Inferring State-based Behavior Models

Davide Lorenzoli
Università di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy
lorenzoli@disco.unimib.it

Leonardo Mariani
Università di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy
mariani@disco.unimib.it

Mauro Pezzè
Università di Milano Bicocca
via Bicocca degli Arcimboldi, 8
20126 Milano, Italy
pezze@disco.unimib.it

ABSTRACT

Dynamic analysis helps to extract important information about software systems useful in testing, debugging and maintenance activities. Popular dynamic analysis techniques synthesize either information on the values of the variables or information on relations between invocation of methods. Thus, these approaches do not capture the important relations that exist between data values and invocation sequences.

In this paper, we introduce a technique, called GK-tail, for generating models that represent the interplay between program variables and method invocations. GK-tail extends the k-tail algorithm for extracting finite state automata from execution traces, to the case of finite state automata with parameters.

The paper presents the technique and the results of some preliminary experiments that indicate the potentialities of the proposed approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*tracing, debugging aids*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software maintenance*

General Terms

Algorithms, Verification

Keywords

dynamic analysis, inference algorithms, behavioral models, finite-state machines, verification.

1. INTRODUCTION

Many software systems present state-dependent behavior, i.e., the results of the execution depend on the internal state of the application. This is the case of many common components and web services that store the information required for the computation. For instance, several popular shopping cart web services record the choices of the customers and behave accordingly [1].

State-dependent behavior introduces new analysis problems: failures may depend on the internal state of the software component, and thus analysis must be able to distinguish different states [7]. For instance, removing an item from a shopping cart may work properly if the item is in the cart, but may fail if the item is not in the cart, if the cart is empty, if the amount of items that we try to remove from the cart is greater than the amount currently in the cart, and so on. Analysis must consider the different behaviors that may derive from different internal states.

Dynamic analysis provides useful information for understanding programs, identifying discrepancies between expected and actual behavior, diagnose faults, manage changes, and compare executions in different contexts [15, 9, 16]. Current techniques for analyzing the run time behavior of programs provide useful information about the value of program variables and sequences of actions, but do not extract integrated information about program states, parameters and transitions [6, 3, 2].

In this paper, we propose a technique for automatically extracting models of state-dependent computation from sequences of program executions. The models are expressed in terms of finite state automata extended with parameters (FSAP). Finite state automata capture states and transitions; parameters associated with transitions indicate the dependencies between transitions and values of the program variables in different states. For example, a model of the dynamic behavior of a shopping cart may indicate that method `create-cart` has always executed first in the analyzed executions, or that method `insert-item` has always executed with a positive quantity.

FSAP models of program behavior can be used in many ways, e.g., to understand the behavior of the program, to check if the actual behavior is consistent with expectations, to compare the behavior observed during testing against the behavior observed in the field and deduce limitations of testing or anomalous uses of the program, to match the behavior of different components [11].

In the next section, we discuss the problems of dynamically extracting state-dependent models from program executions, present the overall approach, and discuss the limits of the many existing algorithms for inferring finite state automata when used for dynamically analyzing program be-

WODA'06, May 23, 2006, Shanghai, China.

haviors. Section 3 defines the finite state automata with parameters, and illustrates how they can describe the behavior of software systems. Section 4 introduces the GK-Tail algorithm that we propose to infer FSAP from sequences of program executions. Section 5 presents early results obtained by applying GK-Tail on simple examples. Section 6 outlines our research agenda.

2. SYNTHESIZING MODELS OF STATEFULL PROGRAMS

Statefull executions are captured as sequences of actions, e.g., sequences of method invocations with corresponding parameters. Monitoring even small software systems produces an enormous quantity of traces that are hard to store and interpret. Dynamic analysis aims at synthesizing general and compact models from sets of traces, thus reducing long term storing requirements and facilitating interpretation and analysis of the collected data. Finite state automata are a simple and efficient formalism for capturing statefull behavior.

Dynamic analysis techniques for synthesizing finite state automata must take into account the specific properties of the domain. Synthesis algorithms can count on many "positive samples", i.e., sequences of actions that have been recorded by program monitors, and must be represented by the synthesized automata. They can count neither on "negative samples", i.e., sequences that must not be represented by the synthesized automata[1] nor on additional information about the expected results, in the form of "teachers", or ordered samples.

