Decompositional Algorithms for Safety Verification and Testing of Aspect-Oriented Systems *

Cheng Li and Zhe Dang**

School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA

Abstract. To efficiently solve safety verification and testing problems for an aspect-oriented system, we use multitape automata to model aspects and propose algorithms for the aspect-oriented system specified by a number of primary labeled transition systems (some of them are black-boxes) and aspects. Our algorithms combine automata manipulations over the aspects and primary systems with black-box testing over each individual black-box, but without generating the woven system.

1 Introduction

Aspect-oriented Programming (AOP) [1] has been considered among "ten emerging areas of technology that will soon have a profound impact on the economy and on how we live and work" [14]. In a software system, a concern is understood as a property of interest. Separation of concerns has long been regarded as a main principle in software engineering. A concern can be implemented as a *component* (if it can be cleanly encapsulated in a generalized procedure or object) or as a cross-cutting *aspect* (if otherwise; e.g., a security aspect interleaved with several components) [1]. In AOP, *primary systems* can be *woven* with aspects into *woven systems* – final executables – by aspect *weavers*. This process is called *weaving*, which has provided a new way to compose a complex system, whose reusability, extensibility and adaptability may also be increased. The successes of AOP at the code level (e.g., AspectJ [2]) have also inspired researchers to study methodologies in aspect-oriented design that bring in cross-cutting concerns even at earlier software development stages [9, 10, 8, 12, 3, 4].

Despite its convenience in addressing cross-cutting concerns, introducing aspects into a system on the other hand raises a quality assurance issue in the woven system: how to assure that a collection of aspects really add the functionality they are supposed to, and moreover, do not invalidate desirable properties of the primary system to which the aspects are woven? That is, we would like to assure that aspects perform their intended behavioral modifications over the primary system without producing any undesirable side effects. Theoretically, it is clear that, once a primary system is given, a well-specified aspect (we assume that the aspect "knows" how to weave) will give us a construction on the woven system. Therefore, the quality assurance problem is essentially a verification problem and verification techniques like model checking [5] can be applied on the woven system directly. However, this direct approach has serious issues:

* The work was supported in part by NSF Grant CCF-0430531.

** Corresponding author (zdang@eecs.wsu.edu)

– Before the model-checking starts on the woven system, one *has to* wait till the woven system is constructed. But when the model-checking actually starts, the state space in the woven system may have already exploded, in particular when nested weaving is involved.
– When the primary system contains components that are black-boxes (such as a COTS component, whose source code or design details are unavailable), a woven system may not even be available.

To address the issues, in this paper, we study fundamental algorithms that are possible to verify/test an aspect-oriented system or design, but without weaving (i.e., without constructing the woven system).

In our study, a system or design is modeled as a labeled transition system. An aspect is a multitape automaton, or more precisely, the tuple language accepted by the automaton. It characterizes how behaviors of several primary systems can be woven into a behavior of the woven system. We then define an aspect-oriented system $\mathcal{A}$ as a tree whose leaves are primary systems and nonterminal nodes are aspects. As defined in the paper, the woven system, also denoted by $\mathcal{A}$, can be constructed through automata manipulations (assuming that the automata for the aspects as well as the primary systems are of finite-state). We study the safety verification problem as follows: Given a regular set Bad (of event sequences), whether the woven system has a behavior in Bad. Our safety verification algorithm is a top-down and then bottom-up process that explores the structure of the tree $\mathcal{A}$ (using automata manipulations), during which a regular $badSet$ is calculated and updated for each node. Once any one of these $badSet$s becomes empty, the algorithm halts. Our algorithm makes it possible to obtain the answer to the safety verification problem before the entire tree is explored. We also study the safety testing problem which is exactly the same as the safety verification problem, except that one or more of the primary systems are black-boxes. Our safety testing algorithm explores the structure of the tree $\mathcal{A}$ and makes use of the white-box primary systems as well as the *test results* of those black-boxes that have been tested in the algorithm. Then, the algorithm computes, through automata manipulations, a $badSet$ for the black-box that is about to test. This $badSet$ has the following property: a behavior of the black-box that is not in the $badSet$ can not cause the woven system $\mathcal{A}$ to have a behavior in the given Bad. Hence, this $badSet$ can be used to further eliminate the *unnecessary tests* that would otherwise be tested on the black-box. The algorithm selects and performs tests for each of the black-boxes in this way. The algorithm halts when one of these $badSet$s becomes empty. Therefore, essentially, our safety testing algorithm is decompositional and dynamic: tests run on a black-box are tailored to the specific safety testing problem instance of $\mathcal{A}$. Furthermore, tests performed over a black-box will be used later in the algorithm to further trim away unnecessary tests performed over other black-boxes.

2 Related Work

Recently, a significant amount of papers have been published to address the modeling and verification problems of aspect-oriented systems.

In [18, 16], model-checking has been used to verify aspect-oriented systems at the source code level by extracting finite-state designs. Unfortunately, such an approach

may cause false negatives on the verification results. References [9, 10, 8, 12, 3] extend the UML (Unified Modeling Language) to support aspect-oriented design, where the primary system and aspects can be woven at the design level. However, since the semantics of UML is not formal in general, the woven design can not be faithfully verified. To address the issue, some researchers seek to translate a subclass of aspect-oriented UML to a formal specification language associated with a formal analysis tool. For instance, in [4], performance is modeled as an aspect using aspect-oriented UML which is translated into Rapide ADL [15] to evaluate if the woven system satisfies a time-response requirement. Reference [17] adapts a role-based aspect-oriented modeling method for aspect-oriented UML design and uses Alloy, a lightweight formal specification language and analysis tool, to verify the woven system. However, as pointed out by authors, the translation from UML to Alloy was done manually and only worked for some special cases.

