An Automata-theoretic Approach for Model-checking Systems with Unspecified Components⋆

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. This paper introduces a new approach for the verification of systems with unspecified components. In our approach, some model-checking problems concerning a component-based system are first reduced to the emptiness problem of an oracle finite automaton, which is then solved by testing the unspecified components on-the-fly with test-cases generated automatically from the oracle finite automaton. The generated test-cases are of bounded length, and with a properly chosen bound, a complete and sound solution is immediate. Particularly, the whole verification process can be carried out in an automatic way. In the paper, a symbolic algorithm is given for generating test-cases and performing the testings, and an example is drawn from an TinyOS application to illustrate our approach.

1 Introduction

In recent years, component-based software development [20, 6] has gained enormous popularity where large systems are built by assembling software components previously developed by the same organization, customized by third-party software vendors, or even purchased as commercial-off-the-shelf (COTS) products. This development method, however, has also posed one serious challenge to the quality assurance issue of component-based software—externally obtained components could be a new source of system failures. And the response to this challenge is greatly complicated by some intrinsic characteristics of component-based software: 1) for copyright or patent reasons, source codes or design details of externally obtained software components are usually not available to system developers; 2) software components are generally built with multiple sets of functionality [15] and with huge state space in their interfaces; 3) in many applications, software components are used for dynamic upgrading or extending running systems [30] that are too expensive or not supposed to be stopped at all. For instance, in practice, testing is almost the most natural resort to solve the problem; and when integrating a component into a system, developers may choose to either extensively test the component in isolation or hook the component with the system and conduct integration testing. The problem with the first choice, however, is that it is usually difficult to know when the testing over the component is adequate, and indiscriminately testing all the functionality of a software component is not only expensive but sometimes also infeasible due to the second characteristic. The second choice is not

⋆ The research was supported in part by NSF Grant CCF-0430531.

always applicable due to the third characteristic. On the other hand, for safety-critical and mission-critical systems, formal verification techniques, like model-checking [9], are usually desired over the testing techniques to establish the solid confidence for a reliable component. Yet, existing formal verification techniques are not always applicable either, due to the first characteristic.

Clearly, this problem plagues both component-based software systems and modularized hardware systems that contain externally obtained components. Generally, we call such systems as *systems with unspecified components* (in spite of the fact that in many cases, the components are partially specified, our approach still applies). In this paper, we study some model-checking problems, i.e., *reachability*, *safety*, and *LTL Model-checking problems*, for systems with unspecified components.

Most of the current work addresses this problem from the viewpoint of component developers, i.e., how to ensure the quality of components before they are released [33, 29, 24, 34]. This view, however, is fundamentally insufficient: an extensively tested component (by the vendor) may still not perform as expected in a specific deployment environment, since the deployment environments of a component could be quite different and diverse such that they may not be thoroughly tried by the vendor. We approach this problem from system developers' point of view: how to ensure that a component functions correctly in a host system where the component is deployed. The idea of our approach is simple: with respect to certain requirements about a system, derive and test the expected behaviors for the unspecified components. Specifically, we first reduce the model-checking problems concerning systems with unspecified components to the emptiness problem of *oracle finite automata* [35], which are finite automata augmented with query tapes and the ability of querying some external oracles during its computation. This is similar to the conventional automata-theoretic approaches for model-checking [32]. The difference, however, is that decision problems in conventional automata-theoretic model-checking approaches generally have analytic solutions, while the emptiness problem of an oracle finite automaton can only be resolved by querying the oracles with query strings of length bounded by some B. Since each query in the oracle automaton is equivalent to running a test-case corresponding to the query string over an unspecified component, so essentially the solution to solve the emptiness problem is a testing process. But the key point is the generation of test-cases. In this paper, we give an efficient testing algorithm that only generates test-cases that are useful to solve the problem, and performs testing on the fly. Moreover, with an appropriately chosen bound B, our approach is both sound and complete.

2 Preliminaries

2.1 The System Model

In this paper, we consider systems consisting of a host system and a collection of components whose design details are not given. Such a system is denoted by

$$Sys = \langle M, X_1, ..., X_k \rangle \tag{1}$$

for some $k \geq 1$, where M is the host system and each X_i, $1 \leq i \leq k$, is an unspecified component. Both the host system M and all the unspecified components X_i's are finite-state transition systems and they communicate synchronously via a finite set of input and output symbols.

Formally, a component X_i can be viewed as a quintuple

$$\langle S_i, s_{init}^i, \Sigma_i, \nabla_i, R_i \rangle, \tag{2}$$

where S_i is a finite set of states, $s_{init}^i \in S_i$ is the initial state, Σ_i is a finite set of input symbols, ∇_i is a finite set of output symbols, and $R_i \subseteq S_i \times (\Sigma_i \cup \nabla_i) \times S_i$ is the transition relation. Transitions in $S_i \times \Sigma_i \times S_i$ are called *input transitions*, while those in $S_i \times \nabla_i \times S_i$ are called *output transitions*. Since X_i is unspecified, its states set S_i and transition relation R_i are not supposed to be given. But we can assume that its sets of input and outputs symbols, Σ_i and ∇_i as well as an upper bound m_i for X_i's number of states $|S_i|$ are always given, we also assume that a special input symbol (not in Σ_i) "reset" always makes X_i return to its initial state s_{init}^i regardless of its current state. Furthermore, for each unspecified component X_i, we assume that it is *input-deterministic*, i.e., for any $\alpha \in \Sigma_i$, if $(s, \alpha, t) \in R_i$ and $(s, \alpha, t') \in R_i$ then $t = t'$; we also assume that X_i is *output-deterministic*, i.e., for any $\beta \in \nabla_i$ and $\beta' \in \Sigma_i \cup \nabla_i$, if $(s, \beta, t) \in R_i$ and $(s, \beta', t') \in R_i$ then $\beta = \beta'$ and $t = t'$. These two latter assumptions ensure that black-box testing can be efficiently performed on X_i. A *behavior* of X_i is a sequence of symbols in $\Sigma_i \cup \nabla_i$: $c_0 c_1 \ldots$, such that there is a sequence of states $s_0 s_1 \ldots$ with $s_0 = s_{init}^i$ and $(s_j, c_j, s_{j+1}) \in R_i$ for each $j \geq 0$.

