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
Email: {feixie, browne}@cs.utexas.edu Fax: +1 (512) 471-8885

Abstract

This paper presents an approach to integrating model checking into component-based development of software systems. This integration enables development of highly reliable component-based software systems and reduces the complexity of verifying these systems by utilizing their compositional structures. In this approach, 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 simpler components, temporal properties of the composed component are verified on an abstraction of the component. The abstraction is constructed from environment assumptions of the component and verified properties of its sub-components. This approach has been applied to improve reliability of run-time images of TinyOS [4], a component-based run-time environment for networked sensors. Results from an initial case study demonstrate the applicability of the integration, the improvement of reliability, and a significant reduction in model checking complexity.

1 Introduction

Component-based development (CBD), developing software systems through composition of components, is one of the most important technical initiatives in software engineering research. Testing is the most commonly used technique for validating software systems being built with CBD. Testing has the inherent test case coverage problem, which hinders development of highly reliable component-based software systems. Application of model checking to software is an important method for improving reliability of software systems. Model checking provides exhaustive state space coverage for the systems being checked. Model checking is particularly effective at detecting coordination errors which frequently result from component compositions and are notoriously difficult to detect by testing. However, model checking often cannot handle large-scale software systems due to state space explosion.

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 can reduce the state spaces model checkers have to handle.

This paper presents an approach to integrating model checking into the CBD of software systems, which contributes to solution of fundamental problems in both CBD and model checking:

- Development of 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 being built, temporal properties of the component are established, verified, and then packaged with the component.
- Selecting a component for reuse considers not only its functionality but also its temporal properties.
- Properties of a composed component are verified by reusing verified properties of its sub-components and applying compositional reasoning [2].

A general component model, which provides a framework for representing components and their properties and for composing components, is defined. 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 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 general component model can be instantiated in many different computation models upon which the syntax and semantics for components, properties, and component compositions are precisely defined. We have instantiated the general component model in an Asynchronous Interleaving Message-passing (AIM) model. In this instantiation, executable representations of components are specified in an executable object-oriented modeling language whose semantics conform to the AIM computation model, such as xUML [6]. This instantiation is of interest due to that although many component-based software systems are not developed on the AIM model, most of them can be readily transformed to systems based on the AIM model.

Components are categorized as *primitive* components (components built from "scratch" and not composed from other components) and *composed* components. Properties of primitive components are verified by directly model checking the object-oriented design models of the components using methods established in our previous work [9, 11]. Properties of a composed component, instead of being model checked on its executable design model, are checked on its *abstraction*. The abstraction is composed of simple automata corresponding to environment assumptions of the composed component and verified properties of its sub-components. A sub-component property is included in the abstraction if it is enabled in the composition and related to the properties to be checked according to cone-of-influence analysis. If the abstraction of the composed component is still too complex to be checked directly, compositional reasoning is applied to decompose the abstraction. If the abstraction is too abstract for verifying the desired properties, it is refined by verifying additional properties of the sub-components so that the necessary properties of the sub-components can be included into the abstraction.

This approach is founded on compositional reasoning [2]. There has been extensive research on compositional reasoning, most of which 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 work 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 approach and verifications of these properties involve compositional reasoning. Research on interface automata [1] is related to our work in that it also explores the compositional structures of software components. It differs from our work in that it focuses on checking interface compatibilities of software components while our work focuses on checking temporal properties of software components. Inscape [8] was pioneering research in use of extended interfaces to support effective composition of systems from components. Rapide [7] was an early project integrating formal methods (in their case, proofs of consistency with specifications) and component-based development of software systems.

Our approach is most suitable for application to a family of software systems that are built from a set of software components. We have identified two major application domains: Families of software systems based on a specific hardware/software architecture, such as the TinyOS [4] runtime environment, and families of distributed large-scale software systems based on component platforms such as CORBA, DCOM, and EJB.

Section 2 defines a general component model for verification and instantiates it on the AIM computation model. Section 3 discusses how to verify properties of components. Section 4 illustrates our approach with the TinyOS case study. Section 5 analyzes the effectiveness of our approach in the context of the case study. Section 6 discusses future work and concludes.