Our approach is totally different from all approaches we mentioned above. Our safety verification and testing algorithms verify and test aspect-oriented systems without constructing the actual woven systems. We also believe that our formal approach of using multitape automata and their manipulations in studying verification problems of aspect-oriented systems is also new: this approach will also make research results that are already established in automata theory be available in analyzing aspects and aspect-oriented systems, e.g., aspects that are of infinite-state.

Our algorithms are also related to our decompositional testing algorithms [6] for concurrent systems containing black-box components. In these latter algorithms which are inspired by the decompositional verification ideas by Giannakopoulou et. al.[7], test sequences are generated and run on a concurrent component that are customized to its specific deployment environment. Since blackbox testing (instead of verification) is used in [6], unlike the framework in [7], the testing algorithms in [6] does not require a complete specification about a component to be incorporated into the concurrent system. On the other hand, we study decompositional testing algorithms for aspect-oriented systems in this paper instead of concurrent systems in [6].

3 Systems, Transactions, and Aspects

In this paper, a system M is a (nondeterministic) labeled transition system, where its labels, called (external) events, are drawn from a given finite alphabet Σ. Formally, $M = (Q, q_0, \Sigma, \delta)$, where Q is a (not necessarily finite) set of states (with $q_0 \in Q$ being the initial state) and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ defines transitions, each of which is in the form of (q, a, q'), for some $q, q' \in S$ and $a \in \Sigma \cup \{\epsilon\}$, indicating that state q transits to state q' while event a is observed (when $a = \epsilon$ (i.e., a is *silent*), nothing is observed). Therefore, when M runs by following the state transitions, one observes a sequence of events, i.e., a word w in Σ^*. Formally, an *execution* of M is, for some n, a sequence $(q_0, a_1, q_1)(q_1, a_2, q_2), \cdots, (q_{n-1}, a_n, q_n)$ of transitions in δ, which starts from the initial state q_0. A word w is a *behavior* of M if, for some execution of M shown above, w is $a_1 \cdots a_n$ (after ignoring all the silent events in $a_1 \cdots a_n$). In particular, when the word ends with a special event $\heartsuit \in \Sigma$, its is called a valid *transaction* of M. Notice that the special symbol is an indication of the end of a transaction and, moreover, there

could be multiple appearances of ♡'s before the last appearance of ♡ in the transaction. As usual, we use $L(M)$ to denote the set of all transactions of M.

The events in Σ serves as the interface of M. Even though M can be an infinite-state system (i.e., the state set Q is infinite), its behaviors over the interface could be simple; e.g., $L(M)$ forms a regular language (such a view of *interface automata* is studied in [11]). Clearly, when M is a finite-state system (i.e., the state set Q is finite), $L(M)$ has to be a regular language.

Labeled transition systems M are a popular abstract representation of a software system and its design. In case when the transition graph δ of M is unknown (but its interface Σ is known), M is considered as a black-box. In this paper, we assume that the black-box can be tested. That is, there is a procedure $\mathbf{BTest}(M, w)$ that returns a definite (yes/no) answer on whether w is a transaction of M. In automata theory, this is called membership testing; i.e., whether $w \in L(M)$. Clearly, in order for one to implement the procedure $\mathbf{BTest}$, a number of requirements of M must be met (e.g., one needs to distinguish input events and output events in M, one might want to assume that M is input deterministic, M has an implementation to run, etc.; see [13] for a comprehensive survey on black-box testing). For ease of presentation, we simply assume that the black-box M has already met all the necessary requirements such that the black-box testing procedure $\mathbf{BTest}$ does exist and is given. As we all know, black-box testing can even run on infinite-state systems.

An important class of verification queries, called the *safety verification problem*, is as follows:

Given: a system M and a set $Bad \subseteq \Sigma^*$,

Question: $L(M) \cap Bad = \emptyset$?

In above, $Bad \subseteq \Sigma^*$ specifies a set of *bad* transactions that are not supposed to be the transactions of M. Clearly, a negative answer to the **Question** indicates an error in the system with respect to its requirement specified as "no Bad transactions". Automata-theoretic model-checking techniques can be used to solve the safety verification problem when both M and Bad are in certain restricted forms. In particular, when M is a finite-state system and Bad is a regular set, the problem can be solved.

When M is a black-box, the safety verification problem can not be solved in general. In this case, black-box testing can be used to obtain an inconclusive answer as follows. We assume that a procedure $\mathbf{GenTests}(M, \Sigma)$ is given which returns a set of words. Each word w that is in the set and in Bad is then run on the testing procedure $\mathbf{BTest}(M, w)$. If one of such w is *successful* (i.e., $\mathbf{BTest}(M, w)$ returns "yes"), then a negative answer to the **Question** in the safety verification problem is identified. Otherwise, the answer is inconclusive. The set of tests that $\mathbf{GenTests}$ generates has to be finite (patience of a test engineer is practically bounded). In practice, it is still an ongoing research issue in Software Engineering on how to define an "adequate" $\mathbf{GenTests}$, in particular when M is a grey-box (with partial information on its transition graph known). Nevertheless, in this paper, we assume that such a $\mathbf{GenTests}$ exists and given (e.g., a straightforward version of $\mathbf{GenTests}$ is to return the set of all words in Σ^* whose length are not longer than 40).

