CTL Model-checking for Systems with Unspecified Components[*]

[Extended Abstract]

Gaoyan Xie and Zhe Dang
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA
{gxie,zdang}@eecs.wsu.edu

ABSTRACT

In this paper, we study a CTL model-checking problem for systems with unspecified components, which is crucial to the quality assurance of component-based systems. We introduce a new approach (called *model-checking driven black-box testing*) that combines model-checking with traditional black-box software testing to tackle the problem in an automatic way. The idea is, with respect to some requirement (expressed in a CTL formula) about the system, to use model-checking techniques to derive a condition (expressed in terms of *witness graphs*) for an unspecified component such that the system satisfies the requirement iff the condition is satisfied by the component. The condition's satisfiability can be established by testing the component. Test sequences are generated on-the-fly by traversing the witness graphs with a bounded depth. With a properly chosen bound, a complete and sound algorithm is immediate.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model-checking*; D.2.5 [**Software Engineering**]: Testing/Debugging—*Black-box testing*; F.4.1 [**Mathematic Logic and Formal Languages**]: Mathematical Logic—*Temporal Logic*

General Terms

Verification, Component-based systems

Keywords

Component-based systems, Model-checking, Black-box testing

1. INTRODUCTION

Although component-based software development [22, 6] enjoys the great benefits of reusing valuable software assets, reducing development costs, improving productivity, etc., it also poses serious challenges to the quality assurance problem [3, 27] of component-based systems. This is because prefabricated components could be a new source of system failures. In this paper, we are interested in one problem that system developers often face:

(*) *how to ensure that a component whose design detail and source code are unavailable will function correctly in a host system.*

This is a rather challenging problem and yet to be handled in a satisfying way by available techniques. For instance, in practice, testing is almost the most natural resort to solve the problem. When integrating a component into a system, system developers may either choose to thoroughly test the component separately or to hook the component with the system and conduct integration testing. However, software components are generally built with multiple sets of functionality [17], and indiscriminately testing all the functionality of a software component separately is not only expensive but also infeasible. Also, integration testing is often not applicable in applications where software components are used for dynamic upgrading or extending a running system [32] that is too costly or not supposed to shut down for testing at all. Even without the above limitations, purely testing techniques are still considered to be insufficient to solve the problem for mission-critical or safety-critical systems where formal methods like model-checking are highly desirable. But, it is very common that design details and source code of an externally obtained component are not available to the developers of its host system. This makes existing formal verification techniques (like model-checking) not directly applicable to these cases.

In this paper, we study how to extend CTL model-checking techniques to solve the problem in (*). Specifically, we consider systems with only one such unspecified component. Denote such a system as $Sys = \langle M, X\rangle$, where M is the host system and X is an unspecified component. Both M and X are finite-state transition systems (the actual specification of X is unknown), which communicate with each other by synchronizing a finite set of given input and output symbols. Then our problem can be further formulated as to check whether $\langle M, X\rangle \models f$ holds, where f is a CTL formula specifying some requirement for Sys.

Our approach to solve the above model-checking problem is a combination of both model-checking and traditional black-box testing techniques (called *model-checking driven black-box testing*). First, a model-checking procedure is used to derive from M and f a condition P over the unspecified components X. The condition P guarantees that the system Sys satisfies the requirement f iff P is satisfied by X. The satisfiability of the condition P over the unspecified component X is then checked through adequate black-

[*]The research was supported in part by NSF Grant CCF-0430531.

box testing on X with test-cases generated automatically from P. Our study shows that the obtained condition P is in the form of a hierarchy of communication graphs (called a witness graph), each of which is a subgraph of M. Test-cases can be generated by traversing witness graphs when the unspecified component X is a finite or infinite state system. In particular, when X is a finite state system (m is an upper bound of its state number), our study shows that traversing the witness graphs up to a depth bounded by $O(k \cdot n \cdot m^2)$ is sufficient to answer the model-checking query, where k is the number of CTL operators in the formula f and n is the state number in the host system M. Thus, in this case, with a properly chosen search depth, a complete and sound solution is immediate.

The advantages of our approach are obvious: a stronger confidence about the reliability of the system can be established through both model-checking and adequate functional testing; system developers can customize the testing of a component with respect to some specific system properties; intermediate model-checking results (the witness graphs) for a component can be reused to avoid (repetitive) integration testing when the component is updated, if only the new component's interface remains the same; the whole process can be carried out in an automatic way.

The rest of this paper is organized as follows. Section 2 defines the system model and introduces some background on black-box testing. Section 3 presents algorithms for deriving the condition as well as testing the condition over the unspecified component. Section 4 compares our research with some related work. Section 5 concludes the paper with discussions on issues to be solved in the future.

Due to space limit, details of most algorithms are omitted in this extended abstract. Readers can find the full version of this paper at `http://www.eecs.wsu.edu/∼gxie/`.

