Verified Systems by Composition from Verified Components *

Fei Xie and James C. Browne
Dept. of Computer Sciences
Univ. of Texas at Austin
Austin, TX 78712, USA
{feixie, browne}@cs.utexas.edu

ABSTRACT

This paper presents an approach to integration of model checking into component-based development of software systems. This approach assists in development of highly reliable component-based software systems and reduces the complexity of verifying these systems by utilizing their compositional structures. Temporal properties of a software component are specified, verified, and packaged with the component. Selection of a component for reuse considers not only its functionality but also its temporal properties. When a component is composed from other components, a property of the component is verified on an abstraction of the component. The abstraction is constructed from environment assumptions of the component and verified properties of its sub-components. A general component model that enables component verification is defined. Component verification is discussed in the context of the instantiation of the general component model on an Asynchronous Interleaving Message-passing computation model. This approach has been applied to improve reliability of instances of TinyOS, a component-based run-time system for networked sensors. A case study on TinyOS is included, which illustrates the applicability of this approach, the detection of a bug, and the reduction in model checking complexity.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Modules and Interfaces; D.2.4 [**Software/Program Verification**]: Formal Methods, Model Checking, Reliability; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical Verification

General Terms

Design, Reliability, Verification

Keywords

Component, composition, verification, model checking, abstraction

*This research was partially supported by NSF grant 010-3725.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

1. INTRODUCTION

Component-Based Development (CBD) [23], developing software systems through composition of components, is one of the most important technical initiatives in software engineering. Model checking [9, 22, 10] is an important method for improving reliability of software systems. It provides exhaustive state space coverage for the systems being checked and is particularly effective at detecting coordination errors which frequently result from component compositions and are notoriously difficult to detect. However, model checking often cannot handle large-scale software systems due to state space explosions. Model checking and CBD are synergistic. Model checking can potentially enable effective development of more reliable component-based software systems. CBD introduces compositional structures, clean component interfaces, and standard composition rules to the systems being built, which may reduce the state spaces that model checkers have to handle.

This paper defines, discusses, and illustrates an approach to integration of model checking into the CBD of software systems, which contributes to solution of the following fundamental problems in CBD and model checking:

- Developing components which can be reused with certainty that their behaviors will meet their specifications in a proper composition;
- Identifying proper components for a composition;
- Establishing that a component composed from "correct" components will meet its specifications;
- Alleviating the state space explosion problem.

This approach can be summarized as follows:

- As a software component is built, temporal properties of the component are specified, verified, and then packaged with the component.
- Selecting a component for reuse considers not only its functionality but also its temporal properties.
- Verification of properties of a composed component reuses verified properties of its sub-components and is based on compositional reasoning [21, 2, 1, 19, 3, 11].

A general component model that enables component verification is defined. This model provides a framework for representing components and their properties and for composing components. In this model, a property of a component is defined with assumptions on the environment of the component. The property is verified on the component under these assumptions. When the component is

reused in the composition of a larger component, the verified property is *enabled* if the environment assumptions made in its verification hold on other components in the composition and/or the environment of the composed component. (The formal definition of an enabled property is given in Section 3.5.1).

The general component model can be instantiated on existing computation models upon which syntax and semantics for components, properties, and component compositions are precisely defined. These computation models also provide semantics for component execution and interaction. We demonstrate the general component model with its instantiation on an Asynchronous Interleaving Message-passing (AIM) computation model. In this instantiation, executable representations of components are specified in xUML [20], an executable dialect of UML, whose semantics conform to the AIM model. This instantiation is of interest because the AIM model captures the essential nature of a broad range of concurrent software systems.

Components are categorized as *primitive* components (components built from "scratch" and not composed from other components) or as *composed* components. A property of a primitive component is verified by directly model checking an executable representation of the component, for instance, checking the executable design model (specified in xUML) of the component using methods established in our previous work [24, 26, 27]. A property of a composed component, instead of being model checked on the executable representation of the component, is checked on an *abstraction* of the component. The abstraction is composed of simple automata corresponding to environment assumptions of the composed component and verified properties of its sub-components. A verified sub-component property is included in the abstraction if it is enabled in the composition, related to the property to be checked by cone-of-influence analysis [10], and not involved in invalid circular dependencies [19] among sub-component properties. If the abstraction is still too complex to be checked directly, compositional reasoning is applied to decompose the abstraction. If the abstraction is too abstract to enable the verification of the desired property, it is refined as follows: decomposing the property into a set of properties of the sub-components, verifying these properties on the sub-components, and then including these verified properties into the abstraction. Algorithms for abstraction construction and refinement are based on compositional reasoning.

Our approach is most suitable for application to a product line of software systems that are built from a growing set of software components. We have identified two major application domains: product lines of software systems based on a specific hardware/software architecture, such as the TinyOS [16] run-time system, and product lines of distributed large-scale software systems based on component platforms such as CORBA, DCOM, and EJB.

The rest of this paper is organized as follows. Section 2 defines the general component model for component verification and instantiates it on the AIM computation model. Section 3 elaborates on how to verify properties of components, either primitive or composed. Section 4 illustrates our approach with a case study on TinyOS. Section 5 analyzes the effectiveness of our approach in the context of the case study. Section 6 presents the related work. Section 7 discusses the future work and concludes.

2. COMPONENT MODEL FOR VERIFICATION

In this section, we first define the general component model, which is a template that can be instantiated on an existing computation model to enhance it with components, properties, and component compositions which enable component verification. We then sketch the AIM computation model. After that, we instantiate the general component model on the AIM model.

2.1 General Component Model

2.1.1 Component

A component, C, is a four-tuple, $(E,\ I,\ V,\ P)$, where

- E is an executable representation of C.
- I is an interface through which C interacts with other components, for instance, a messaging interface or a procedural interface.
- V is a set of variables defined in E and referenced by the properties defined in P.
- P is a set of temporal properties that are defined on I and V, and have been verified on E. A temporal property is denoted by a pair, $(p,\ A(p))$, where p is a temporal formula defined on I and V, and $A(p)$ is a set of temporal formulas defined on I and V. The property, p, holds on C if the temporal formulas in $A(p)$ hold on the *environment* of C (see the next paragraph for the definition of the environment of a component). P is extended incrementally by including properties that are newly verified. A property is included in P only when it is verified.