Before we proceed further, we present a simple banking system (modified from [12]) shown in Figure 1, which will be used throughout this paper. With this simple

banking system, a customer can `open` and `close` a bank account. With a bank account, the customer can `login` to the system and perform a number of *atomic* accesses on the bank account, then `logout` the system. An atomic access can be any one of `withdraw`, `deposit` or `getBlance` on the account. According to Figure 1,

$$\texttt{open}, \texttt{deposit}, \texttt{getBlance}, \texttt{logout}, \heartsuit$$

is a valid transaction: the customer `open`s an account and `deposit`s some money on the account, then `getBlance` of the new created account before `logout`. However,

$$\texttt{open}, \texttt{withdraw}, \texttt{getBlance}, \texttt{logout}, \heartsuit$$

is not valid transaction: the figure specifies that any costumer should deposit some cash to the account first, before withdrawing from the account.

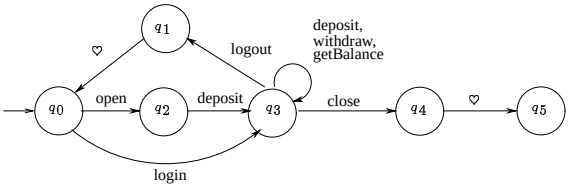

Fig. 1. A simple banking system

In aspect-oriented software development, an aspect can be understood as a structural transformer (e.g., a program transformer in AspectJ) or a behavioral transformer (a relation between event sequences). We use the latter understanding in this paper and thus an aspect is called a *behavioral aspect*. The semantics of the aspect, which is specified by the relation, is independent of the syntax (i.e., the transition graph) and the semantics (i.e., the behaviors) of a primary system M. Therefore, even without the primary system M, one can still design an aspect. Also, it guarantees that the semantics of the woven system does not change whenever the semantics of the primary system does not change. In the following, we will present a formal definition of an aspect, which can be applied to several primary systems (e.g., "interleaving" can be considered as an aspect that weaves two systems into one where the two systems run concurrently).

Formally, a k-ary *behavioral aspect* A is a relation $A \subseteq (\Sigma^*)^k \times \Sigma^*$, which specifies how to weave k primary behaviors into a woven behavior. Let $M_1, \cdots, M_k$ be labeled transition systems over events Σ. The set of *woven transactions*, written $A(L(M_1), \cdots, L(M_k))$, is the set of all words $w\heartsuit$ such that there are transactions $w_1\heartsuit, \cdots, w_k\heartsuit$ in $L(M_1), \cdots, L(M_k)$, respectively, satisfying $(w_1\heartsuit, \cdots, w_k\heartsuit, w\heartsuit) \in A$. To further abuse the notation, we simply use $A(M_1, \cdots, M_k)$ to denote the set. For a given behavioral aspect A, a *weaving function* is a function $W_A(M_1, \cdots, M_k)$ that maps from $M_1, \cdots, M_k$ (called primary systems) to some system M, called a woven system, such that $L(M) = A(M_1, \cdots, M_k)$. Notice that, even though a behavioral aspect is independent of the transition graphs of the primary systems, as an exercise in

computability theory, one can show that a computable weaving function always exists and can be constructed for a given recursively enumerable behavioral aspect, when the primary systems are given as Turing machines (or any other universal computing devices). That is, the existence of such a computable weaving function tells us that, in the most general sense, a woven system can be constructed automatically from primary systems using a behavioral aspect.

4 Finite-state Behavioral Aspects and Weaving

We now study finite-state behavioral aspects that are tuple languages accepted by multitape finite automata. A (nondeterministic) multitape finite automaton consists of a finite control and n (for some n) input tapes. Each tape has a one-way and read-only head. The automaton starts in its initial state with all the heads on the leftmost cells of their tapes. Each transition is of the form $(q, a_1, \cdots, a_n, p)$ where q and p are states and $a_1, \cdots, a_n$ are symbols (in $\Sigma \cup \{\epsilon\}$). On firing the transition, the automaton can, when in state q, for each i, read a_i from the i-th tape, and enter state p. The automaton accepts the tuple of n input strings if each head reaches the right end of its tape while entering a designated final state. It is known that multitape finite automata are essentially different from (one-tape) NFA; e.g., the equivalence problem (whether two automata accept the same language) is undecidable for multitape finite automata.

A k-ary behavioral aspect A is of *finite-state* if there is a $(k+1)$-tape finite automaton M such that A equals the $(k+1)$-tuple language accepted by M. In this case, we sometimes abuse the A as the M.

Now let us go back to the simple banking system example. As the simple banking system evolves, the requirement changes. Developers might be asked to add a new feature to the system: Every atomic access to an account should be logged by recording the name of the accessing customer and the type of the access in a log file. This logging feature is a typical example of a crosscutting concern, which can not be easily represented in an object-oriented design as it interleaves the same feature into every atomic access in the original simple banking system. Adding such a feature is best supported by aspect-oriented software development. In this example, we use a `logging` aspect to implement this feature.

