-monotonic with respect to language containment), then given a software system and a property to be checked, compositional reasoning in the software semantics is conducted as follows. (1) The system is decomposed into components on the software semantics level. (2) The component properties are formulated. The components and their properties are translated into the formal semantics. A compositional reasoning rule in the formal semantics is reused. The conditions of the rule are checked. (3) If the conditions hold, then it can be concluded on the software semantics level that the system property holds.

TBCR has been realized for software specified in xUML. The xUML-to-S/R translation implements the semantic mapping from the AIM semantics to the ω-automaton semantics. Based on this translation, we have reused, for verification of xUML models, a rule [8] that has been established in the ω-automaton semantics, Rule 1.

Rule 1 *For ω-automata P_1 and P_2 modeling two components of a system, and Q modeling a property of the system, to show that $P_1 \otimes P_2 \models Q$, find ω-automata Q_1 and Q_2 modeling the component properties such that the following conditions are satisfied.*[1]

C1: $P_1 \otimes Q_2 \models Q_1$ *and* $P_2 \otimes Q_1 \models Q_2$
C2: $Q_1 \otimes Q_2 \models Q$
C3: *Either* $P_1 \otimes CL(Q) \models (Q \oplus Q_1 \oplus Q_2)$ *or* $P_2 \otimes CL(Q) \models (Q \oplus Q_1 \oplus Q_2)$

5.2 Translation-Based Compositional Reasoning for Co-Verification

The translation-based nature of TBCR enables its natural extension to support compositional reasoning for co-verification. Given an embedded system and a system property in the unified property specification language, compositional reasoning for co-verification can be conducted as follows: (1) The system is partitioned into its hardware and software components. (2) The properties of the hardware and software components are formulated. The hardware and software components and their properties are translated into a formal language with their corresponding translators. The conditions of a compositional reasoning rule in the formal semantics are checked. (3) If the conditions hold, it can be concluded that the system property holds. As shown in Figure 2,

[1] An additional condition of Rule 2 is that Q_1 (or Q_2) does not block P_2 (or P_1). A process Q does not block process P iff (i) any initial state of P can be extended to an initial state of $P \otimes Q$, and (ii) for any reachable state of $P \otimes Q$, any transition of P from that state can be extended to a transition of $P \otimes Q$. The condition holds trivially in the ω-automaton semantics.

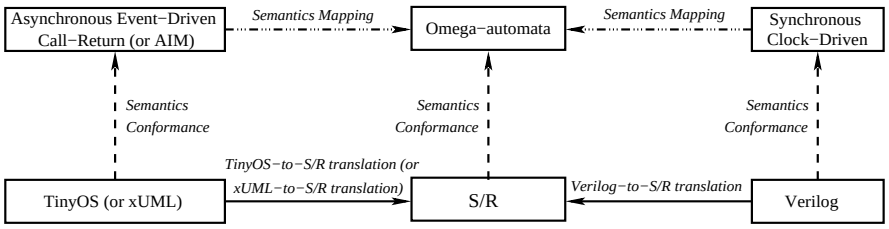

Fig. 2. Model translations realize semantic mappings for co-verification.

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, therefore, enables compositional reasoning for systems with hardware in Verilog and with software in the C subset for TinyOS (or in xUML).

This extension of TBCR requires that the hardware and software translations preserve the validity of the hardware and software properties, e.g., TinyOS-to-S/R and Verilog-to-S/R translations are linear-monotonic with respect to language containment.

5.3 Compositional Reasoning with Components That Share Sub-Components

Compositional reasoning for co-verification requires new rules that support reasoning about components that share sub-components. Simulating a software semantics with the common formal semantics often requires modeling of a scheduler in the formal semantics. The translation of a TinyOS system into S/R inserts in the resulting S/R system a scheduler that interacts with the automata simulating each software component. The translation of an xUML system into S/R inserts a scheduler that interacts with each automaton simulating an object instance. (A component in xUML may contain multiple object instances.) These schedulers make scheduling decisions based on interactions with hardware. When each software component is verified, it is often the case that the scheduler must be included in the verification since using assumptions to abstract the scheduler is often difficult. Therefore, the scheduler becomes a shared sub-component. Rule 1 does not apply here since it does not allow components to share sub-components.

