Component-Based Abstraction and Refinement [⋆]

Juncao Li[1], Xiuli Sun[2], Fei Xie[1], and Xiaoyu Song[2]

[1] Dept. of Computer Science, Portland State University, Portland, OR 97207
[2] Dept. of ECE, Portland State University, Portland, OR 97207

Abstract. In this paper, we present a comprehensive approach to model checking component-based systems (including software, hardware, and embedded systems) through abstraction and refinement. This approach is based on assume-guarantee compositional reasoning and features two synergistic techniques: (1) an automatic algorithm to component-based abstraction and (2) a mechanized assistant for abstraction refinement. The key insight to the abstraction algorithm is that a verified property is a natural abstraction of a component. The abstraction algorithm automatically determines which component properties can be included in the abstraction for verifying a system property by determining whether the assumptions of the component properties hold in the context of the system. If the abstraction fails to establish the system property, the refinement assistant determines the causes of the failure, e.g., why a component property is not included, and provides automatic remedies or requests manual remedies. This approach has been applied in component-based hardware/software co-verification of embedded systems. Case studies have shown that this approach is very effective in abstracting component-based embedded systems and guiding abstraction refinement.

1 Introduction

A common trend in both hardware and software industries is component-based development (CBD): developing systems via assembly of components [1, 2]. (In the hardware industry, CBD is also known as IP-based development [1].) Embedded systems are also increasingly component-based and include only the necessary hardware and software components for their missions, due to their diverse applications and often stringent physical constraints. CBD introduces compositional structures and standard component interfaces into systems and promotes reuse of design and development efforts. As verification becomes increasingly important, it is also desired to reuse verification efforts.

Reuse of verification efforts is further made possible by the increasing practice of assertion-based verification (ABV) [3]. ABV was initially developed for hardware verification, however, it is gaining popularity in software verification and embedded system verification. ABV requires component developers to specify temporal correctness properties of components as they are developed. Component properties are often specified in standard property specification languages such as the IEEE Property Specification Language (PSL) [4], which facilitates reuse of component properties.

Model checking [5] is a formal verification method that has great potential in system verification. It provides exhaustive state space coverages for the systems being verified. A stumbling block to scalable application of model checking is its intrinsic

[⋆] This research was supported by Semiconductor Research Corporation Contract 1356.001 and National Science Foundation Grant 0720546.

complexity. The number of possible states and execution paths in a real-world system can be extremely large, which requires state space reduction. Compositional reasoning [6], as applied in model checking, is a powerful state space reduction algorithm and accomplishes verification of a property on a system by decomposing the system into modules, checking the module properties locally, and deriving the system property from the module properties. Compositional system structures introduced by CBD may greatly simplify application of compositional reasoning in system verification.

To leverage the collective effectiveness of CBD, ABV, model checking, and compositional reasoning in system verification, the following challenges need to be addressed:

- How to determine which component properties should be considered in deriving a system-level property? ABV tends to introduce a large number of component properties. Automation is needed in managing these properties and extracting the appropriate component properties for verifying a system-level property.
- How to determine which component properties can be used in verifying a system-level properties? Many properties have enabling assumptions. Automation is needed in determining whether a component property is *enabled*, i.e., whether its assumptions hold in the context of the system.
- How to determine the causes for a compositional reasoning failure, i.e., failure to derive a system property from component properties that have been established, and identify remedies for the problems? To address these problems, manual efforts are sometimes needed. It is desired to reduce the manual efforts when possible.

In this paper, we present a comprehensive approach to model checking component-based systems (including software, hardware, and embedded systems) through abstraction and refinement. This approach is based on assume-guarantee compositional reasoning [7–13] and features two synergistic techniques: (1) an automatic algorithm to component-based abstraction and (2) a mechanized assistant for abstraction refinement. The key insight to the abstraction algorithm is that a verified property is a natural abstraction of a component. This algorithm automatically determines which component properties should be considered in constructing the abstraction for verifying a system property by dependency analysis and which component properties can be included in the abstraction by determining whether the assumptions of these properties hold in the context of the system. If the abstraction fails to establish the system property, the refinement assistant determines the causes of the failure, e.g., why a component property is not included, and provides automatic remedies or requests manual remedies.