Efficient dynamic analysis techniques can take advantage from the regularity of the applicative environment: methods cannot be invoked in any order and with arbitrary values for the parameters, but follow precise design and implementation rules. Thus, we expect many subsequences shared among several traces and relations between parameters and method invocations. For example, in a shopping cart service, we expect many subsequences of the form $\langle create, add \ldots add, get\text{-}price \rangle$, and a specific relation between the price returned by the get-price method and the cost of the items added to the cart. Many existing analysis techniques focus on either one of these aspects. Some approaches, e.g., Daikon, provide useful information on the relation between the values of the variables at different execution points, but do not consider invocation sequences [6, 17]. Other approaches synthesize models of invocation sequences independently from the value of the parameters [9, 15]. Modeling the relations between method invocations and parameter values provides additional information that can be very useful for understanding and analyzing the program behavior.

In this paper, we propose a method for generating finite state models of program behaviors in the form of finite state automata augmented with information about the values of the parameters in the different states. The method borrows from algorithms for generating finite state automata and uses Daikon for deriving constraints on parameter values [6].

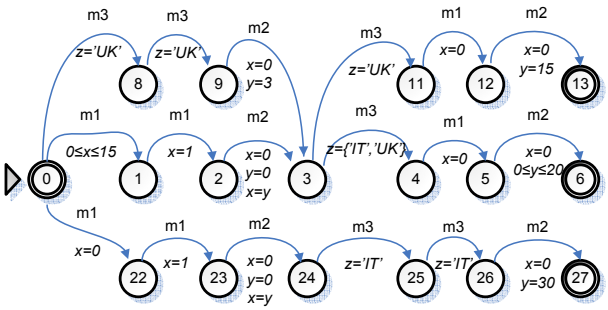
There are many algorithms for deriving finite state automata from sets of traces. Several are based on hypotheses that are not satisfied in the application domains, e.g., the presence of negative samples [5], the availability of teachers [12] or information on the order of the traces [13]. We developed our algorithm by extending the k-Tail algorithm and its many variants that work well on positive samples only [3, 4, 14]

The *GK-Tail* algorithm proposed in this paper generates finite state automata augmented with parameters (FSAP) from sets of program traces in three steps. It first merges traces that correspond to the same sequences of method invocations albeit with different parameter values. Then, it identifies the constraints on the values of the parameters that characterize the invocation of methods in different states. Finally, it iteratively merges states that have the same k-future, i.e., that are followed by the same behaviors up to a depth of k steps.

3. FSAP

Classic finite state automata can capture the dependency of method invocations from the state of the components, but not the constraints on the values of the parameters. In this section, we define *Finite State Automata with Parameters (FSAP)* and *interaction traces*. *FSAPs* extend classic finite state automata (FSA) to model constraints on the values of the parameters. *Interactions traces* represent sequences of method invocations, and formalize the concept of execution traces.

Figure 1 shows a simple *FSAP*: transitions are associated with method names and constraints. Names indicate the methods that can be invoked in each state; constraints indicate the values allowed for the parameters of the method in the different state.



LEGEND
The state with the triangle is the initial state and the states with the thick border are final states.

Figure 1: A simple Finite State Automaton with Parameters (FSAP).

Formally, a (non-deterministic) *finite state automaton with parameters* is a 8-tuple $(Q, \Sigma, D, F, \delta, \varphi, q_0, Q_E)$, where

- Q is a finite non-empty set of states,
- Σ is a finite non-empty set of input symbols,
- D is an n-dimensional space $D_1 \times ... \times D_n \cup \{\emptyset\}$,

[1]Sequences that expose program failures may be considered negative samples, but are special cases, since they are actual albeit incorrect executions, and not impossible execution sequences.

- F is a set of *enabling functions* f_i, $f_i : D \to \{0, 1\}$,
- δ is the *transition function*, $\delta : Q \times \Sigma \times D \to \wp(Q)$,
- φ is the *selecting function*, $\varphi : Q \times \Sigma \to F$ with
  - $\forall q \in Q, Q' \subseteq Q \setminus \{\emptyset\}, m \in \Sigma, d \in D,\ \delta(q, m, d) = Q' \Rightarrow \varphi(q, m)(d) = 1$
  - $\forall q \in Q, m \in \Sigma, d \in D$ s.t. $\varphi(q, m)(d) = 1 \Rightarrow \exists Q' \subseteq Q \setminus \{\emptyset\},\ \delta(q, m, d) = Q'$
- q_0 is the initial state, $q_0 \in Q$,
- Q_E is the set of the final states, $Q_E \subseteq Q \setminus \{\emptyset\}$.