We propose a new compositional reasoning rule, Rule 2, addressing this problem:

Rule 2 *For ω-automata P_1 and P_2 modeling two components of a system, S modeling a common component, and Q modeling the system property, to show that $S \otimes P_1 \otimes P_2 \models Q$, find ω-automata Q_1 and Q_2 modeling the component properties such that the following conditions are satisfied.*

C1': $S \otimes P_1 \otimes Q_2 \models Q_1$ *and* $S \otimes P_2 \otimes Q_1 \models Q_2$
C2': $S \otimes Q_1 \otimes Q_2 \models Q$
C3': *Either* $S \otimes P_1 \otimes CL(Q) \models (Q \oplus Q_1 \oplus Q_2)$ *or* $S \otimes P_2 \otimes CL(Q) \models (Q \oplus Q_1 \oplus Q_2)$

Lemma 2. $\mathcal{L}_*(S \otimes P_1 \otimes P_2) \Rightarrow \mathcal{L}_*(S \otimes Q_1 \otimes Q_2)$

Proof of Lemma 2: Follows from C1' by induction on length of finite prefixes. □

Theorem 1. *(Soundness) For ω-automata S, P_1, P_2, Q_1, Q_2, and Q satisfying the conditions of Rule 2, $S \otimes P_1 \otimes P_2 \models Q$.*

We decompose the proof of Theorem 1 into a safety proof and a liveness proof according to the decomposition of $\mathcal{L}(Q)$ into its safety part and liveness part, $\mathcal{L}(Q) \equiv cl(\mathcal{L}(Q)) \wedge (\neg cl(\mathcal{L}(Q)) \vee \mathcal{L}(Q))$. The safety proof shows that $\mathcal{L}(S \otimes P_1 \otimes P_2) \Rightarrow cl(\mathcal{L}(Q))$ while the liveness proof shows that $\mathcal{L}(S \otimes P_1 \otimes P_2) \wedge cl(\mathcal{L}(Q)) \Rightarrow \mathcal{L}(Q)$.

Proof of Safety Part of Theorem 1:

$$
\begin{aligned}
& \mathcal{L}(S \otimes P_1 \otimes P_2) \\
\Rightarrow\ & cl(\mathcal{L}(S \otimes P_1 \otimes P_2)) && \{\text{Closure is weakening}\} \\
\equiv\ & lim\ \mathcal{L}_*(S \otimes P_1 \otimes P_2) && \{\text{Lemma 1}\} \\
\Rightarrow\ & lim\ \mathcal{L}_*(S \otimes Q_1 \otimes Q_2) && \{lim \text{ is monotonic; Lemma 2}\} \\
\equiv\ & cl(\mathcal{L}(S \otimes Q_1 \otimes Q_2)) && \{\text{Lemma 1}\} \\
\Rightarrow\ & cl(\mathcal{L}(Q)) && \{\text{Closure is monotonic; Condition C2'}\}
\end{aligned}
$$

□

Proof of Liveness Part of Theorem 1:

$$
\begin{aligned}
& \mathcal{L}(S \otimes P_1 \otimes P_2) \wedge cl(\mathcal{L}(Q)) \\
\equiv\ & \mathcal{L}(S \otimes P_1 \otimes P_2) \wedge \mathcal{L}(CL(Q)) && \{\text{Closure represents language closure}\} \\
\equiv\ & \mathcal{L}(S) \wedge \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}(CL(Q)) \\
& \{\text{Composition is conjunction of languages}\} \\
\Rightarrow\ & \mathcal{L}(S) \wedge \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}(Q \oplus Q_1 \oplus Q_2) && \{\text{Condition C3'}\} \\
\equiv\ & \mathcal{L}(S) \wedge \mathcal{L}(P_1) \wedge \mathcal{L}(P_2) \wedge (\mathcal{L}(Q) \vee \mathcal{L}(Q_1) \vee \mathcal{L}(Q_2)) \\
& \{\text{Cartesian sum is disjunction of languages}\} \\
\Rightarrow\ & \mathcal{L}(Q) \vee (\mathcal{L}(S) \wedge \mathcal{L}(P_1) \wedge (\mathcal{L}(Q_2)) \vee (\mathcal{L}(S) \wedge \mathcal{L}(P_2) \wedge \mathcal{L}(Q_1)) \\
& \{\text{Distribution of } \wedge \text{ over } \vee\text{; dropping conjuncts}\} \\
\Rightarrow\ & \mathcal{L}(Q) \vee (\mathcal{L}(S) \wedge \mathcal{L}(Q_1) \wedge \mathcal{L}(Q_2)) && \{\text{Condition C1'}\} \\
\Rightarrow\ & \mathcal{L}(Q) && \{\text{Condition C2'}\}
\end{aligned}
$$