The host system M, formally, is also defined as a tuple $\langle S, \Gamma, R_{env}, R_{comm}, s_{init} \rangle$, where

- S is a finite set of states;
- Γ is a finite set of event symbols;
- $R_{env} \subseteq S \times \Gamma \times S$ defines a set of *environment transitions*, where each transition $(s, a, s') \in R_{env}$ makes M move from state s to state s' upon receiving an event (symbol) $a \in \Gamma$ from the outside environment[1];
- $R_{comm} \subseteq S \times \bigcup_{1 \leq i \leq k} (\Sigma_i \cup \nabla_i) \times S$ defines a set of *communication transitions*, where each transition $(s, \alpha, s') \in R_{comm}$ with $\alpha \in \Sigma_i$ (called an *output transition*) makes M move from state s to state s' as well as send α to the unspecified component X_i, while each transition $(s, \beta, s') \in R_{comm}$ with $\beta \in \nabla_i$ (called an *input transition*) makes M move from state s to state s' upon receiving β from X_i; and,
- $s_{init} \in S$ is M's initial state.

For the Sys defined above, we assume that the set Γ, all Σ_i's and ∇_i's are pairwise disjoint. This assumption excludes broadcasting communications in our system model. For simplicity's sake, we also require that each unspecified component X_i is *closed*[2] in the sense that it communicates only with the host system M; i.e., X_i only receives input symbols sent by M and sends output symbols only to M. Finally, it is worthwhile to

[1] We assume that Γ always includes a special symbol ϵ such that (s, ϵ, s') makes M move from state s to state s' without receiving any event symbol.

[2] Note that this assumption does not limit the expressiveness of our model, since two communicating components can be regarded as one component.

notice that the system model defined here covers both systems that are sequential compositions of components and systems that are just collections of concurrently running components. This model is also flexible enough to characterize the two-way communications between the host system and a component in the form of function calls or in the form of synchronized events.

A *behavior* of the system Sys is a sequence τ of symbols in $\Gamma \cup \bigcup_{1 \leq i \leq k} (\Sigma_i \cup \nabla_i)$: $c_0 c_1 \ldots$ such that: 1) there exists a sequence θ of states $s_0 s_1 \ldots$, where $s_0 = s_{init}$ and $(s_j, c_j, s_{j+1}) \in R_{env}$ (resp.$(s_j, c_j, s_{j+1}) \in R_{comm}$) if $c_j \in \Gamma$ (resp. $c_j \in \Sigma_i \cup \nabla_i$ for some $1 \leq i \leq k$); 2) for each $1 \leq i \leq k$, let τ_i denote the subsequence of τ consisting of symbols only in Σ_i and ∇_i, then τ_i is a behavior of X_i. The combination of the behavior τ and the state sequence θ is also a sequence: $s_0 c_0 s_1 c_1 s_2 \ldots$, called a *computation* of Sys. For any given state $s \in S$, we say that the system Sys can *reach* s iff Sys has a computation on which s appears (i.e., $s_0 c_0 s_1 c_1 \ldots s$). Note that, in the case when X_i is fully specified, the system can be regarded as an I/O automaton [23]. As

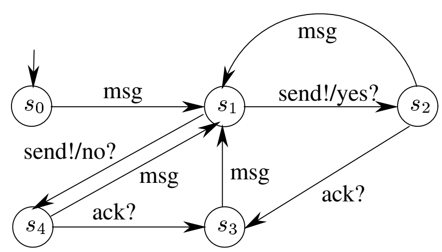

Fig. 1. A simple communication system

an illustrating example, we consider a simple system $Sys = \langle M, X \rangle$ that has only one unspecified component X. In this system, M keeps receiving messages from the outside environment and then transmits the message through X. The unspecified component X accepts only one input symbol *send*, but has three output symbols *yes*, *no* and *ack*. The transition graph of M is depicted in Figure 1, where a suffix ? denotes an input transition (e.g., *ack?*), a suffix ! denotes an output transition (e.g. *send!*), and an infix / is an abbreviation (to save space) for a pair of consecutive output/input transitions (e.g., *send!/yes?*), while a symbol without any suffix denotes an event transition (e.g. *msg*).

