Automatic Generation of Software Behavioral Models

Davide Lorenzoli
Department of Informatics, Systems and Communication
viale Sarca, 336
20126 Milan, Italy
lorenzoli@disco.unimib.it

Leonardo Mariani
Department of Informatics, Systems and Communication
viale Sarca, 336
20126 Milan, Italy
mariani@disco.unimib.it

Mauro Pezzè
University of Milano Bicocca
Department of Informatics, Systems and Communication
viale Sarca, 336
20126 Milan, Italy
pezze@disco.unimib.it

ABSTRACT
Dynamic analysis of software systems produces behavioral models that are useful for analysis, verification and testing.

The main techniques for extracting models of functional behavior generate either models of constraints on data, usually in the form of Boolean expressions, or models of interactions between components, usually in the form of finite state machines. Both data and interaction models are useful for analyzing and verifying different aspects of software behavior, but none of them captures the complex interplay between data values and components interactions. Thus related analysis and testing techniques can miss important information.

In this paper, we focus on the generation of models of relations between data values and component interactions, and we present GK-tail, a technique to automatically generate extended finite state machines (EFSMs) from interaction traces. EFSMs model the interplay between data values and component interactions by annotating FSM edges with conditions on data values. We show that EFSMs include details that are not captured by either Boolean expressions or (classic) FSM alone, and allow for more accurate analysis and verification than separate models, even if considered jointly.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids, Monitors, Tracing

General Terms
Verification, Algorithms

Keywords
Model synthesis, Dynamic analysis, GK-tail

1. INTRODUCTION
Dynamic analysis techniques are extensively used to generate models that support testing and verification of software systems. Dynamic models of software behavior are used to dynamically detect anomalous behaviors [30, 12], statically and dynamically check compatibility between software components [19, 18], verify protocols [2], generate test cases [13, 17], capture unexpected event sequences [34] and verify program properties [8, 20].

The main techniques for dynamically extracting models of software behavior focus on either constraints on data values or interactions between software components. Techniques that focus on data values, for instance Daikon, extract models in the form of Boolean expressions [10]; techniques that focus on component interactions extract models in the form of Finite State Machines (FSMs) [5, 6].

Both data and interaction models are useful for analyzing and verifying specific aspects of system execution: Daikon describes important properties of the monitored variables, and FSM inference engines capture and generalize component interaction patterns. However, none of these models captures the complex interplay between data values and component interactions.

Let us consider, for example, a set of traces recorded from a generic implementation of the builder design pattern. The sequence of methods that are invoked when an object is instantiated depends on the values of the parameter \( p \) passed to method \( \text{build} \). For instance, when invoked with value \( A \) for \( p \), method \( \text{build} \) may invoke methods \( m1 \) and \( m2 \) in this order, while when invoked with value \( B \) for \( p \), it may invoke methods \( m3 \) and \( m4 \). Daikon can automatically generate models that indicate the range of values of the parameter of method \( \text{build} \), for instance, \( p \in \{ A, B \} \); algorithms for inferring FSMs can capture the set of invocations sequences triggered by invoking method \( \text{build} \), for instance \( \{ m1, m2, m3, m4 \} \); however, neither class of models, even if considered jointly, can capture the relation between data values and invocation sequences, that is \( \langle m1, m2 \rangle \) is invoked with value \( A \) for \( p \), while \( \langle m3, m4 \rangle \) is invoked with value \( B \) for \( p \).

The information not fully captured by existing dynamic models limits the precision of related analysis and verification techniques. For example, in the case illustrated above, verification techniques based on classic models cannot detect incorrect invocation sequences triggered by legal values of parameters. For instance, they would not detect an invocation of method \( \text{build} \) with value \( A \) for \( p \) that incorrectly invokes methods \( m3 \) and \( m4 \), since Daikon would notice a legal value for \( p \), and FSM-based techniques would notice a
legal invocation sequence. Even when models of both types are used jointly, verification techniques would not signal the problem, since different models capture the constraints on the data values and the invocation sequences, but not the interplay among them.