Our approach to component-based abstraction and refinement has been applied in hardware/software co-verification of embedded systems. Case studies have shown that this approach is very effective in abstracting component-based embedded systems and guiding abstraction refinement. In particular, this abstraction and refinement approach can be applied across the hardware/software boundaries smoothly.

The reminder of this paper is organized as follows. In Section 2, we provide the background of this work. In Section 3, we introduce the algorithm to component-based abstraction and the procedure for mechanizing abstraction refinement. In Section 4, we present application of component-based abstraction and refinement in hardware/software co-verification and evaluate its effectiveness. In Section 5, we discuss related work. In Section 6, we conclude this paper and touch on future work.

2 Background

2.1 ω-Automaton Semantics

We adopt the L-process model of ω-automaton semantics, details of which can be found in [14]. Only the concepts necessary for this paper are given below. For an L-process, ω, its language, $\mathcal{L}(\omega)$, is the set of all infinite sequences accepted by ω. For L-processes, $\omega_1, \ldots, \omega_n$, their synchronous parallel composition, $\omega = \omega_1 \otimes \ldots \otimes \omega_n$, is an L-process and $\mathcal{L}(\omega) = \cap \mathcal{L}(\omega_i)$, and their Cartesian sum, $\omega' = \omega_1 \oplus \ldots \oplus \omega_n$, is also an L-process and $\mathcal{L}(\omega) = \cup \mathcal{L}(\omega_i)$. 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, $\mathcal{L}(CL(\omega)) = cl(\mathcal{L}(\omega))$. In [14], $cl(\mathcal{L})$ is termed as the smallest limit prefix-closed language containing $\mathcal{L}$. Given L-processes ω_1 and ω_2, ω_1 implements ω_2 (denoted by $\omega_1 \models \omega_2$) if $\mathcal{L}(\omega_1) \subseteq \mathcal{L}(\omega_2)$.

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, $\mathcal{L}(\omega_1 \otimes \ldots \otimes \omega_n) \subseteq \mathcal{L}(\omega)$. A realization of the ω-automaton semantics is the S/R language [15]. S/R is the input formal language of the COSPAN model checker [15], which we utilize in this research.

2.2 Assume-Guarantee Compositional Reasoning

Assume-guarantee compositional reasoning, that each module guarantees certain properties based on properties of the other modules, was introduced by Chandy and Misra [7] and Jones [8] for analyzing safety properties. Abadi and Lamport [9], Alur and Henzinger [10], and McMillan [11] extended it to liveness properties. These extensions are incomplete, i.e., there exist properties of systems which are true but not provable using these extensions [12]. Amla, Emerson, Namjoshi, and Trefler proposed a sound and complete compositional reasoning rule for both safety and liveness properties [13]. This rule, Rule 1, has been realized in the ω-automaton semantics as shown below.

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.*

C0: $V^i(Q_1) \subseteq V^i(P_1)$ *and* Q_1 *does not block* P_2*, and vice versa for* Q_2
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* $P_2 \otimes CL(Q) \models (Q \oplus Q_1 \oplus Q_2)$

$V^i(P)$ is the set of interface variables of P. 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 $P \otimes Q$. An additional restriction on $P's$ and $Q's$, which is not shown in Rule 1, is that P_1 (or Q_1, respectively) and P_2 (or Q_2) modify disjoint sets of variables.

Note that checking Condition C3 is not needed if one of Q_1, Q_2, and Q is a safety property since its safety closure is itself. In [16], Rule 1 has also been extended to support compositional reasoning with components that have shared sub-components.

3 Component-Based Abstraction and Refinement

In this section, we first present a key observation that leads to our component-based approach to abstraction and refinement: verified properties of a component can serve as abstractions of the component if their assumptions are satisfied. Then, we introduce our automatic algorithm that constructs abstractions of a composite component (a system is a composite component) from verified properties of its sub-components. After that, we introduce a mechanized assistant to refinement of component-based abstraction.

3.1 Verified Properties as Component Abstractions

Once a property p is verified on a component C, we have $C \models p$, i.e., all behaviors that C exhibits are a subset of the behaviors allowed by p, and p is usually structurally less complex than C. Therefore, verified properties are natural abstractions of components. In the ω-automaton semantics, this is more appealing since systems, components, properties, and assumptions are modeled uniformly as ω-automata. In the rest of this paper, systems, components, properties, and assumptions are all modeled as ω-automata.