2 Component Model for Verification

Section 2.1 presents a general component model designed to enable component verification. The general component model is a template that can be instantiated on computation models to enhance them with component and component composition which enable verification of components. The computation models provide semantics and abstract syntax for the executable representations and the interfaces of components and provides semantics for component execution and interaction. Composition rules are also realized on the computation models. Section 2.2 sketches the AIM computation model. Section 2.3 instantiates 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, such as 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 defined on I and V, and verified on E. A temporal property in P is represented by a pair, $(p,\ A)$, where p is a temporal formula defined on I and V and A is a set of temporal formulas defined on I and V. The property, p, holds on C if temporal formulas in A hold on the environment of C (see next paragraph for the definition of the environment of a component). Temporal formulas in A are referred to as the environment assumptions of C for enabling p.

Under this model, an applicable system is a component. The environment that the system interacts with is also modeled as components. The set of components that a component interacts with is referred to as the environment of the component. The environment of a component varies as the component is reused in different compositions.

2.1.2 Component Composition

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

- E is constructed from E_0, ..., E_{n-1} by connecting $E_0, \ldots, E_{n-1}$ through their interfaces.
- I is derived from I_0, ..., 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, ..., 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, $(V,\ M,\ Q,\ I)$, where V is a set of variables, each of which is of a bounded type; M is an extended Moore state model; Q is an infinite FIFO message queue; I is an initial condition that consists of an initial value for each variable in V, an initial state for M, and the empty status for Q.

An extended Moore state model is a three-tuple, $(S,\ E,\ T)$, where S is a set of states, each of which has an associated state action; E is a set of message types (assuming that $E's$ of different processes are disjoint); T is a set of state transitions each of which is a three-tuple, $(r,\ t,\ m)$, where r and t are two states in S and m is a message type in E. A state action is a program segment composed from executable statements as follows:

- An empty statement;
- An assignment statement;
- A messaging statement that outputs a message;
- An if statement;
- 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 it just entered a state and has not yet executed the state action associated with the state or if one of its state transition is enabled by the first message in its message queue.
- At any given moment, exactly one *ready* process is non-deterministically scheduled to execute from among all the *ready* processes.
- When scheduled, a *ready* process executes an enabled state transition or an enabled state action in a run-to-completion fashion.

Processes in a system interact via messages. 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, T, defined in E of P_2, and the parameters required by T. When the state action is executed, a message of the type, T, with its parameters is put in the message queue of P_2.

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

- E is constructed from $E_0, \ldots, E_{n-1}$, by mapping output message types in S_0, ..., S_{n-1} to input message types in R_0, ..., R_{n-1}. If there is a mapping defined between an output message type, s, in S_i, $0 \leq i < n$, to 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.

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

  - 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, each 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, ..., 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 via messages. 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 an AIM process in the component or its recursively nested sub-components.

3 Verification of Components

This section discusses verification of components in the instantiation of the general component model on the AIM computation model. First, we introduce how AIM systems are 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.

3.1 Background: Verification of AIM Systems

There are many software design specification languages whose semantics conform to the AIM model, such as xUML [6], an executable dialect of UML, and SDL [5]. In our previous research [9, 11], we designed and implemented an approach to model checking system designs specified in xUML. This section briefly sketches this approach and discusses how compositional reasoning, which plays a key role in verifying components, is applied in this approach.

3.1.1 Model Checking xUML Specifications

The approach to model checking software system designs specified in xUML can be summarized as follows:

- A system design is specified in xUML as an executable model.
- A property to be checked on the design is specified in an xUML level logic.
- The xUML model and the property are automatically translated to a model and a query in the S/R [3] automaton language.
- The S/R query is checked on the S/R model by the COSPAN [3] model checker.
- If the query fails, an error track is generated by COSPAN and is automatically translated to an error report in the name space of the xUML model.

The S/R automaton language employs synchronous parallel execution semantics and shared-variable communication paradigm. The AIM computation model is simulated in the synchronous parallel variable-sharing computation model of S/R. This approach requires a system design to be model checked be a closed system. A system is made closed by modeling its environment as part of the system.