An application system is also a component. The environment with which the system interacts is also modeled as a set of components. The set of components with which a component interacts is referred to as the environment of the component. The environment of a component varies as the component is reused in different compositions. Given a component and its property, $(p,\ A)$, the temporal formulas in A are referred to as the environment assumptions of the component for enabling p.

2.1.2 Component Composition

A component, $C = (E,\ I,\ V,\ P)$, can be composed from a set of simpler components, $(E_0,\ I_0,\ V_0,\ P_0), \ldots, (E_{n-1},\ I_{n-1},\ V_{n-1},\ P_{n-1})$, as follows:

- E is constructed from $E_0, \ldots, E_{n-1}$ by connecting $E_0, \ldots, E_{n-1}$ through their interfaces.
- I is derived from $I_0, \ldots, I_{n-1}$: An operation in I_i, $0 \leq i < n$, is included in I if and only if it is used when C interacts with other components.
- V is a subset of $\bigcup_{i=0}^{n-1} V_i$. A variable in $\bigcup_{i=0}^{n-1} V_i$ is included in V if and only if the variable is referenced by the properties defined in P.
- P is a set of temporal properties defined on I and V, and verified on E. Properties in P are verified on E by utilizing the properties in $P_0, \ldots, P_{n-1}$.

2.2 AIM Computation Model

The AIM computation model is a commonly used computation model for software systems. Under our definition of the AIM model, a system is a finite set[1] of interacting processes. A process is a four-tuple, $(X,\ \Sigma,\ Q,\ \Delta)$, where X is a set of variables each of which is of a bounded type; Σ is an extended Moore state model; Q is an infinite FIFO message queue; Δ is an initial condition that consists

[1]If not specified explicitly, a set referred to in this paper is finite.

of an initial value for each variable in X, an initial state for Σ, and the empty state for Q. An extended Moore state model is a three-tuple, $(\Phi,\ M,\ T)$, where Φ is a set of states, each of which has an associated state action; M is a set of message types (assuming that $M's$ of different processes are disjoint); T is a set of state transitions defined on Φ and M, each of which is a three-tuple, $(r,\ t,\ m)$, where r and t are two states in Φ and m is a message type in M. A state action is a program segment composed from the following executable statements: an empty statement, an assignment statement, a messaging statement that outputs a message, an if statement, or a composite statement that is a sequential composition of statements.

An execution of an AIM system is an interleaving of state transitions and state actions of the processes in the system.

- A process is *ready* if and only if either it just entered a state and has not yet executed the state action associated with the state or one of its state transitions is enabled by the first message in its message queue.
- At any given moment, exactly one *ready* process is nondeterministically scheduled to execute from among all the *ready* processes. When scheduled, a *ready* process executes an enabled state transition or an enabled state action run-to-completion.

Processes in a system interact through asynchronous message-passing. Given two processes, P_1 and P_2, to send a message from P_1 to P_2, a messaging statement is included in a state action of P_1. The messaging statement includes a message type, m, defined in M of P_2, and the parameters required by m. When the state action is executed, a message of the type, m, with its parameters is put in the message queue of P_2.

2.3 Instantiation of General Component Model on AIM Computation Model

This section discusses the instantiation of the general component model on the AIM computation model with the focus on how to derive the executable specification and the interface of a composed component from its sub-components.

2.3.1 Component

A component, C, is a four-tuple, $(E,\ I,\ V,\ P)$, where

- E is an executable representation of C with syntax and semantics conforming to the AIM model. E can be either an AIM specification that consists of a set of interacting AIM processes, or an implementation of the AIM specification in a programming language.
- I is the messaging interface through which C interacts with other components and is a pair, $(R,\ S)$, where R (or S, respectively) is a set of input (or output) message types whose instances may be input (or output) by C, more precisely by processes in C, when C interacts with other components.
- V and P inherit their definitions from the general component model and reference semantic entities in the instantiations of E and I on the AIM model.

2.3.2 Component Composition

A component, $C = (E,\ I,\ V,\ P)$, is composed from a set of sub-components, $(E_0,\ I_0,\ V_0,\ P_0), \ldots, (E_{n-1},\ I_{n-1},\ V_{n-1},\ P_{n-1})$, as follows:

- E is constructed from $E_0, \ldots, E_{n-1}$, by mapping output message types in $S_0, \ldots, S_{n-1}$ to input message types in $R_0, \ldots, R_{n-1}$. If there is a mapping defined between an output message type, s, in S_i, $0 \leq i < n$, and an input message type, r, in R_j, $0 \leq j < n$, the following steps are executed:
  - A conformance check is performed on the parameter lists of s and r;
  - All occurrences of s in E_i are replaced by r, except the occurrences where messages of the type, s, are output to components outside C;
  - If s is mapped to more than one input message type, a messaging statement that outputs s is replicated for each input message type.
- $I = (R,\ S)$ is derived from $I_i = (R_i,\ S_i)$, $0 \leq i < n$. R (or S, respectively) is a subset of $\bigcup_{i=0}^{n-1} R_i$ (or $\bigcup_{i=0}^{n-1} S_i$). A message type in $\bigcup_{i=0}^{n-1} R_i$ (or $\bigcup_{i=0}^{n-1} S_i$) is included in R (or S) if and only if messages of that type may be input (or output) by C when C interacts with other components.
- V is derived by following the corresponding rule given in the general component model.
- Formulation of the properties in P and verification of these properties by utilizing the properties in $P_0, \ldots, P_{n-1}$ are discussed in Section 3 in detail.

2.3.3 Component Execution and Interaction

The execution semantics of components are defined recursively (assuming bounded recursion). When a component executes, if the component has no sub-components, then at any given moment exactly one AIM process in the component executes; if the component has sub-components, then at any given moment exactly one sub-component executes.

Components interact with each other through message-passing. A component can only input (or output, respectively) messages of the types listed in its input (or output) messaging interface. Messages input (or output) by a component are consumed (or generated) by AIM processes in the component or its recursively nested sub-components.

3. VERIFICATION OF COMPONENTS

This section discusses verification of components under the instantiation of the general component model on the AIM computation model. First, we introduce how a closed AIM system is verified. Then, we discuss how component properties are formulated. Finally, we differentiate components into two categories, *primitive* and *composed*, and present procedures for verifying components of the two categories respectively.