However, when verifying component properties, assumptions are often introduced due to the dependencies of a component to its environment. For instance, if p is verified on C under a set A(p) of assumptions, we have $A(p) \Rightarrow C \models p$. In this case, to utilize p as the abstraction of C, we must show that $A(p)$ can be satisfied. We introduce the concept of an *enabled* component property in a component composition as follows:

Definition 1. *Given a composition of components $C = C_1 \otimes \ldots \otimes C_n$. A property $(p, A(p))$ of $C_i,\ 1 \leq i \leq n$, is enabled in C if and only if $C_1 \otimes \ldots \otimes C_n \models A(p)$.*

Checking whether p is enabled by checking $C_1 \otimes \ldots \otimes C_n \models A(p)$ is often as expensive as, if not more expensive than, directly checking $C_1 \otimes \ldots \otimes C_n \models p$. On the other hand, it is often the case that many other properties of $C_1, \ldots, C_n$ have already been verified. Therefore, it is desirable to determine whether $(p, A(p))$ is enabled, through analyzing the verified properties of $C_1, \ldots, C_n$.

Determining whether a sub-component property is enabled is further complicated by the fact that there may exist circular dependencies among the sub-component properties. It must be shown that these circular dependencies do not cause circular reasoning, before the sub-component properties can be deemed as enabled.

Not all enabled sub-component properties are necessary in verifying a property of the composition since properties are asserted on certain aspects of a component. Therefore, we only need to determine whether the sub-component properties related to the property of the composition are enabled. To ensure all the related sub-component properties are included, a straightforward approach is to apply cone-of-influence analysis [5] based on the component interfaces and their connections. This may bring in unnecessary properties. More accurate dependency analysis are needed to exclude such properties.

Based on the above discussion, we define an abstraction of a component as follows:

Definition 2. *Given a component $C = C_1 \otimes \ldots \otimes C_n$ and a property $(p, A(p))$ to be verified on C, an abstraction for checking $(p, A(p))$ is the composition of all verified properties of $C_1, \ldots, C_n$ that are related to p by dependency analysis and can be shown to be enabled through analyzing $A(p)$ and the verified properties of $C_1, \ldots, C_n$.*

The abstraction is conservative since each enabled property of a sub-component is a conservative abstraction of the sub-component. Composition of conservative abstractions is a conservative abstraction due to the language intersection property of ω-automata. Therefore, if $(p, A(p))$ holds on the abstraction, it also holds on C.

According to [17], properties can be categorized as safety properties, liveness properties, and their hybrids. (The same categorization is also applicable to assumptions.) Any property p can be represented as the intersection of a safety property and a liveness property. In the ω-automaton semantics, this is represented as $p \equiv CL(p) \wedge (\neg CL(p) \vee p)$ where $CL(p)$ is the safety closure [17] of p. In this study, we specify properties and assumptions using the ω-automata assertion templates in [16]. Based on these templates, it is easy to identify a safety or liveness assertion and decompose a hybrid assertion into its safety and liveness parts. Being able to identify safety and liveness assertions enables us to better determine whether circular dependencies among component properties can cause circular reasoning. We assume that the components and properties involved in component-based abstraction and refinement meet the restrictions imposed by Rule 1. Therefore, cycles with safety properties will not cause circular reasoning. We only need to consider pure liveness property cycles for possible circular reasoning.

3.2 Automatic Component-Based Abstraction

There are two major tasks for the component-based abstraction algorithm: (1) deciding which sub-component properties should be considered in constructing the abstraction for a composite component and (2) determining whether these sub-component properties are enabled. The efficiency and effectiveness of this algorithm lies in whether unnecessary properties can be excluded from the abstraction and necessary properties can be shown to be enabled as possible only by analyzing the sub-component properties.