2.2 Black-box Testing

Black-box testing is a technique to test a system without knowing its internal structure. The system is regarded as a "black-box" in the sense that its behavior can only be determined by observing (i.e., testing) its input/out sequences. Each unspecified component X_i defined in the previous subsection can be regarded as a "black-box". But, our definition of an unspecified component in (2) is not the Mealy machine as used in traditional black-box testing. So, for the purpose of testing over X_i, we assume that whenever X_i is sent an input symbol in Σ_i, it immediately outputs a special output symbol (not in

∇_i) "yes" or "no" to indicate whether the input symbol is accepted or not. Also we assume that X_i has a special input symbol (not in Σ_i) "probe" that always makes M_i execute an output transition $(s, \beta, s') \in R_i$ if s is its current state, or just output the special symbol "no" if there are no such transitions. Let π^j denote the j-th element of a string π, then the following algorithm **BlkBoxTest**(X_i, π) is used in this paper to test whether π is an behavior of X_i:

Algorithm 1 **BlkBoxTest**(X_i, π)

send "reset" to X_i;
for$(j := 0, j < |\pi|, j++)$
    if π^j is an input symbol $\alpha \in \Sigma_i$
        if send(X_i, α)="No"
            return "No";
    else if π^j is an input symbol $\beta \in \nabla_i$
        if send(X_i, "probe")$\neq \beta$
            return "no";
return "yes";

3 Oracle Finite Automata

In a recent paper [35], we studied oracle finite automata that are finite automata augmented with queries to some oracles. In that paper, we show that, in many cases, the emptiness problems (whether an oracle finite automaton accepts an empty language) are bounded testable; i.e., one can calculate a number B (called query bound) such that querying the oracles with query strings not longer than B is sufficient to solve the emptiness problems. We have obtained computable query bounds B for various classes of oracles for various restricted forms of oracle finite automata (e.g., regular oracles, context-free oracles, commutative semilinear oracles, etc.). However, efficient algorithms for solving the problem were not given in [35]. In this section, after we recall some basic definitions on oracle finite automata,we given an efficient dynamic testing algorithm to solve the emptiness problem of oracle automata, which in turn will be used to solve the model-checking problems concerning systems with unspecified components in the next section.

3.1 Definitions

An oracle automaton is a finite automaton augmented with a finite number of query tapes (that are initially empty) and the power of querying some oracles during its computation. Let $\mathcal{O}$ be a class of languages over alphabet Σ. An oracle O is a language in $\mathcal{O}$ whose definition is unknown, but querying the oracle with a word w on Σ (i.e., $w \in \Sigma^*$) always gives a definite "yes" or "no" answer (depending on whether w is a word of O). Formally, an *oracle finite automaton* (OFA) with k oracles drawn from $\mathcal{O}$ (written as $M^{\mathcal{O}}$), is a tuple

$$\langle S, \Sigma, R, s_{\text{init}}, F, k \rangle, \tag{3}$$

where Σ is the given alphabet, S is a finite set of *states* with $s_{\rm init}$ being the *initial state* and $F \subseteq S$ being a set of *accepting states*. R is a (finite) set of *transitions*, each of which is one of the following:

– a *input transition*, $s \stackrel{a}{\rightarrow} s'$, which makes M move from state s to state s' after reading an input symbol a;
– a *write transition*, $s \stackrel{\texttt{write}(i,a)}{\rightarrow} s'$, which makes M move from state s to state s' after appending a symbol a to the end of the i-th query tape;
– a *positive-query transition*, $s \stackrel{\texttt{query}(i)}{\rightarrow} s'$, which makes M move from state s to state s' when querying the i-th oracle (with the i-th query tape content as the query string) returns a "yes" answer;
– a *negative query transition*, $s \stackrel{\neg\texttt{query}(i)}{\rightarrow} s'$, which makes M move from state s to state s' when $\texttt{query}(i)$ returns a "no" answer;
– or a *reset transition*, $s \stackrel{\texttt{reset}(i)}{\rightarrow} s'$, which makes M move from state s to state s' and resets the i-th query tape content to be empty,

where $s, s' \in S$, $a \in \Sigma$, and $1 \leq i \leq k$. Note that the syntactical definition of $M^{\mathcal{O}}$ does not include any definition of its oracles, except that they should be drawn from $\mathcal{O}$. So $M^{\mathcal{O}}$ actually defines a collection of OFAs, and in the following, we shall use $M^{\mathcal{O}}(O_1, \ldots, O_k)$ to denote the specific OFA associated with k oracles $O_1, \ldots, O_k$ drawn from $\mathcal{O}$.

The semantics of an oracle finite automaton $M^{\mathcal{O}}(O_1, \ldots, O_k)$ can be defined as usual. A word w is *accepted* by $M(O_1, \ldots, O_k)$ if there is an accepting run over w. The language accepted by $M^{\mathcal{O}}(O_1, \ldots, O_k)$, $L(M^{\mathcal{O}}(O_1, \ldots, O_k))$, is the set of all words accepted by $M^{\mathcal{O}}(O_1, \ldots, O_k)$.

Syntactically, an *oracle Buchi automaton* (ω-OFA) $M_\omega^{\mathcal{O}}$ is an oracle finite automaton $M^{\mathcal{O}}$ in (3). But they are semantically different. An ω-word $w_\omega \in \Sigma^\omega$ is *accepted* by $M_\omega^{\mathcal{O}}(O_1, \ldots, O_k)$ if there is an ω-run on w_ω such that some accepting state in F appears infinitely often. The ω-language $L_\omega(M_\omega^{\mathcal{O}}(O_1, \ldots, O_t))$ is still the set of ω-words accepted by the ω-OFA$M_\omega^{\mathcal{O}}(O_1, \ldots, O_t)$.