2. PRELIMINARIES

2.1 The System Model

In this paper, we consider systems with only one unspecified component, which is denoted by

$$Sys = \langle M, X \rangle,$$

where M is the host system and X is the unspecified component. Both M and X are finite-state transition systems communicating synchronously with each other via a finite set of input and output symbols.

Formally, the unspecified component X is viewed as a deterministic Mealy machine whose internal structure is unknown (but an implementation of X is available for testing). We write X as a triple $\langle \Sigma, \nabla, m \rangle$, where Σ is the set of X's input symbols, ∇ is the set of X's output symbols, and m is an upper bound for the number of states in X (the m is given). Upon receiving an input symbol, X may perform some internal actions, move to a new state, and then send back an output symbol immediately. Assume that X has an initial state s_{init} and X is always in this initial state when the system starts to run. A *run* of X is a sequence of alternating symbols in Σ and ∇: $\alpha_0\beta_0\alpha_1\beta_1...$, such that, starting from the initial state s_{init}, X outputs exactly the sequence $\beta_0\beta_1...$ when it is given the sequence $\alpha_0\alpha_1...$ as input. In this sense, we say that the input sequence is accepted by X.

The host system M is defined as a 5-tuple

$$\langle S, \Gamma, R_{env}, R_{comm}, I \rangle$$

where

- S is a finite set of states;
- Γ is a finite set of events;
- $R_{env} \subseteq S \times \Gamma \times S$ defines a set of *environment transitions*, where $(s, a, s') \in R_{env}$ means that M moves from state s to state s' upon receiving an event (symbol) $a \in \Gamma$ from the outside environment;
- $R_{comm} \subseteq S \times \Sigma \times \nabla \times S$ defines a set of *communication transitions* where $(s, \alpha, \beta, s') \in R_{comm}$ means that M moves from state s to state s' when X outputs a symbol $\beta \in \nabla$ after M sends X an input symbol $\alpha \in \Sigma$; and,
- $I \subseteq S$ defines M's initial states.

Without loss of generality, we further assume that, there is only one transition between any two states in M (but in general, M could still be nondeterministic).

An *execution path* of the system $Sys = \langle M, X \rangle$ is a (potentially infinite) sequence τ of states and symbols, $s_0c_0s_1c_1...$, where each $s_i \in S$, each c_i is either a symbol in Γ or a pair $\alpha_i\beta_i$ (called a *communication*) with $\alpha_i \in \Sigma$ and $\beta_i \in \nabla$. Additionally, τ satisfies the following requirements:

- s_0 is an initial state of M, i.e., $s_0 \in I$;
- for each $c_i \in \Gamma$, (s_i, c_i, s_{i+1}) is an environment transition of M;
- for each $c_i = \alpha_i\beta_i$, $(s_i, \alpha_i, \beta_i, s_{i+1})$ is a communication transition of M.

The *communication trace* of τ, denoted by τ_X, is the sequence obtained from τ by retaining only symbols in Σ and ∇ (i.e., the result of projecting τ onto Σ and ∇). For any given state $s \in S$, we say that the system Sys can *reach* s iff Sys has an execution path τ on which s appears and τ_X (if not empty) is also a run of X. In the case when X is fully specified, the system can be modeled as an I/O automaton [26] (which is not input-enabled) or as two interface-automata [10]. In the latter case, the state number m can also be considered as the number of states in an interface automaton of X (instead of the state number in X itself).

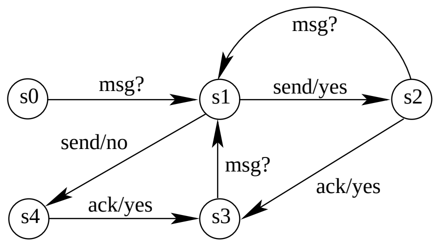

Figure 1: An example system

As an illustrating example, consider a system $Sys = \langle M, X \rangle$ where the host system M keeps receiving messages from the outside environment and then sends the message through the unspecified component X. The only event symbol in M is msg, while X has two input symbols $send$ and ack that make X send a message and ask X for an acknowledge respectively. X also has two output symbols yes and no that indicate whether the internal actions related with a previous input symbol succeeded or not (i.e., whether a message is sent and whether an ack is available). The transition graph of M is depicted in Figure 1 where we use a suffix ? to denote events from the outside environment (e.g., msg?), and use an infix / to denote communications of M with X (e.g., $send/yes$).

2.2 Black-box Testing

Black-box testing is a technique to test a system without knowing its internal structure. A system is regarded as a "black-box" in the sense that its behaviors can only be determined by observing its implementation's the input/output sequences, and a test over a black-box is simply to run its implementation with a given input sequence.