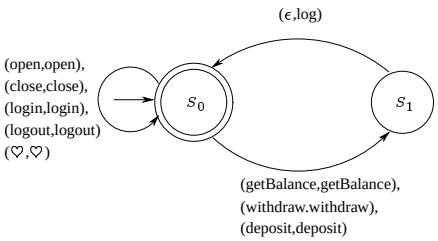

Fig. 2. The `logging` aspect modeled as a two-tape finite automaton

The `logging` aspect is quite simple. Figure 2 shows how the `logging` aspect can be modeled as a deterministic two-tape automaton A. A has two states S_0 and S_1 and two transitions between them. For transition from S_0 to S_1, the two tapes of A read same input; for the transition from S_1 to S_0, the first tape reads nothing and the second tape reads `log` as input. As a result, whenever there is an atomic access in primary behavior, there is a same atomic access appended by a `log` event in woven behavior. It should be noticed that our `logging` aspect does not `log` the events `open`, `close`, `login`, `logout` since they are not atomic accesses. Therefore, the aforementioned primary behavior `open`, `deposit`, `getBlance`, `logout`, ♡ becomes the following woven behavior after weaving the `logging` aspect and the simple banking system: `open`, `deposit`, `log`, `getBlance`, `log`, `logout`, ♡. Indeed, the `logging` aspect defines a relationship between the behavior of a primary system and the behavior of a woven system.

Let A be a k-ary finite-state behavioral aspect and $M_1, \cdots, M_k$ be finite-state systems. In this case, a woven system $M = W_A(M_1, \cdots, M_k)$ can be constructed as follows (sketch). M is a finite automaton that simulates the multitape automaton A. During the simulation, the tape contents of the first k tapes in A are guessed and also run over the systems $M_1, \cdots, M_k$, respectively. The content of the last tape in A is fed by the input tape content of M itself. M accepts when A accepts. It can be shown that, in worst case, the size (state number) of the woven system is $O(|A| \cdot |M_1| \cdots |M_k|)$. Apply the weaving process to the simple banking system in Figure 1 and the `logging` aspect in Figure 2, the woven system is shown in Figure 3.

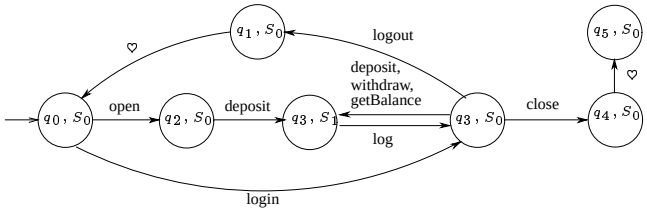

Fig. 3. The simple banking system woven with the `logging` aspect

5 Safety Verification and Testing of Aspect-Oriented Systems

At the heart of aspect-oriented software development methodology, aspects are used along with multiple primary systems to construct a final woven system through (nested) weaving. One can raise the same safety verification problem for the woven system. However, one of the difficulties now is how to deal with the case when some of the primary systems are black-boxes (a white-box can also be marked as a black-box when its behaviors are hard to analyze; e.g., some infinite-state systems.). Our solution is a decompositional algorithm that combines model-checking with black-box testing. Be-

fore we proceed further, we first formally define aspect-oriented systems. To simplify our presentation (but WLOG), we assume that a behavioral aspect is 2-ary.

Let $M_1, \cdots, M_n$ be some given primary systems, and $A_1, \cdots, A_m$ be some given (2-ary) behavioral aspects. An *aspect-oriented system* $\mathcal{A}$ is a binary tree T where each node is either a leaf or a nonterminal node (with two children). There are n leaves in T, which are labeled with $M_1, \cdots, M_n$, respectively. Each nonterminal node is labeled with an aspect A_i for some $1 \leq i \leq m$. Notice that distinct nonterminal nodes could have the same label. For a nonterminal node u, we use $u.left$ and $u.right$ to indicate its left and right children, respectively. The semantics of the aspect-oriented system $\mathcal{A}$ is defined recursively as follows. We associate a system M_u to each node u in T. When u is a leaf, M_u is simply the system M_i originally labeled on u. Then, recursively, when u is a nonterminal node, M_u is the woven system $W_A(M_{u.left}, M_{u.right})$, where A is the behavioral aspect originally labeled on u. The final woven system of $\mathcal{A}$ is then specified by the woven system associated with the root node $root$; i.e., M_{root}. Sometimes, we simply use $\mathcal{A}$ itself to indicate the M_{root}. Figure 4 (a) shows an $\mathcal{A}$ with four primary systems and three aspects.

5.1 Safety Verification Algorithm for Aspect-Oriented Systems

The *safety verification problem for aspect-oriented systems* is to decide whether an aspect-oriented system $\mathcal{A}$ has a bad transaction in a given regular set Bad; i.e., $L(\mathcal{A}) \cap Bad = \emptyset$?. Suppose that all the primary systems $M_1, \cdots, M_n$ as well as all the behavioral aspects $A_1, \cdots, A_m$ in $\mathcal{A}$ are of finite-state. To solve the problem, a naive approach would be to construct the final woven system $\mathcal{A}$ (which is still a finite-state system) directly and then use this $\mathcal{A}$ to check against the emptiness of $L(\mathcal{A}) \cap Bad$. However, there is an issue with this approach. Calculating the final woven system $\mathcal{A}$ is expensive: in worst case, the size of the woven system is $O(N^n \alpha^n)$ where N is a state number bound for the primary systems $M_1, \cdots, M_n$, and α is a state number bound for the aspects $A_1, \cdots, A_m$. But the real issue is that one has to perform such an expensive calculation before the verification result on the emptiness of $L(\mathcal{A}) \cap Bad$ could be obtained (whose time complexity is $O(N^n \alpha^n |M_{Bad}|)$ where $|M_{Bad}|$ is the size of a finite automaton accepting Bad). Therefore, it is desirable to design a verification algorithm where the verification result can be established earlier (e.g., before the entire woven system $\mathcal{A}$ is calculated) whenever it is possible. To this end, we present a safety verification algorithm **verifyAOS**($\mathcal{A}$, Bad). For each node u in the tree $\mathcal{A}$, the algorithm