□

Theorem 2. *(Completeness) For ω-automata S, P_1, P_2, and Q, if $S \otimes P_1 \otimes P_2 \models Q$, there exist Q_1 and Q_2 that satisfy the conditions of Rule 2.*

Proof of Theorem 2: By choosing P_1 and P_2 as Q_1 and Q_2, the proof is trivial. □

5.4 Dependency Refinement

Both Rule 1 and Rule 2 share the same intuition: using Conditions C3 and C3' to prevent circular reasoning by showing that at least one component will take the first step

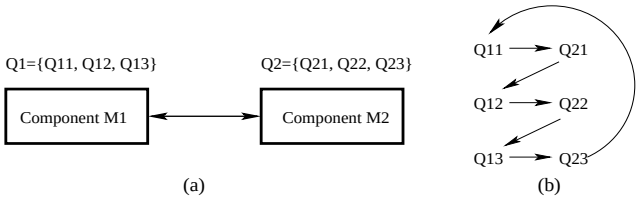

Fig. 3. A motivating example for dependency refinement

voluntarily. However, naive application of these rules will fail to establish system properties in many cases where the interaction between the two components of a system has more than two steps which form a dependency cycle. Suppose that a system has two components, M_1 and M_2, as shown in Figure 3(a). The property of M_1 (or M_2, respectively), Q_1 (or Q_2), is actually the conjunction of a set of sub-properties, Q_{11}, Q_{12}, and Q_{13} (or Q_{21}, Q_{22}, and Q_{23}), each of which asserts on a step of the interaction. The circular dependency among Q_1 and Q_2, in fact, consists of more complicated dependencies among Q_{11}, Q_{12}, Q_{13}, Q_{21}, Q_{22}, and Q_{23} as shown in Figure 3(b). If Q_1 and Q_2 are used straightforwardly in Rule 1 (or Rule 2, respectively), C3 (or C3') does not hold since the left-hand side of C3 (or C3') implies the sub-properties of Q_1 or Q_2 asserted on the first step of the interaction, but not those asserted on the other steps.

Our solution to the above problem is dependency refinement: (1) decompose the component properties into their sub-properties, derive the refined dependency graph of the sub-properties, and identify the cycles in the refined graph; (2) apply C3 or C3' to break each of the identified cycles; (3) if all cycles in the refined dependency graph can be broken, the compositional reasoning is sound and the component properties can be established. For the example in Figure 3, suppose that we can establish C3 for Q_{11}, i.e., M_1 takes the first step. We can then conclude that the component properties Q_1 and Q_2 hold since there is a single cycle. Currently, manual efforts are required to decompose the component properties into their sub-properties, refine the dependency graph, and identify the first sub-property in a dependency cycle for which the conditions C3 or C3' should be checked first. We are exploring heuristics that can automate these steps.

6 Case Studies

Our approach to compositional reasoning for co-verification has been applied to networked sensors with hardware specified in Verilog and software specified in xUML following the asynchronous interleaving message-passing semantics or in C following the asynchronous event-driven call-return semantics. In this section, we illustrate our approach with its application to a sensor instance with software in xUML. The approach is applied to sensor instances with software in C the same way. (Translation of sensor software in C to S/R currently requires manual efforts due to the unfinished translator.)

The architecture of the sensor instance with software in xUML is shown in Figure 4. Its software is partitioned into two components: software sensor (*S-SEN*) and software network (*S-NET*) and its hardware is partition into three components: hardware clock (*H-CLK*), hardware sensor (*H-SEN*), and hardware network (*H-NET*). The

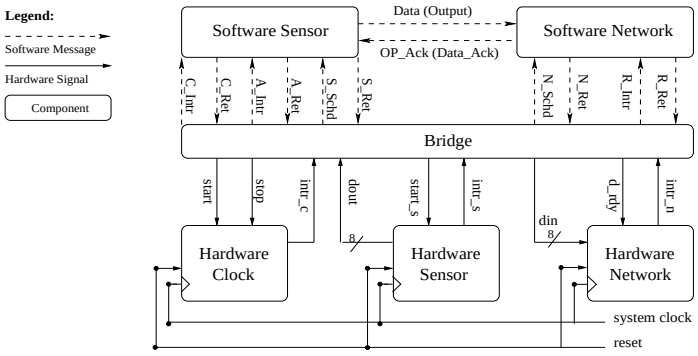

Fig. 4. Architecture of a sensor instance with software in xUML and hardware in Verilog

software components execute on a generic processor while the hardware components are implemented as application specific integrated circuits (ASICs). The software and hardware components are connected through a bridge component (*BRDG*) which interacts with the software components following the software semantics and with the hardware components following the hardware semantics and propagates events such as software messages and hardware interrupts across the hardware/software boundary.