Studies [33] have shown that if only an upper bound for the number of states in the system and the system's input/output symbols are known, then its (equivalent) internal structure can be recovered through black-box testing. Clearly, a naive solution to the CTL model-checking problem over the system Sys is to first recover the full structure of the component X through black-box testing, and then solve the classic model-checking problem over the fully specified system composed from M and the recovered X. Notice that, in the naive solution, when black-box testing is performed over X, the selected test sequences have nothing to do with the host system M. Therefore, it is desirable to find more sophisticated solutions such as the algorithms introduced in below, which only select "useful" test sequences w.r.t. the M as well as its temporal requirement.

The unspecified component X in this paper can be treated as a black-box. And as a common practice in black-box testing, X is assumed to always have a special input symbol $reset$ which always makes it return to the initial state s_{init} regardless of its current state. Throughout this paper, we use $Experiment$ to denote a test over the unspecified component X. Specifically, we use $Experiment(X, reset\pi)$ to denote the output sequence obtained by testing X with the input sequence $reset\pi$ (i.e., run X from its initial state with input sequence π). Suppose after testing X with the input sequence $reset\pi$, we continue to run X by feeding it with an input symbol α. Corresponding to this α, we may obtain an output symbol β from X, and we use $Experiment(X, \alpha)$ to denote this β. Notice that this $Experiment(X, \alpha)$ is actually a shorthand for "the last output symbol in $Experiment(X, reset\pi\alpha)$".

3. CTL MODEL-CHECKING DRIVEN BLACK-BOX TESTING

In this section, we introduce algorithms for CTL model-checking driven black-box testing for the system $Sys = \langle M, X\rangle$.

3.1 Ideas

Recall that the CTL model-checking problem is, for a Kripke structure $K = (S, R, L)$, a state $s_0 \in S$, and a CTL formula f, to check whether $K, s_0 \models f$ holds. The standard algorithm [9] to solve this problem operates by exhaustively searching the structure and, during the search, labeling each state s with the set of subformulas of f that are true at s. Initially, labels of s are just $L(s)$. Then, the algorithm goes through a series of stages—during the i-th stage, subformulas with the $(i-1)$-nested CTL operators are processed. When a subformula is processed, it is added to the labels of each state where the subformula is true. When all the stages are completed, the algorithm returns $true$ when s_0 is labeled with f, or $false$ otherwise.

However, for a system like Sys that contains an unspecified component, the standard algorithm does not work, since transitions of the host system M may depend on communications with the unspecified component X, which cannot be statically resolved. For instance, for the system depicted in Figure 1, a simple check like $\langle M, X\rangle, s_1 \models EX\ s_2$ (i.e., whether s_2 is reachable from s_1) cannot be done by the standard algorithm. In this section, we adapt the standard CTL model-checking algorithm [9] to handle systems like Sys; i.e., to check whether

$$\langle M, X\rangle, s_0 \models f \tag{1}$$

holds, where s_0 is an initial state in M and f is a CTL formula.

Our new algorithm follows a similar structure to the standard one. It also goes through a series of stages to search M's state space and label each state during the search. The labeling of a state, however, is far more complicated when processing a subformula during each stage. The central idea of our algorithm can be summarized as follows. When the truth of a subformula h at a state s cannot be statically decided (due to communications), we construct some communication graph (called a *witness graph*, written as $[\![h]\!]$) by picking up all the communications that shall witness the truth of h at state s and then label s with the witness graph. The witness graph serves as a sufficient and necessary condition for h to be true at s, and this condition shall be later evaluated by testing the unspecified component X.

Actually, we do not have to construct one witness graph for every subformula at every state. Instead, we construct one witness graph only for a subformula h that contains a CTL operator, and this witness graph encodes all the witnesses (communications) to the truth of the formula at every state in M. Thus, totally we shall construct k witness graphs where k is the number of CTL operators in f, and we associate each witness graph with a unique ID number that ranges from 2 to $k+1$. Let $\mathcal{I}$ be the mapping from the witness graphs to their IDs; i.e., $\mathcal{I}([\![h]\!])$ denotes the ID number of h's witness graph, and $\mathcal{I}^{-1}(i)$ denotes the witness graph with i as its ID number, $2 \leq i \leq k+1$. Notice that the witness graph to different CTL operators shall be evaluated differently, so we call $[\![h]\!]$ as an *EX graph*, an *EU graph*, or an *EG graph* when h takes the form of $EX\ g$, $E[g_1\ U\ g_2]$, or $EG\ g$, respectively.

Specifically, we label a state s with 1 (resp. nothing) if h is true (resp. false) at s regardless of the communications between M and X. Otherwise, we shall label s with $i = \mathcal{I}([\![h]\!])$ when h takes the form of $EX\ g$, $E[g_1\ U\ g_2]$, or $EG\ g$, which means that h could be true at s and the truth would be witnessed by some (communication) paths starting from s in $\mathcal{I}([\![h]\!])$. When h takes the form of a Boolean combination of subformulas using $\neg$ and $\vee$, the truth of h at state s shall also be a logic combination of the truths of its component subformulas at the same state. To this end, we shall label s with an *ID expression* ψ defined as follows:

- $ID := 1 \mid 2 \mid \ldots \mid k+1$;
- $\psi := ID \mid \neg\psi \mid \psi \vee \psi$.