The complex interplay between data values and invocation sequences, or more generally event sequences that may depend on data values, can be effectively modeled by extending finite state machines with annotations on edges that capture the constraints on data values that characterize the invocations in the different states. Automatically generating Extended Finite State Machines (EFSMs) from execution samples is an extremely challenging problem, which has been addressed to some extent by Berg et al. [4]. The approach by Berg et al. has limited applicability due to strong requirements, like the possibility of performing membership and equivalence queries and the support limited to Boolean parameters only.

In this paper, we approach the problem of generating EFSMs from execution samples from a general viewpoint. We introduce GK-tail, an algorithm that generates EFSMs from execution samples by combining classic algorithms for generating FSMs and Daikon. The main contributions of our work are

- the GK-tail algorithm, an approach that generates EFSMs from execution samples without particular limitations on the analyzed system,
- a prototype for Java programs that allows to evaluate the approach,
- a preliminary evaluation of the applicability and early data on the scalability of the approach through the analysis of an initial set of sample applications,
- a preliminary evaluation of using dynamically inferred EFSMs instead of simple FSMs for test case selection.

The overall ideas underlying GK-tail have been outlined in a preliminary presentation at the Workshop on Dynamic Analysis (WODA) in 2006 [16]. This paper completes the early ideas by defining the model and approach in all details, and by providing some experimental evidence of its applicability.

The paper is organized as follows. Section 2 introduces models for representing EFSMs and traces. Section 3 presents the GK-tail technique. Section 4 discusses the preliminary empirical work on several applications of different size and complexity and analyzes the effectiveness of the different configurations of GK-tail when addressing inference of complex models. Section 5 investigates the use of EFSMs for testing purposes. Section 6 discusses related work. Section 7 concludes summarizing the main contributions of this paper and ongoing related research work.

2. EXTENDED FINITE STATE MACHINES

In this section, we introduce models for interaction traces and extended finite state machines. These models are used in the next section to define the GK-tail algorithm.

Interaction traces represent sequences of method invocations annotated with values for parameters and variables. Data are parameters, variables or other information related to the invocation of methods, and their respective values.

For example, an invocation to method addUser(String user, String pwd) can be associated with data user="John", pwd="MyPwd", numUsers=4 and timestamp=1099999999333 that indicate the current values of the parameters user and pwd, of the object variables numUsers, and of other associated information, in this case timestamp that records the time of the method invocation.

Other information can be added either automatically by monitoring platforms or manually by testers. Interaction traces can be automatically extracted from program executions by several monitoring and logging platforms, such as Aspectwerkz [1], TPTP [15], and Apache Log4j [8].

We do not impose specific requirements on the structure of interactions traces, like maximum length, absence of cycles and determinism. The only requirement is that each interaction trace is sequential. For instance, if a concurrent application is monitored, interaction traces are recorded either thread-by-thread, thus each thread produces a different interaction trace, or by application, thus invocations issued by different threads are sequentially recorded in a unique interaction trace.

Monitoring even small software systems produces enormous amounts of traces that require huge amount of memory and are hard to analyze and interpret. Techniques for synthesizing models generate compact and general models from sets of traces, to reduce long term storage requirements, and to simplify the analysis and interpretation of the data. EFSMs can suitably represent sequences of method invocations annotated with data.

In a previous work [16], we introduced Finite State Automata with Parameters (FSAPs), to represent execution sequences together with parameter values. In this paper, we use Extended Finite State Machines (EFSMs) that extend FSAPs by including context variables and have been widely used in modeling and testing [28].

Since GK-tail generates predicates without distinguishing between input and output parameters, differently from classic definitions of EFSMs, we do not need to distinguish inputs from outputs.