Suppose that C is a composite component with sub-components $C_1, \ldots, C_n$. The interface I_i of C_i is a pair (V_i^I, V_i^O) where V_i^I is the set of variables C_i imports and V_i^O is the set of variables C_i exports. We assume all V_i^O's are disjoint. When C_i is composed with other components, the input variables in V_i^I are mapped to the output variables in V_i^O's of other components. P_i is a set of properties of C_i that are defined on I_i and has been verified. Each property in P_i is of the form $(p_{ij}, A(p_{ij}))$, $1 \leq j \leq m_i$ and m_i is the number of properties in P_i. Our abstraction algorithm constructs the abstraction for verifying $(p, A(p))$ on C from the sub-component properties, where $(p, A(p))$ is defined on the interface I of C and I is also a pair (V^I, V^O). To simplify the presentation of our algorithm, let $P = P_1 \cup \ldots \cup P_n \cup \{(true, p), (A(p), \emptyset)\}$, i.e., P is the set of all sub-component properties with addition of $(true, p)$ and $(A(p), \emptyset)$ which are derived from $(p, A(p))$. $(A(p), \emptyset)$ is marked as enabled since $A(p)$ are assumptions on the environment of C. Our automatic abstraction algorithm is shown in Figure 1:

Step 1: Build the property dependency graph G. To determine which sub-component properties should be considered in abstraction construction, we first construct a dependency graph based on the potential enabling relations among the sub-component properties. We initiate the graph with a single node $(true, p)$ and expand the graph from it. For each node $(p_g, A(p_g))$ in the graph, which has not been expanded, we first find all the sub-component properties $(p_h, A(p_h))$ based on the direct variable dependencies between $A(p_g)$ and p_h and then find all the sub-component properties $(p_k, A(p_k))$

```
Inputs: $P = \{(true, \{p\}), (p_1, A(p_1)), \ldots, (p_s, A(p_s)), (A(p), \emptyset)\}$
        where $s$ is the sum of the numbers of properties in $P_1, \ldots, P_n$
Outputs: "$p$ holds" or "refinement needed"

Build the property dependency graph $G$ from $P$;

foreach node $t \in G$ do  /* via DFS or BFS */
  Find all nodes $N$ in $G$ that $t$ depends on;  /* via dependency arcs from $t$ */
  if !($P(N) \models A(t)$) then  /* "!" representing logic negation */
    Mark $t$ as DU (directly unsatisfied);
    enqueue (Que, $t$);
  endif
endfor

while !empty(Que) do
  $t$ = dequeue (Que);
  Find all unmarked nodes $N$ in $G$ that depend on $t$;  /* via arcs to $t$ */
  foreach $t'$ in $N$
    Find all unmarked nodes $N'$ that $t'$ depends on;  /* via dependency arcs from $t'$ */
    if !($P(N') \models A(t')$) then
      Mark $t'$ as IU (indirectly unsatisfied);
      enqueue (Que, $t'$);
    endif
  endfor
endwhile

if $(true,\ \{p\})$ is marked DU or IU then return "refinement needed";
else
  SCSs = {strong connected subgraphs of unmarked liveness properties};
  if !empty(SCSs) then return "refinement needed";
  else return "$p$ holds";
  endif
endif
```

Fig. 1. Component-Based Abstraction Algorithm

based on the direct or indirect dependencies between p_h and p_k through examining only the property part (but not the assumption part) of each node along the dependency chain. If any $(p_h, A(p_h))$ or $(p_k, A(p_k))$ is not in the graph, include it in the graph, add a directed arc from $(p_g, A(p_g))$ to it, and put it in the queue for nodes to be expanded; otherwise, just add the arc.

Optimization based on ω-automata assertion templates. The above approach to building the property dependency graph may involve a lot of unnecessary component properties since it only considers variable dependencies. This may lead to significant overhead in abstraction construction and refinement. We optimize this approach using heuristics based on the semantic meanings of the ω-automata assertion templates in [16].

Example. Consider a system S with two components C_1 and C_2. C_1 outputs a variable a and inputs a variable b. C_2 outputs two variables b and c and inputs a variable a. The properties of C_1 and C_2 are shown in Figure 2. (Note that all assertions in a set, e.g., A_{12}, are conjunctive.) A system property to be verified is p: `Repeatedly`(c) with no assumption. The property dependency graph constructed for verifying p is in Figure 2.