3.1 Background: Verification of a Closed AIM System

There are many software design specification languages whose semantics conform to the AIM model, such as xUML [20], an executable dialect of UML, and SDL [17]. In previous research [26, 27], we designed and implemented an approach to model checking executable software system designs specified in xUML, which can be summarized as follows:

- A system design is specified in xUML as an executable model and a property to be checked on the design is specified in an xUML level property specification language. (Details of this property specification language are given in the appendix.)

- The xUML model and the property are automatically translated to a model and a property in the S/R [15] automaton language, the input language of COSPAN [15] model checker.
- The S/R property is checked on the S/R model by COSPAN. In COSPAN, both the S/R model and the S/R property are represented as compositions of ω-automata [18]. (A composition of ω-automata is also an ω-automaton.) COSPAN conducts a language containment check, checking whether the language of the ω-automaton corresponding to the S/R model is a subset of the language of the ω-automaton corresponding to the S/R property [18]. The property holds on the model if the language containment check returns true.
- If the property does not hold on the model, an error trace is generated by COSPAN and is automatically translated to an error report in the name space of the xUML model.

This approach requires that a system design to be model checked specify a closed system. A system is made closed by modeling its environment as part of the system.

This approach suffers from the state space explosion problem. To verify large-scale software system designs, we extended the approach with top-down application of compositional reasoning, where model checking a property on a system is accomplished by decomposing the system into modules, checking module properties locally on the modules, and deriving the system property from the module properties [24]. We applied compositional reasoning in model checking xUML specifications by following the Translation-Based Compositional Reasoning approach proposed in [25], where compositional reasoning rules are established in the semantics of software systems, but are proved and implemented based on translation of software systems to formal representations for which compositional reasoning rules have already been established, proved, and implemented.

3.2 Formulation of Component Properties

After the AIM specification and the messaging interfaces of a component are constructed, properties of the component can then be formulated. Properties are mainly derived from functional specifications of the component such as input and output relationships through domain analysis. However, it is not required that all properties of the component be packaged initially. Additional properties may be introduced incrementally as the component is reused in composing other components. Verification of a property on a composed component may require top-down application of compositional reasoning to decompose the property into a set of properties on its sub-components. The sub-component properties are then verified on the sub-components and packaged with the sub-components for future reuse. This top-down application of compositional reasoning requires that system/component developers manually guide the property decomposition, however, it enables verification of large-scale components that cannot otherwise be verified.

Since our approach targets a product line of software systems constructed from a set of components, we assume that the AIM specifications of the components are available to system/component developers, which facilitates incremental introduction and verification of component properties. The set of properties of a component is expected to become quite stable after a few reuses.

3.3 Formulation of Environment Assumptions

Our approach requires that a property of a component be specified with its assumptions on the environment of the component. We have investigated both automatic generation and manual formulation of environment assumptions. Given a component and a property, an assumption that enables the property on the component can be automatically generated by taking the complement of the product of the property and its cone-of-influence on the component. (Various optimizations are possible.) Construction of the complement, the product, and the cone-of-influence is supported by COSPAN. The assumption generated is the weakest assumption that enables the property on the component, however, it is usually a complex and non-intuitive assumption which is difficult to check on other components composed with the component in a composition. A desired set of assumptions for the property is a set of simple and intuitive assumptions formulated on the interface of the component. Each of these assumptions is weaker than the automatically generated assumption, however, the conjunction of these assumptions is stronger than the automatically generated assumption. These assumptions are often easier to check on other components. Domain-specific knowledge of system/component developers is expected to facilitate the formulation of such a set of assumptions. Therefore, currently in our approach environment assumptions are formulated manually. Investigation of heuristics that can reduce or decompose the automatically generated assumption by utilizing domain-specific knowledge is in progress.

3.4 Verification of Primitive Components

A primitive component often has limited functionality. As a result, the state space of a primitive component is often of modest size and suitable for direct application of model checking. The approach in Section 3.1 is employed to verify a primitive component. However, the AIM specification of a primitive component often does not specify a closed system and the approach cannot be readily applied. Therefore, we construct a closed system from the AIM specification and the environment assumptions of the component.

Given a primitive component, $C = (E,\ I,\ V,\ P)$, and a property, $(p,\ A(p))$, specified on I and V, in order to check whether p holds on E assuming that assumptions in $A(p)$ hold on the environment of C, the following steps are executed:

1. Create an AIM process, ENV, whose input message types are the same as the output message types defined in I and whose state model outputs messages of the input message types defined in I;
2. Build an AIM system from ENV and the AIM processes in E and translate the system into S/R;
3. Free all variables of the automata corresponding to ENV in the S/R model obtained in Step 2 so that these variables may obtain any value in their domains non-deterministically; (Discussions on freeing variables in an S/R model can be found in [15, 18].)
4. Translate assumptions in $A(p)$ to S/R automata and compose them with the S/R model obtained in Step 3 so that the free variables introduced in Step 3 are now constrained by the assumptions in $A(p)$;
5. Translate p to an S/R property and check the S/R property on the S/R model gotten in Step 4.

These steps construct a closed system by using the ENV process as a translation stub and replacing ENV with the assumptions in $A(p)$ in the resulting S/R model, and then verify p on the closed system. Construction of the closed system is simplified by the fact that in S/R, models, properties, and assumptions are all specified as automata that are of the same form and can be trivially composed.

3.5 Verification of Composed Components

In this section, we present a method for verification of a property, $(p,\ A(p))$, on a composed component, $C = (E,\ I,\ V,\ P)$, where C is composed from $C_0 = (E_0,\ I_0,\ V_0,\ P_0)$ and $C_1 = (E_1,\ I_1,\ V_1,\ P_1)$. This method reuses the properties that have been verified on the sub-components, C_0 and C_1. It can be readily extended to the case that C is composed from $C_0, \ldots, C_{n-1}$.

3.5.1 Component Abstraction Construction

Since the AIM specification of a composed component often has a large state space that hinders direct application of model checking, we construct an abstraction of the component based on the composition, the environment assumptions of the component, the messaging interfaces of the sub-components, and the verified properties of the sub-components.

Before discussing how to construct the abstraction, we first elaborate on the concept of *enabled property*. A property of a component is defined with assumptions on the environment of the component. The property is verified on the component under these assumptions. When the component is reused in the composition of a larger component, the property is *enabled* if the environment assumptions made in its verification hold on other components in the composition and/or the environment of the composed component. We now formally define an *enabled property*.