Let Ψ denote the set of all ID expressions. For each subformula h, in addition to the possible witness graph of h, we also construct a labeling (partial) function $L_h : S \rightarrow \Psi$ to record the ID expression labeled to each state during the processing of the subformula h. The labeling function is returned when the subformula is processed.

In summary, our new algorithm to solve the model-checking problem $\langle M, X\rangle, s_0 \models f$ can be sketched as follows:

```
Procedure $CheckCTL(M, X, s_0, f)$
    $L_f := ProcessCTL(M, f)$
    If $s_0$ is labeled by $L_f$ Then
        If $L_f(s_0) = 1$ Then
            Return $true$;
        Else
            Return $TestWG(X, reset, s_0, L_f(s_0))$;
    Else
        Return $false$.
```

In the above algorithm, a procedure $ProcessCTL$ (will be introduced in Section 3.2) is called to process all subformulas of f, and it returns a labeling function L_f for the outer-most subformula (i.e., f itself). The algorithm returns $true$ when s_0 is labeled with 1 by L_f or $false$ when s_0 is not labeled at all. In other cases, a procedure $TestWG$ (will be introduced in Section 3.3) is called to test whether the ID expression $L_f(s_0)$ could be evaluated true at s_0.

3.2 Process a CTL Formula

Processing a CTL formula h is implemented through a recursive procedure $ProcessCTL$. Recall that any CTL formula can be expressed in terms of $\vee$, $\neg$, EX, EU, and EG. Thus, at each intermediate step of the procedure, depending on whether the formula h is atomic or takes one of the following forms: $g_1 \vee g_2$, $\neg g$, $EX\ g$,$E[g_1\ U\ g_2]$, or $EG\ g$, the procedure has six cases to consider. When it finishes, a labeling function L_h is returned for the formula h.

3.2.1 Process atom

When h is an atomic formula, $ProcessCTL$ simply returns a function that labels each state where h is true with 1.

3.2.2 Process negation

When $h = \neg g$, we first process g by calling $ProcessCTL$, then construct a labeling function L_h for h by "negating" g's labeling function L_g as follows:

- For every state s that is not in the domain of L_g, let L_h label s with 1;
- For each state s that is in the domain of L_g but not labeled with 1 by L_g, let L_h label s with ID expression $\neg L_g(s)$.

3.2.3 Process union

When $h = g_1 \vee g_2$, we first process g_1 and g_2 respectively by calling $ProcessCTL$, then construct a labeling function L_h for h by "merging" g_1 and g_2's labeling functions L_{g_1} and L_{g_2} as follows:

- For each state s that is in both L_{g_1}'s domain and L_{g_2}'s domain, let L_h label s with 1 if either L_{g_1} or L_{g_2} labels s with 1 and label s with ID expression $L_{g_1}(s) \vee L_{g_2}(s)$ otherwise;
- For each state s that is in L_{g_1}'s domain (resp. L_{g_2}'s domain) but not in L_{g_2}'s domain (resp. L_{g_1}'s domain), let L label s with $L_{g_1}(s)$ (resp. $L_{g_2}(s)$).

3.2.4 Process an EX subformula

When $h = EX\ g$, subformula g is processed first by recursively calling $ProcessCTL$. Then, the procedure $ProcessEX$ is called with g's labeling function L_g to create a witness graph for h and to construct a labeling function L_h.

In $ProcessEx$, the witness graph for $h = EX\ g$, called an EX graph, is created as a triple: $[\![h]\!] = \langle N, E, L_g\rangle$, where N is a set of nodes and E is a set of annotated edges. It is created as follows:

- Add one node to N for each state that is in the domain of L_g.
- Add one node to N for each state that has a successor in the domain of L_g.
- Add one edge between two nodes in N to E when M has a transition between two states corresponding to the two nodes respectively; if the transition involves a communication with X then annotate the edge with the communication symbols.

The labeling function L_h is constructed as follows. For each state s that has a successor s' in the domain of L_g, if s can reach s' through an environment transition and s' is labeled with 1 by L_g then let L_h also label s with 1, otherwise let L_h label s with $\mathcal{I}([\![h]\!])$.

3.2.5 Process an EU subformula

The case when $h = E[g_1\ U\ g_2]$ is more complicated. We first process g_1 and g_2 respectively by calling $ProcessCTL$, then call the procedure $ProcessEU$ with g_1 and g_2's labeling functions L_{g_1} and L_{g_2} to create a witness graph for h and to construct a labeling function L_h.