Let $X$ be a finite set of methods, $R$ a finite set of parameters, $V$ a finite set of variables that represent extra information associated with methods, for instance system variables, $D_R$ a set of evaluation domains for parameters and $D_V$ a set of evaluation domains for variables. Given a method $x \in X$, $R_x \subseteq R$ is the set of input parameters associated with $x$ and $D_{R_x} \subseteq D_R$ is the evaluation domains of the parameters in $R_x$. For instance, if $x$ is method addUser, $R_{addUser}$ is the set of parameters of addUser, $(\text{user, pwd})$, and $D_{R_{addUser}}$ is the set of possible values of the parameters $(\text{user, pwd})$, for instance the set $\text{String} \times \text{String}$.

An Extended Finite State Machine (EFSM) over $X$, $R$, $V$ and the associated evaluation domains is a tuple $(S, T, s_0, s_F)$, where $S$ is a finite set of states, $T$ is a finite set of transitions between states in $S$, $s_0 \in S$ is the initial state, and $s_F \subseteq S$ is the set of final states. A transition $t \in T$ is a tuple $(s, x, P, s')$, where

- $s, s' \in S$ are the source and destination states of the transition, respectively.

\[1\] If input and output values are recorded separately, GK-tail generates distinct predicates for input and output parameters. This extension can be obtained by classifying predicates according to the variables used in their expressions.
• \(x \in X\) is the method invocation modeled by the transition.

• \(P : D_{Rx} \times D_v \rightarrow \{True, False\}\) is the predicate of the transition; it specifies the values of input parameters and variables accepted by the method modeled by the transition.

For example, a transition \(t\) can be associated with the method \(x=\text{addUser}\) and the predicate \(P=\text{length(user)} \geq 0 \land \text{length(pwd)} > 0\) to indicate the invocation of the method \(\text{addUser}\), and the constraints on its parameters.

An EFSM can be:

• **predicate complete** if for each transition \(t = (s, x, P, s')\) and for each value \(r \in D_{Rx} \times D_v\), there is at least one transition \(t' = (s, x, P', s'')\) for which \(P'(x, r)\) is True. That is, for each method that can be invoked from a state and for each value of the parameters and variables associated with this method, it exists at least a transition enabled by these values.

• **input complete** if, for each pair \((s, x) \in S \times X\) exists at least one transition from state \(s\) with input \(x\). That is, all methods can be invoked in all states for at least some values of parameters and variables.

• **deterministic** if the predicates of all pairs of transitions exiting the same state are mutually exclusive with respect to any input \(x\). That is, there is at most one transition that exits any state with a given method invocation and a set of values for parameters and local variables.

In general, EFSM models of software systems are non-deterministic, since states may model different conditions that may enable different method invocations. For example the invocation of method \(m_1\) with parameter \(x = 0\) in the initial state of the EFSM in Figure 6 triggers 2 transitions, since the system may later evolve in two different ways represented by different paths in the EFSM. While a complete EFSM model of a software system can be predicate and input complete, when all methods can be invoked in all states with all possible combinations of values for parameters and variables, the EFSMs dynamically generated by the GK-tail algorithm are neither input nor predicate complete, since they model the executions observed with a finite amount of inputs that are a subset of all possible executions.

We now define **interaction traces**. Given a finite set of methods \(X\), a (possibly empty) set of input parameters \(R\) and corresponding domains \(D_{Rx}\), and a (possibly empty) set of variables \(V\) and corresponding domains \(D_v\), a **parametrized trace** is a tuple \((x, p_x, v)\), where \(x \in X\), \(p_x \in D_{Rx}\), and \(v \in D_v\). An **interaction trace** it is a sequence of parametrized traces \(it = (x_1, p_{x_1}, v_1) \ldots (x_n, p_{x_n}, v_n)\), where \(x_i \in X\), \(p_{x_i} \in D_{Rx}\), and \(v_i \in D_v\). Figure 1 shows some examples of invocation sequences. Figure 2 shows the corresponding set of interaction traces.