3.2 Testability of The Emptiness Problem

For the OFAs and ω-OFAs defined in the previous subsection, various decision problems can be considered. In this paper, we study the *emptiness problem*, which is to decide whether an OFA $M^{\mathcal{O}}(O_1, \ldots, O_k)$ accepts an empty language. In the next section, we will show that this emptiness problem is closely related with some model-checking problems for systems with unspecified components. Obviously, since the behavior of an OFA depends on the query results with its oracles and the definitions of the oracles are unknown, we can not analytically solve the problem only from the definition of M itself.

Recall that an oracle O is a language drawn from some class of languages $\mathcal{O}$. Suppose that $\mathcal{O}$ is the class of languages accepted by deterministic finite automata (DFA) with at most m states. Then a finite automaton (without oracles) $\mathcal{T}$ can be constructed to solve the emptiness problem of the OFA $M^{\mathcal{O}}(O_1, \ldots, O_k)$ as follows:

1. for each oracle O_i, $\mathcal{T}$ constructs a DFA $\mathcal{A}_i$ that accepts exactly the language O_i by querying O_i with all words on Σ with length less than $2m-1$ [31], and saves each $\mathcal{A}_i$ on its working tape;
2. $\mathcal{T}$ starts to faithfully simulate M except that when M queries an oracle O_i with a query string w, $\mathcal{T}$ runs the DFA $\mathcal{A}_i$ on its working tape (whose length is bounded by $2m-1$) with w as $\mathcal{A}_i$'s input, and the query is considered successful if $\mathcal{A}_i$ accepts w, or vice versa.

Obviously, $M^{\mathcal{O}}(O_1, \ldots, O_k)$ accepts an empty language iff $\mathcal{T}$ accepts an empty language. Since the emptiness problem of $\mathcal{T}$ can be analytically solved (after $\mathcal{T}$ constructs all the $\mathcal{A}_i$'s.), so does the the emptiness problem of $M^{\mathcal{O}}(O_1, \ldots, O_k)$. Additionally, because the above construction involves pre-querying all oracles with query strings not longer than $2m-1$, which can be viewed as a testing process, we also say that the emptiness problem of $M^{\mathcal{O}}(O_1, \ldots, O_k)$ is $(2m-1)$-*testable*.

Let $\mathrm{OFA}^{\mathrm{DFA}(m)}$ denote the OFAs whose oracles are drawn from the class of languages accepted by deterministic finite automata (DFA) with at most m states. Then we have the following conclusion:

Theorem 1. *The emptiness problem for* $\mathrm{OFA}^{\mathrm{DFA}(m)}$ *is* $(2m-1)$*-testable.*

Similarly, let $\mathrm{OFA}_{\omega}^{\mathrm{DFA}(m)}$ denote a oracle Buchi automata whose oracles are drawn from the class of languages accepted by deterministic finite automata (DFA) with at most m states. Then we have the following conclusion:

Theorem 2. *The emptiness problem for* $\mathrm{OFA}_{\omega}^{\mathrm{DFA}(m)}$ *is* $(2m-1)$*-testable.*

Clearly, not every OFA's emptiness problem can be solved in this way; i.e., not every OFA's emptiness problem is testable. For instance, OFAs associated with oracles from context-free languages are proved to be not testable (for a detailed exposition about the testability of oracle automata, see [35]).

3.3 A Dynamic Testing Algorithm

The solution to the emptiness problem for $\mathrm{OFA}^{\mathrm{DFA}(m)}$ and $\mathrm{OFA}_{\omega}^{\mathrm{DFA}(m)}$ in the previous subsection involves pre-querying the oracles indiscriminately with all possible strings with length shorter than $2m-1$. This would be extremely inefficient in practice, considering the fact that there are an exponential number ($|\Sigma|^{2m-1}$) of such strings.

In this subsection, we introduce a more efficient algorithm to solve the emptiness problem for $\mathrm{OFA}^{\mathrm{DFA}(m)}$ and $\mathrm{OFA}_{\omega}^{\mathrm{DFA}(m)}$. The new algorithm only queries the oracles with query strings that could be "generated" by the OFAs. Since each query to an oracle can also be viewed as a test over the oracle where the query string is a test-case, this algorithm can also be viewed as a dynamic testing process where test-cases are generated on-the-fly.

Suppose that $M^{\mathrm{DFA}(m)}$ is an OFA as defined in (3). Without loss of generality, we assume that M is associated with only one oracle (i.e., $k=1$); generalization to multiple oracles is straightforward. Consequently, there will be only one query tape in M. Then we write instructions $\texttt{reset}(i)$, $\texttt{write}(i, a)$, $\texttt{query}(i)$, and $\neg\texttt{query}(i)$ as $\texttt{reset}$,

$\mathtt{write}(a)$, $\mathtt{query}$, and $\neg\mathtt{query}$, respectively. A transition relation r is a subset of $S \times S$, where S is the state set of M. We use $r_1 \circ r_2$ to denote the relation obtained from composing relation r_1 with relation r_2 , $\mathtt{Intersect}$ to denote the intersection operator, and $\mathtt{TransClosure}(r)$ to denote the transitive closure of a relation r, respectively. We also use $\mathtt{Empty}(r)$ to test whether a relation r is empty. Then, from the definition of M, we define the following transition relations:

$r_{\mathtt{input}} = \{\langle s, s'\rangle : \exists a, s \stackrel{a}{\to} s' \in R\}$,

$r_{\mathtt{reset}} = \{\langle s, s'\rangle : s \stackrel{\mathtt{reset}}{\to} s' \in R\}$,

$r_{\mathtt{write}(a)} = \{\langle s, s'\rangle : s \stackrel{\mathtt{write}(a)}{\to} s' \in R\}$,

$r_{\mathtt{query}} = \{\langle s, s'\rangle : s \stackrel{\mathtt{query}}{\to} s' \in R\}$,

$r_{\neg\mathtt{query}} = \{\langle s, s'\rangle : s \stackrel{\neg\mathtt{query}}{\to} s' \in R\}$.

