http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 11 (10): 1391-1399, 2012
ISSN 1812-5638 / DOI: 10.3923/itj.2012.1391.1399

Specifications Mining Based on Adjusted Automata Learning Algorithm

Xiao Jianyu and DAI Jingguo
Department of Computer, Hunan International Economics University,
Changsha Hunan, 410003, China

Abstract: The problem of software's specifications mining was to automatically acquire temporal safety properties of a software based on its execution traces, which was represented as a DFA-enforceable security automata. It is similar to the problem of automata learning which had been studied in the field of theoretical computer science, assuming that the training set of software's trace was expressed as regular language. We proposed to use Angluin's algorithm which was a classical algorithm to learn Deterministic Finite Automata (DFA) to solve the problem specifications mining. Observing that a security automaton was a prefix-closed DFA, we adjusted the Angluin algorithm to check whether the prefixes of the target string being processed were accepted before the member query was asked, so, as to avoid unnecessary member queries. In the implementation of the Angluin algorithm, we used model checker to simulate the oracles of member queries and equivalence queries. Experiments showed that our implementation of Angluin algorithm was practical to solve the problem of specifications mining.

Key words: Specifications mining, automata learning, query oracle, enforceable security automata, prefix-closed regular language

INTRODUCTION

Software model checking (Clarke and Kroening, 2004; Kimour and Boudour, 2005; Mei *et al*., 2009; Xu and Jiangu, 2008) is a kind of formal method to verify software's temporal correctness, which has attracted much attention in recent years (Ball *et al*., 2004; Beyer *et al*., 2005; Andrews *et al*., 2004). The main obstacle for this technique to be accepted by the community of software engineering is that it is very difficult to acquire the formal specifications of software's properties (Mashiyat *et al*., 2008; Lian *et al*., 2008; Umapathi and Raja, 2008). It is not practical for common programmers to write these specifications because they don't know how to deal with this time consuming and error prone task (Medikonda and Panchumarthy, 2009). Ammons *et al*. (2002) introduced the new concept of specifications mining and proposes to use an off-the-shelf probability automata learning algorithm to automatically induce property specifications with dynamic execution traces of the assumed correct program as its input. Ammons' method is based on the frequency information of the events occurred in the program's execution traces and the number of the acquired specifications is very large. But many of the specifications induced by Ammons' method are redundant and their reliability can not be guaranteed. Whaley *et al*. (2002) proposed to append data members representing type state in the target class and then use model checking method to judge a given specification is worth to be accepted basing on whether it should trigger a selected exception. Whaley's method has a low degree of automatization which requires users to provide candidate specifications. Weimer and Necula (2005) propose to learn temporal safety properties from the information of error handling. The automatization of Weimer's method is high and the specifications of its output are simple, but their domain is very limited. The works mentioned above have contributed to solve the problem of specifications mining from different angles, but due to its difficulty and importance, the problem needs further studies (Srinivasan and Thambidurai, 2007).

Automata learning (Angluin, 1987; Fellah *et al*., 2007) (also called regular language learning) is to automatically infer an automaton's structure from the analysis of its behavior, or to say, it is to learn an unknown regular language from examples of its members and nonmembers. The problem of automata learning has been studied for over 20 years in the field of theoretical computer science and there is a basic conclusion that the problem can be solved in polynomial time if there exists an oracle which can answer membership queries and equivalence queries about the unknown regular language. The Angluin algorithm (Angluin, 1987) is the classical one for

Corresponding Author: Xiao Jianyu, Department of Computer, Hunan International Economics University, Changsha Hunan, 410003, China

Automata learning. The difficulty in the implementation of Angluin algorithm is how to implement the oracle, which has different style for different regular language. The complexity bottle of the algorithm is the number of membership queries (Gavalda, 1994).

Observing the similarities between the problem of specifications mining and automata learning, we propose to use the Angluin algorithm to mine software's temporal safety properties given the precondition that the training set of software's execution traces is a regular language. Considering the fact that a security automata which expresses temporal safety properties is a prefix-closed finite automaton, in order to reduce unnecessary membership queries, we propose to check the following two facts before membership queries in the implementation of the Angluin algorithm: (1) If a string has been accepted, all its prefixes must be accepted; (2) If a string has been rejected, all its extensions must be rejected. In the implementation of the Angluin algorithm, we propose to use model checker to simulate the oracle of membership queries and equivalence queries. Experiments showed that the adjusted Angluin algorithm is practical to solve the factual problem of specifications mining.

Software's temporal safety properties: The current tool of software model checking can not verify all kinds of the program's properties. In order to regulate the domain of specifications mining, we give the concept of program's property and a kind of its classification.