Definition – Enabled Property: *A property $(p_i,\ A(p_i))$ of C_i, where $i \in \{0,\ 1\}$ and $(p_i,\ A(p_i)) \in P_i$, is enabled in the composition of C_0 and C_1 if and only if either $A(p_i)$ is empty or for each q, $q \in A(p_i)$, q is implied by the assumptions in $A(p)$ and the properties in P_{1-i} that are enabled in the composition.*

The function in Figure 1 can be applied to determine whether a

```
boolean function enabled ( (p_i, A(p_i)) ) begin
  while ( !empty( A(p_i) ) ) do
    q = remove-an-element ( A(p_i) ); P'={ };
    foreach ((p', A(p')) ∈ cone(A(P) ∪ P_{1-i}, q) ∩ P_{1-i})
      if ( enabled ( (p', A(p')) ) then
        P'=P' ∪ {(p', A(p'))};
      endif;
    endfor;
    if (q is implied by cone(A(p) ∪ P', q)) then continue;
    else return false;
    endif;
  endwhile;
  return true;
end;
```

Figure 1: The "enabled" function

property $(p_i,\ A(p_i))$ of C_i, $i \in \{0,\ 1\}$, is enabled in the composition of C_0 and C_1 assuming that the assumptions in $A(p)$ hold on the environment of the composition. For each assumption q, $q \in A(p_i)$, the function first identifies the set P' of properties that are in P_{1-i}, related to q according to cone-of-influence analysis, and enabled. (In Figure 1, $cone(A(p) \cup P_{1-i},\ q)$ denotes the cone-of-influence of q on $A(p) \cup P_{1-i}$. $cone(A(p) \cup P_{1-i},\ q)$ includes assumptions in $A(p)$ and properties in P_{1-i}, which reference the semantics entities in C that influence the semantics entities referenced by q.) If q is implied by $cone(A(p) \cup P',\ q)$, then the function continues with the next assumption in $A(p_i)$; otherwise, the function returns false. The implication can be decided by either matching q to an element of $cone(A(p) \cup P',\ q)$, or model checking q on the product of the elements of $cone(A(p) \cup P',\ q)$. If a property, $(p_i,\ A(p_i))$, of C_i is not currently enabled in the composition of C_0 and C_1, it does not indicate that p_i does not hold on C_i under the composition. $(p_i,\ A(p_i))$ can become enabled when all assumptions in $A(p_i)$ become enabled, which may require checking additional properties of C_0 and C_1.

The abstraction of the composed component, C, upon which the property, *p*, is to be verified, is derived as follows:

- Realize the output message interfaces of C_0 (or C_1, respectively) in the context of C by replacing the output message types of C_0 (or C_1) with the corresponding input message types of C_1 (or C_0) according to the mappings among the output message types of C_0 (or C_1) and the input message types of C_1 (or C_0);
- Create an AIM system, *SYS*, which consists of three stub AIM processes, CP_0, CP_1, and ENV, where: (i) CP_0 (or CP_1, respectively) is corresponding to C_0 (or C_1), whose variables have the same names and domains as the variables in V_0 (or V_1), whose input message types are the same as the input message types of C_0 (or C_1), and whose state model outputs messages of the output message types of C_0 (or C_1); (ii) ENV is corresponding to the environment of C, whose input message types are the same as the output message types of C and whose state model outputs messages of the input message types of C;
- Run the "enabled" function in Figure 1 on each property in P_0 and P_1, and include the property into *SYS* if the function returns true;
- Include the assumptions in $A(p)$ into *SYS*;
- Run the cone-of-influence analysis on *SYS* to exclude properties and assumptions not related to p.

There may exist circular dependencies among the properties of sub-components. Circular dependencies among the liveness properties of sub-components may be invalid. Suppose we have verified that the property of *Eventually(X)* holds on C_0 assuming that the property of *Eventually(Y)* holds on C_1 and vice versa. We cannot conclude that the property of *Eventually(X) and Eventually(Y)* holds on C unless we can show that there is no execution of C in which *X* and *Y* are always false. Validity of such circular dependencies has to be checked with compositional reasoning rules that support such validity checks, for instance, rules presented in [3] and [19]. Sub-component properties involved in invalid circular dependencies are excluded from the abstraction. Validation of circular dependency can be readily included in the "enabled" function.

3.5.2 Verification of Component Abstraction

Instead of checking the property, p, on the AIM specification of C, we check p on the abstraction, *SYS*:

- Translate *SYS* into S/R;
- Free all variables of the automata corresponding to CP_0, CP_1, and ENV in the resulting S/R model; (These free variables are now constrained by properties from P_0 and P_1 and assumptions from $A(p)$.)
- Translate p into S/R and check the S/R query corresponding to p on the S/R model corresponding to *SYS*;
- Include $(p,\ A(p))$ in P if p holds on *SYS*; otherwise, refine *SYS* as discussed in Section 3.5.3.

The complexity of model checking p on the abstraction, *SYS*, is often much lower than the complexity of directly checking p on the AIM specification of C (see Section 5.2).

3.5.3 *Refinement of Component Abstraction*

If p does not hold on the abstraction, *SYS*, then either p does not hold on C assuming assumptions in $A(p)$ hold on the environment of C, or *SYS* is too abstract. It is often possible to differentiate the two cases by analyzing the error traces generated by the model checker. If p does not hold on C under the assumptions in $A(p)$, then either more assumptions have to be added to $A(p)$ or C has to be re-composed. If *SYS* is too abstract, it must be refined.