Property of C_1:
p_{11}: **After** (a) **Never** (a) **UnlessAfter** (b);
A_{11}: {**Never** (b) **UnlessAfter** (a);
After (b) **Never** (b) **UnlessAfter** (a);}

p_{12}: **Repeatedly** (a);
A_{12}: {**After** (a) **Eventually** (b);
Never (b) **UnlessAfter** (a);
After (b) **Never** (b) **UnlessAfter** (a)};

Properties of C_2:
p_{21}: **Never** (b) **UnlessAfter** (a);
After (b) **Never** (b) **UnlessAfter** (a);
A_{21}: {**After** (a) **Never** (a) **UnlessAfter** (b);}

p_{22}: **After** (a) **Eventually** (b);
A_{22}: {**After** (a) **Never** (a) **UnlessAfter** (b);}

p_{23}: **Repeatedly** (c);
A_{23}: {**Repeatedly** (a);
After (a) **Never** (a) **UnlessAfter** (b);}

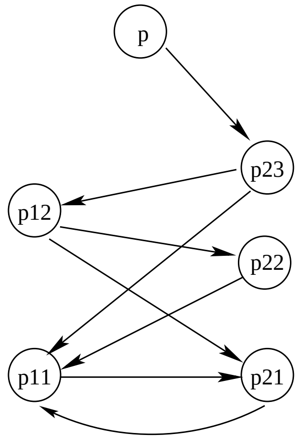

Fig. 2. Component Properties and Property Dependency Graph

Step 2: Determine enabled properties optimistically. After the dependency graph is constructed, we determine, in an optimistic way, whether a sub-component property in the graph is enabled. It is optimistic since we assume, at this point, that dependency cycles do not cause circular reasoning. We will deal with these cycles in the next step.

We first conduct a breadth-first or depth-first search on the graph. For each node t in the graph, we find the set N of all nodes to which t has dependency arcs. We check $P(N) \models A(t)$, i.e., whether the property assertions from all the nodes in N can conjunctively satisfy the assumption assertions in $A(t)$. If no, we mark t as "directly unsatisfied", i.e., even if all the nodes in N are enabled, t will still not be enabled.

Starting from the set of nodes marked as "directly unsatisfied", we recursively identify nodes that are unsatisfied due to their dependencies to nodes that have been marked as unsatisfied. That an unmarked node t' has a dependency arc to an unsatisfied node t does not imply that t' is unsatisfied. We still need to check if the unmarked nodes which t' depends on can satisfy t'. This process terminates when there are no more unsatisfied properties to mark (assuming that dependency cycles do not cause circular reasoning). The unsatisfied nodes identified in this phase are marked "indirectly unsatisfied".

If $(true, p)$ is marked as unsatisfied, directly or indirectly, the abstraction algorithm returns and requests refinement; otherwise, the algorithm moves on to Step 3.

Step 3: Detect liveness circular dependencies. In this step, we detect the existence of circular dependencies among the unmarked liveness sub-component properties in the graph G. The circular dependency detection is by finding the strongly connected sub-graphs of the unmarked liveness sub-component properties. If there exist such sub-graphs in G, the abstraction algorithm returns and requests refinement.

Remarks: The component-based abstraction constructed by our algorithm includes all the sub-component properties which $(true, p)$ depends on and are identified as enabled. In this algorithm, we determine whether a node t can be satisfied by the nodes in N, by applying model checking, specifically, applying COSPAN with $P(N)$ as the system and $A(t)$ as the property to be checked. The complexity of such a check depends on the property and assumption assertions involved. Since we specify these assertions using the templates in [16], each assertion is simple and has only a few states. Therefore, the number of assertions is the deciding factor. The overall complexity of our algorithm also depends on the number of calls to COSPAN. The number of calls to COSPAN is, in the worst case, the sum of the number of nodes and the number of arcs in G. Other complexities of this algorithm include that of building the property dependency graph and that of detecting strongly connected sub-graphs of unmarked liveness properties.

3.3 Mechanized Abstraction Refinement

Our abstraction algorithm may fail to verify a property $(p, A(p))$ for the following two reasons: (1) the sub-component properties are insufficient to verify $(p, A(p))$ and (2) there exist liveness property dependency cycles that, before being validated to be free of circular reasoning, preclude inclusion of the involved sub-component properties in the abstraction. Below, we present our mechanized approaches to addressing the problems.

Insufficient sub-component properties. When our abstraction algorithm reports that it fails to establish $(p, A(p))$ due to insufficient sub-component properties, our refinement assistant conducts a breadth-first search through the dependency graph generated by the abstraction algorithm to identify all nodes that are marked "directly unsatisfied" and reachable from $(p, A(p))$ through only nodes marked "indirectly unsatisfied". For each such node, the assistant outputs the node, the nodes it depends on, and the error trace of the COSPAN call on this node. The user is asked to modify existing sub-component properties and introduce new sub-component properties. These modified or new sub-components properties need to be verified. If a sub-component is a primitive component, its modified or new properties are directly checked on the component; otherwise, the properties are checked again through component-based abstraction.