In $ProcessEU$, the witness graph for $h = E[g_1\ U\ g_2]$, called an EU graph, is created as a 4-tuple: $[\![h]\!] := \langle N, E, L_{g_1}, L_{g_2}\rangle$, where N is a set of nodes and E is a set of edges. N is constructed by adding one node for each state that is in the domain of L_h, while E is constructed in the same way as that of $ProcessEX$.

We construct the labeling function L_h recursively. First, let L_h label each state s in the domain of L_{g_2} with $L_{g_2}(s)$. Then, for state s that has a successor s' in the domain of L_h, if s (resp. s') is labeled with 1 by L_{g_1} (resp. L_h) and s can reach s' through an environment transition, then let L_h also label s with 1, otherwise let L_h label s with $\mathcal{I}([\![h]\!])$. Notice that, in the latter step, if a state s can be labeled with both 1 and $\mathcal{I}([\![h]\!])$, let L_h label s with 1. Thus, we can guarantee that the constructed L_h is indeed a function.

3.2.6 Process an EG subformula

To handle formula $h = EG\ g$, we first process g by calling $ProcessCTL$, then call the procedure $ProcessEG$ with g's labeling function L_g to create a witness graph for h and to construct a labeling function L_h.

In $ProcessEG$, the witness graph for h, called an EG graph, is created as a triple: $[\![h]\!] := \langle N, E, L_g\rangle$, where N is a set of nodes and E is a set of annotated edges. The graph is constructed in the same way as that of $ProcessEU$.

The labeling function L_h is constructed as follows. For each state s that can reach a loop C through a path p such that every state (including s) on p and C is in the domain of L_g, if every state (including s) on p and C is labeled with 1 by L_g and no communications are involved on the path and the loop, then let L_h also label s with 1, otherwise let L_h label s with $\mathcal{I}([\![h]\!])$.

3.3 Evaluate an ID Expression

As seen from the previous subsection, the $ProcessCTL$ procedure labels states with ID expressions for each subformula h, which are essentially conditions under which the subformula h is true at a state. Also, as seen in Section 3.1, the $CheckCTL$ procedure either gives a definite $true$ or $false$ answer to the CTL model-checking problem, i.e., $\langle M, X\rangle, s_0 \models f$, or it reduces the problem to checking whether the ID expression $\psi = L_f(s_0)$ can be evaluated true at state s_0. The evaluation is carried out by a recursive procedure $TestWG$, which is essentially a testing process.

According to the definition of an ID expression, $TestWG$ only needs to consider six cases. When the ID expression ψ is the value 1, $TestWG$ returns $true$; when $\psi = \neg\psi_1$, $TestWG$ returns $false$ (resp. $true$) if ψ_1 is evaluated true (resp. false) at s_0; when $\psi = \psi_1 \vee \psi_2$, $TestWG$ returns $true$ if either ψ_1 or ψ_2 can be evaluated true at s_0, and returns $false$ if neither can be evaluated true at s_0. The remaining three cases are when ψ represents an EX graph, an EU graph, or an EG graph. We discuss the evaluation for these three cases as follows.

3.3.1 Evaluate an EX graph

To check whether an EX graph $G = \langle N, E, L_g \rangle$ can be evaluated true at a state s_0 is simple. We just test whether the system M can reach from s_0 to another state $s' \in \mathbf{dom}(L_g)$ along one edge in G such that the ID expression $L_g(s')$ can be evaluated true at s'.

3.3.2 Evaluate an EU graph

To check whether an EU graph $G = \langle N, E, L_{g_1}, L_{g_2} \rangle$ can be evaluated true at a state s_0, we need to traverse all paths p in G with length less than mn,[1] and test the unspecified component X to see whether the system can reach some state $s' \in \mathbf{dom}(L_{g_2})$ through one of those paths. In here, m is the given upper bound for the number of states in the unspecified component X and n is the number of nodes in G. In the meantime, we should also check whether $L_{g_2}(s')$ can be evaluated true at s' and whether $L_{g_1}(s_i)$ can be evaluated true at s_i for each s_i on p (excluding s') by calling $TestWG$.