The property shown in Figure 5 is to be verified on the entire system. This property

Repeated (H-NET.flag = true); **Repeated** (H-NET.flag = false);

Fig. 5. Repeated transmission property

asserts that the sensor system transmits on the network repeatedly. Repeated setting and clearing of a flag in *H_NET* indicates repeated transmission. The system property is manually decomposed into the properties of its components as shown in Figure 6. Note that as *S-SEN* and *S-NET* are composed, the *Output* (or *Output_Ack*, respectively) message type of *S-SEN* is mapped to the *Data* (or *Data_Ack*) message type of *S-NET*. The *ADC.Pending* variable in *S-SEN* and the *RFM.Pending* variable in *S-NET* are mapped to the *start_s* signal in *H-SEN* and the *d_rdy* signal in *H-NET* via BRDG, respectively. *STQ.Empty* (or *NTQ.Empty*, respectively) is a variable in *S-SEN* (or *S-NET*).

We verify the system property with compositional reasoning in two steps. In Step 1, we establish the property of the composite component *S&B* that is composed of *S-SEN*, *S-NET*, and *BRDG*. In this step, we apply Rule 2 since although not shown in Figure 4, *S-SEN*, *S-NET*, and *BRDG* share a scheduler that schedules the execution of the xUML object instances in each component. The scheduler is inserted when the xUML model is translated into S/R. The properties of *S-SEN*, *S-NET*, and *BRDG* can be directly verified by assuming the properties of the others hold. (*BRDG* also has assumptions on the hardware components.) Therefore, Condition C1' holds. Since we define the property of *S&B* as the conjunction of the properties of its sub-components, Condition C2' holds trivially. The dependencies among the sub-properties of the component properties are shown in Figure 7. It can be observed that on the component property level, there is a dependency cycle among the property P_{SS} of *S-SEN*, the property P_{SN} of *S-NET*,

Property of S-SEN, P_{SS}:
$P_{SS}(1)$:
IfRepeatedly (C_Intr) **Repeatedly** (Output);
$P_{SS}(2)$:
After (C_Intr) **Eventually** (C_Ret); **After** (A_Intr) **Eventually** (A_Ret); **After** (S_Schd) **Eventually** (S_Ret);
$P_{SS}(3)$:
After (Output) **Never** (Output) **UnlessAfter** (OP_Ack);
Never (Output) **UnlessAfter** (S_Schd); **After** (Output) **Never** (Output) **UnlessAfter**(S_Schd);
Never (S_Ret) **UnlessAfter** (OP_Ack); **After** (S_Ret) **Never** (S_Ret) **UnlessAfter**(OP_Ack);
Never (C_Ret) **UnlessAfter** (C_Intr); **After** (C_Ret) **Never** (C_Ret) **UnlessAfter** (C_Intr);
Never (A_Ret) **UnlessAfter** (A_Intr); **After** (A_Ret) **Never** (A_Ret) **UnlessAfter** (A_Intr);
After (ADC.Pending) **Never** (ADC.Pending) **UnlessAfter** (A_Ret);
Never (S_Ret) **UnlessAfter** (S_Schd); **After** (S_Ret) **Never** (S_Ret) **UnlessAfter** (S_Schd);
After (STQ.Empty=False) **Never** (STQ.Empty=False) **UnlessAfter**(S_Ret);