We define three equality criteria between interaction traces. Given \(it_1 = (x_1, p_{x_1}, v_1) \ldots (x_n, p_{x_n}, v_n)\) and \(it_2 = (z_1, p_{z_1}, w_1) \ldots (z_m, p_{z_m}, w_m)\), we say that:

\[it_1 \equiv_{\text{input-equal}} it_2\ (it_1 \text{ input-equal to } it_2)\]

iff \(n = m\) and \(\forall i = 1 \ldots n, x_i = z_i\)

\[it_1 \equiv_{\text{param-equal}} it_2\ (it_1 \text{ parameter-equal to } it_2)\]

iff \(n = m\) and \(\forall i = 1 \ldots n, x_i = z_i\) and \(p_{x_i} = p_{z_i}\)

\[it_1 \equiv_{\text{equal}} it_2\ (it_1 \text{ equal to } it_2)\]

iff \(n = m\) and \(\forall i = 1 \ldots n, x_i = z_i\) and \(p_{x_i} = p_{z_i}\) and \(v_i = w_i\)

![Figure 1: An example of four invocation sequences. Annotations below labels indicate parameter values (above the line) and context variables (below the line). Context variables are reported only when present.](image1)

![Figure 2: The interaction traces corresponding to the invocation sequences shown in Figure 1. We omit parameter names for simplicity.](image2)
(2) generate predicates associated with traces, (3) create an initial EFSM, and (4) merge equivalent states to obtain the final EFSM.

As defined in the former section, input-equivalent traces are invocation sequences that differ on data values only. Intuitively, a set of input-equivalent traces represents a behavioral pattern, that is the same sequence of methods invoked with different inputs. Figure 3 shows some interaction traces that correspond to the behavioral pattern that adds a new user. In the first step, GK-tail merges input-equivalent traces to create a unique trace annotated with multiple data value.

### Deriving Predicates

In the second phase, GK-tail generates predicates from data sets. A predicate generalizes and summarizes the conditions under which the corresponding method invocation can be accepted. GK-tail uses Daikon to generate the predicates [10].

Daikon works on a set of (variable, value) pairs, and automatically generates relations that are satisfied by all input pairs. The generated relations are filtered by probability thresholds to exclude incidental relations.

In our framework, we can formally define Daikon as a function $\text{Daikon}: (D_{Rs}, D_V) \rightarrow P$, where $D_{Rs}$ is a set of input parameter valuations for any $x \in X$, $D_V$ is a set of valuations for context variables and $P: D_{Rs} \times D_V \rightarrow \{\text{True}, \text{False}\}$ is a predicate on parameters and/or context variables. Daikon guarantees that if $\text{Daikon}(D_{Rs}, D_V) = P$, then $\forall p \in D_{Rs}$ and $d \in D_V, P(p_d, d) = \text{True}$. Figure 5 shows an example of the predicates that can be automatically generated with Daikon.

**Figure 3:** An example of two input-equivalent interaction traces.

In the second step, GK-tail generates predicates associated with transitions from sets of multiple data values. For instance, the set of data values associated with method addUser in Figure 3, is characterized by strings of length greater than or equal to 4 and 5 as values for the parameters user and pwd, respectively.

In the third step, GK-tail combines interaction traces annotated with predicates into a tree-like structure.

In the fourth step, GK-tail iteratively merges equivalent states, that are states with the same “tail”. States have the same “tail” if they cannot be distinguished by looking only at the outgoing sequences. Hereafter, we describe in details the four phases.

### Merging Input-Equivalent Traces

Interaction traces represent sets of executions. Several interaction traces often correspond to a same behavioral pattern, that is the same sequence of methods invoked with different values for the parameters. GK-tail takes advantage from this characteristic to compact input-equivalent interaction traces into data sets.