3.3.3 Evaluate an EG graph

To check whether an EG graph $G = \langle N, E, L_g \rangle$ can be evaluated true at a state s_0, we need to find an infinite path in G, along which the system can run forever. The following procedure $TestEG$ first decomposes G into a set of SCCs. Then, for each state s_f in the SCCs, it calls another procedure $SubTestEG$ to test whether the system can reach s_f from s_0 along a path not longer than mn,[1] as well as whether the system can further reach s_f from s_f for $m - 1$ times[1]. Here, m is the same as before while n is the number of nodes in G.

```
Procedure $TestEG(X, \pi, s_0, G = \langle N, E, L_g \rangle)$
    $SCC := \{C | C$ is a nontrivial SCC of $G\}$;
    $T := \bigcup_{C \in SCC} \{s | s \in C\}$;
    For each $s \in T$ Do
        $Experiment(X, reset\pi)$;
        If $SubTestEG(X, \pi, s_0, s, G, level = 0, count = 0)$;
            Return $true$;
    Return $false$.
```

The maximal length of the paths that the above evaluation process shall traverse depends on how many witness graphs are involved in an ID expression, the sizes of the witness graphs, and the number of states of the unspecified component. One can show that the maximal length is bounded by $O(k \cdot n \cdot m^2)$, where k is the number of CTL operators in the formula f, m is the upper bound for the number of states in the unspecified component X, and n is the number of states in the host system M.

3.4 Example

To better understand how our algorithms work, consider such a model-checking problem for the system depicted in Figure 1: starting from the initial state s_0, whenever the systems reaches state s_2, it would eventually reach s_3; i.e., the problem is to check whether $(M, X), s_0 \models AG(s_2 \rightarrow AFs_3)$ holds. Taking the negation of the original problem, we describe how the problem $(M, X), s_0 \models f$, where $f = E[true\ U(s_2 \wedge EG\neg s_3)]$ is solved by our algorithms.

Step 1. Atomic subformula s_2 (in f) is processed by *ProcessCTL*, which returns a labeling function $L_1 = \{(s_2, 1)\}$.

Step 2. Atomic subformula s_3 is processed by *ProcessCTL*, which returns a labeling function $L_2 = \{(s_3, 1)\}$.

Step 3. Subformula $\neg s_3$ is processed by $Negation$ (see Section 3.2.2), which returns a labeling function $L_3 = \{(s_0, 1), (s_1, 1), (s_2, 1), (s_4, 1)\}$.

[1]Since the unspecified component X is treated as a finite state transition system, these bounds can be easily obtained from a Cartesian product of M and X.

Step 4. Subformula $EG\neg s_s$ is processed by $ProcessEG$ (see Section 3.2.6), which constructs an EG graph $G_1 = \langle N, E, L_3 \rangle$ with an ID 2 (see Figure 2) and returns a labeling function $L_4 = \{(s_0, 2), (s_1, 2), (s_2, 2)\}$.

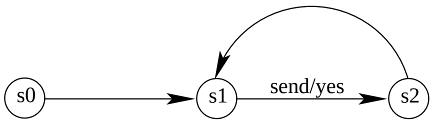

Figure 2: The witness graph for $EG\neg s_s$

Step 5. Subformula $s_2 \wedge EG\neg s_3$ is processed by $Negation$ and $Union$ (see Section 3.2.3), which return a labeling function $L_5 = \{(s_2, 2)\}$.

Step 6. Finally, the formula $E[true\ U(s_2 \wedge EG\neg s_3)]$ is processed by procedure $ProcessEU$ (see Section 3.2.5), which constructs an EU graph $G_2 = \langle N, E, L_{true}, L_5 \rangle$[2] with an ID 3 (see Figure 3) and returns a labeling function $L_f = \{(s_0, 3), (s_1, 3), (s_2, 3), (s_3, 3), (s_4, 3)\}$.

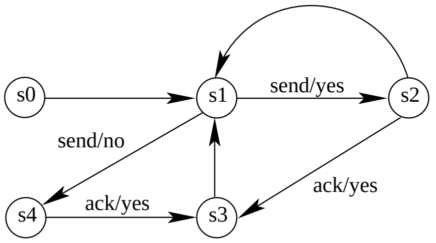

Figure 3: The witness graph for $E[true\ U(s_2 \wedge EG\neg s_3)]$

When $ProcessCTL$ finishes, s_0 is labeled by L_f with an ID expression 3 instead of 1 (i.e., $true$). This indicates that the original model-checking problem can not be statically decided and its truth depends on a condition that the ID expression 3 be evaluated true at s_0. Hence, procedure $TestWG$ must be called to test the condition as follows.

Step 7. The ID expression 3 is evaluated by $EvaluateEU$ since the witness graph with ID 3, G_2 constructed in **Step 6**, is an EU graph.

Step 8. $EvaluateEU$ traverses every path p of G_2 that is between s_0 and some state in the domain of L_5 (recall that L_5 is the fourth component of G_2 in **Step 6** and s_2 is the only state in its domain) to see:

- Whether the annotations (communication symbols) on p constitute a run of the unspecified component X. For instance, if $p = s_0 s_1 s_2 s_1 s_3 s_1 s_2$, then we need to test the "black-box" X with an input sequence "$send\ ack\ send$" to see whether the corresponding output sequence is "$yes\ yes\ yes$".

- Whether the ID expression $L_{true}(s)$ (recall that L_{true} is the third component of G_2 in **Step 6**) can be evaluated true at each state s along p. Obviously, that is true from the definition of L_{true}.