Property of S-NET, P_{SN}:
$P_{SN}(1)$:
IfRepeatedly (Data) **Repeatedly** (RFM.Pending); **IfRepeatedly** (Data) **Repeatedly** (RFM.Pending=False);
$P_{SN}(2)$:
After (Data) **Eventually**(Data_Ack); **After** (N_Schd) **Eventually** (N_Ret); **After** (R_Intr) **Eventually** (R_Ret);
$P_{SN}(3)$:
Never (Data_Ack) **UnlessAfter** (Data); **After** (Data_Ack) **Never** (Data_Ack) **UnlessAfter** (Data);
Never (N_Ret) **UnlessAfter** (N_Schd); **After** (N_Ret) **Never** (N_Ret) **UnlessAfter** (N_Schd);
After (NTQ.Empty=False) **Never**(NTQ.Empty=False) **UnlessAfter**(N_Ret);
Never (R_Ret) **UnlessAfter** (R_Intr); **After** (R_Ret) **Never** (R_Ret) **UnlessAfter** (R_Intr);
After (RFM.Pending) **Never** (RFM.Pending) **UnlessAfter** (R_Ret);

Property of BRDG, P_B:
$P_B(1)$:
IfRepeatedly (intr_c) **Repeatedly** (C_Intr);
IfRepeatedly (RFM.Pending) **Repeatedly** (d_rdy);
IfRepeatedly (RFM.Pending=False) **Repeatedly** (d_rdy=False);
$P_B(2)$:
After (ADC.Pending) **Eventually** (A_Intr); **After** (STQ.Empty=False) **Eventually** (S_Schd);
After (NTQ.Empty=False) **Eventually** (N_Schd); **After** (RFM.Pending) **Eventually** (R_Intr);
$P_B(3)$:
After (C_Intr) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UnlessAfter** (C_Ret);
Never (A_Intr) **UnlessAfter** (ADC.Pending);
After (A_Ret) **Never** (A_Intr) **UnlessAfter** (ADC.Pending);
After (A_Intr) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UnlessAfter** (A_Ret);
Never (S_Schd) **UnlessAfter** (STQ.Empty=False);
After (S_Ret) **Never** (S_Schd) **UnlessAfter** (STQ.Empty=False);
After (S_Schd) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UnlessAfter** (S_Ret);
Never (N_Schd) **UnlessAfter** (NTQ.Empty=False);
After (N_Ret) **Never** (N_Schd) **UnlessAfter** (NTQ.Empty=False);
After (N_Schd) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UnlessAfter** (N_Ret);
Never (R_Intr) **UnlessAfter** (RFM.Pending);
After (R_Ret) **Never** (R_Intr) **UnlessAfter** (RFM.Pending);
After (R_Intr) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UnlessAfter** (R_Ret);

Property of H-CLK, P_{HC}:
$P_{HC}(1)$: **Repeatedly** (intr_c);

Property of H-SEN, P_{HS}:
$P_{HS}(1)$:
After (start_s) **Eventually** (intr_s);
Never (intr_s) **UnlessAfter** (start_s); **After** (intr_s) **Never** (intr_s) **UnlessAfter** (start_s);

Property of H-NET, P_{HN}:
$P_{HN}(1)$:
IfRepeatedly (d_rdy) **Repeatedly** (flag); **IfRepeatedly** (d_rdy=False) **Repeatedly** (flag=False);
$P_{HN}(2)$:
After (d_rdy) **Eventually** (intr_n);
Never (intr_n) **UnlessAfter** (d_rdy); **After** (intr_n) **Never** (intr_n) **UnlessAfter** (d_rdy);

Fig. 6. Component properties and their sub-properties

$P_{SS}(1) \rightarrow \{P_{SN}(2), P_{SN}(3), P_B(2), P_B(3)\}$
$P_{SS}(2) \rightarrow \{P_{SN}(2), P_{SN}(3), P_B(3)\}$
$P_{SS}(3) \rightarrow \{P_{SN}(3), P_B(3)\}$

$P_{SN}(1) \rightarrow \{P_{SS}(3), P_B(2), P_B(3)\}$
$P_{SN}(2) \rightarrow \{P_{SS}(3), P_B(3)\}$
$P_{SN}(3) \rightarrow \{P_{SS}(3), P_B(3)\}$

$P_B(1) \rightarrow \{P_{SS}(2), P_{SS}(3), P_{HS}(1), P_{HN}(2)\}$
$P_B(2) \rightarrow \{P_{SS}(2), P_{SS}(3), P_{SN}(2), P_{SN}(3), P_{HS}(1), P_{HN}(2)\}$
$P_B(3) \rightarrow \{P_{SS}(3), P_{SN}(3), P_{HS}(1), P_{HN}(2)\}$