Liveness property circular dependencies. When our abstraction algorithm reports liveness property circular dependencies, our refinement assistant provides to the user all strongly connected sub-graphs of unmarked liveness properties. An automatic remedy the assistant can provide is to check the additional conditions (such as C3 in Rule 1) dictated by the rules in Sec 2.2, which, if established, can prevent circular reasoning. These conditions are essentially additional properties to be checked on the involved sub-components. If a sub-component is primitive, its additional property can be checked directly; otherwise, component-based abstraction is recursively applied. If these rules fail, the user needs to manually validate that the sub-graphs be free of circular reasoning using techniques such as temporal induction [11], modify the existing sub-component properties, or introduce new sub-component properties. (Note that modification of existing properties and introduction of new properties may lead to new circular dependencies.) If all sub-graphs are shown to be free of circular reasoning, $(p, A(p))$ holds.

Remarks: The abstraction/refinement loop terminates when $(p, A(p))$ is verified or the user aborts this loop. The user aborts this loop when an error is found or she has difficulty in modifying or introducing sub-component properties for verifying $(p, A(p))$. Errors in component composition are detected through the user examining the unsatisfied nodes in the dependency graph. Errors in sub-components are detected when verification of sub-component properties fails.

4 Application in Hardware/Software Co-Verification

An Illustrative Example. We illustrate component-based abstraction and refinement with its application in hardware/software (HW/SW) co-verification of a sensor system as shown in Figure 3. Its software is partitioned into two components: software sensor

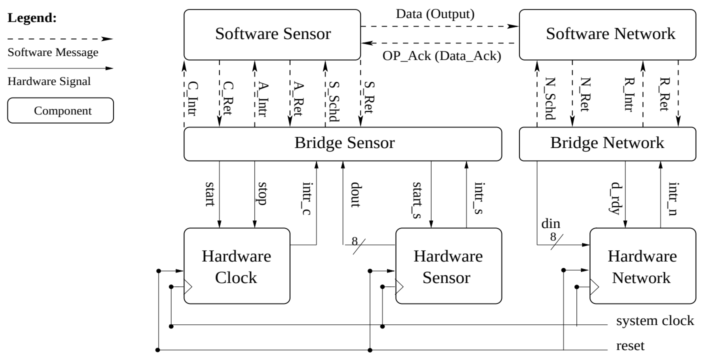

Fig. 3. Architecture of a sensor instance with software in xUML and hardware in Verilog

(*S-SEN*) and software network (*S-NET*) and its hardware is partitioned into three components: hardware clock (*H-CLK*), hardware sensor (*H-SEN*), and hardware network (*H-NET*). The software components are specified in xUML [18] while the hardware components are specified in Verilog [19]. The software and hardware components are connected by bridge components (*B-SEN* and *B-NET*), which interact with the software components following the software semantics and with the hardware components following the hardware semantics and propagate events such as software messages and hardware interrupts across the HW/SW boundary. A property to verify on this system is shown in Figure 4. This property asserts that the sensor system transmits on the net-

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

Fig. 4. Repeated transmission property

work repeatedly. Repeated setting and clearing of a flag in *H_NET* indicates repeated transmission. (Space limitation precludes presentation of the component properties.)

We apply component-based abstraction to verify the system property. The abstraction algorithm constructs an assume-guarantee dependency graph as shown in Figure 5. The abstraction algorithm is able to enable each property optimistically by ignoring

/* Properties of *S-SEN* */
$P_{SS}(1) \rightarrow \{P_{SN}(1)\}$
$P_{SS}(2) \rightarrow \{P_{BS}(1)\}$
$P_{SS}(3) \rightarrow \{P_{SN}(4), P_{SN}(1), P_{BS}(1)\}$
$P_{SS}(4) \rightarrow \{P_{SN}(4), P_{SN}(1), P_{BS}(3), P_{BS}(2), P_{BS}(1)\}$