3.1.2 Compositional Reasoning

The above approach suffers from the state space explosion problem. To verify large-scale software system designs, we extend the approach with compositional reasoning [2], 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. We apply compositional reasoning in model checking xUML specifications by following the Multi-Semantics Compositional Reasoning approach proposed in [10], 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 be formulated. Properties are mainly derived from functional specifications of the component such as input and output relationships. Additional properties may also be introduced incrementally when the component is reused in composing other components. Verifying a property of a composed component may require formulating and verifying additional properties of its sub-components.

3.3 Verification of Primitive Components

A primitive component often has specific 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 3.1.1 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, $(p,\ A) \in P$, in order to check whether p holds on E assuming assumptions in A 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 automaton corresponding to ENV in the S/R model obtained in Step 2 so that these variables will take on any value in their domains nondeterministically (Detailed discussion on freeing variables in an S/R model can be found in [3]);
4. Translate assumptions in A 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;
5. Translate p to an S/R automaton and check p on the S/R model gotten in Step 4.

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

3.4 Verification of Composed Components

This section presents a method for verifying a property, $(p,\ A^p) \in P$, 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 $C_0, \ldots, C_{n-1}$.

3.4.1 Component Abstraction Construction

Since the AIM specification of a composed component often has a large state space that cannot be directly model checked, we construct an abstraction of the component based on the composition, the environment assumptions of the component, and the messaging interfaces and the verified properties of the sub-components.

Before discussing how to construct the abstraction, we first introduce 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.

Definition – Enabled Property: *A property $(p_i,\ A_i)$ of C_i, where $i \in \{0,\ 1\}$ and $(p_i,\ A_i) \in P_i$, is enabled in the composition of C_0 and C_1 if and only if:*

- *Either A_i is empty;*
- *Or for each formula, q, in A_i, either there exists a property, $(q,\ A')$, which is defined, verified, and enabled on P_{1-i}, or $q \in A^p$.*

That a property, $(p_i,\ A_i)$, of C_i is not currently enabled in the composition of C_0 and C_1 does not imply that q does not hold on C_i under the composition. $(p_i,\ A_i)$ can become enabled when all assumptions in A_i become enabled, which may require checking additional properties of C_0 and C_1. The function in Figure 1 can be applied to determine

```
boolean function enabled ( $A, i$ ) begin
  if ( empty( $A$ ) ) then return true;
  else
    while ( ! empty( $A$ ) ) do
      $q$ = Remove( $A$ );
      if ( $q \in A^p$ ) then continue;
      elseif ( $< q,\ A' > \in P_{1-i}$ ) then
        if ( enabled( $A', 1-i$ ) ) then continue;
        endif;
        return false;
      endif;
    endwhile;
    return true;
  endif;
end;
```

Figure 1. The "enabled" function

whether $(p_i,\ A_i)$ is enabled in the composition of C_0 and C_1, assuming that the environment assumptions of p in A^p hold on the environment of the composition.

The abstraction of the composed component, C, is derived according to the following steps:

- 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) so that output message types of C_0 and C_1 only appear when used to communicate with the environment of C.

- Create an AIM system, S, which consists of three stub AIM processes:
  - CP_0 (or CP_1, respectively), 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);
  - ENV, 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, $(\hat{p}, \hat{A})$, in P_0 and P_1, and include $\hat{p}$ into S if the function returns true;
- Include the temporal formulas in A^p into S;
- Run the cone-of-influence analysis on S to exclude properties and assumptions not related to p.

There often exist circular dependencies among the properties of sub-components. Validity of these circular dependencies has to be checked with compositional reasoning rules [10] that support such validity checks. Sub-component properties involved in invalid circular dependencies are excluded from the abstraction.

3.4.2 Verification of Component Abstraction

Instead of checking p on the AIM specification of C, we check p on the abstraction, S:

- Translate S and p into S/R by applying the approach described in Section 3.1.1;
- Free all variables of the automata corresponding to CP_0, CP_1, and ENV in the S/R model;
- Check the S/R query corresponding to p on the S/R model corresponding to S;
- Include (p, A^p) in P if p holds on S; Otherwise, refine S as discussed in Section 3.4.3.