Definition 1: Software/program system: A software/program system can be represented as $S = (A,\Sigma)$, where A is the set of the program's actions (including events and operations); $\Sigma \epsilon A^*$ is the set of all possible execution traces of the program where a trace σ is a sequence of program's actions: $\{\alpha 1, \alpha 2,....\alpha n,....\}$.

Definition 2: Program's property: A property of a program is a computable predicate p defined on a single execution trace of the program system (A,Σ) which has the following form:

$$p(\Sigma) = \forall \sigma \epsilon \Sigma.\ P(\sigma)$$

where, P is the set of computable predicates on A^*.

Intuitively, program's property is a kind of policy or constraint on the actions of the program, which includes the following: (1) Access Control policies specify that no execution may operate on certain resources such as files or sockets, or invoke certain system operations; (2) Availability policies specify that if a program requires a resource during an execution, then it must release that resource at some arbitrary later point in the execution; (3) Bounded availability policies specify that if a program acquires a resource during an execution, then it must release that resource by some fixed point later in the execution; (4) Application specific constraints on the order of invocation of functions.

Definition 3: Safety property: A safety property is a kind of program's properties which specify that nothing bad happens. That is, P is a safety property of program system (A,Σ) iff for all $\sigma \epsilon \Sigma$, $\neg P(\sigma) \Rightarrow \forall \sigma \epsilon \Sigma.\ (\sigma \prec \sigma \Rightarrow \neg P(\sigma'))$.

Intuitively, once a bad action which violates the safety property has taken place there is no extension of that execution that can remedy the situation. Access control policies are typical safety properties. Another important character of safety properties is that its violation should appear in the program's finite execution traces.

Definition 4: Liveness property: A liveness property is a kind of program's properties in which nothing exceptionally bad can happen in any finite amount of time. That is, P is a liveness property of program system (A,Σ) iff for all $\sigma \epsilon \Sigma$, $\forall \sigma \epsilon \Sigma. \exists \sigma' \epsilon \Sigma. (\sigma \prec \sigma \Rightarrow P(\sigma))$.

Availability is a typical kind of liveness property.

According to Alpern and Schneider (1987), any property can be decomposed into the conjunction of a safety property and a liveness property.

Definition 5: Exception predicate: An exception predicate is a predicate defined on the set of program's error states.

Definition 6: Temporal safety property: A Temporal safety property is a subset of safety properties, which usually specify legal order of relevant operations and can be specified through exception predicates.

Essentially, temporal specifications of operations on system's resources are the principal part of a program. Violations of temporal safety properties can lead to fatal system error such as synchronization error (deadlock, resource competition etc.), memory leakage and null pointer reference. According to Schneider (2000), temporal safety properties are enforceable properties which can be enforced through a monitor paralleling with the target program. Whenever the target program wishes to execute a security relevant operation, the monitor first checks its policy to determine whether or not that operation is allowed. If the operation is allowed, the target program continues operation and the monitor does not change the program's behavior in any way. If the operation is not allowed, the monitor terminates execution of the program.

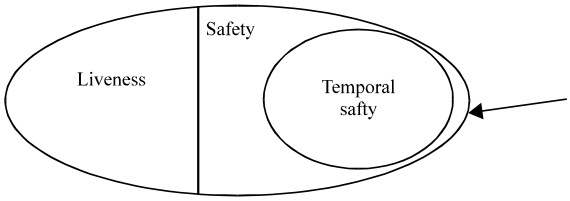

Fig. 1: The Classification of software's properties

The inclusion relation among software properties defined in this paper can be illustrated in Fig. 1.

Software's properties can be specified in two kind of languages: Temporal logic (Sistla, 1994) or Buchi automata (Vardi, 1996). Temporal logic is the common language to specify properties used in currently available model checkers, which includes Linear Temporal Logic (LTL) and Computational Temporal Logic (CTL). Buchi automata are strongly expressive which can cover program's safety properties. The security automata defined below is a subset of Buchi automata. According to Latvala (2003), temporal logic formula and Buchi automata can be translated from each other.

Definition 7: Security automata: A security automaton is a DFA $(Q,q0,F,\delta)$ which is defined on a program system, (A,Σ) where countable set Q specifies the possible automaton states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of acceptable states and the partial function $\delta: Q \times A \rightarrow Q$ specifies the transition function for the automaton.

The security automaton is consistent with the state-transition model of program's operational semantics. It is a simplified version of the model of program's operational semantics which captures a profile of the model. Viewing from another angle, the security automata is a recognizer of a program's actions and can enforce each temporal safety properties of a program. The currently available tools of software model checking can only verify temporal safety properties of programs now (Ball and Rajamani, 2001a). SLIC language (Ball and Rajamani, 2001b) in the SLAM (Ball *et al*., 2004) system and Query language (Beyer *et al*., 2004) in the BLAST (Beyer *et al*., 2005) system are all concrete implementations of security automata.