Q is the set of the states of the software component.

Σ is the set of methods that can be invoked. For instance, Σ = {ClassA.m1, ClassB.m2, ClassB.m3, ClassC.m4}.

D is the set of tuples of parameter values. Each method in Σ corresponds to a set of tuples, e.g., the set of tuples of the method $ClassA.m1(p1 : int, p2 : string)$ is the cartesian product $\mathbb{N} \times String$.

F is the set of conditions that can be associated with transitions. For instance, if `addItem(id, qt)` is one of the methods that can be invoked, and $String \times \mathbb{N}$ is its domain, a transition labeled with `addItem` can be associated with the enabling condition $qt > 0$.

δ models transitions between states: $\delta(q_1, m, x) = q_2$ indicates that the invocation of a method m in a state q_1 with parameters x brings the FSAP in the new state q_2. If δ maps every triple state-symbol-input to a single state, the automaton is deterministic.

φ is a function that represents the dependency between a pair state-transition and the corresponding constraint. For instance, φ(`empty cart`, `addItem`) $= (qt > 0)$ indicates that method `addItem(id, qt)` can be invoked in state `empty state` only with positive quantities.

We now define *interaction traces*, which denote sequences of method invocations as pairs of sequences. The first element is a sequence of method names, and the second is the sequence of values of the parameters in the invocations. Figure 2 shows an example of invocation sequences and Figure 3 shows the corresponding interaction traces.

```
 m1    ->   m1    ->  m2   ->   m3    ->   m1    ->  m2
x=0         x=1        x=0       z='IT'       x=0        x=0
                       y=0                               y=0

 m3    ->   m3    ->  m2   ->   m3    ->   m1    ->  m2
z='UK'      z='UK'      x=0       z='UK'       x=0        x=0
                       y=3                               y=15

 m1    ->   m1    ->  m2   ->   m3    ->   m1    ->  m2
x=15        x=1        x=0       z='UK'       x=0        x=0
                       y=0                               y=15

 m1    ->   m1    ->  m2   ->   m3    ->   m3    ->  m2
x=0         x=1        x=0       z='IT'       z='IT'     x=0
                       y=0                               y=30
```

Figure 2: Examples of invocation sequences.

Formally, let Σ^* and D^* be the set of input strings over Σ and the set of sequences of n-dimensional tuples, respectively. Given $\alpha \in \Sigma^*$, $|\alpha|$ denotes the length of α; similarly, given $\beta \in D^*$, $|\beta|$ denotes the length of β.

An interaction trace t is a pair (t_Σ, t_D) where $t_\Sigma \in \Sigma^*$, $t_D \in D^*$ and $|t_\Sigma| = |t_D|$. If $t_1 = (t^1_\Sigma, t^1_D)$ and $t_2 = (t^2_\Sigma, t^2_D)$, we say that

```
it1 (<m1,m1,m2,m3,m1,m2>, <0,1,(0,0),'IT',0,(0,0)>)
it2 (<m3,m3,m2,m3,m1,m2>, <'UK','UK',(0, 3),'UK',0,(0,15)>)
it3 (<m1,m1,m2,m3,m1,m2>, <15,1,(0,0),'UK',0,(0,20>)
it4 (<m1,m1,m2,m3,m3,m2>, <0,1,(0,0),'IT','IT',(0,30)>)
```

Figure 3: Interaction traces corresponding to the invocation sequences shown in Figure 2

- $t_1 = t_2$ iff $t^1_\Sigma = t^2_\Sigma \wedge t^1_D = t^2_D$
- $t_1 =_\Sigma t_2$ iff $t^1_\Sigma = t^2_\Sigma$.

For instance, $it1 \neq it3$ and $it1 =_\Sigma it3$ hold for traces in Figure 3.

The transition function δ applies to single invocations, and not to interaction sequences. We thus define δ^* that extends δ to interaction sequences. δ^* is computed by iteratively extracting the first invocation from the input interaction trace and processing it with δ.

δ^* can be inductively defined as follows:

- $\delta^*(q, \pi, \phi) = q$, if π is an empty sequence of symbols and ϕ is an empty sequence of tuples;
- $\delta^*(q, i\alpha, x\beta) = \delta^*(\delta(q, i, x), \alpha, \beta)$ if $q \in Q$, $i \in \Sigma$, $\alpha \in \Sigma^*$, $x \in D$, and $\beta \in D^*$.