- Whether the ID expression $L_5(s_2) = 2$ can be evaluated true at s_2 by calling $EvaluateEG$ (since the witness graph with ID 2, G_1 constructed in **Step 4**, is an EG graph).

[2]L_{true} labels every state with 1 (i.e., $true$).

- $EvaluateEG$ tries to find in G_1 a loop C as well as a path p_1 from s_2 to C such that the annotations (communication symbols) on the concatenated path pp_1C constitute a run of the unspecified component X. As we can see from Figure 2, the only loop in G_1 is $s_1s_2s_1\cdots$. So, if $p = s_0s_1s_2s_1s_3s_1s_2$, then we need to test the "black-box" X with an input sequence "$send\ ack\ send\ send\ send\cdots$" to see whether the output sequence is "$yes\ yes\ yes\ yes\ yes\cdots$".

Step 9. If none of such paths satisfies the conditions in **Step 8**, then $false$ is returned to indicate that the original model-checking problem is true. Otherwise, $true$ is returned. In this case, the maximal length of test sequences generated is bounded by $5m + 3(m - 1)$ according to the evaluation algorithms for EU graphs and EG graphs.

It is easy to see that, in this example, $TestWG$ essentially would be testing whether some communication trace (of bounded length) of sys with two consecutive symbol pairs $(send\ yes)$ is a run of the unspecified component X.

Note. Notice that the condition L_f, a sufficient and necessary condition on the unspecified component X to ensure the truth of the model-checking problem $(M, X), s_0 \models f$, does not depend on the state number m of X. Therefore, even when X is an *infinite-state* system, the condition can also be useful in generating test cases for X and a testing procedure similar to $TestWG$ could be formulated to answer the model-checking query conservatively.

4. RELATED WORK

The quality assurance problem for component-based software has attracted lots of attention in the software engineering community. However, most work are based on the traditional testing techniques and they consider the problem from component developers' point of view; i.e., how to ensure the quality of components before they are released.

Voas [34, 35] proposed a component certification strategy with the establishment of independent certification laboratories performing extensive testing of components and then publishing the results. Technically, this approach would not provide much improvement, since independent certification laboratories can not ensure the sufficiency of their testing either. Some researchers [28] suggested an approach to augment a component with additional information to increase the customer's understanding and analyzing capability of the component behavior. A related approach [36] is to automatically extract a finite-state machine model from the interface of a software component, which is delivered along with the component. This approach can provide some convenience for customers to test the component, but again, how much a customer should test is still a big problem.

Bertolino et. al. [4] recognized the importance of testing a software component in its deployment environment. They developed a framework that supports functional testing of a software component with respect to customer's specification, which also provides a simple way to enclose with a component the developer's test suites which can be re-executed by the customer. Yet their approach requires the customer to have a complete specification about the component to be incorporated into a system, which is not always possible.

In the formal verification area, there has been a long history of research on verification of systems with modular structure. A key idea [24, 23, 20] in modular verification is the *assume-guarantee* paradigm: A module should guarantee to have the desired behavior once the environment with which the module is interacting has the assumed behavior. There have been a variety of implementations for this idea (see, e.g., [19, 1, 29, 11, 8, 37]). The assume-guarantee ideas can be applied to our problem setup if we consider the unspecified component as the host system's environment (though this is counter-intuitive). But the key issue with the assume-guarantee style reasoning is how to obtain assumptions about the environment. Giannakopoulou et. al. [16, 15] introduced a novel approach to generate assumptions that characterize exactly the environment in which a component satisfies its property. Their idea is the closest to ours, still there are non-trivial differences: (1) theirs is a purely formal verification technique (model-checking) while we combine both model-checking and black-box testing to handle systems with unspecified components; and (2) theirs uses a labeled transition system to specify the reachability property of a system while we use CTL formulas, which are more expressive and harder to manipulate. Although not within the assume-guarantee paradigm, Fisler et. al. [13, 25] introduced a similar idea of deducing a model-checking condition for extension features from the base feature for model-checking feature-oriented software designs. Unfortunately, their algorithms are not sound (have false negatives). Furthermore, their approach is not applicable to component-based systems where unspecified components exist. This paper is also different from our previous work [38] where an automata-theoretic approach is used to solve a similar LTL model-checking problem.

In the past decade, there has also been lots of research on combining model-checking and testing techniques for system verification, which can be grouped into a broader class of techniques called specification-based testing. But many of the work only utilizes model-checkers' ability of generating counter-examples from a system's specification to produce test cases against an implementation [7, 21, 12, 14, 2, 5], and they do not generalize the problem setup in this paper. Peled et. al. [31, 18, 30] studied the issue of checking a black-box against a temporal property (called black-box checking). But their focus is on how to efficiently establish an abstract model of the black-box through black-box testing , and their approach requires a clearly-defined property (LTL formula) about the black-box, which is not always possible in component-based systems.