A data set is defined as a sequence $(x_1, d_{p1}, d_{v1}) \ldots (x_n, d_{pn}, d_{vn})$, where $x_i \in X$ is a method, $d_{pi} \in \Phi(D_{Rs})$ is a set of input parameter evaluations, and $d_{vi} \in \Phi(D_V)$ is a set of variable evaluations. Given a set $h_1 = (x_1, d_{p1}, d_{v1}) \ldots (x_m, d_{pm}, d_{vm})$, the action traces often correspond to a same behavioral pattern, that is the same sequence of methods invoked with different inputs. Figure 3 shows some interaction traces that correspond to the behavioral pattern that adds a new user. In the first step, GK-tail merges input-equivalent traces to create a unique trace annotated with multiple data value.

**Figure 4:** Examples of data sets obtained by merging input-equivalent interaction traces from Figure 1.

In the second step, GK-tail generates predicates associated with transitions from sets of multiple data values. For instance, the set of data values associated with method addUser in Figure 3, is characterized by strings of length greater than or equal to 4 and 5 as values for the parameters user and pwd, respectively.

In the third step, GK-tail combines interaction traces annotated with predicates into a tree-like structure.

In the fourth step, GK-tail iteratively merges equivalent states, that are states with the same “tail”. States have the same “tail” if they cannot be distinguished by looking only at the outgoing sequences. Hereafter, we describe in details the four phases.

### Deriving Predicates

In the second phase, GK-tail generates predicates from data sets. A predicate generalizes and summarizes the conditions under which the corresponding method invocation can be accepted. GK-tail uses Daikon to generate the predicates [10].

Daikon works on a set of (variable, value) pairs, and automatically generates relations that are satisfied by all input pairs. The generated relations are filtered by probability thresholds to exclude incidental relations.

In our framework, we can formally define Daikon as a function $\text{Daikon}: (D_{Rs}, D_V) \rightarrow P$, where $D_{Rs}$ is a set of input parameter valuations for any $x \in X$, $D_V$ is a set of valuations for context variables and $P: D_{Rs} \times D_V \rightarrow \{\text{True}, \text{False}\}$ is a predicate on parameters and/or context variables. Daikon guarantees that if $\text{Daikon}(D_{Rs}, D_V) = P$, then $\forall p \in D_{Rs}$ and $d \in D_V, P(p_d, d) = \text{True}$. Figure 5 shows an example of the predicates that can be automatically generated with Daikon.

**Figure 5:** An example predicate generated by Daikon. Daikon requires a larger set of values than the one shown on the left hand side of the figure, to infer the predicates on the right hand side. Small sets of values would produce simpler invariants given as enumerative sets of values.

We transform data sets into sequences of interactions annotated with predicates by using Daikon: a data set $ds = (x_1, p_{1x}^{1}, v_{1x}^{1}, \ldots, p_{nx}^{n}, v_{nx}^{n})$ is mapped to a sequence $seq = (x_1, P_1) \ldots (x_n, P_n)$. The generated relations are filtered by probability thresholds to exclude incidental relations.

In our framework, we can formally define Daikon as a function $\text{Daikon}: (D_{Rs}, D_V) \rightarrow P$, where $D_{Rs}$ is a set of input parameter valuations for any $x \in X$, $D_V$ is a set of valuations for context variables and $P: D_{Rs} \times D_V \rightarrow \{\text{True}, \text{False}\}$ is a predicate on parameters and/or context variables. Daikon guarantees that if $\text{Daikon}(D_{Rs}, D_V) = P$, then $\forall p \in D_{Rs}$ and $d \in D_V, P(p_d, d) = \text{True}$. Figure 5 shows an example of the predicates that can be automatically generated with Daikon.

**Figure 4:** Examples of data sets obtained by merging input-equivalent interaction traces from Figure 1.
Merging Equivalent States

Two state are equivalent if they share the same k-future, i.e., exiting paths match completely, and predicates associated with inputs are equivalent. Since EFSMs are generated from incomplete traces, requiring the complete matching of both methods and parameters may limit generalization and produce an overrestricted EFSM.