ANALYSIS OF ANGLUIN AUTOMATA LEARNING ALGORITHM

Basic idea of Angluin algorithm: Exact learning of the target DFA from an arbitrary presentation of labeled examples is a hard problem (Gold, 1978; Angluin, 1981). One of the main positive results in the area of computational learning is the fact that DFA can be learned from membership and equivalence queries (Angluin, 1987). In a membership query, the learning algorithm asks the oracle whether a string σ accepted by the unknown target DFA. The oracle answers yes or no. In an equivalence query, the learning algorithm asks the oracle whether a conjectured DFA is equal to the target DFA. The oracle answers yes or no, where in the later situation, it will supply a counterexample which is accepted by the conjectured DFA or the target DFA but rejected by another.

The kernel data structure of the Angluin algorithm is an observation table which is defined as follows.

Definition 8: Observation table: Let the unknown regular language accepted by the target DFA is L and assume that it is over a fixed known finite alphabet Σ. At any given time, the learning algorithm has information about a finite collection of strings over Σ, classifying them as members or nonmembers of L. This information is organized into an observation table, (S,E,T) where: (1) S is a finite set of strings and $\varepsilon \in S$; (2) E is a finite set of strings and $\varepsilon \in E$; (3) $T:(S \cup S.\Sigma)E. \rightarrow \{0.1\}$ is a finite function, defined such that $T(w) = 1$ iff $w \in L$.

In the Angluin algorithm, the observation table is expressed as a two-dimensional array with rows labeled by elements of $S \cup S.\Sigma$ and columns labeled by elements of Σ, with the entry for row s and column e equals to T(se). If $s \in (S \cup S.\Sigma$, then row(s) denotes the finite function $f:E \rightarrow \{0.1\}$ defined by $f(e) = T(se)$ for every $e \in E$.

Definition 9: Completeness: An observation table (S,E,T) is called complete, if $\forall t \in S.\Sigma$, $\exists s \in S$, row(t) = row(s).

Definition 10: Consistence: An observation table (S,E,T) is called consistent if:

$$(s1,s2 \in S, \text{row}(s_1) = \text{row}(s_2) \Rightarrow \forall \sigma \in \Sigma, \text{row}(s_1\sigma) = \text{row}(s_2\sigma)$$

The typical behavior of the Angluin algorithm is to start by asking a sequence of membership queries and gradually builds a conjectured DFA M using the obtained answers. When M is stable (the corresponding observation table satisfying completeness and consistence), it makes an equivalence query to find out whether M is correct. If the result is successful, the algorithm has succeeded, otherwise it uses the returned counterexample to revise M and perform subsequent membership queries until arriving at a new conjectured DFA, etc.

Complexity bottle of Angluin's algorithm: According to Angluin (1987), the time complexity of the Angluin algorithm is bounded by a polynomial in n, $|\Sigma|$ and m, where n is the number of states in the unknown target minimum DFA, $|\Sigma|$ is the size of the alphabet and m is the maximum length of any counterexample string presented by the query oracle. In the implementation of the algorithm, the main factor affecting its efficiency is the number of queries which is called query complexity (Gavalda, 1994) because the query oracle is implemented as a slow equipment. In the original implementation of the Angluin algorithm, the worst complexity of equivalence query is $O(n)$ and the worst complexity of membership query is $O(|\Sigma|mn^2)$. So, the complexity bottle of the Angluin algorithm is the number of membership queries. In addition, the implementations of membership query and equivalence query have no determined criterion, which is the main difficulty of the application of the algorithm.

ANGLUIN ALGORITHM TO MINE SOFTWARE'S SPECIFICATIONS

Problem of specifications mining: The problem of specifications mining is first introduced by (Ammons *et al*., 2002) whose definition is described in Def.4.1. We can see that the problem of specifications mining is similar to the problem of automata learning.

Definition 11: Problem of specifications mining: Let l be the set of all traces of interactions with an API (Application Programming Interface) or ADT (Abstract Data Type), and C⊆I be the set of all correct traces of interactions with the API or ADT. Given an unlabelled training set T of interaction traces from I, find an automaton A that generates exactly the traces in C. A is called a specification. The problem of how to find A is called the problem of specifications mining.

According to definition 8, intuitively, software's specifications we concern are the rules of temporal and data-dependence relationships that a program follows when it interacts with an application programming interface (API) or abstract datatype (ADT). These rules are concisely summarized as state machines that capture both temporal and data dependence. The specifications mining problem can not be solved if there is no restrictions placed on the set C. If C is not recursively enumerable, then A does not exist (Sipser, 1997). Because what we concern is the specifications of the program's temporal safety properties which can be verified by currently available software model checkers and there is only ready-made algorithm for learning DFA in the field of automata learning, we give the following hypothesis.