An interaction trace is accepted by a FSAP, if it leads to a final state. Formally, an interaction trace (α, β) is *accepted* by an FSAP $(Q, \Sigma, D, F, \delta, \varphi, q_0, Q_E)$ if $\delta^*(q_0, \alpha, \beta) \cap Q_E \neq \emptyset$. For instance, all traces shown in Figure 3 are accepted by the FSAP in Figure 1.

4. THE GK-TAIL ALGORITHM

We synthesize *FSAP* from interaction traces by matching "similar" traces and "equivalent" states.

Traces are similar if they correspond to the same sequence of method invocations, independently from the values of the parameters. Intuitively, similar traces represent the same behavior pattern with different data values. GK-Tail merges similar traces into a single interaction trace, to obtain a general representation of the behavior pattern. The set of possible parameter values are indicated by constraints associated with transitions. GK-Tail works incrementally starting with an initial FSAP obtained by linking all interaction traces to a common initial state, as shown in Figure 7.

States are equivalent if they have the same future, i.e, the FSAs rooted in equivalent states generate the same set of behaviors. Equivalent states cannot be distinguished by an external observer, thus they can be conservatively merged into a unique state without altering the behavior of the FSA. The future of a state is often infinite, hence a finite number of traces seldom allow conservative merging of states. To overcome the problem of partial observation of future behaviors, we approximated the future of a state by considering only behavior of length k, where k is a parameter of the technique. Fixing the length of the considered future allows the identification of (likely) equivalent states even if a finite number of traces is available. For instance, the heuristic above allows the reconstruction of loops into the FSA even if a sequence of infinite length is not part of processed traces. This heuristic is inherited from kTail [3], where it

has been defined for standard FSA. Here, we extended the heuristic to include suitable management of conditions over parameters and state merging. Two states are merged in a new one, by linking the input and output transitions of both states to the new one. For instance, the FSAP shown in Figure 1 can be obtained from the FSAP shown in Figure 7 by merging state 10 with state 3.

m1	->	m1	->	m2	->	m3	->	m1	->	m2
x=0, x=15		x=1, x=1		x=0 y=0, x=0 y=0		z='IT', z='UK'		x=0, x=0		x=0 y=0, x=0 y=15

Figure 4: Identical traces has been merged.

The *GK-Tail* algorithm works in three steps. In the first step, we merge similar traces. In the second step, we derive constraints on the parameters of the calls annotating the transitions. In the last step, we iteratively merge all the k-equivalent states.

Step one: merge similar traces. In the first step, the algorithm merges similar traces, i.e., traces satisfying the $=_\Sigma$ relation. A merged set of interaction traces results in a data set, which is an interaction trace annotated with sets of data values, instead of single values. For instance, Figure 4 shows a data set obtained by merging traces $it1$ and $it3$ in Figure 3.

Data sets can be formally defined as pairs $ds = (t_\Sigma, ds_D)$, where $t_\Sigma \in \Sigma^*$ and $ds_D \in \wp(D^*)$. It follows that traces are data sets, and that the $=_\Sigma$ relation can be extended from traces to data sets. Merging two label-identical data sets (hence merging two traces) $ds_1 = (t_\Sigma, ds_D^1)$ and $ds_2 = (t_\Sigma, ds_D^2)$ results in the data set $ds_3 = (t_\Sigma, ds_D^1 \cup ds_D^2)$.

Step two: derive constraints. In the second step, GK-Tail generates constraints for transitions from the values collected in the data sets. A constraint generalizes and summarizes the condition under which the corresponding transition can be executed. Constraints are generated with the Daikon inference engine [6], which automatically derives relations on sets of variables. Daikon works on a set of variables, each associated with a set of values. It starts with a set of constraints syntactically legal for the considered variables, and incrementally considers the input values. At each step, it eliminates the constraints violated by the values to obtain a set of constraints satisfied by all inputs. Statistic considerations allow to identify constraints that are verified incidentally [6]. Figure 5 shows an example of how constraints can be obtained from data sets.

This inference step can be formally specified by the function INF_{DAIKON} that maps a set of values to the corresponding constraint:

$$INF_{DAIKON} : \wp(D) \rightarrow F$$

with

$$f = INF_{DAIKON}(d) \Rightarrow f(d_i) = 1 \ \forall d_i \in d$$

The INF_{DAIKON} function can be applied to data sets to replace values with conditions. Figure 6 shows the definition of the extension of the INF_{DAIKON} function to data sets.