Fig. 7. Dependencies among component sub-properties

and the property P_B of *BRDG*. If P_{SS}, P_{SN}, and P_B are used straightforwardly in Rule 2, Condition C3' does not hold. However, if we conduct dependency refinement and examine the dependencies among the component sub-properties, we can successfully establish the property of *S&B* using Rule 2. $P_{SS}(3)$, $P_{SN}(3)$, and $P_B(3)$ forms a dependency cycle on which C3' holds since $P_{SS}(3)$, $P_{SN}(3)$, and $P_B(3)$ are safety properties. Thus, $P_{SS}(3)$, $P_{SN}(3)$, and $P_B(3)$ holds. Following the dependencies backward, we can show all other sub-properties hold. Therefore, the property of *S&B* holds.

In Step 2, we derive the system property by applying Rule 1 to *S&B*, *H-CLK*, *H-SEN*, and *H-NET* since these components do not share any sub-component. The properties of *H-CLK*, *H-SEN*, and *H-NET* are verified directly, thus C1 holds. Since the system property is implied by $P_{HC}(1)$, $P_B(1)$, $P_{SS}(1)$, $P_{SN}(1)$, and $P_{HN}(1)$, C2 holds. There is no need to check C3 since there are no circular dependencies among the properties of *S&B*, *H-CLK*, *H-SEN*, and *H-NET*. Therefore, the system property holds.

If the system property is verified using the straightforward translation-based co-verification approach in [19]: translating the entire system into S/R and verify it using COSPAN, 50800 seconds and 730.54 megabytes are needed. The time and memory usages for establishing C1' in Step 1 and C1 in Step 2: model checking the component properties, are shown in Table 1. In Step 1, C2' holds trivially and since the sub-

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

properties for which C3' must be checked are safety properties, C3' also holds trivially. In step 2, C2 can be established by checking the system property on the component properties using 0.1 seconds and 3.4 megabytes. It can be observed that our component-based approach to co-verification achieved order-of-magnitude reduction on verification complexities over the translation-based approach in verifying this sensor instance.

7 Conclusions and Future Work

In this paper, we have presented a novel approach to compositional reasoning for co-verification. Its key contributions include integration of compositional reasoning for hardware and software based on translation, development of a new compositional reasoning rule allowing components to share sub-components, and extending applicability of compositional reasoning rules via dependency refinement. Case studies on networked sensors have shown that our approach is very effective. Future work will be focused on automation of compositional reasoning with heuristics that explore architectural patterns of embedded systems to formulate system and component properties, decompose system properties into component properties, and facilitate dependency refinement.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. of Logic of Programs Workshop. (1981)
2. Quielle, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proc. of Symposium on Programming. (1982)
3. Chandy, K.M., Misra, J.: Proofs of networks of processes. IEEE Transaction on Software Engineering **7**(4) (1981)
4. Jones, C.B.: Development methods for computer programs including a notion of interference. PhD thesis, Oxford University (1981)
5. Abadi, M., Lamport, L.: Conjoining specifications. TOPLAS **17**(3) (1995)
6. Alur, R., Henzinger, T.: Reactive modules. FMSD **15**(1) (1999)
7. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Cadence Design Systems Technical Reports (1999)
8. Amla, N., Emerson, E.A., Namjoshi, K.S., Trefler, R.: Assume-guarantee based compositional reasoning for synchronous timing diagrams. In: Proc. of TACAS. (2001)
9. de Roever, W.P., de Boer, F., Hanneman, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods. Cambridge University Press (2001)
10. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System architecture directions for networked sensors. In: Proc. of ASPLOS. (2000)
11. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley (2002)
12. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Proc. of CAV. (2000)
13. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)
14. Alpern, B., Schneider, F.: Defining liveness. Information Processing Letters **21**(4) (1985)
15. Hardin, R.H., Har'El, Z., Kurshan., R.P.: COSPAN. In: Proc. of CAV. (1996)
16. Kurshan, R.P.: FormalCheck User Manual. Cadence (1998)
17. Xie, F., Levin, V., Browne, J.C.: Objectcheck: A model checking tool for executable object-oriented software system designs. In: Proc. of FASE. (2002)
18. Xie, F., Browne, J.C., Kurshan, R.P.: Translation-based compositional reasoning for software systems. In: Proc. of FME. (2003)
19. Xie, F., Song, X., Chung, H., Nandi, R.: Translation-based co-verification. In: Proc. of MEMOCODE. (2005)