The complexity of model checking p on the abstraction, S, is often much lower than the complexity of directly model checking p on the AIM specification of C.

3.4.3 Refinement of Component Abstraction

If p does not hold on the abstraction, S, then either p does not hold on C assuming assumptions in A^p hold on the environment of C or S is too abstract. With the help of domain specific knowledge, it is often possible to differentiate the two cases by analyzing the error tracks generated by the model checker. If p does not hold on C under the assumptions in A^p, then either C has to be re-composed or more assumptions have to be added to A^p. If S 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 yet 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 procedure enables the property, $(p,\ A)$, of C_i

```
procedure enable ( $A$, $i$ ) begin
  while ( !empty( $A$ ) ) do
    $q$ = remove( $A$ );
    if ( $q \in A^p$ ) then continue;
    elseif ( $< q,\ A' > \in P_{1-i}$ ) then
      if ( ! enabled ( $A'$, $1 - i$ ) ) then
        enable ( $A'$, $1 - i$ );
      endif;
    elseif ( $q$ is supposed to hold on $C_{1-i}$ ) then
      $A'$ = { assumptions of $q$ };
      if ( ! verify ( $q$, $A'$, $1 - i$ ) ) then abort; endif;
      enable ( $A'$, $1 - i$ );
    else abort;
    endif;
  endwhile;
end;
```

Figure 2. The "enable" procedure

by enabling all its assumptions on C_{1-i}. If an assumption, q, is a property that has been verified in P_{1-i}, but is not enabled, the "enable" procedure is called for q recursively. If q has not been verified in C_{1-i}, 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 A' recursively. The recursive execution of the "enable" procedure is aborted 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 supposed to hold on C_{1-i}. Circular dependencies among properties, introduced by the refinement, must be validated as discussed in Section 3.4.1.

4 Case Study: Verification of TinyOS Components

We illustrate the approach to integrating model checking into CBD by applying it to improve the reliability of TinyOS [4] run-time images. TinyOS is a component-based run-time environment 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 functional modules into a single run-time image and different requirements of these systems require different configurations of TinyOS modules, which make CBD an appropriate development approach for TinyOS.

TinyOS run-time images are usually loaded to a large number of deeply embedded systems such as networked sensors, which makes correction of software bugs very expensive. Hardware constraints of TinyOS prohibit using locks and monitors which are computationally expensive. This combination of complexity and the requirement for high reliability justifies the application of model checking to improve the reliability 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 Sensor

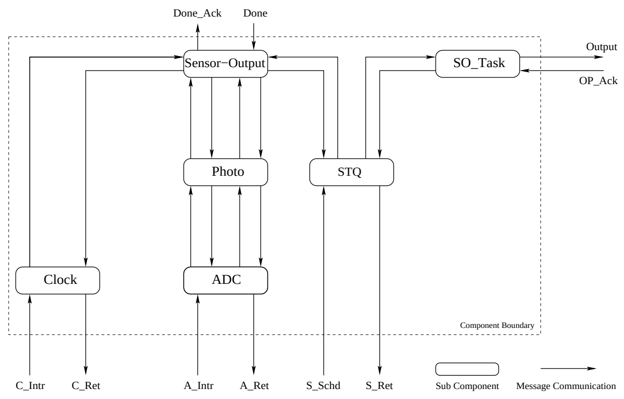

Figure 3. Sensor component

component consists of six AIM processes that interact with each other and the environment of the component via 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 needs to handle and *C_Ret*, *A_Ret*, and *S_Ret* are the corresponding replies. The Sensor component outputs Sensor readings as messages of the type, *Output*, and gets messages of the type, *OP_Ack*, back as the replies. The properties to be checked on the Sensor component are listed in Figure 4 with their assumptions. These properties assert that the component repeatedly

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

outputs sensor readings and correctly handles the signal-and-reply relationship between *Output* and *OP_Ack* and between *Done* and *Done_Ack* assuming 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.