The data shown on the left hand side are a sample of the set of input values. Daikon requires a larger set of data than the one shown to infer the equation on the right hand side. Small sets of values would produce simpler invariants that indicate the enumerative set of values.

Figure 5: Deriving constraints with Daikon.

The result of applying INF_{DAIKON} to a data set is a FSAP, modulo the following correspondence rules:

- $Q = \{q_1, \ldots, q_{|t_\Sigma|+1}\}$
- $\Sigma = \{$ labels in $t_\Sigma\}$
- $D = \bigcup(v | \exists f_i$ with $f_i(v) = 1)$
- $F = \bigcup_i f_i$
- $\varphi(q_i, t_i) = f_i \ \forall i = 1 \ldots n$
- $\delta(q_i, i, d) = q_{i+1}$ only if $i = t_i$ and $f_i(d) = 1$
- $q_0 = q_1$
- $Q_E = q_{|t_\Sigma|+1}$

Step three: merge equivalent states. In step three, we merge k-equivalent states to build a general and compact FSAP. The initial FSAP is obtained by linking all interaction traces to a common initial state. The GK-Tail algorithm compares the k-tails of the states to identify states to be merged. The algorithm considers three possible criteria: *equivalence*, *weak subsuming* and *strong subsuming*. Two states are *equivalent* if their k-tails contain the same set of behaviors annotated with equivalent constraints on transitions. A state q *strongly subsumes* a state q' if the k-tails contain exactly the same set of behaviors, and the transitions in the k-tail of q are annotated with conditions less restrictive than the corresponding conditions in the k-tail of q'. A state q *weakly subsumes* a state q' if the k-tail of q contains the k-tail of q'.

The choice of the criterion is parametric and depends on the completeness of the set of the available traces. Equivalence is preferred if the available traces are a good sample of the program behavior. Strong subsumption is advised when traces sample well the execution space, but parameters represent only a partial sample. Weak subsumption is often chose when the available sample is partial.

We merge two states q and q' into a state $\bar{q}$ with a set of input and output transitions given by the union of the input and output transitions of states q and q'. Redundant transitions are also eliminated. Transitions are redundant if they have the same label, and same input and output state,

Let $ds = (t_\Sigma, (< ds_1^1, ds_2^1, ds_3^1, \ldots, ds_n^1 >, < ds_1^2, ds_2^2, ds_3^2, \ldots, ds_n^2 >, \ldots < ds_1^m, ds_2^m, ds_3^m, \ldots, ds_n^m >))$, $INF_{DAIKON}(ds) = (t_\Sigma, < INF_{DAIKON}(ds_1^1 \cup ds_1^2 \cup \ldots, \cup ds_1^m), INF_{DAIKON}(ds_2^1 \cup ds_2^2 \cup \ldots \cup ds_2^m), \ldots, INF_{DAIKON}(ds_n^1 \cup ds_n^2 \cup \ldots \cup ds_n^m) >) = (t_\Sigma, < f_1, f_2, \ldots, f_n >) \in (\Sigma^*, F^*)$, where F^* denotes sequences of enabling conditions.

Figure 6: The extension of the INF_{DAIKON} function to data sets.

and one constraint implying the other. We eliminate the transition with the less general constraint.

The merging process terminates when all equivalent states are merged, modulo the chosen relation.

As an example, Figure 8 shows the FSAP obtained by reducing the FSAP in Figure 7 with $k = 2$ and weak subsumption. The final FSAP is obtained by incrementally merging states 22 with 1, 23 with 2, 10 with 3, and 11 with 4.

As an example of merging, consider states 2 and 23.

2-Tail(2)=$\{\langle m2, m3\rangle, \langle x = 0 \wedge y = 0 \wedge x = y, z =$"$IT$"$\vee z =$"$UK$"$\rangle\}$

2-Tail(23)=$\{\langle m2, m3\rangle, \langle x = 0 \wedge y = 0 \wedge x = y, z =$"$IT$"$\rangle\}$.

The two tails share the same invocation sequence: $\langle m2, m3\rangle$. The constraints sequence of state 2 subsumes the sequence of state 23[2], in fact

$x = 0 \wedge y = 0 \wedge x = y \Rightarrow x = 0 \wedge y = 0 \wedge x = y$, and
$z =$"IT"$\Rightarrow z =$"IT"$\vee z =$"UK".