We first present the algorithm, **TestEmptiness**(B), for testing the emptiness of $M^{\mathrm{DFA}(m)}$, where the query strings are not longer than B. Later, we will describe an algorithm for testing the emptiness of $M_\omega^{\mathrm{DFA}(m)}$.

Algorithm 2 **TestEmptiness**(B)

1: $l := 0$;
2: $\Theta := \{(\{\langle s, s\rangle : s \in S\}, \Lambda)\}$;
3: $E = \{\langle s_{\mathtt{init}}, s_{\mathtt{init}}\rangle\}$;
4: $\Theta' := \Theta$;
5: **for each** (r, w) in Θ with $|w| = l$
6:     $r' := r \circ \mathtt{TransClosure}(r_{\mathtt{input}})$;
7:     **if** $\neg\mathtt{Empty}(r' \circ r_{\mathtt{query}})$ or $\neg\mathtt{Empty}(r' \circ r_{\neg\mathtt{query}})$
8:         query the oracle with query string w;
9:         **if** the query returns yes
10:             $r' := r \circ \mathtt{TransClosure}(r_{\mathtt{input}} \cup r_{\mathtt{query}})$;
11:         **if** the query returns no
12:             $r' := r \circ \mathtt{TransClosure}(r_{\mathtt{input}} \cup r_{\neg\mathtt{query}})$;
13:     replace the entry (r, w) in Θ with (r', w);
14:     $r'' := r' \circ r_{\mathtt{reset}}$;
15:     **if** $\neg\mathtt{Empty}(r'')$
16:         $E := \mathtt{TransClosure}(E \cup r'' \cup r_{\mathtt{input}})$;
17:     **for each** $a \in \Sigma$
18:         $r'' := r' \circ r_{\mathtt{write}(a)}$;
19:         **if** $\neg\mathtt{Empty}(r'')$
20:             add (r'', wa) to Θ;
21: $l := l + 1$;
22: **for each** (r, w) in Θ
23:     $r' := \mathtt{Intersect}(E \circ r, \{s_{\mathtt{init}}\} \times F)$;
24:     **if** $\neg\mathtt{Empty}(r')$
25:         **return** “unsuccessful”;
26: **if** Θ' and Θ are equal or $l > B$
27:     **return** “successful”;
28: **goto** 4;

The **TestEmptiness** algorithm works as follows. We maintain a finite set Θ of pairs of a relation r and a word w. For two states s and s', $\langle s, s' \rangle$ is in r iff, starting from state s and with empty query tape, there is some input word such that state s' is reached (after running M on the input) with the query tape content w, during which no reset occurs. The algorithm also maintains a relation E: for two states s and s', $\langle s, s' \rangle$ is in E iff, starting from state s and with empty query tape, there is a run of M that brings to state s' and also with empty query tape. After initializing Θ and E, the entire algorithm works as a loop from statement 4 to statement 28 and back. In the l-th round (l starts with 0), the algorithm updates an element (r, w) in Θ with $|w| = l$, realized by changing w into wa (i.e., $\texttt{write}(a)$ on the query tape). However, transitions like reading input symbols and querying the oracle can happen before this write, and obviously, the query result matters. This is shown in statements 6–13 where an updated version (r', w) of (r, w) is replaced in Θ (i.e., statement 13). Notice that, a query is performed when necessary shown in statement 8. Then, $\texttt{write}(a)$ is implemented in statements 17–20 to add longer query strings wa into Θ. Clearly, w can also be changed into an empty string through a `reset`, which causes an update on E (recalling the meaning of E mentioned earlier) shown in statements 14–16. Finally in the round, statements 22–27 are used to check whether M accepts an empty language. Clearly, according to the semantics of Θ, if it has a (r, w) where r contains the pair of the initial state and an accepting state, then obviously M accepts a nonempty language — an "unsuccessful" is returned as the result of statements 23–25. If the set Θ does not change in the round (so further rounds are not necessarily) or the level l is higher than the given bound B, then M must accept an empty language (i.e., returns "successful" as in statement 27).

It's not hard to show that the above algorithm is both sound and complete, if one chooses a bound $B \geq m \cdot |M|$. It shall also be noted that the algorithm can be implemented symbolically. This is because a relation can be represented symbolically as a Boolean formula whose satisfying assignments can be further encoded with a BDD [7]. Operations like `TransClosure`, `Intersect`, $\circ$, `Empty` are all standard operations in existing BDD libraries [28].

We can construct another algorithm ω**-TestEmptiness** for testing the emptiness problem of $M_\omega^{\mathrm{DFA}(m)}$, using **TestEmptiness**. This algorithm works as follows. It first constructs an OFA M' from the ω-OFA M that works as follows. M' first guesses an accepting state in F (the set of accepting states in the ω-OFA M) and faithfully simulates M. M' accepts an input word if M enters the guessed accepting state for m times. Clearly, M' is an OFA (instead of an ω-OFA), and it is not hard to show that M' accepts an empty language iff the ω-OFA M does. Then ω**-TestEmptiness** calls algorithm **TestEmptiness**(B) running on M' with $B \geq |F| \cdot |M| \cdot m^2$. One can also show that the algorithm ω**-TestEmptiness** is both sound and complete.