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 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 and included in P are listed in Figure 6 with their assumptions. The properties assert 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

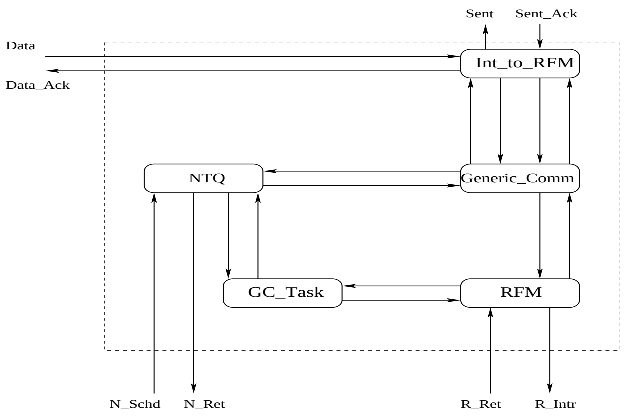

Figure 5. Network component

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

component consists of two variables, *RFM.Pending* and *NTQ.Empty*, referenced by the properties and the assumptions listed in Figure 6.

4.3 Sensor-to-Network Component

This section introduces how a run-time image of TinyOS, the Sensor-to-Network component, is composed from the Sensor component and the Network component, and then 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 and Network components. 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 (Output message type)" denotes the replacement of an output message type of a component with 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*}.

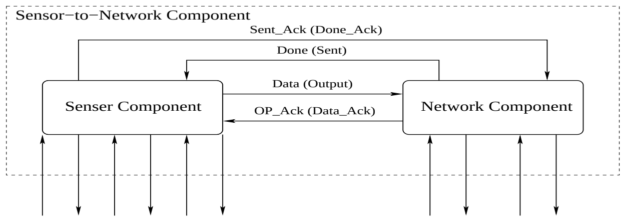

Figure 7. Sensor-to-Network component

The properties to be checked on the Sensor-to-Network component are listed in Figure 8 with their assumptions. These properties assert that the Sensor-to-Network component repeatedly transmits on the physical network if the assumptions hold on its environment. The set, V, of the Sensor-to-Network component consists of four variables, *ADC.Pending*, *STQ.Empty*, *RFM.Pending*, and *NTQ.Empty*, which are referenced by the properties and the assumptions listed in Figure 8.

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

In order to check the properties of the Sensor-to-Network component, we construct an abstraction of the component following the steps given in Section 3.4.1:

- Replace the output message types of the Sensor (or Network, respectively) components with the corresponding input message types of the Network (or Sensor) components as shown in Figure 7;

- Create an AIM system, SN, which consists of three stub AIM processes:
  - 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 $Empty$;
  - NP, whose input message types are *N_Schd*, *R_Intr*, *Sent_Ack*, and *Data*, whose state model outputs messages of types *N_Ret*, *R_Ret*, *Done*, and *OP_Ack*, and whose variables are $Pending$ and $Empty$;
  - 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 include the properties in Figure 9 into the abstraction.

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

IfRepeatedly (Data) **Repeatedly** (NP.Pending);
IfRepeatedly (Data) **Repeatedly** (Not NP.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

We then check the properties in Figure 8 on the AIM system, SN, by following the steps given in Section 3.4.2. It is easy to observe that the properties hold on the abstraction under the assumptions in Figure 8. Therefore, we can conclude that the properties also hold on the executable representation of the Sensor-to-Network component under the given assumptions.

4.4 Verification through Abstraction Refinement

An abstraction of a composed component may be refined by establishing, 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 hence after.

Property 1 *The Sensor-to-Network component never transmits any hardware Sensor reading duplicately.*

An abstraction of the Sensor-to-Network component for checking *Property 1* is constructed. Model checking of *Property 1* on the abstraction returns false. By analyzing the error trace from the model checker, we observe that the abstraction is too abstract for verifying *Property 1* and has to be refined. To refine the abstraction, we introduce and check *Property 2* on the Network component.