The abstraction can be refined by including additional properties of C_0 and C_1. These properties are either properties that are newly introduced, but have not been verified, or properties that have been verified, but are not currently enabled in the composition. If a property to be included has not been verified, it is first verified. If a property to be included has been verified, but is not currently enabled, the procedure in Figure 2 is applied to enable the property. The

boolean procedure enable ($(p_i,\ A(p_i))$) **begin**
  while (!empty ($A(p_i)$)) **do**
    q = remove-an-element ($A(p_i)$); $P' = \{\ \}$;
    foreach $((p',\ A(p')) \in cone(A(p) \cup P_{1-i},\ q) \cap P_{1-i})$
      if (enabled $((p',\ A(p')))$) **then** $P' = P' \cup \{(p',\ A(p'))\}$;
      elseif (enable $((p',\ A(p')))$) **then**
        $P' = P' \cup \{(p',\ A(p'))\}$;
      endif;
    endfor;
    if (q is implied by $cone(A(p) \cup P',\ q)$) **then** continue;
    elseif (q is expected to hold on C_{1-i}) **then**
      A' = { assumptions of q };
      if (!verify ($(q,\ A'), 1-i$)) **then** return false; **endif**;
      return enable ($(q,\ A')$);
    else return false;
    endif;
  endwhile;
return true;
end;

Figure 2: The "enable" procedure

procedure enables a property, $(p_i,\ A(p_i))$, of C_i by enabling all its assumptions. For each assumption, q, the procedure first attempts to enable the properties that are in $cone(A(p) \cup P_{1-i},\ q) \cap P_{1-i}$ and not enabled, by calling itself recursively. After the *foreach* loop, P' contains all the properties in $cone(A(p) \cup P_{1-i},\ q) \cap P_{1-i}$ that have been enabled. If q is implied by $cone(A(p) \cup P',\ q)$, the procedure continues with the next assumption; otherwise, if q is expected to hold on C_{1-i} [2], a set of assumptions, A', of q is introduced and $(q,\ A')$ is verified on C_{1-i}. If $(q,\ A')$ is successfully verified, the "enable" procedure is called on $(q,\ A')$ recursively. The "enable" procedure returns false if a call to the "verify" procedure returns false or if q is neither an assumption of the composed component, C, on its environment nor q is a property that is expected to hold on C_{1-i}. Circular dependencies among properties, introduced by the refinement, must be validated as discussed in Section 3.5.1.

4. CASE STUDY: VERIFICATION OF TINYOS COMPONENTS

We have applied the approach to integration of model checking into CBD to improve reliability of instances of TinyOS [16]. We now illustrate this approach with a case study on TinyOS. TinyOS is a component-based run-time system designed to provide support for deeply embedded systems which require concurrency-intensive operations while constrained by minimal hardware resources. Hardware constraints of deeply embedded systems prohibit loading all TinyOS modules into a single instance and different requirements of these systems require different configurations of TinyOS modules, which makes CBD an appropriate development approach for TinyOS. TinyOS instances are usually loaded to a large number of deeply embedded systems such as networked sensors, which makes correction of software bugs very expensive. Locks and monitors, which are often used to safeguard concurrent operations, are not used in TinyOS due to their computational expenses and the hardware constraints of TinyOS. This combination of complexity and the requirement for high reliability justifies the application of our approach to improve reliability of instances of TinyOS.

4.1 Sensor Component

We sketch how primitive components are specified and verified with the Sensor component. We first introduce the $(E,\ I,\ V,\ P)$ specification of the Sensor component. The executable representation, E, of the Sensor component is specified in xUML. The communication diagram of the Sensor component is shown in Figure 3. (Space limitations prohibit showing all xUML diagrams of E.) The

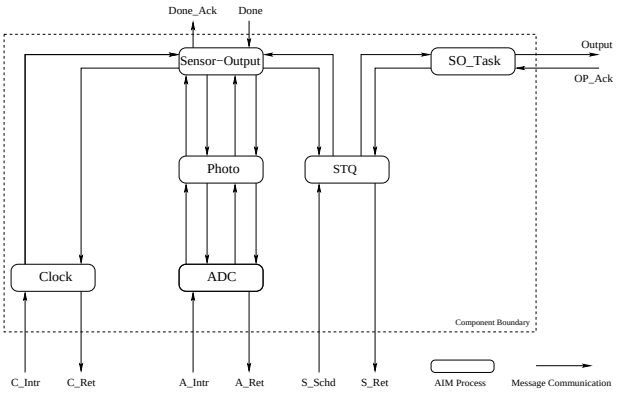

Figure 3: Sensor component

component consists of six AIM processes that interact with each other and the environment of the component through messages. The messaging interface, I, of the component is as follows:

- R={*C_Intr, A_Intr, S_Schd, OP_Ack, Done*};
- S={*C_Ret, A_Ret, S_Ret, Output, Done_Ack*}.

Message types in R are defined in the AIM processes of the Sensor component and the message types in S are to be realized when the component is composed with other components. *C_Intr*, *A_Intr*, and *S_Schd* are the hardware interrupts the Sensor component must handle while *C_Ret*, *A_Ret*, and *S_Ret* are the corresponding replies. The Sensor component outputs Sensor readings as messages of the type, *Output*. The properties to be checked on the Sensor component are listed in Figure 4 with their assumptions. (In Figure 4, the "+" operator denotes a logical OR. Detailed discussions of the property specification language are given in the appendix.) These properties assert that the component repeatedly outputs sensor readings and correctly handles the signal-and-reply relationship between *Output* and *OP_Ack* and between *Done* and *Done_Ack* assuming that the assumptions hold on its environment. The set, V, consists of two variables, *ADC.Pending* and *STQ.Empty*, referenced by the properties and the assumptions listed in Figure 4.

[2]If q is expected to hold on the conjunction of C_{1-i} and $A(p)$, it is first decomposed into sub-assumptions on C_{1-i} and $A(p)$ respectively, which is guided by system/component developers.

Properties:
Repeatedly (Output);

After (Output) **Never** (Output) **UntilAfter** (OP_Ack);

After (Done) **Eventually** (Done_Ack);
Never (Done_Ack) **UntilAfter** (Done);
After (Done_Ack) **Never** (Done_Ack) **UntilAfter**(Done);

Assumptions:
After (Output) **Eventually** (OP_Ack);
Never (OP_Ack) **UntilAfter** (Output);
After (OP_Ack) **Never** (OP_Ack) **UntilAfter** (Output);

After (Done) **Never** (Done) **UntilAfter** (Done_Ack);

Repeatedly (C_Intr);
After (C_Intr) **Never** (C_Intr + A_Intr + S_Schd)
UntilAfter (C_Ret);

After (ADC.Pending) **Eventually** (A_Intr);
After (A_Intr) **Never** (C_Intr + A_Intr + S_Schd)
UntilAfter (A_Ret);

After (STQ.Empty = FALSE) **Eventually** (S_Schd);
After (S_Schd) **Never** (C_Intr + A_Intr + S_Schd)
UntilAfter (S_Ret);

Figure 4: Properties of Sensor Component

The Sensor component has a state space of modest size. The properties listed in Figure 4 were successfully verified on the component by following the steps in Section 3.4 and were included into P for future reuse.