Algorithm 1 verifyAOS($\mathcal{A}$, Bad)

```
1: initialize(A, Bad)
2: checkNode(root)
3: return "no"          //A does have Bad transactions
```

maintains and updates a set, denoted by $u.badSet$, which is always a regular set in Σ^*. Initially (line 1), only the set in the root node, $root.badSet$, is set to be the given Bad; the sets in other nodes are all Σ^*. Then (line 2), the algorithm updates all the $badSet$s

in the tree starting from the root, during which the main algorithm **verifyAOS**($\mathcal{A}$, Bad) could halt with "yes" (i.e., $\mathcal{A}$ does not have Bad transactions) returned (otherwise, as in line 3, "no" is returned).

We now explain the procedure checkNode($root$) in line 2 in a little more detail. For each node u starting from the root, it "projects" its current $u.badSet$ down to its left child; i.e., $(u.left).badSet$ is set to be $\mathrm{Project}_A(\downarrow, \Sigma^*, u.badSet)$, where A is the aspect that u is labeled with. [1] Similarly, u also projects $u.badSet$ down to its right child. In case when u is a leaf, it intersects its current $u.badSet$ with the transaction set of the system M that u is labeled with and obtains a new $u.badSet$. Then, for each nonterminal node u (from the lowest level up to the root), the procedure will project the new $badSet$s of u's children up to u itself; i.e.,

$$u.badSet := \mathrm{Project}_A((u.left).badSet, (u.right).badSet, \downarrow)$$

It shall be noticed that, during the procedure, once a $badSet$ becomes empty (this could happen at an earlier stage of the execution), we can conclude that $\mathcal{A}$ does not have Bad transactions – no further execution of the algorithm is necessary. In the following, we present the recursive procedure checkNode(NODE u):

Procedure 2 checkNode (NODE u)

```
 1: if u is a leaf then
 2:    M be the primary system that u is labeled with
 3:    u.badSet := u.badSet ∩ L(M)
 4:    if u.badSet is empty then
 5:       return "yes"     // A does not have Bad transactions
 6:       exit     // the main algorithm verifyAOS halts
 7:    end if
 8: else
 9:    let A be the aspect that u is labeled with
10:    (u.left).badSet := Project_A(↓, Σ*, u.badSet)
11:    (u.right).badSet := Project_A(Σ*, ↓, u.badSet)
12:    checkNode(u.left)
13:    checkNode(u.right)
14:    u.badSet := Project_A((u.left).badSet, (u.right).badSet, ↓)
15:    if u.badSet is empty then
16:       return "yes" and exit
17:    end if
18: end if
```

Due to space limitation, we omit the correctness proof of the algorithm. Notice that, in our algorithm presentation, set operations, such as emptiness testing, intersection, and $\mathrm{Project}_A$, are used. In fact, one can use finite automata to represent $badSet$s and multitape finite automata to represent aspects A. It should be straightforward that all

[1] For a 2-ary aspect A, and sets X and Y, we define $\mathrm{Project}_A(\downarrow, X, Y)$ to be the set of all $w\heartsuit$ such that there are $x\heartsuit \in X$ and $y\heartsuit \in Y$ satisfying $x\heartsuit$ and $y\heartsuit$ can be woven into $w\heartsuit$ using A; i.e., $(w\heartsuit, x\heartsuit, y\heartsuit) \in A$. Accordingly, $\mathrm{Project}_A(X, \downarrow, Y)$ and $\mathrm{Project}_A(X, Y, \downarrow)$ can be defined.

these set operations can be implemented using the corresponding automata manipulations. One can also prove that, in worst case, the time complexity of our algorithm is $O(|M_{Bad}| \cdot N^n \cdot \alpha^n \cdot |M_{Bad}|^n \cdot \alpha^{n \log n})$, comparing to the naive algorithm's time complexity $O(|M_{Bad}| \cdot N^n \cdot \alpha^n)$ mentioned earlier. Notice that N (the state number in primary systems) is the dominate parameter which is usually $\gg$ all the other parameters (specifications for Bad and for aspects are typically simple and n is also small). So, as long as $N \gg$ the slow down factor $|M_{Bad}|^n \cdot \alpha^{n \log n}$, our algorithm's worst-case time complexity is the same as the naive one, not to mention the additional benefit of possible earlier termination when worst-cases do not happen.