5. CONCLUSIONS

In this paper, we studied the CTL model-checking problem

$$(M, X), s_0 \models f$$

where X is an unspecified component. Our approach is a combination of both model-checking and traditional black-box testing techniques. For such a problem, our algorithm $CheckCTL$ in Section 3.1 either gives a definite $true/false$ answer or gives a sufficient and necessary condition in the form of ID expressions and witness graphs. The condition is evaluated through black-box testing over the unspecified component X. Test sequences are generated by traversing the witness graphs with bounded depth as we evaluate the condition. The evaluation process terminates with a $true/false$ answer. One can show that our algorithm is both complete and sound with a properly chosen search depth (as the ones given in this paper). Basically only theoretic results on the approach are presented in this paper, and in the future we plan to continue investigating the following issues that are important to the implementation of our approach.

- Symbolic Algorithms. The algorithms presented in this paper are essentially explicit state-space searches, which may not scale well to large systems. So it would be interesting our approach can be implemented with symbolic algorithms.

- Scalability. Another issue concerning the scalability of our approach is the choice of the search depth for the generations test sequences. In practice we could sacrifice the completeness of the algorithm by choosing a smaller search depth.
- More Complex Models. The system model considered in this paper is rather restricted. At the present, we are working to extend our approach to more complex system models that allow multiple unspecified components, asynchronous communications between unspecified components and the host system as well as among unspecified components, and unspecified components with an infinite state space.

6. REFERENCES

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *CAV'98*, volume 1427 of *LNCS*, pages 521–525. Springer, 1998.

[2] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM'98*, pages 46–. IEEE Computer Society, 1998.

[3] B. Balzer. Living with cots. In *ICSE'02*, pages 5–5. ACM Press, 2002.

[4] A. Bertolino and A. Polini. A framework for component deployment testing. In *ICSE'03*, pages 221–231. IEEE Computer Society, 2003.

[5] P. E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *ASE'00*, pages 81–. IEEE Computer Society, 2000.

[6] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, Sep/Oct 1998.

[7] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *SPIN'96*, 1996.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE'03*, pages 385–395. IEEE Computer Society Press, 2003.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[10] L. de Alfaro and T. A. Henzinger. Interface automata. In *ASE'01*. ACM Press, 2001.

[11] J. Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *ICSE'03*, pages 138–148. IEEE Computer Society Press, 2003.

[12] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS'97*, volume 1217 of *LNCS*, pages 384–398. Springer, 1997.

[13] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *FSE'01*, pages 152–163. ACM Press, 2001.

[14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE'99*, volume 1687 of *LNCS*, pages 146–163. Springer, 1999.

[15] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *ICSE'04*, pages 211–220. IEEE Press, 2004.

[16] D. Giannakopoulou, C. S. Psreanu, and H. Barringer. Assumption generation for software component verification. In *ASE'02*, pages 3–13. IEEE Computer Society, 2002.

[17] I. Gorton and A. Liu. Software component quality assessment in practice: successes and practical impediments. In *ICSE'02*, pages 555–558. ACM Press, 2002.

[18] A. Groce, D. Peled, and M. Yannakakis. Amc: An adaptive model checker. In *CAV'02*, volume 2404 of *LNCS*, pages 521–525. Springer, 2002.

[19] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, 1994.

[20] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 1998.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[22] W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, Sep/Oct 1998.

[23] O. Kupferman and M. Vardi. Module checking revisited. In *CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 1997.

[24] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[25] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):89–98, 2002.

[26] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[27] B. Meyer. The grand challenge of trusted components. In *ICSE'03*, pages 660–667. IEEE Computer Society Press, 2003.

[28] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. volume 1999 of *LNCS*, pages 129–144, 2001.

[29] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*, pages 168–183, 1999.

[30] D. Peled. Model checking and testing combined. In *ICALP'03*, volume 2719 of *LNCS*, pages 47–63. Springer, 2003.

[31] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE/PSTV'99*, pages 225–240. Kluwer, 1999.

[32] C. Szyperski. Component technology: what, where, and how? In *ICSE'03*, pages 684–693. IEEE Computer Society, 2003.

[33] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite automata; behavior and synthesis*. North-Holland Pub. Co., 1973.

[34] J. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, June 1998.

[35] J. Voas. Developing a usage-based software certification process. *IEEE Computer*, 33(8):32–37, August 2000.

[36] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA'02*, pages 218–228. ACM Press, 2002.

[37] F. Xie and J. C. Browne. Verified systems by composition from verified components. In *FSE'03*, pages 277–286. ACM Press, 2003.

[38] G. Xie and Z. Dang. An automata-theoretic approach for model-checking systems with unspecified components. In *FATES'04*, LNCS. Springer, to appear.