Hypothesis 1: C is regular language and A is DFA in the problem of specifications mining defined in definition 11. According to Ammons (2003), although most of the execution traces of a program can not form a regular language, its traces contain subtraces that do. So, the hypothesis 1 is basically reasonable. On the precondition of hypothesis 1, the related works of Hagerer *et al.* (2002) and Groce *et al.* (2002) which also attempt to generate automata model of software system by algorithms originally for automata learning, we conclude that the problem of specifications mining can be approximately solved by the Angluin algorithm.

THE CONSISTENCE BETWEEN SECURITY AUTOMATA AND TEMPORAL SAFETY PROPERTIES

We consider a class in a object oriented program, where $C(M,V,V_r, Init, S,T_m)$ is the finite set of methods' names, V is the finite set of variables in the class, V_r is the set of valuations of return variables of all the methods, Init is the initial predicate on V, S is the set of states predicates on V, T_m is the set of transitions predicates on V related to method m. What we concern here is the temporal safety rules a program must obey when it interacts with the class C, so as to keep C from the error states defined by a exception predicate E. When a program interacts with the class E, it invokes a sequence of methods $\{m_1,.....,m_l)$ and gets return values $\{v_1,....v_l\}$ where $v_{1\le i\le l}\in V_r$. Essentially, a temporal safety specification related to C is a function $I:(M\times V_r)^*\to 2^M$, where the input is a history of interaction and the output is a set of methods that can be safely called after this interaction. The program's trace $\pi = S_0, m_0,s_1,m_1,....(S_{i\ge 0}\in S,mi_{\le 0}\in M)$ which obeys the specification I should satisfy the following: (1). $S_0| = Initm_0\in I(\varepsilon)$; (2) $\forall i \ge 0, s_l \xrightarrow{m_i} S_{i+1}$ (3) $\forall I\ge 0, m_{i+l}\in I((m_0,S_0[V_r]),..,(m_i,S_i[Vr])$ According to Def.2.6, a temporal safety property can also be specified as an exception predicate. The trace $\pi = S_0,m_0,S_l,m_l,...$ is considered safe with respect to the exception predicate E iff it satisfies: $\forall i\ge 0, s_i|\ne E$. The temporal safety specification I on the class C is considered safe with respect to the exception predicate E iff all the traces obeying I is safe with respect to E.

Aiming at class $C(M,V,V_r,Init, S,T_m)$, we can define a security automata $J = (Q,q_0,F,\delta)$ on the input alphabet, (M,V_r) where, Q = S representing the set of C's states, q_0 = Init representing C's initial state, F⊆S representing the final states accepted by C and $\delta: S\times(M\times V_r)\to S$ is the set of transition predicates and satisfies $\delta \subseteq \bigcup_{m\in M} T_m$. Intuitively,

if $\delta(q,(m,r)) = q$, it means that when at state q, the specifications corresponding to the target security automata allow the invocation of method m and if it receives the return value r, it moves to state q.

We can use the example of a Java class Signature from Weimer and Necula (2005) to illustrate the consistence between temporal safety specifications and security automata. The Signature class provides the functionality of a digital signature algorithm which has five methods: initSign (), initVerify (), sign (), verify () and update (). Users sign by invoking sign () and check the input signature using verify (). Both operations need initialization via initSign () and initVerify (), respectively. Once such initialization method is invoked, the user can also update the data to be signed or verified by update (). The security automata expressing the rule illustrated in Fig. 2, where $Vr = \varnothing$ and we use M instead of $M \times V_r$. All states in the figure are final and missing transitions indicate disallowed calls. For instance, I(initSighn, initVerify) specifies the set of allowed methods to be invoked, that is just all methods on the outgoing transitions from q_1: {initSign, initVerify, update, verify}.

From the consistence between security automata and temporal safety properties, we can conclude that if we can get a regular training set of a program's traces, then we can automatically learn the security automata that accepts the regular traces through the Angluin algorithm which just represents the program's temporal safety specifications.

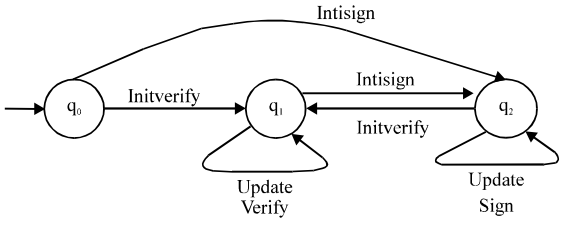

Fig. 2: The security automaton of class signature

IMPROVEMENT OF ANGLUIN ALGORITHM FOR MINING SPECIFICATIONS AND ITS IMPLEMENTATION

Improvements of Angluin Algorithm for Mining Specifications: Def.5.1 Prefix: Σ is an alphabet. If $x,y,z \in \Sigma^*$ and $x = yz$ then y is called a prefix of X, represented as $y \prec x$ or $x \succ y$.