5.2 Safety Testing Algorithm for Aspect-Oriented Systems

When some of the primary systems are black-boxes (whose state number could be infinite), the *safety testing problem* for $\mathcal{A}$ is exactly the safety verification problem for $\mathcal{A}$ in which each black-boxes M is replaced with a finite-state system M' whose transactions are exactly those in $\mathbf{GenTests}(M, \Sigma)$. We shall emphasize that, even though $\mathbf{GenTests}(M, \Sigma)$ could return a huge set of tests (such as strings on Σ not longer than 40), the safety testing problem is to seek a *definite* yes/no answer. In this case, one would follow the naive approach by first testing each black-box M using the tests generated from $\mathbf{GenTests}(M, \Sigma)$ and then replacing the M with a system whose behavior is exactly those successful tests. However, exhaustive testing of the entire test set $\mathbf{GenTests}(M, \Sigma)$ is not feasible. It is desirable to have an algorithm using the tree $\mathcal{A}$ as well as the set Bad to trim the test set $\mathbf{GenTests}(M, \Sigma)$ before actual tests are run on the M (i.e., tests on a black-box are tailored to the specific safety testing problem of $\mathcal{A}$). Furthermore, successful tests themselves are valuable information on the actual behavior of M. This information should be used to further trim away unnecessary tests performed over other black-boxes. To this end, we propose a safety testing algorithm **testAOS**($\mathcal{A}$,Bad) as follows:

Algorithm 3 testAOS($\mathcal{A}$, Bad)

```
1: initialize(A, Bad)
2: trim(root)
3: for each leaf node u labeled with a black box do
4:     propagate(root)
5:     test(u)
6:     trim(root)
7: end for
8: return "no"
```

Each node u in $\mathcal{A}$ is associated with $u.badSet$, $u.flag$ (which is $white$ or $black$), and Boolean value $u.updated$. Initially (line 1), only the set in the root node, $root.badSet$, is set to be the given Bad; the sets in other nodes are all Σ^*. Also, for each leaf node u, if it is labeled by a black-box then its flag is $black$ else the flag is $white$. The rest of initialize($\mathcal{A}$,Bad) in line 1 is to run init($root$), which is defined recursively in Procedure 4.

Procedure 4 init(Node u)

1: **if** $u.badSet$ is empty **then**
2: return "yes" and exit //the main algorithm **testAOS** halts
3: **end if**
4: **if** u is a leaf with a $white$ flag **then**
5: let M be the primary system that u is labeled with
6: $u.badSet := u.badSet \cap L(M)$
7: **if** $u.badSet$ is empty **then**
8: return "yes" and exit
9: **end if**
10: set $u.updated$ to be $True$
11: **else**
12: let A be the aspect that u is labeled with
13: $(u.left).badSet := \text{Project}_A(\downarrow, \Sigma^*, u.badSet)$
14: $(u.right).badSet := \text{Project}_A(\Sigma^*, \downarrow, u.badSet)$
15: init($u.left$)
16: init($u.right$)
17: set $u.updated$ to be $False$
18: **if** each of u's two children has a $white$ flag **then**
19: set the flag of u to be $white$
20: **else**
21: set the flag of u to be $black$
22: **end if**
23: **end if**

Procedure 5 trim(Node u)

1: **if** u is not a leaf **then**
2: trim($u.left$)
3: trim($u.right$)
4: **end if**
5: **if** u has at least a child whose $updated$ is $True$ **then**
6: let A be the aspect that u is labeled with
7: $u.badSet := \text{Project}_A((u.left).badSet, (u.right).badSet, \downarrow)$
8: **if** $u.badSet$ is empty **then**
9: return "yes" and exit
10: **end if**
11: set $u.updated$ to be $True$
12: **if** each of u's two children has a $white$ flag **then**
13: delete these two children (so u is a leaf now)
14: **end if**
15: **end if**

Roughly speaking, init($root$) recursively "projects down" the Bad set to the $badSet$ of each nonterminal node and leaf, much the same as checkNode($root$) does in **verifyAOS**. When a leaf is a $white$ primary system M, an updated $badSet$ is calculated by intersecting it with $L(M)$. Additionally, for a $black$ node, all its ancestors are also flagged $black$.

In line 2 of **testAOS**, trim($root$) "projects up" all the "updated" $badSet$s at leaf nodes to all their ancestors by updating the ancestors' $badSet$s. In the mean time, a $white$ node becomes a $white$ leaf (i.e., children are trimmed away) whenever the children are also $white$ nodes. The procedure is presented in Procedure 5.

Now, the for-loop of **testAOS** (lines 4,5,6) is to test each black-box primary system one by one. Suppose that we are currently processing black-box M that is labeled on some leaf node u. We first use propagate($root$) in line 4 to "project down" the updated $badSet$ of the root all the way to every black-box which then obtains a new (and smaller) $badSet$. Later in line 5, the black-box M at node u is tested using tests that are in both $\mathbf{GenTests}(M, \Sigma)$ and the new $u.badSet$. All the successful tests are collected and form the "updated" $u.badSet$ now. At this time, the black-box node u is flagged $white$ (the black-box M is finished processing). Finally in line 6, this newly added $white$ node u and the test results (recorded in the "updated" $u.badSet$) are used to "trim" the tree (as well as update all the $badSet$s of its ancestors). When the for-loop continues, the next black-box picked will again first "propagate" the root's updated $badSet$ (as a result of the previous black-box's test results), and so on. Details of propagate($root$) and test(u) are shown in Procedures 6 and 7.