4.2 Network Component

The communication diagram of the Network component is shown in Figure 5. The messaging interface, I, of the Network component

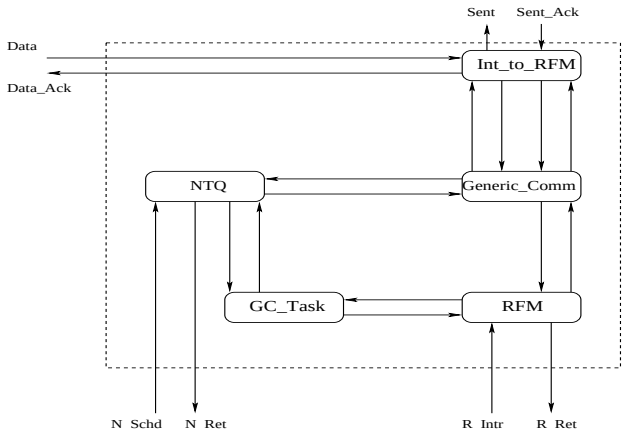

Figure 5: Network component

is as follows:

- R={*N_Schd, R_Intr, Sent_Ack, Data*};
- S={*N_Ret, R_Ret, Sent, Data_Ack*}.

The properties that have been verified on the Network component are listed in Figure 6 with their assumptions. These properties as-

Properties:
IfRepeatedly (Data) **Repeatedly** (RFM.Pending);
IfRepeatedly (Data) **Repeatedly** (Not RFM.Pending);

After (Data) **Eventually**(Data_Ack);
Never (Data_Ack) **UntilAfter** (Data);
After (Data_Ack) **Never** (Data_Ack) **UntilAfter** (Data);

After (Sent) **Never** (Sent) **UntilAfter** (Sent_Ack);

Assumptions:
After (Data) **Never** (Data) **UntilAfter** (Data_Ack);

After (Sent) **Eventually** (Sent_Ack);
Never (Sent_Ack) **UntilAfter** (Sent);
After (Sent_Ack) **Never** (Sent_Ack) **UntilAfter** (Sent);

After (NTQ.Empty = FALSE) **Eventually** (N_Schd);
After (N_Schd) **Never** (N_Schd + R_Intr) **UntilAfter**(N_Ret);

After (RFM.Pending) **Eventually** (R_Intr);
After (R_Intr) **Never** (N_Schd + R_Intr) **UntilAfter** (R_Ret);

Figure 6: Properties of Network Component

sert that the Network component transmits on the physical network repeatedly if it receives inputs repeatedly, and it correctly handles the signal-and-reply relationship between *Data* and *Data_Ack* and between *Sent* and *Sent_Ack*. The set, V, of the Network component consists of two variables, *RFM.Pending* and *NTQ.Empty*.

4.3 Sensor-to-Network Component

This section introduces how an instance of TinyOS, the Sensor-to-Network component, is composed from the Sensor component and the Network component, and discusses how properties of the composed component are verified by utilizing the properties that have been verified on its sub-components.

The executable representation, E, of the Sensor-to-Network component is composed from the executable representations of the Sensor component and the Network component. The abstracted communication diagram of the Sensor-to-Network component is shown in Figure 7, where an annotation of the form of "input message type

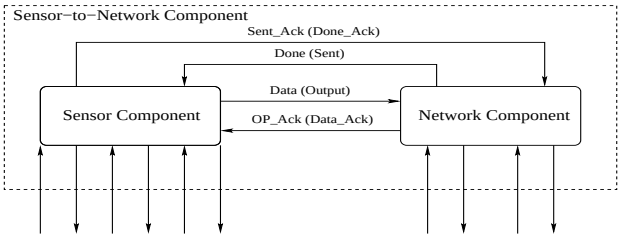

Figure 7: Sensor-to-Network component

(output message type)" denotes the mapping of an output message type of a component to an input message type of the other component. The messaging interface, I, of the Sensor-to-Network component is as follows:

- R={*C_Intr, A_Intr, S_Schd, N_Schd, R_Intr*};
- S={*C_Ret, A_Ret, S_Ret, N_Ret, R_Ret*}.

The properties to be checked on the Sensor-to-Network component are listed in Figure 8 with their assumptions. These properties as-

Properties:
Repeatedly (RFM.Pending);
Repeatedly (Not RFM.Pending);

Assumptions:
Repeatedly (C_Intr);
After (C_Intr) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UntilAfter** (C_Ret);

After (ADC.Pending) **Eventually** (A_Intr);
After (A_Intr) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UntilAfter** (A_Ret);

After (STQ.Empty = FALSE) **Eventually** (S_Schd);
After (S_Schd) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UntilAfter** (S_Ret);

After (NTQ.Empty = FALSE) **Eventually** (N_Schd);
After (N_Schd) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UntilAfter** (N_Ret);

After (RFM.Pending) **Eventually** (R_Intr);
After (R_Intr) **Never** (C_Intr + A_Intr + S_Schd + N_Schd + R_Intr) **UntilAfter** (R_Ret);

Figure 8: Properties of Sensor-to-Network Component

sert that the Sensor-to-Network component repeatedly transmits on the physical network if the assumptions hold on its environment. We refer to these properties together as the "repeated transmission" property hereafter. The set, V, of the Sensor-to-Network component consists of four variables, *ADC.Pending*, *STQ.Empty*, *RFM.Pending*, and *NTQ.Empty*.