Definition 12: Prefix-closed regular language: A regular language L with the input alphabet Σ is called prefix-closed if it satisfies $\forall \alpha \in \Sigma^*$, $\forall \sigma \in \Sigma, \alpha\sigma \in L \Rightarrow \alpha \in L$. The DFA accepting a prefix-closed regular language is called a prefix-closed DFA.

As discussed earlier, a security automaton representing a program's temporal safety properties is a simplified version of the state-transition model of the program's operational semantics where every states reachable from the initial state is the system's accepted states. So, security automata are prefix-closed DFA and the training set of traces are prefix-closed regular language. The characteristics of a prefix-closed language can be summarized as: (1) $x \in L(A) \Rightarrow \forall y \prec x, y \in L(A)$; (2) $x \notin L(A) \Rightarrow \forall y \succ x, y \notin L(A)$. These two characteristics can be utilized to improve the Angluin algorithm to reduce the query complexity. When studying a string that is possibly accepted by the target prefix-closed DFA, we can make the following simple but important observations: (1) prefixes of accepted strings are accepted, no membership queries are needed, (2) suffixes of rejected strings are rejected, no membership queries are needed. The pseudo code of the adjusted Angluin algorithm is illustrated in Fig. 3, where the conjectured DFA M(S,E,T) is computed as follows: (1) the set of states Q = {row(s):s$\in$S (2) the initial state q_0 = row(ε); (3) the set of accepting states F = {row{row(s):s$\in$S$\wedge$T(s) = 1}; (4) the transition function δ(row(s),σ) = row(sσ) = row (sσ).

```
Initialization: S←E←{ε};
Ask me membership queries for each σ∈Σ;
Construct the intial observation table {S, E, T};
Repeat
While {S, E, T} is not complete or not consistent DO
    If ∃s1, s2∈S, ∃σ∈Σ, ∃e∈E, row(s1) = row(s2) and T (s1σe)≠T(s2σe)
    Then W←EU{σe} and extend T using prefix/suffix checking or membership queries;
    If ∃s1∈S, ∃σ∈Σ⇒∀s∈S, row(s1σ)≠row(s1)
    Then S←S∪{s1σ} and extended T using prefix/suffix checking or membership queries; OD
```

Fig. 3: The pseudo code of the improved Angluin algorithm

Implementation of Angluin's algorithm to mine specifications: The implementation of the Angluin algorithm aiming at specifications mining has to fulfill the following two tasks: (1) Preparation of the regular training set of software's execution (sub)traces, that is, the extraction, simplification and standardization of software's traces; (2) Implementations of query oracles including membership query and equivalence query.

Software trace's extraction, simplification and standardization: Extraction of software's traces can be conducted dynamically or statically. In the method of dynamic trace extraction, the source code is first instrumented in which some sentences (usually print commands) inserted to record the input/output data of the concerned API/ADT. Then a selected set of data is input to the program and the traces of interactions to the API/ADT are acquired during the run of the program. Ammons (2003) gave the detail of a scheme of dynamic trace extraction. In the method of static trace extraction, the potential sequence of legal operations is acquired by static analysis of the program's source code, which does not depend upon input data. In study of Eisenbarth *et al.* (2002), a detailed description of the procedure of static trace extraction is given. In our algorithm, traces of a program are only used as input data, and there is no limit on how to get them.

The recorded data of a program's execution traces are usually very large and most of which are irrelevant to the concerned properties' specifications. Flow dependence analysis is often utilized to simplify the recorded data of traces (Ferrante *et al*., 1987). Flow dependencies connect elements that change the state of a variable (that is, elements that define the object) to elements that depend on the value of the variable (that is, elements that use the variable), which helps to decide which functions/methods should be considered in a group, which functions/methods should be considered separately and which should not be considered at all. From another angle, flow dependence analysis is a kind of static analysis of programs which can be combined in the procedure of static trace extraction (Eisenbarth *et al*., 2002).

Due to the requirement of the Angluin algorithm that the input strings representing traces of a program must be regular expressions (Pradel and Gross, 2009), the simplified traces should be further processed to be standardized, that is, they should be expressed as the regular expression defined on the alphabet ($M\times V_r$) (take the class $C(M,V,V_r,Init,S,T_m)$ as the example) with the form $\{(m_0,r_0), (m_1,r_1),\ldots,(m_n,r_n)\}$.