/* Properties of *S-NET* */
$P_{SN}(1) \rightarrow \{P_{SS}(1)\}$
$P_{SN}(2) \rightarrow \{P_{SN}(1), P_{SS}(1), P_{BN}(1)\}$
$P_{SN}(3) \rightarrow \{P_{SN}(1), P_{SS}(1), P_{BN}(1)\}$
$P_{SN}(4) \rightarrow \{P_{SN}(1), P_{SS}(1), P_{BN}(1)\}$
$P_{SN}(5) \rightarrow \{P_{SN}(1), P_{SS}(4), P_{SS}(1), P_{BN}(2), P_{BN}(1)\}$

/* Properties of *B-SEN* and *B-NET* */
$P_{BS}(1) \rightarrow \{P_{SS}(2), P_{HS}(1)\}$
$P_{BS}(2) \rightarrow \{P_{SS}(3), P_{SS}(2), P_{HS}(2), P_{HS}(1)\}$
$P_{BS}(3) \rightarrow \{P_{SS}(3), P_{SS}(2), P_{HC}(1), P_{HS}(2), P_{HS}(1)\}$
$P_{BN}(1) \rightarrow \{P_{SN}(2), P_{HN}(1)\}$
$P_{BN}(2) \rightarrow \{P_{SN}(3), P_{SN}(2), P_{HN}(2), P_{HN}(1)\}$
$P_{BN}(3) \rightarrow \{P_{SN}(5), P_{SN}(3), P_{SN}(2), P_{HN}(2), P_{HN}(1)\}$

/* Properties of *H-NET* */
$P_{HN}(3) \rightarrow \{P_{BN}(3)\}$

Fig. 5. Dependencies among component properties

dependency cycles. In this graph, there are only dependency cycles involving safety properties, there is no need to check for additional conditions and all these properties hold and they enable all other properties that depend on them. All the involved properties form the abstraction on which the system property is successfully verified.

To evaluate the effectiveness of our refinement assistant, we intentionally omit the properties $P_{BN}(3)$ and $P_{BS}(3)$ and their assumptions since these are properties of the bridge components that cannot be automatically generated from their designs. We then apply abstraction and refinement. The assistant reports that the property $P_{HN}(3)$ is not enabled since one of its assumptions is not satisfied due to the omission of $P_{BN}(3)$. When $P_{BN}(3)$ and its assumptions are introduced, the assistant then reports that the property $P_{SS}(4)$ is not enabled since one of its assumptions is not satisfied due to the omission of $P_{BS}(3)$. The key here is that the user is only notified when manual remedies are necessary. This refinement assistant is effective in locating unsatisfied assumptions and reports them to the user as hints for further property modification and introduction.

Experimental Results. Table 1 shows statistics from verification of the property in Figure 4 on three sensor systems of increasing complexity using three different approaches. The “Basic” system refers to the system discussed above. The “Multi” system and the “Encrypting” system are more complex than the “Basic” system. (See [20] for details of these systems.) TBCV denotes translation-based co-verification [21] which translates an entire system into S/R and then verifies the entire system with COSPAN. In the manual component-based co-verification (CBCV) approach [20], the component-based abstraction of a system is manually constructed. (Manually created abstractions serve as guidance in optimizing our automatic algorithm.) In the automatic CBCV approach, our automatic abstraction algorithm is applied to construct the abstraction. The time (or memory, respectively) usage of verifying a system using CBCV is the sum (or max) of the time (or memory) usages of verifying the new components and the abstraction. The component properties are verified by translating the properties and the corresponding

	Usages	Basic	Multi	Encrypting
TBCV	Time (Sec)	31272.8	-	-
TBCV	Memory (MB)	1660.62	Out of memory	Out of memory
Manual CBCV	Time (Sec)	41.89	10.34	0.77
Manual CBCV	Memory (MB)	9.11	6.05	3.57
Manual CBCV	# of COSPAN Calls	8	2	4
Automatic CBCV	Time (Sec)	205.93	10.45	12.97
Automatic CBCV	Memory (MB)	27.57	4.44	3.54
Automatic CBCV	# of COSPAN Calls	39	24	39

Table 1. Time and memory usage comparison

components into S/R and applying COSPAN. (Translation of hardware components in Verilog utilizes FormalCheck [22] while translation of software components in xUML and bridge components utilizes ObjectCheck [23].) It can be observed that the time and memory usages of automatic CBCV are order-of-magnitude smaller than those of TBCV in verifying the first system and TBCV fails to verify the other systems due to out-of-memory while automatic CBCV finishes the verification using little time and memory (which include those for graph construction and strongly connected sub-graph detection). Although automatic CBCV uses more time and memory than manual CBCV, it is automatic and requires no manual effort in abstraction construction.