In order to check the "repeated transmission" property on the Sensor-to-Network component, we constructed an abstraction of the component following the steps given in Section 3.5.1:

- Replace the output message types of the Sensor (or Network, respectively) component with the corresponding input message types of the Network (or Sensor) component as shown in Figure 7;
- Create an AIM system, SN, which consists of the following three stub AIM processes: (i) SP, whose input message types are *C_Intr*, *A_Intr*, *S_Schd*, *OP_Ack*, and *Done*, whose state model outputs messages of the types, *C_Ret*, *A_Ret*, *S_Ret*, *Data*, and *Sent_Ack*, and whose variables are $Pending$ and $Empty$; (ii) NP, whose input message types are *N_Schd*, *R_Intr*, *Data*, and *Sent_Ack*, whose state model outputs messages of types *N_Ret*, *R_Ret*, *OP_Ack*, and *Done*, and whose variables are $Pending$ and $Empty$; (iii) ENV, whose input message types are *C_Ret*, *A_Ret*, *S_Ret*, *N_Ret*, and *R_Ret*, and whose state model outputs messages of the types, *C_Intr*, *A_Intr*, *S_Schd*, *N_Schd*, and *R_Intr*;
- Execute the cone-of-influence analysis, the "enabled" function in Figure 1, and the validity check of circular dependencies on the properties of the Sensor component and the Network component, and execute the cone-of-influence analysis on the assumptions in Figure 8, which leads to inclusion of the properties in Figure 9 into the abstraction.

We then checked the "repeated transmission" property on SN by following the steps given in Section 3.5.2. It is easy to observe that the property holds on the abstraction under the assumptions in Figure 8. Therefore, we concluded that the property also holds on the executable representation of the Sensor-to-Network component under the given assumptions.

Repeatedly (Data);
After(Data) **Never** (Data) **UntilAfter** (OP_Ack);

IfRepeatedly (Data) **Repeatedly** (RFM.Pending);
IfRepeatedly (Data) **Repeatedly** (Not RFM.Pending);
After (Data) **Eventually**(OP_Ack);
Never (OP_Ack) **UntilAfter** (Data);
After (OP_Ack) **Never** (OP_Ack) **UntilAfter** (Data);

Figure 9: Properties included in abstraction

4.4 Verification via Abstraction Refinement

An abstraction of a composed component may be refined by introducing, verifying, and enabling properties of the sub-components of the composed component or even by revising and re-verifying the sub-components. We illustrate how an abstraction is refined with the verification of *Property 1* on the Sensor-to-Network component. (Space limitations prohibit showing the formal specifications of the properties given hereafter.)

PROPERTY 1. *The Sensor-to-Network component transmits any hardware sensor reading exactly once.*

An abstraction of the Sensor-to-Network component for checking *Property 1* was constructed. Model checking of *Property 1* on the abstraction returned false. By analyzing the error trace from COSPAN, we observed that the abstraction is too abstract to enable model checking of *Property 1* and has to be refined. To refine the abstraction, we introduced and checked *Property 2* on the Network component.

PROPERTY 2. *The Network component transmits any of its inputs exactly once assuming that a new input arrives only after it outputs a Sent message to indicate its last input has been successfully transmitted.*

Property 2 was successfully verified on the Network component, but it was not enabled in the composition of the Sensor-to-Network component. To enable the property on the Network component, we introduced and verified *Property 3* on the Sensor component.

PROPERTY 3. *The Sensor component outputs any hardware sensor reading exactly once and after an output, it will not output again until after a message of the type, Done, is received.*

The verification of *Property 3* on the sensor component returned false due to a bug of the Sensor component. In the Sensor component, each time a hardware sensor reading is put in the output buffer, a thin thread [16] is created to output the data. There is a flag that should be set when a sensor reading has been output and a *Done* message has not been received. However, the thin thread fails to set the flag correctly. When the physical sensor outruns the physical network, since the output flag is not set, the sensor component may output again before it receives the *Done* message for its last output. This violates the second assertion in *Property 3*.

The bug was corrected and all properties of the Sensor component, including *Property 3*, were re-verified. A new Sensor-to-Network component was composed from the corrected Sensor component and the Network component. An abstraction of the newly composed component was constructed, on which *Property 1* was successfully verified.

Remarks: In this example, the verification of *Property 1* on the Sensor-to-Network component requires introducing and verifying additional properties of its sub-components, which is only for the purpose of demonstrating abstraction refinement and does not indicate that we frequently need to introduce and verify additional

properties of a component. Since our approach targets product lines of software systems, careful domain analysis and a few reuses often could lead to a quite stable set of properties for a component.

5. ANALYSIS OF CASE STUDY

Application of our approach to integration of model checking into CBD to the TinyOS components leads to the detection of a coordination error which is related to component composition, and a significant reduction in model checking complexity.

5.1 Detection of Coordination Error

By model checking of the "repeated transmission" property and *Property 1* on the Sensor-to-Network component, we have detected a coordination error as described in Section 4.4. This error would be hard to detect with conventional testing methods such as test-case based testing.

5.2 Model Checking Complexity Reduction

Direct verification of a property on a composed component with model checking is often infeasible due to state space explosions. In our approach, model checking of a property on a composed component is reduced to three sub-tasks: model checking of the properties of the sub-components, construction and refinement of an abstraction of the component, and model checking of the property on the abstraction. Complexities of these sub-tasks are often significantly lower than the complexity of directly model checking the property on the component. Verified properties of the sub-components can often be reused. The complexity of model checking a newly introduced property of a sub-component is lower since the sub-component has a smaller state space, and may be further reduced if the sub-component is also a composed component. Since the abstraction construction usually only involves a few environment assumptions and verified sub-component properties, it often has a modest complexity. Although the abstraction refinement may require user interactions, it is expected to be facilitated by domain-specific knowledge. An abstraction of a component only captures the aspect of the component required for verification of a specific property and usually consists of a few simple automata, therefore the verification of the property on the abstraction finishes fairly fast.

We illustrate the reduction attained in our approach on model checking complexity with the statistics from the TinyOS case study. Table 1 shows four model checking runs for verifying the "Repeated transmission" property on the Sensor-to-Network component. Run 1 checks the property on the composed component directly for comparison purposes. Run 2 (or Run 3, respectively) checks the properties in Figure 4 (or Figure 6) on the Sensor (or Network) component. Run 4 checks the "Repeatedly transmission" property on the abstraction of the Sensor-to-Network component. The complexities for model checking the sub-components and the abstraction are an order-of-magnitude lower than the complexity of directly checking the composed component. Furthermore, the verification results for the Sensor and Network components were reused from previous studies. The statistics shown in Table 1 only involves one level of composition. In a multi-level composition, this approach can model check higher level composed components that cannot be directly model checked due to state space explosions.