4 The Model-checking Problems

As mentioned in Section 1, this paper studies some model-checking problems, i.e., the reachability, safety, and LTL model-checking problems for systems with unspecified components. In this section, we show that these model-checking problems can be first reduced to the emptiness problem of oracle finite automata, and then be solved by testing the unspecified components with the algorithms defined in the previous section.

Suppose that $Sys = \langle M, X_1, ..., X_k \rangle$ is defined in (1). Let $m = \max_{1 \leq i \leq k} m_i$ (recall that m_i is an upper bound for the number of states in X_i). The *reachability problem* is to decide: starting from its initial state, whether Sys can reach some state in a given set Bad of states; i.e., whether Bad is reachable in Sys (in practice, Bad may specify some "bad" states that are not supposed to be reached).

To solve this problem, we first construct an OFA, $M_{\mathrm{OFA}}^{\mathrm{DFA}(m)}(O_1, \ldots, O_k)$ in (3) from the definition of Sys as follows:

1. for each $1 \leq i \leq k$, let oracle O_i denote the set of behaviors of the unspecified component X_i (remember that an oracle is a language without detailed definition);
2. let M_{OFA} have the same set of states and same initial state as M;
3. let M_{OFA}'s Σ be the union of Γ and all Σ_i's and ∇_i's in Sys;
4. let Bad be M_{OFA}'s accepting states;
5. for each transition (s, a, s') in M with $a \in \Gamma$, add a transition $(s \xrightarrow{a} s')$ to M_{OFA};
6. for each transition (s, α, s') in M with $\alpha \in \Sigma_i$ for some $1 \leq i \leq k$, add a transition $(s \xrightarrow{\mathtt{write}(\alpha,i)} s')$ to M_{OFA};
7. for each transition (s, β, s') in M with $\beta \in \nabla_i$ for some $1 \leq i \leq k$, add a new state s'', as well as two transitions $(s \xrightarrow{\mathtt{write}(\beta,i)} s'')$ and $(s'' \xrightarrow{\mathtt{query}(i)} s')$ to M_{OFA}.

For instance, from the system depicted in Figure 1, we can construct an OFA as shown in Figure 2 (since this OFA has only one query tape, in Figure 2, we write instructions $\mathtt{write}(i, a)$ and $\mathtt{query}(i)$ as $\mathtt{write}(a)$ and $\mathtt{query}$ respectively).

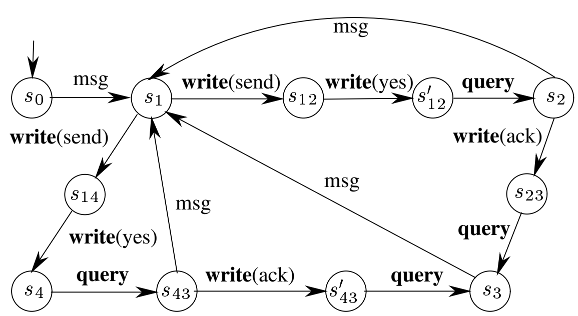

Fig. 2. An oracle finite automaton

Now it is easy to see that Bad is not reachable in the system Sys iff the constructed M_{OFA} accepts a nonempty language. Then we have,

Theorem 3. *The reachability problem for the system Sys is testable.*

The *safety problem* is to decide whether every behavior of the system Sys is contained in a given regular language R. Assume that the complement of R can be accepted by a finite automaton M_R and let $\bar{M}$ be the Cartesian product of M_R and M. Notice that each state in $\bar{M}$ is a pair of states in M_R and M respectively and $\bar{M}$ totally has $|M_R|\cdot|M|$ number of states; i.e. $|\bar{M}| = |M_R|\cdot|M|$. Let F denote the set of states in $\bar{M}$, each of which contains a final state of M_R. Similar to the construction in the reachability problem (except that F would be the OFA's accepting states), we can construct an OFA, $\bar{M}_{\mathrm{OFA}}^{\mathrm{DFA}(m)}(O_1,\ldots,O_k)$ from this $\bar{M}$ as well as the unspecified components X_i, $1 \le i \le k$. Then it shall be noticed that the safety problem is true iff the constructed $\bar{M}_{\mathrm{OFA}}$ accepts an empty language. Hence we have,

Theorem 4. *The safety problem for the system Sys is testable.*

Next, we consider the model-checking problems concerning ω-behaviors of the system Sys; i.e., the LTL model-checking problem.