5 Related Work

Our approach builds on and extends compositional reasoning [6], in particular, assume-guarantee compositional reasoning [7–13]. It combines assume-guarantee compositional reasoning with abstraction/refinement [14] by utilizing component properties as abstractions. It integrates compositional reasoning and abstraction/refinement with component-based development and leverages assertion-based verification to address the component property formulation problem in application of compositional reasoning.

Abstraction techniques [5, 14], as applied in model checking, reduce a system to a less complex system while preserving correctness of the property to be checked. Major approaches to abstraction that have been practically useful include (but are not limited to) localization reduction [14], data abstraction [5], and predicate abstraction [24]. In [11], McMillan integrated data abstraction, assume-guarantee compositional reasoning, and theorem proving techniques in the context of the Cadence SMV system [11]. Our approach, although more restricted compared to McMillan's approach, is more lightweight and is more closely integrated with component-based development.

6 Conclusions and Future Work

In this paper, we have presented a comprehensive approach to component-based abstraction and refinement. This approach is generally applicable although our implementation is based on the ω-automaton semantics and the COSPAN model checker, since its foundation is compositional reasoning. It advances compositional reasoning via integration with component-based development and assertion-based verification.

The accuracy and efficiency of our abstraction algorithm and refinement assistant is affected significantly by the dependency graphs that are constructed over component properties. The dependency graphs are conservative in that they do not omit any true dependency. However, there may be false dependencies introduced by dependency analysis. False dependencies may prevent the abstraction algorithm from including properties that should have been included in an abstraction and may also prevent the refinement assistant from providing an accurate description of the causes for a compositional reasoning failure. We will research better methods for removing false dependencies.

References

1. Jacome, M.F., Peixoto, H.P.: A survey of digital design reuse. IEEE Design and Test of Computers **18**(3) (2001)
2. Szyperski, C.: Component Software - Beyond Object-Oriented Programming. Addison Wesley (2002)
3. Maliniak, D.: Assertion-based verification smooths the road to IP reuse. Electronic Design (September 2002)
4. IEEE: IEEE Property Specification Language (PSL) (IEEE Std 1850-2005). IEEE (2005)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)
6. 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)
7. Chandy, K.M., Misra, J.: Proofs of networks of processes. IEEE Transaction on Software Engineering **7**(4) (1981)
8. Jones, C.B.: Development methods for computer programs including a notion of interference. PhD thesis, Oxford University (1981)
9. Abadi, M., Lamport, L.: Conjoining specifications. TOPLAS **17**(3) (1995)
10. Alur, R., Henzinger, T.: Reactive modules. FMSD **15**(1) (1999)
11. McMillan, K.L.: A methodology for hardware verification using compositional model checking. Cadence Design Systems Technical Reports (1999)
12. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Proc. of CAV. (2000)
13. Amla, N., Emerson, E.A., Namjoshi, K.S., Trefler, R.: Assume-guarantee based compositional reasoning for synchronous timing diagrams. In: Proc. of TACAS. (2001)
14. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)
15. Hardin, R.H., Har'El, Z., Kurshan., R.P.: COSPAN. In: Proc. of CAV. (1996)
16. Xie, F., Yang, G., Song, X.: Compositional reasoning for hardware/software co-verification. In: Proc. of ATVA. (2006)
17. Alpern, B., Schneider, F.: Defining liveness. Information Processing Letters **21**(4) (1985)
18. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley (2002)
19. IEEE: IEEE Standard for Verilog (IEEE Std 1364-2005). IEEE (2005)
20. Xie, F., Yang, G., Song, X.: Component-based hardware/software co-verification. In: Proc. of MEMOCODE. (2006)
21. Xie, F., Song, X., Chung, H., Nandi, R.: Translation-based co-verification. In: Proc. of MEMOCODE. (2005)
22. Kurshan, R.P.: FormalCheck User Manual. Cadence (1998)
23. Xie, F., Levin, V., Browne, J.C.: Objectcheck: A model checking tool for executable object-oriented software system designs. In: Proc. of FASE. (2002)
24. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. (1997)