The two states can thus be merged.

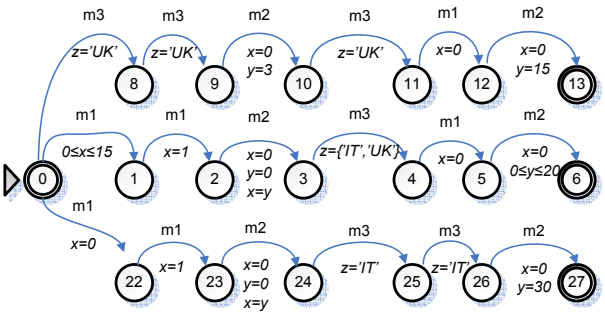


Figure 7: Merging of the initial state.

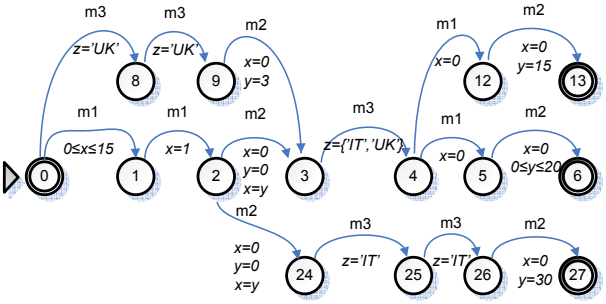


Figure 8: Further state merging.

[2]In this case the subsumption is both weak and strong.

Formally, the k-Tail of a state q, denoted with $tail^k(q)$ is equal to $((t_\Sigma^1, F^1), (t_\Sigma^2, F^2), \ldots, (t_\Sigma^n, F^n)) \in \wp(\Sigma^k, F^k)$ iff $\forall i = 1 \ldots n \; \exists q_i' \in Q$ and $\beta \in D^k$ such that $\delta^*(q, t_\Sigma^i, \beta) = q_i'$ and $f_i(\beta_i) = 1 \; \forall i = 1 \ldots k$. In other words, a k-tail of q represents the set of behaviors of length k that can be generated from that state.

Let us consider

$tail_1^k(q) = ((t_\Sigma^1, F^1), (t_\Sigma^2, F^2), \ldots, (t_\Sigma^{n_1}, F^{n_1}))$, and
$tail_2^k(q') = ((s_\Sigma^1, G_1^1), (s_\Sigma^2, G^2), \ldots, (s_\Sigma^{n_2}, G^{n_2}))$,

we say that

- (weak subsumption) $tail_1^k(q) =_{\Rightarrow w} tail_2^k(q')$ iff $\forall i = 1 \ldots n_1, \exists j \in 1 \ldots n_2, s.t., t_\Sigma^i = s_\Sigma^j$ and $F^i \Rightarrow G^j$;
- (strong subsumption) $tail_1^k(q) =_{\Rightarrow} tail_2^k(q')$ iff $n_1 = n_2$ and $tail_1^k(q) =_{\Rightarrow w} tail_2^k(q')$;
- (equivalence) $tail_1^k(q) =_{\Leftrightarrow} tail_2^k(q')$ iff $n_1 = n_2$ and $\forall i = 1 \ldots n_1 \; \exists j \in 1 \ldots n_2, s.t., t_\Sigma^i = s_\Sigma^j$ and $F^i \Leftrightarrow G^j$.

Note that $F^i \Rightarrow G^j$ iff $F^i = f_1 \ldots f_k$, $G^j = g_1 \ldots g_k$, and $f_u \Rightarrow g_u \; \forall u = 1 \ldots k$; $F^i \Leftrightarrow G^j$ iff $F^i \Rightarrow G^j$ and $G^j \Rightarrow F^i$.

5. EARLY EXPERIMENTS

To demonstrate the effectiveness of the GK-Tail algorithm in inferring state- and parameter-dependent behaviors, we apply the technique to an implementation of the Builder design pattern [8]. We selected this design pattern because it represents an interesting set of state-dependent applications, and guarantees a reasonable degree of generality, since it can be found in many commercial systems. Moreover, the design pattern is simple enough to allow manually execute some steps of the algorithm that can be automatized but have not been implemented in tools yet.