The linear-time temporal logic (LTL) views the behaviors of a finite-state system as a set of paths, i.e., infinite words on an alphabet Σ. And LTL formulas, which are interpreted over paths, are defined as follows:

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid X\phi \mid \phi \mathbf{U} \phi,$$

where $a \in \Sigma$ is an *atomic proposition*. X is the *next* operator, and $\mathbf{U}$ is the *until* operator. We interpret each atomic proposition a as the singleton set $\{a\}$. Intuitively, a path σ satisfies an atomic proposition a if the first symbol of σ is symbol a. A path σ satisfies $X\phi$ if σ^1 (by deleting the first symbol in σ) satisfies ϕ. σ satisfies $\phi\mathbf{U}\psi$ if there is a suffix σ^i (by deleting the first i symbols) of σ such that (1). the suffix satisfies ψ and (2). ϕ is consistently satisfied on each σ^j with $0 \le j < i$. Notice that our treatment of atomic propositions here is essentially equivalent to a standard LTL definition [10] (though the appearance of ours is a little different). LTL is capable of expressing many interesting properties of a reactive system. For instance, the property "the pump is on for infinitely many times" can be expressed as $\Box\Diamond\texttt{pumpOn}$ (where $\Diamond\phi$ (eventually ϕ) is an abbreviation for $\texttt{true}\ \mathbf{U}\ \phi$, and $\Box\phi$ (always ϕ) stands for $\neg\Diamond\neg\phi$.). We use $[f]$ to denote the set of ω-words that satisfy f. It is known that $[f]$ can be accepted by a Buchi automaton (an ω-OFA without the query tapes) with $O(2^{|f|})$ number of states, where $|f|$ is the length of f).

The *LTL model-checking* problem is to decide whether every ω-behavior of the system Sys satisfies a given LTL formula f. Similar to the standard LTL model-checking approach [32], we define $\bar{M}$ to be the Cartesian product of M and the Buchi automaton that accepts $[\neg f]$. Similar as before, we construct an ω-OFA, $\bar{M}_{\omega}^{\mathrm{DFA}(m)}(O_1,\ldots,O_k)$ from this $\bar{M}$ as well as the unspecified components X_i s. Observe that the LTL model-checking problem is equivalent to checking the emptiness of $\bar{M}_{\omega}^{\mathrm{DFA}(m)}(O_1,\ldots,O_t)$. Hence, we have the following result:

Theorem 5. *The LTL model-checking problem for the system Sys is testable.*

Note that in the above constructions, the oracles actually characterize the behaviors of the unspecified components. Therefore, when we apply the **TestEmptiness** algorithm to the constructed OFAs (OFA_{ω}s), line 8 of **TestEmptiness**, i.e., "query the oracle with string w" should be replaced with **BlkBoxTest**(X, w). That is, the model-checking problems for the systems Sys are finally reduced to testings over the unspecified components in Sys.

Remark. As we have seen from the above reductions from the model-checking problems on Sys to the emptiness problems for the constructed M_{OFA}s, there are no `reset` and negative query transitions in M_{OFA}. This implies that the reduction and the algorithms still work when we understand each m_i, instead of being the number of states in component X_i, to be the number of states in a *nondeterministic* finite automaton that accepts the behaviors of X_i. This will greatly bring down the bound B for query strings for the algorithms **TestEmptiness** and ω**-TestEmptiness**. Also, the above argument still applies, if we further allow "reset" to be an ordinary input symbol of X_i, i.e., "reset" can appear on a transition in Sys. Clearly, the transition containing a "reset" in sys corresponds to a `reset` transition in the OFA to which Sys is reduced.

5 Applications

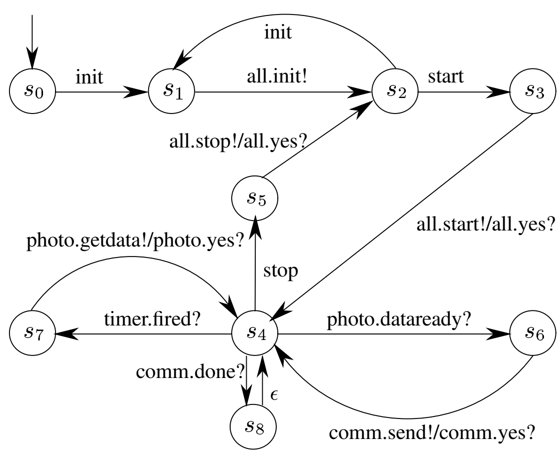

Fig. 3. A data acquirement system

In this section, we consider a TinyOS application. TinyOS is a lightweight operating system for networked sensors [18]. It is designed with a highly modularized architecture such that a specific application can be easily built by assembling just the software components required to synthesize the application from the hardware components. In a TinyOS application, components are glued together through *interfaces*. An interface consists of a set of commands and a set of events, and each component declares the interfaces it provides to other components and the interfaces it shall use from other components. The provider of an interface must implement a command handler for each command in the interface; and the user of an interface must implement an event handler for each event in the interface. The command handlers return a boolean value indicating a success or failure. A TinyOS application is event-driven; i.e., it executes by synchronizing events and commands between its components.

For instance, consider a data acquirement application which periodically transmits the reading of a photo sensor via some underlying communication network. This application consists of a host system and three components: *timer*, *photo* and *comm* whose functionality are as implied by their names. All three components respond to three standard commands: *init*, *start*, and *stop*. Particularly, the timer component also signals an event *fired* when the time interval set runs out. The photo component always responds to a command *getdata*, but it signals an event *dataready* only when the sensor's reading is ready. The comm component always responds to a command *send*, but it signals an event *done* only when the data is successfully sent.

Suppose the internal specifications of the three components are not available, but we know they are all finite state transition systems. Then each of them can be treated as an unspecified component in (2), and the system can be viewed as a system in (1). The transition graph of the host system is depicted in Figure 3, where s_0 is the initial state, suffix "?" is used to denote an event coming from a component, suffix "!" is used to denote a command sent to a component, infix "/" is used as an abbreviation (to save space) for a pair of consecutive command and event, symbols without any suffix denote commands from the outside world, and ϵ denotes an empty symbol. Notice that, in Figure 3, we use two additional events "yes" and "no" to indicate the return value of a command handler (i.e., the success or failure of a command), and we also used "all" as an abbreviation for all of the three components. [3]

Then we can consider the following LTL model-checking problem for the system Sys (which can be expressed in the LTL formalism defined earlier):

– $Sys, s_0 \models AG(s_6 \rightarrow X((\neg s_6)\ U\ s_7))$, i.e., on all computations of Sys, no two $photo.dataready$ outputs can be sent without receiving a $photo.getdata$ message.