Run	Component	Time	Memory
1	Sensor-to-Network	89m15.45s	208.48M
2	Sensor	10m41.01s	33.673M
3	Network	18.0s	6.8239M
4	Abstraction of SN	0.1s	0.1638M

Table 1: Verification Complexity Comparison

6. RELATED WORK

There has been extensive research [21, 2, 1, 19, 3, 11] on compositional reasoning in the formal methods community. Most of the prior work applies compositional reasoning in a top-down approach: To check properties of a large system, the system is decomposed into modules recursively in a top-down fashion. Our research is based on the prior work, but combines the top-down approach with the bottom-up component composition process of CBD. Properties of components are verified as they are composed from simpler components in a bottom-up fashion and verification of these properties is based on compositional reasoning.

A closely related work to our research is Compositional Reachability Analysis (CRA) by Graf and Steffen [14], Yeh and Young [28], Cheung and Kramer [5, 6, 7, 4, 8], et. al. CRA analyzes a system or its modules in the context of the system. Modules are represented by Labeled Transition Systems (LTSs) or similar compositional representations of state space graphs. It is assumed that there exist LTSs of the lowest level modules. The LTS of a higher level module is composed from LTSs of its sub-modules and is minimized according to the property to be checked and context constraints. Property decomposition has not yet been supported in CRA, therefore a property involving multiple modules may lead to a complex global LTS. Another related work is Modular Feature Verification by Fisler and Krishnamurthi [12], which targets systems developed by Feature-Oriented Programming (FOP). In FOP, components are features that are orthogonal to the component concepts used in this paper. Our approach supports both component verification in the context of a system, and component verification with only environment assumptions and without a specific composition context. A component is represented by a set of verified temporal properties. Properties of a primitive component are obtained by directly model checking its original representation that can be in any model-checkable languages such as model-checkable subsets of UML, SDL, JAVA, or C/C++. Properties of a composed component are obtained by model checking its abstractions constructed from its environment assumptions and verified properties of its sub-components. Our approach also supports property decomposition by applying top-down compositional reasoning.

There is also related research on automatic generation of assumptions in compositional reasoning, such as [13]. [13] proposes an approach to automatic generation of assumptions for safety properties in the context of representing systems and their components using LTSs. In this approach, an automatically generated assumption can be quite complex and hard to check on other components.

7. CONCLUSIONS AND FUTURE WORK

An approach to integration of model checking into the CBD of software systems has been presented. The case study on TinyOS demonstrates the applicability of this approach, the detection of a coordination error, and a significant reduction in model checking complexity. The reduction in model checking complexity seems scalable. This approach can be readily applied on many software computation models.

The next steps for the research are to execute further and more detailed case studies on several software product lines and to automate formulation of component properties and environment assumptions as possible. A more in-depth case study with TinyOS is in progress. Two directions for automatic property and assump-

tion formulation are application of domain-specific heuristics and application of theorem proving to help derive the properties of a composed component from the properties of its sub-components.

Acknowledgment

We gratefully acknowledge contributions of Robert P. Kurshan to this research. We also acknowledge Don Batory, Thomas Graser, and Dewayne Perry for their valuable suggestions. We also thank the anonymous referees for their valuable suggestions.

8. REFERENCES

[1] M. Abadi and L. Lamport. Conjoining specifications. In *Proc. of ACM TOPLAS*, 1995.

[2] R. Alur and T. Henzinger. Reactive modules. In *Proc. of IEEE LICS*, 1996.

[3] N Amla, E. A. Emerson, K. S. Namjoshi, and R. Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In *Proc. of TACAS*, volume 2031 of *LNCS*, 2001.

[4] Shing Chi Cheung, Dimitra Giannakopoulou, and Jeff Kramer. Verification of liveness properties using compositional reachability analysis. In *Proc. of ESEC / SIGSOFT FSE*, 1997.

[5] Shing Chi Cheung and Jeff Kramer. Enhancing compositional reachability analysis with context constraints. In *Proc. of SIGSOFT FSE*, 1993.

[6] Shing Chi Cheung and Jeff Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proc. of SIGSOFT FSE*, 1995.

[7] Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM TOSEM 5(4)*, 1996.

[8] Shing Chi Cheung and Jeff Kramer. Checking safety properties using compositional reachability analysis. *ACM TOSEM 8(1)*, 1999.

[9] 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.

[10] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[11] W. P. de Rover, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods*. Cambridge Univ. Press, 2001.

[12] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. of SIGSOFT FSE*, 2001.

[13] D. Gannakopoulou, C. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of ASE*, 2002.

[14] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proc. of CAV*, volume 531 of *LNCS*, 1990.

[15] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. of CAV*, volume 1102 of *LNCS*, 1996.

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

[17] ITU. *ITU-T Recommendation Z.100 (03/93) - Specification and Description Language (SDL)*. ITU, 1993.

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

[19] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Cadence TR*, 1999.

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

[21] A. Pnueli. In transition from global to modular reasoning about programs. In *Proc. of Logics and Models of Concurrent Systems*. NATO ASI Series, 1985.

[22] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of Symposium on Programming*, 1982.

[23] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[24] F. Xie and J. C. Browne. Integrated state space reduction for model checking executable object-oriented software system designs. In *Proc. of FASE*, 2002.

[25] F. Xie, J. C. Browne, and R. P. Kurshan. Translation-based compositional reasoning for software systems. In *Proc. of FME*, 2003.

[26] F. Xie, V. Levin, and J. C. Browne. Model checking for an executable subset of UML. In *Proc. of ASE*, 2001.

[27] 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.

[28] W. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proc. of Symposium on Testing, Analysis, and Verification*, 1991.

APPENDIX

A Property Specification Language for xUML

An xUML level property specification language has been defined for model checking of xUML models. This language is presented in terms of a set of property templates that have intuitive meanings and also have rigorous mappings to a property specification language written in S/R, the native language of the COSPAN model checker. An example of one such template is

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

where the *enabling* condition e and the *discharging* condition d are propositional logic predicates over semantic entities of an xUML model. 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.)

Our property specification language is linear time, with the expressiveness of ω-automata. The templates define parameterized automata. Additional templates could be formulated in terms of the given ones if doing so simplifies property specification. A property formulated in this language consists of declarations of propositional logic predicates over semantic entities of an xUML model and declarations of temporal predicates. A temporal predicate is declared by instantiating a property specification template: each argument of the template is replaced by a propositional logic expression composed from previously declared propositional predicates.