We applied the GK-Tail algorithm to the pattern with $k = 2$ and weak subsumption. A value of k between 2 and 4 is recognized as a reasonable choice for state-based inference in k-Tail in absence of additional information [14]. We choose 2 because the analyzed system is small. We use weak subsumption because we apply the algorithm to a limited set of execution traces. To evaluate the GK-Tail algorithm, we built also a classic FSA using k-Tail algorithm, and we compared the two FSAs.

In the implementation considered in this case study, we use the Builder design patter to implement the system that assembles a PC configuration. For simplicity, we suppose that there are only three types of PCs: server station, workstation and game station. Each type of PC can have many configurations, e.g., a server can be assembled using different kind of CPUs and motherboards. A typical execution of this system (shown in Figure 9) involves a Client, which asks for a particular configuration, a Director, which manages the assembling process, and a ServerBuilder, which creates the configuration. We record interactions between the Director and the ServerBuilder and we automatically build a model of these interactions.

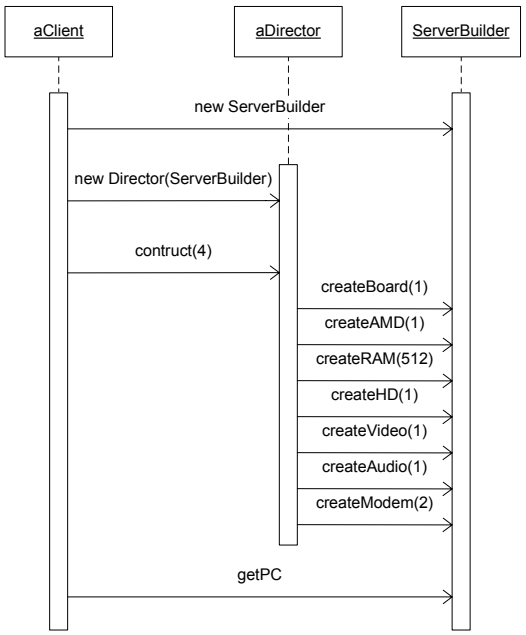


Figure 9: A sequence diagram of a typical execution of the PC assembling process.

Figure 10 shows a sample of the traces obtained by executing a set of test cases. Each trace indicates the sequence of invoked methods indicated with an intuitive shorthand, and the values of the parameters and the variables relevant for the execution and recorded by the monitor: `c` indicates the value of the parameter of method `construct` (its scope is the entire execution), and `t` is the value of the parameter of the method of `ServerBuilder`.

```
cBrd -> cAMD -> cRAM -> cHD -> cVid -> cAud -> cMdm
t=1     t=1     t=512   t=1     t=1     t=1     t=2
c=3     c=3     c=3     c=3     c=3     c=3     c=3

cBrd -> cAMD -> cRAM -> cHD -> cVid -> cAud -> cLAN
t=1     t=1     t=512   t=1     t=2     t=2     t=1
c=3     c=3     c=3     c=3     c=3     c=3     c=3

cBrd -> cInt -> cRAM -> cHD -> cVid -> cAud -> cLAN
t=1     t=1     t=512   t=2     t=1     t=2     t=1
c=3     c=3     c=3     c=3     c=3     c=3     c=3

cBrd -> cAMD -> cAMD -> cRAM -> cHD -> cVid -> cLAN
t=1     t=1     t=1     t=512   t=2     t=1     t=1
c=2     c=2     c=2     c=2     c=2     c=2     c=2
```

Figure 10: Invocation sequences

Figure 11 and Figure 12 show the FSAP produced by GK-Tail and the FSA produced by k-Tail, respectively.

The FSA produced by k-Tail contains less states and transition than the FSAP generated by GK-Tail, but is less accurate, i.e., the FSA accepts more sequences that do not correspond to feasible execution traces than the FSAP. For instance, the FSA accepts the sequence `<createBoard`, `createAMD`, `createRAM`, `createHD`, `createVideo`, `createLAN>`, which cannot be observed from the implementation, and is not accepted by the FSAP. Moreover, the FSA accepts sequences independently from the value of the parameter `choice`, thus we cannot use the FSA to recognize problems