Procedure 6 propagate(Node u)

1: **if** $u.badSet$ is empty **then**
2: return "yes" and exit
3: **end if**
4: set $u.updated$ to be $False$
5: **if** u is not a leaf and $u.left$ has a $black$ flag **then**
6: $(u.left).badSet := \text{Project}_A(\downarrow, \Sigma^*, u.badSet) \cap (u.left).badSet$
7: propagate($u.left$)
8: **end if**
9: **if** u is not a leaf and $u.right$ has a $black$ flag **then**
10: $(u.right).badSet := \text{Project}_A(\Sigma^*, \downarrow, u.badSet) \cap (u.right).badSet$
11: propagate($u.right$)
12: **end if**

Figure 4 shows an example execution of the safety testing algorithm $\mathbf{testAOS}$ over the aspect-oriented system $\mathcal{A}$ shown in Figure 4 (a), where M_1, M_2, M_3, M_4 are primary systems (in which M_2 and M_4 are black-boxes), and A_1, A_2, A_3 are behavioral aspects.

At any time when the algorithm $\mathbf{testAOS}$ runs, if the $badSet$ at some node becomes empty, then the algorithm halts and return a "yes" answer to the safety testing problem. When this happens before any black-box primary system is tested, we simply do not need test any black-box at all for the safety testing problem. When this happens after some black-boxes have already been tested, all the remaining black-boxes are not

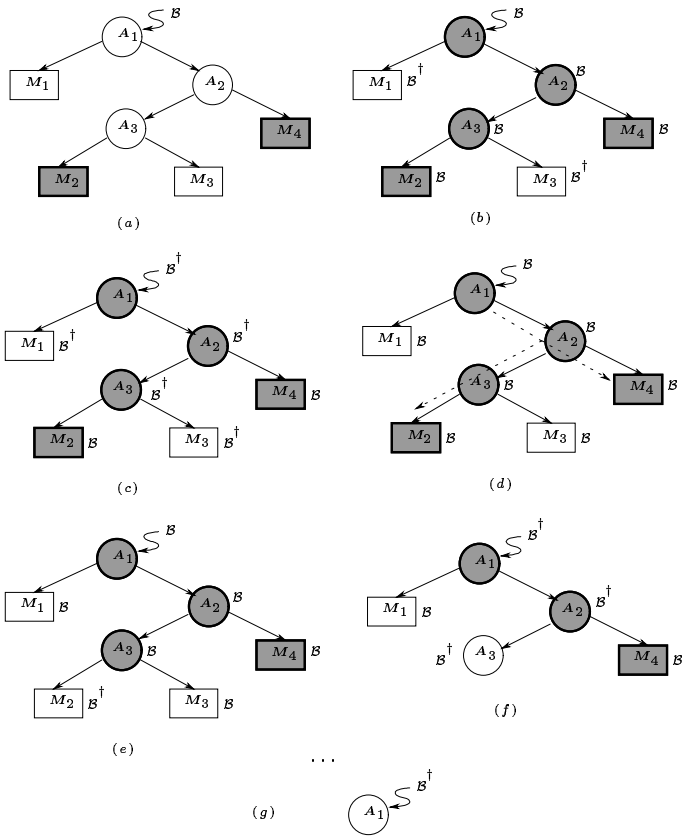

Fig. 4. An example run of safety testing algorithm **testAOS** over the aspect-oriented system in (a). The black boxes M_2 and M_4 are lightly shaded and (the root node labeled with aspect) A_1 is associated with a $badSet$ initially being the regular set Bad. In the sequel, each $badSet$ at a node is simply denoted by a special symbol $\mathcal{B}$ in the figure. (b) the result of running init(A_1) at line 1 of **testAOS**. Each non-root node is associated with a $\mathcal{B}$ using Project operations. When the node is a white-box node, i.e. M_1 or M_3, its $\mathcal{B}$ is further $updated$ by $\mathcal{B} \cap M$, which is denoted as $\mathcal{B}^\dagger$. The flag (i.e., in the figure, a shaded/clear circle corresponds to a $black$/$white$ flag) of each nonterminal node is set according to the flags of the two children. (c) The result of running trim(A_1) at line 2 of **testAOS**. The $\mathcal{B}$ of each node is updated to $\mathcal{B}^\dagger$ using Project operations if one of the children is associated with $\mathcal{B}^\dagger$. In our example, since M_3 was associated with $\mathcal{B}^\dagger$ in (b), the $\mathcal{B}$ of the parent A_3 is therefore updated to $\mathcal{B}^\dagger$. Similarly, the $\mathcal{B}$ of A_2 as well as A_1 is also updated to $\mathcal{B}^\dagger$. (d) The result of running propagate(A_1) at line 4 of **testAOS**. From (c), a new $\mathcal{B}$ is associated with the root A_1, then all shaded nodes (flagged with $black$) are associated with new $\mathcal{B}$'s recursively using Project operations starting from the root A_1. In this step, $updated$ is reset to $False$ in all nodes; i.e., all updated $\mathcal{B}^\dagger$ is renamed as un-updated $\mathcal{B}$. (e) The result of running test(M_2) at line 5 of **testAOS**. Through testing, the black-box M_2 is associated with an updated $\mathcal{B}^\dagger$ and its flag is set to $white$ (M_2, after testing, is a white-box now). (f) trim(A_1) again at line 6 of **testAOS**. Notice that the flags of A_3's children (M_2 and M_3) are both $white$ now. In this case, both children are deleted from the tree after $\mathcal{B}$ of A_3 is updated. (g) Repeat procedures from d to f (i.e., the for-loop in **testAOS**) until all the black-boxes are tested.