From the results presented in this paper, this LTL model-checking problems can be reduced to the emptiness problem of an ω-OFA constructed from the system. And the emptiness problem of the ω-OFA can be solved by querying (testing) the oracles (unspecified components) with strings of bounded length.

6 Related Work

In the formal verification area, there has been a long history of research on verification of systems with modular structure (called modular verification [27]). A key idea [21, 17] in modular verification is called 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., [1]). However, the assume-guarantee idea does not fit with our problem setup since it requires users to have clear assumptions about a module's environment. Although Giannakopoulou et. al.[14] introduced a novel approach to generate assumptions that characterize exactly the environment in which a component satisfies its property. Their donot generalize to systems with unspecified components where a purely formal method is not applicable.

[3] This example comes from the TinyOS distribution [4], and we abstracted its original nesC source code into this automaton form.

In the past decade, there also has been some research on combining model-checking and testing techniques for system verification, which can be classified into a broader class of techniques called specification-based testing. But most of the work [8, 19, 11, 13, 3, 5, 2] just utilizes model-checkers' ability of producing counter-examples from a system's specification to generate test-cases against an implementation. In spirit, our work is closely related with the series of work by Peled et. al. [26, 16, 25] where they studied the issue of checking a black-box against a temporal property (called black-box checking). But our problem setup is on the verification of component-based systems, and we focus on how to derive test-cases for unspecified components from the host system. Their approach requires a clearly-defined property (LTL formula) about the black-box, which is not always possible in component-based systems. Fisler et. al. [12, 22] introduced an idea of deducing a model-checking condition for extension features from the base feature to study model-checking feature-oriented software designs. Unfortunately, their approach relies totally on model-checking techniques; their algorithms have false negatives and do not handle LTL formulas.

References

1. Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *CAV'98*, volume 1427 of *LNCS*, pages 521–525. Springer, 1998.
2. Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. NIST-IR 6777, National Institute of Standards and Technology, 2002.
3. Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *ICFEM'98*, page 46, 1998.
4. UC Berkeley. Tinyos 1.1.0, Sep. 2003. http://webs.cs.berkeley.edu/tos/download.html.
5. Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *ASE'00*, page 81, 2000.
6. A.W. Brown and K.C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, Sep/Oct 1998.
7. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
8. J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *Proceedings 1996 SPIN Workshop*, 1996.
9. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop of Logic of Programs*, volume 131 of *LNCS*. Springer, 1981.
10. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1990.
11. A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS'97*, volume 1217 of *LNCS*, pages 384–398. Springer, 1997.
12. Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE'01*, pages 152–163. ACM Press, 2001.
13. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE'99*, volume 1687 of *LNCS*, pages 146–163. Springer, 1999.
14. Dimitra Giannakopoulou, Corina S. Psreanu, and Howard Barringer. Assumption generation for software component verification. In *ASE'02*, page 3. IEEE Computer Society, 2002.

15. Ian Gorton and Anna Liu. Software component quality assessment in practice: successes and practical impediments. In *ICSE'02*, pages 555–558. ACM Press, 2002.
16. Alex Groce, Doron Peled, and Mihalis Yannakakis. Amc: An adaptive model checker. In *CAV'02*, volume 2404 of *LNCS*, pages 521–525. Springer, 2002.
17. Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98*, volume 1427 of *LNCS*, pages 440–451. Springer, 1998.
18. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.
19. 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.
20. W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, Sep/Oct 1998.
21. Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.
22. Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):89–98, 2002.
23. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Proc. 6th ACM Symp. on Principles of Distributed Computing, pp. 137–151, 1987.
24. A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. *LNCS*, 1999:129–144, 2001.
25. Doron Peled. Algorithmic testing methods. In *CAV'03*, volume 2725 of *LNCS*. Springer-Velag, july 2003.
26. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *FORTE/PSTV'99*, pages 225–240. Kluwer, 1999.
27. A. Pnueli. In transition from global to modular temporal reasoning about programs, 1985. In K.R. Apt, editor, Logics and Models of Concurrent Systems, sub-series F: Computer and System Science.
28. F. Somenzi. Cudd: Cu decision diagram package release, 1998.
29. J. Stafford and A. Wolf. Annotating components to support component-based static analyses of software systems, September 2000. In Grace Hopper Celebration of Women in Computing, Hyannis, Massachusetts.
30. C. Szyperski. Component technology: what, where, and how? In *ICSE'03*, pages 684–693. IEEE Computer Society Press, 2003.
31. B. A. Trakhtenbrot and Ya. M. Barzdin. *Finite automata; behavior and synthesis*. North-Holland Pub. Co., 1973.
32. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS'86*, pages 332–344. IEEE Computer Society Press, 1986.
33. J. Voas. Developing a usage-based software certification process. *IEEE Computer*, 33(8):32–37, August 2000.
34. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA'02*, July 2002.
35. Gaoyan Xie, Cheng Li, and Zhe Dang. Testability of oracle automata, 2004. To Appear in the Proceedings of CIAA'04.