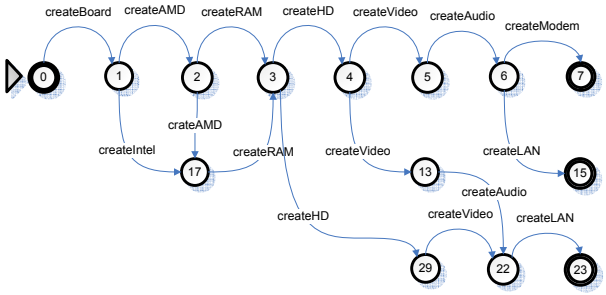


Figure 11: FSM obtained with the K-Tail algorithm

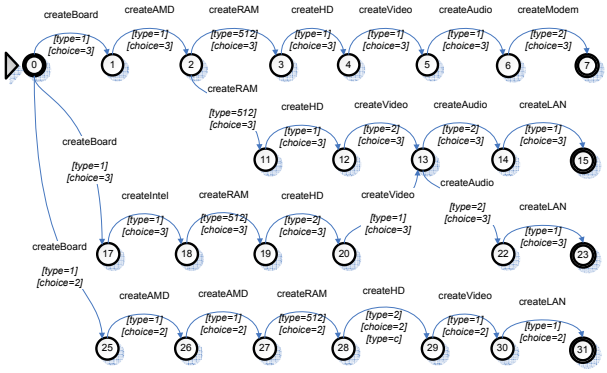


Figure 12: EFSM obtained with the GK-Tail algorithm

such as the sequence associated to `choice=3` has been erroneously executed with `choice=2`. The lack of accuracy of the FSA depends on the information lost by ignoring dependencies on the parameter values. This information can be particularly valuable when the inferred model is used to check correctness of the implementation and to identify behavioral differences among different version of the same component. Both automata accept all feasible program behaviors.

In this particular case, the language generated by the FSAP produced by GK-Tail fully corresponds to the set of possible behaviors of the program. This excellent result depends on the accuracy of the test suite and on the simple design of the program. In general, the FSAP approximates the program behavior, i.e., can accepts sequences that do not correspond to feasible traces, and can refuse feasible traces. This latter case happens when the traces produced by the test suite are not a complete sample of the execution space. Although the merging mechanism may result in different acceptance sets between the two approaches, and may coincidentally favor one over the other, e.g., because a non-sampled behavior is coincidentally accepted due to the approximation of the merging, in general we expect more accurate results of the FSAP than of the FSA.

6. CONCLUSIONS

Our experience with the analysis of component based systems suggests that state-dependent behaviors are common and important. In previous work, we experimented with techniques that infer two distinct models, one for the the

constraints on the values and the other for the constraints on invocations sequences [11]. The experience with two distinct models provided useful information about the interplay between values and invocation sequences, and suggested investigating joint models.

This paper presents a method for inferring integrated models, and reports preliminary results that confirm the needs of such new models and the effectiveness of the algorithm. Experiments have been conducted with prototypes that implement only some phases, but require non-trivial manual elaboration. We are currently developing an integrated prototype to be able to run large scale experiments to fully validate the approach.

The results of the analysis depends on the accuracy of the test cases. The possibility of choosing different criteria to merge states help us tuning the approach and overcome limitations of poor test suites. We are studying the relations between the criteria to generate models and the quality of the test suites, and the possibility of integrating the technique with methods for refining test suites [10, 17].

7. REFERENCES

[1] Amazon. Amazon web services. www.amazon.com/gp/aws/landing, 2006.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *proceedings of the 29th Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.

[3] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computer*, 21:592–597, June 1972.

[4] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[5] P. Dupont. Incremental regular inference. In L. Miclet and C. Higuera, editors, *proceedings of the 3rd International Colloquium on Grammatical Inference*, volume 1147 of *LNCS*, pages 222–237. Springer-Verlag, 1996.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[7] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Computing Series. Addison-Wesley Professional, 1995.

[9] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, May 2003. ACM Press.

[10] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *in Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.

[11] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[12] R. Parekh and V. Honavar. An incremental interactive algorithm for regular grammar inference. In L. Miclet and C. Higuera, editors, *proceedings of the 3rd International Colloquium on Grammatical Inference*, volume 1147 of *LNCS*, pages 238–250. Springer-Verlag, 1996.

[13] S. Porat and J. Feldman. Learning automata from ordered examples. *Machine Learning*, 7:109–138, 1991.

[14] S. P. Reiss and M. Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001.

[15] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*, Portland, USA, May 2003. ACM Press.

[16] T. Xie and D. Notkin. Exploiting synergy between testing and inferred partial specifications. In *proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, May 2003. ACM Press.

[17] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006 (to appear).