Implementation of oracle of membership query: Take the $C(M,V,V_r, Init, S,T_m)$ class as the example with the exception predicate being E and the target security automaton to be learned being A. Given a string of a standardized trace, the oracle of membership query should answer the following question:

- Whether $\sigma\in L(A)$.....(Question-1)

Because A is unknown, question-1 can not be answered directly. Due to the fact that L(A) is a prefix-closed regular language, our adjusted learning algorithm has excluded the possibility of any prefixes of σ not belonging to L(A). So, we can construct a DFA B that just accepting σ and all of its prefixes. The Question-1 can now be transformed to the following question:

- Whether $L(B)\subseteq L(A)$.....(Question-2)

Question-2 is also difficult to answer. Because L(A) includes all the strings representing the safe traces with respect to E, question-2 can be transformed to the following question:

- Whether B is a safety specification with respect to E?.....(Question-3)

A model checker can answer question-3. We construct a model $B\|C$ representing the interactions between B and C[26], whose state space is defined as $S_{B\|C} = \{t_B,t_C\}\times M\times S\times Q$. A state means that at state $s = (t_C,m,s_C,q)\in S_{B\|C}$ class C has turn, the currently executing method is m and the class C and the automaton B are in states S_A and q, respectively. Transitions of $B\|C$ are defined as follows: (1) Initially, B has turn and selects a legal method m_0 which obeys the rule of r_0 to be executed. Then, it passes turn to the class C; (2) When the class C gets turn, the method m_0 is simulated and the return value r_0 is gotten. The system's state is now changed to be ($s = (tC,m_0,S_C,[r_0],q)$) from the original state ($s = (t_C, m_0,S_C,q)$) and then the turn and r_0 are passed to B; (3) B changes the system's state from $s = (t_B,m_0,S_C[r_0],q)$ to $s = (t_C,m_0,S_C[r_0].q)$ where and picks a new legal method m_1. It then passes turn to the class and the interactions continue as before. To answer the question-3, a model checker checks whether all the sequences of method calls and return values allowed in B keep the class C away from states that satisfy E by checking whether $B\|C$ satisfies the LTL temporal logic formula $\Box(\neg E)$. If the model checker answers yes, then the answer to the Question-1 (viz. the membership query) is true. Otherwise, the answer to the membership query is false.

Implementation of oracle of equivalence query: We again take the $C(M,V,V_r)Init,S,T_m)$ class as the example with the exception predicate being E and the target security automaton to be learned being A. Given a conjectured DFA D, the oracle of equivalence query should answer the following question:

- Whether D = A?.....(Question-4)

Due to the fact that A is unknown, question-4 can not be answered directly. We can transform the question to:

- Whether L(D⊆L(A)and L(D)⊇LA?....(Question-5)

In question-5, the problem of L(D)⊆L(A) deciding the satisfiability. In order to answer the equivalence query, the oracle need only to decide the satisfiability of L(D)⊇L(A). That is to say, if L(D)⊆L(A) is true, question-1 can be transformed to:

- Whether L(D)⊇L(A)?.....(Question-6)

According to [6], question-6 is NP-hard. Question-6 can also be described as:

If, $\forall\sigma\in(M\times V_r)^*$ whether $\sigma\notin L(D)$? That is, for each sequence of method calls $\sigma\notin L(D)$, does there exist some runs of class C which lead states of C to satisfy exception predicate:

- E?..... (Question-7)

In order to answer question-7, we adopt a method of conservative approximation which poses a stronger constraint on question-7:

For each sequence of method calls:

- Whether all runs of class $\sigma\notin L(D)$ which lead states of C to satisfy exception predicate E?.....(Question-8)

If the answer to question 8 is true, the answer to the equivalence query (question-1) is true. Otherwise, the equivalence query can not be answered precisely now and we can assume that the answer is false and returns σ as the counterexample. A model checker can answer question-8. We construct a model D‖C representing the interactions between D and C (Alur *et al*., 1998), whose state space is defined as $S_{D|C} = \{t_D,t_C\}\times M\times S\times Q\times L$ where $L = \{0,1\}$ is the set of legal bits which indicates whether the currently selected method obeys the rule of D. Transitions of D‖C are defined as follows: (1) Initially, D has turn and selects a method m_0, if m_0 is legal with respect to D, the legal bit of l∈L is set to 1, otherwise 0; (2) When the class C has turn, if l = 1, the method m_0 is simulated with the return value r_0 being gotten and the system's state is changed from to $s = (t_C,m_0,s_C,q_1)$ tos = $(t_C,m_0,s_C[r_0],q1)$. If, l = 0,D‖C will stop running after the simulation of m_0 on C; (3) If D gets back turn with the return value r_0, it will change the system's state to be $s = (t_C,m_C,s_C[r_0],q,1)$ where, $q = \delta(q, (m_0,r_0)$ pick a new method m_1, sets legal bit l according to whether m_1 obeys the rule of D and pass turn to the class and the interactions continues as before. To answer the question-8, a model checker checks whether D‖C satisfies the LTL temporal logic formula $\Box((l = 0)\Rightarrow\Diamond E)$. If the model checker answers yes, then the answer to the Question-4 (viz. the equivalence query) is true. Otherwise, the model checker will return a sequence of method calls σ as the counterexample of the equivalence query.