Property 2 *The Network component never transmits any of its input duplicately assuming that its next input only arrives until after it outputs a Sent message to indicate its last input has been successfully transmitted.*

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

Property 3 *The Sensor component never outputs any hardware reading duplicately and never outputs again until after a message of the type, Done, is received.*

The verification of *Property 3* returns false due to a bug of the Sensor component. In the Sensor component, each time a Sensor reading is put in the output buffer, a thin thread [4] 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, the sensor component may output again before it receives the *Done* message for its last output.

This bug in the Sensor component is corrected. All properties of the Sensor component, including *Property 3*, are re-verified on the corrected Sensor component. A new Sensor-to-Network component is composed from the corrected Sensor component and the Network component. An abstraction of the newly composed component is constructed and *Property 1* is successfully verified on the abstraction.

5 Analysis of Case Study

Application of our approach for integrating model checking into the CBD of software systems to TinyOS components demonstrated both detection of a coordination error, which is resulting from component composition and is hard to formulate and detect with testing, and reduction of model checking complexity.

5.1 Detection of Coordination Error

Model checking of the "repeated output" property and the "non-duplication" property on the Sensor-to-Network component detected a coordination error as described in Section 4.4. Complete simulation test of these properties is not feasible due to the fact that it is not possible to construct test cases of infinite length.

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 explosion. In our approach, the verification is reduced into three sub-tasks: model checking of the properties of the sub-components composing the component, construction and refinement of the 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 composed component. Furthermore, verification of the properties of the sub-components can often be reused from previous efforts. Even if the properties of the sub-components are newly introduced and need to be model checked, the complexity is lower due to the reduced state spaces and can often be further reduced if the sub-components are composed components. The procedures for constructing component abstractions run much faster than model checking procedures. Although abstraction refinement involves user interactions, it is often facilitated by domain-specific knowledge. An abstraction of a component often only captures an aspect of the component and consists of several simple automata of 2-4 states, which makes verifications on the abstraction run 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 "Repeatedly Output" 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 listed in Figure 4 (or Figure 6) on the Sensor (or Network) component. Run 4 checks the "Repeatedly Output" 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 model checking the composed component. Furthermore, the verification results for the Sensor and Network components are 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 explosion.

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 Conclusions and Future Work

This paper defines, discusses, and illustrates an approach to integrating model checking into the CBD of software systems. In this approach, model checking improves reliability of software systems constructed with CBD and compositional structures of these systems, introduced by CBD, significantly reduce model checking complexity. This approach can be readily instantiated and applied on many software computation models.

Currently, our approach only considers integrated application of model checking in CBD. There are other formal software reliability methods such as theorem proving, which have potential in improving reliability of component-based software systems. For instance, theorem proving may help us derive the properties of a composed component from the properties of its sub-components in many cases.

Acknowledgment

We gratefully acknowledge the intellectual contribution of Robert P. Kurshan to this research. We also thanks Don Batory, Thomas Graser, and Dewayne E. Perry for their valuable suggestions. This research was partially supported by NSF grant 010-3725.

References

[1] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, and F. Mang. Interface Compatibility Checking for Software Modules. *Proc. of CAV'02*, 2002.

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

[3] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. *Proc. of CAV'96*, 1996.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. *Proc. of ASPLOS-IX*, 2000.

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

[6] Kennedy Carter. *http://www.kc.com/html/xuml.html*. Kennedy Carter, 2001.

[7] D. C. Luckham, J. J. Kenney, and et al. Specification and Analysis of System Architecture Using Rapide. *IEEE Transaction on Software Engineering*, 21(4), 1995.

[8] D. E. Perry. The Inscape Environment. *Proc. of ICSE'89*, 1989.

[9] F. Xie and J. C. Browne. Integrated State Space Reduction for Model Checking Executable Object-oriented Software System Designs. *Proc. of FASE 2002*, 2002.

[10] F. Xie and J. C. Browne. Multi-Semantics Compositional Reasoning for Software Systems. *To appear in Proc. of ISSRE 2002*, 2002.

[11] F. Xie, V. Levin, and J. C. Browne. ObjectCheck: A Model Checking Tool for Executable Object-oriented Software System Designs. *Proc. of FASE 2002*, 2002.