Procedure 7 test(Node u)

1: let M be the black-box primary system labeled on leaf u
2: **for** each test $w\heartsuit$ in **GenTests**(M, Σ)$\cap u.badSet$ **do**
3: run black-box testing **BTest**(M, $w\heartsuit$)
4: **end for**
5: set $u.badSet$ to be the set all successful tests $w\heartsuit$
6: set the flag of u to be $white$
7: set $u.updated$ to be $True$

needed to test. Also in the algorithm, procedures trim($root$) and propagate($root$) work together to make sure that, after a black-box is tested, the test results (the successful tests) are used to create a smaller test set for each of the remaining black-boxes yet to be tested.

Again, due to space limitation, the correctness prove of the algorithm is omitted. Similarly, in the algorithm, all the set operations can be implemented through automata manipulations. It is hard to conduct a precise complexity analysis for the safety testing algorithm, since the test results for a black-box affect the test sets that will be run over the other black-boxes. At least when there is no black-box, **testAOS** does not perform worse than **verifyAOS**. It is reasonable to assume that black-box testing is expensive, in particular when one exhaustively runs every test from a huge (e.g., 10^{24} in [6]) test set generated from **GenTests**. The saved testing time resulted from eliminating a large number of unnecessary tests from the test set would well make up the overhead of calculating the unnecessary tests using our algorithm **testAOS**. For instance, concurrent composition (through interleaving) can be considered as a concurrency aspect (though it is very special). The case-study performed in [6] is a very special case of our safety testing algorithm that runs over one white-box and three black-boxes and with only one 4-ary concurrency aspect (which is the root). The case-study shows that a huge test set with 10^{24} tests is reduced into a set with 10^5 tests after removing all unnecessary tests. On the other hand, state-space explosion seems unavoidable when a even larger test set is selected. In that case-study, automata manipulations (for the concurrency aspect and tests results) failed to complete. We would anticipate similar experimental results for our safety testing algorithm **testAOS**.

6 Conclusions

In this paper, we use multitape automata to model aspects and study verification and testing algorithms for an aspect-oriented system specified by a number of primary labeled transition systems (some of them are black-boxes) and aspects. Our algorithms combine automata manipulations with black-box testing over each individual black-box, but without generating the woven system.

In a forthcoming paper, we are going to implement the algorithms and perform case-studies in order to justify the real-world efficiency of the algorithms. The authors thank Anneliese Andrews and Curtis Dyreson for discussions.

References

1. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *ECOOP'97*, LNCS 1241, pages 220–242, Springer, 1997.
2. R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.
3. E. Barra, G. Gnova, and J. Llorens. An Approach to Aspect Modeling with UML 2.0. In *5th AOSD Modeling With UML Workshop*, San Francisco, California, USA, October 2004.
4. K. Cooper, L. Dai and Y. Deng. Modeling Performance as an Aspect: a UML Based Approach. In *4th AOSD Modeling With UML Workshop*, San Francisco, California, USA, October 2003.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
6. G. Xie and Z. Dang. Testing Systems of Concurrent Black-boxes: a Decompositional Approach. In *FATES'05*, LNCS 3997, pages 170–189, Springer, 2006.
7. D. Giannakopoulou, C. Pasareanu, and H. Barringer. Assumption Generation for Software Component Verification. In *ASE'02*, IEEE Computer Society, 2002.
8. M. E. Fayad and A. Ranganath. Modeling Aspects using Software Stability and UML. In *4th AOSD Modeling With UML Workshop*, San Francisco, California, USA, October 2003.
9. R. France, I. Ray, G. Georg, and S. Ghosh. An Aspect-Oriented Approach to Early Design Modeling. *IEE Proceedings - Software*, 151(4):173–185, 2004.
10. I. Hammouda, M. Pussinen, M. Katara, and T. Mikkonen. UML-based Approach for Documenting and Specializing Frameworks Using Patterns and Concern Architectures, In *4th AOSD Modeling With UML Workshop*, San Francisco, California, USA, October 2003.
11. L. Alfaro and T. A. Henzinger. Interface Automata. In *FSE'01*, pages 109–120, ACM Press, 2001.
12. M. M. Kande, J. Kienzle, and A. Strohmeier. *From AOP to UML - a Bottom-up Approach*. In *3rd AOSD Modeling With UML Workshop*, San Francisco, California, USA, October 2002.
13. D. Lee, M. Yannakakis. Principles and Methods of Testing Finite State Machines - a Survey. *Proceedings of the IEEE* 84(8):1090-1126, 1996.
14. http://www.globalfuture.com/mit-trends2001.htm
15. http://pavg.stanford.edu/rapide/rapide-pubs.html
16. M. Sihman and S. Katz. Model Checking Applications of Aspects and Superimpositions. In *FOAL'03*, Boston, Massachusetts, USA, March 2003.
17. S. Nakajima and T. Tamai. Lightweight Formal Analysis of Aspect-Oriented Models. In *5th AOSD Modeling With UML Workshop*, San Francisco, California, USA, October 2004.
18. N. Ubayashi and T. Tamai. Aspect-oriented Programming with Model Checking. In *AOSD'02*, Enschede, The Netherlands, April 2002.