EXPERIMENT

The aim of the experiment is to test the practicability of our adjusted Angluin algorithm to solve the real problem of specifications mining. We select five samples of state-transition systems: the Peterson mutex protocol (P-Mutex), a simple buffer control protocol (Buf), a simple ATM system (ATM), a Java class Signature and a Java class ListItr. The number of the sampling systems' states is between 2 and 20 and the length of input alphabet is between 2 and 6. The experiment consists of three steps: (1) The training set of strings of sampling systems' standardized traces is generated according to the procedure described earlier, where we use the tool of static trace extraction (Eisenbarth *et al*., 2002). (3) Take the training set of strings as input and use our adjusted Angluin algorithm whose pseudo code is illustrated in Fig. 3 to learn the corresponding automaton, where the oracles of membership and equivalence queries are implemented as runs of model checking with the tool of model checking being Mocha (Alur *et al*., 1998). The number of member queries and equivalence queries and the running time of our algorithm (not including the running time of the two oracles) are recorded. (3) We take the output automaton of step 2 with respect to the Java class Signature as the input property of the software model checker BLAST (Beyer *et al*., 2005) to verify an application program which uses Signature.

The experiment is performed using a 2.4 GHz Pentium processor with 256 M RAM. The O S is RedHat Linux 9.0. The result is shown in Table 1 and in step 3, we find three sequences of method calls which violate the specification.

Table 1: The adjusted Angluin's algorithm to solve the real problem of specifications mining

Sampling systems	No. of states	Length of alphabet	No. of membership queries	No. of equivalence queries	Running time (msec)
P-Mutex	2	3	5	1	427
Buf	6	2	110	2	965
ATM	20	6	968	8	58064
Signature	11	4	564	5	39783
ListItr	9	3	412	5	20021

The experiment results shows that our adjusted Angluin algorithm can acquire software's temporal safety specifications in acceptable running time and reasonable queries complexities.

CONCLUSIONS

In this study, we propose to use the classic Angluin automata learning algorithm in the field of theoretic computer science to solve the problem of specifications mining, in which we give a new scheme of implementations of oracles of membership and equivalence queries. Considering the fact that the security automata which express the temporal safety properties of a program accept prefix-closed language, we propose a scheme to adjust the Angluin algorithm to reduce its complexity of membership query. The experiment shows that the adjusted Angluin algorithm is practical to undertake the real task of specifications mining.

The study of automata learning is still active now. A recent paper (Lo, 2010) proposes an scheme to improve the Angluin algorithm based on domain-specific knowledge, and another paper (Goues and Weimer, 2009) proposes to make use of method of artificial intelligence to reconstruct algorithms of automata learning. Our future work will investigate the applicability of these new algorithms to solve the problem of specifications mining.

ACKNOWLEDGMENTS

A project supported by Hunan Provincial Natural Science Foundation of China (07JJ3119) and the Planned Science and Technology Project of Hunan Province (2011SK3146).

REFERENCES

Alpern, B. and F. Schneider, 1987. Recognizing safety and liveness. Distrib. Comput., 2: 117-126.

Alur, R., T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani and S. Tasiran, 1998. MOCHA: Modularity in model checking. Proc. Int. Conf. Computer-aided Verication (CAV), LNCS, 1427: 521-525.

Ammons, G., 2003. Strauss: A specification miner. Ph.D Thesis, University of Wisconsin, Madison

Ammons, G., R. Rodik and J.R. Larus, 2002. Mining specifications. Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2002, January 16-18, 2002, Portland, OR, USA.

Andrews, T., S. Qadeer, S.K. Rajamani, J. Rehof and Y. Xie, 2004. Zing: A model checker for concurrent software. Comp. Aided Verifica., Vol. 3114. 10.1007/978-3-540-27813-9_42

Angluin, D., 1981. A note on the number of queries needed to identify regular languages. Inf. Control, 51: 76-87.

Angluin, D., 1987. Learning regular sets from queries and counterexamples. Inf. Comput., 75: 87-106.

Ball, T. and S.K. Rajamani, 2001a. Automatically validating temporal safety properties of interfaces. Proceedings of the 8th International SPIN Workshop on Model Checking of Software, May 19-20, 2001, Toronto, Canada, pp: 103-122.

Ball, T. and S.K. Rajamani, 2001b. SLIC: A specification language for interface checking (of C). TechReport, MSR-TR-2001-21, Pages: 12, http://research. microsoft. com/apps/pubs/default. aspx?id=69906

Ball, T., B. Cook, V. Levin and S.K. Rajamani, 2004. SLAM and static driver verifier: technology transfer of formal methods inside microsoft. Integrated Formal Methods, LNCS, 2999: 1-20.

Beyer, D., A.J. Chlipala, T.A. Henzinger, R. Jhala and R. Majumdar, 2004. The blast query language for software verification. Proceedings of the 11th International Static Analysis Symposium, Aug. 26-28, Verona, Italy, Springer-Verlag, pp: 2-18.

Beyer, D., T.A. Henzinger, R. Jhala and R. Majumdar, 2005. Checking memory safety with blast. Proc. Int. Conf. Fundamental Approaches Software Eng. LNCS, 3442: 2-18.

Clarke, E.M. and D. Kroening, 2004. Tutorial: Software model checking. Lect. Notes Comput. Sci., 3308: 9-10.

Eisenbarth, T., R. Koschke and G. Vogel, 2002. Static trace extraction. Proceedings of the Working Conference on Reverse Engineering, IEEE Computer Society Press, October, 2002, Washington, DC, USA.

Fellah, A., Z. Friggstad and S. Noureddine, 2007. Deterministic timed AFA: A new class of timed alternating finite automata. J. Comput. Sci., 3: 1-8.

Ferrante, J., K. Ottenstein and J. Warren, 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9: 319-349.

Gavalda, R., 1994. The complexity of learning with queries. Proceedings of the 9th IEEE Structures in Complexity Theory Conference, IEEE Press, Junr 28-July 1, 1994, University Politecnica de Catalunya, Barcelona, pp: 324-337.

Gold, E.M., 1978. Complexity of automaton identification from given data. Inf. Control, 37: 302-320.

Goues, C.L. and W. Weimer, 2009. Specification mining with few falsepositives. Proceedings of the 15th International Conference Tools and Algo-rithms for the Construction and Analysis of Systems, March 22-29, 2009, York, UK, pp: 292-306.

Groce, A., D. Peled and M. Yannakakis, 2002. Adaptive model checking. Int. Conf. Tools Algorithms Construction Anal. Syst., 2280: 357-370.

Hagerer, A., H. Hungar, O. Niese and B. Steffen, 2002. Model generation by moderated regular extrapolation. Int. Conf. Fundamental Approaches Software Eng., LNCS, 2306: 80-95.

Kimour, M.T. and R. Boudour, 2005. SYMTC: Towards a symbolic model checking for the codesign. Asia J. Inform. Technol., 4: 1055-1060.

Latvala, T., 2003. Efficient model checking of safety properties. Proc. Int. SPIN Workshop Model Checking Software, LNCS, 2648: 74-88.

Lian, W., H. Fan, Y.Y. Du and Y. Liang, 2008. Petri net methods of constructing Kleene-closure operations of regular languages. Inform. Technol. J., 7: 689-693.

Lo, D., 2010. Scenario-based and value-based specification mining: Better together. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, September 20-24, 2010, Antwerp, Belgium, pp: 387-396.

Mashiyat, A.S., K.H. Talukder and R. Rahman, 2008. Design and implementation of a model of a specification language for formal verification. Res. J. Anim. Sci., 3: 288-293.

Medikonda, B.S. and S.R. Panchumarthy, 2009. An approach to modeling software safety in safety-critical systems. J. Comput. Sci., 5: 311-322.

Mei, J., H. Miao and P. Liu, 2009. Applying SMV for security protocol verification. Infom. Technol. J., 8: 1065-1070.

Pradel, M. and T.R. Gross, 2009. Automatic generation of object usagespecifications from large method traces. Proceedings of the 24th International Conference Automated Software Engineering, November 16-20, 2009, Auckland, New Zealand, pp: 371-382.

Schneider, F.B., 2000. Enforceable security policies. ACM Transa. Inf. Sys. Secur., 3: 30-50.

Sipser, M., 1997. Introduction to the Theory of Computation. 1st Edn., PWS Publishing Company, Boston.

Sistla, A.P., 1994. Safety, liveness and fairness in temporal logic. Formal A. Comput., 6: 495-511.

Srinivasan, N. and P. Thambidurai, 2007. On the problems and solutions of static analysis for software testing. Asia J. Inform. Technol., 6: 258-262.

Umapathi, C. and J. Raja, 2008. Discovering frequent patterns and trends by applying web mining technology in web log data. Int. J. Soft Comput., 3: 99-105.

Vardi, M.Y., 1996. An automata-theoretic approach to linear temporal logic. Logics Concurrency, LNCS, 1043: 238-266.

Weimer, W. and G.C. Necula, 2005. Mining temporal specifications for error detection. Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, April 4-8, 2005, Edinburgh, UK, pp: 461-476.

Whaley, J., M. Martin and M. Lam, 2002. Automatic extraction of object-oriented component interfaces. Proc. Int. Sympos. Software Test. Anal., Vol. 27. 10.1145/566171.566212

Xu, Z. and X. Jianyu, 2008. Combination of model checking and theorem proving to develop and verify embedded software. Inform. Technol. J., 7: 623-630.