To better address incomplete traces, we can use either weak or strong subsumption. A state s1 weakly subsumes a state s2 if the method sequences in the k-future of s1 and s2 match completely, but the predicates in the k-future of s1 are more general than the corresponding predicates in the k-future of s2. A state s1 strongly subsumes a state s2, if the k-future of s1 includes the k-future of s2, that is the method sequences in the k-future of s1 includes the method sequences in the k-future of s2, and the predicates in the k-future of s1 are more general than the corresponding predicates in the k-future of s2.

Formally, given two sequences of length k \( seq_1 = (x_1^1, P_1^1) \) \( \ldots (x_k^1, P_k^1) \) and \( seq_2 = (x_1^2, P_1^2) \) \( \ldots (x_k^2, P_k^2) \), we say that

- \( \forall i \in 1, \ldots, k \), \( x_i^1 = x_i^2 \) and \( P_i^1 \) \( \iff \) \( P_i^2 \)
- \( \forall i \in 1, \ldots, k \), \( x_i^1 = x_i^2 \) and \( P_i^1 \) \( \Rightarrow \) \( P_i^2 \)

Given two k-futures \( f_1 = \{ seq_1^1, \ldots, seq_{n_1}^1 \} \), with \( seq_1^i = (x_1^i, P_1^i) \) \( \ldots (x_k^i, P_k^i) \) and \( f_2 = \{ seq_1^2, \ldots, seq_{n_2}^2 \} \), with \( seq_2^i = (x_1^i, P_1^i) \) \( \ldots (x_k^i, P_k^i) \), we say that

- \( f_1 \) is equivalent to \( f_2 \) if \( n_1 = n_2 \), \( \forall i = 1, \ldots, n_1 \) \( \exists j = 1, \ldots, n_1 \) s.t. \( seq_1^i = seq_2^j \)
- \( f_1 \) weakly subsumes \( f_2 \) if \( n_1 = n_2 \), \( \forall j = 1, \ldots, n_1 \) \( \exists i = 1, \ldots, n_1 \) s.t. \( seq_2^j \subseteq seq_1^i \), and vice versa \( \forall i = 1, \ldots, n_1 \) \( \exists j = 1, \ldots, n_1 \) s.t. \( seq_1^i \subseteq seq_2^j \)
- \( f_1 \) strongly subsumes \( f_2 \) if \( \forall j = 1, \ldots, n_2 \) \( \exists i = 1, \ldots, n_1 \) s.t. \( seq_2^j \subseteq seq_1^i \)

Figure 8 shows few examples of k-future that satisfy the different criteria.

- **equivalence**, state 1 is equivalent to state 8
- **weak subsumption**, state 1 weakly subsumes state 8
- **strong subsumption**, state 1 strongly subsumes state 8

Figure 8: An example of the merging criteria with \( k = 2 \).

Given a value for \( k \), GK-tail iteratively merges pairs of states according to a merging criterion, until there are no
states to be merged. \textit{GK-tail} merges states $s$ and $s'$ by removing state $s$, adding to $s'$ all transitions entering or exiting $s$ that do not correspond to transitions entering or exiting $s'$, extending the predicates of the transitions entering or exiting $s$ to include the predicates of the corresponding transitions in $s'$, and moving attributes associated with $s$ to $s'$. For instance, if $s$ is either initial or final, $s'$ is marked by initial or final, respectively.

Figure 9 shows the EFSM obtained from the initial EFSM shown in Figure 6 with $k = 2$ and weak subsumption.

4. PRELIMINARY EVALUATION

To evaluate the technique proposed in this paper, we designed a prototype implementation based on Aspectwerkz [1], Simplify [9], and Daikon [10]. Our prototype implements the GK-tail algorithm using Aspectwerkz to monitor systems and record traces, the Simplify theorem prover to check for equivalence and for implication between annotations, and the Daikon inference engine to derive the constraints that annotate the edges of the EFSM.

To measure feasibility and usefulness of EFSMs as models of the dynamic behavior of software systems, we generated the EFSM models for a set of open source software applications of different sizes and natures, and for the implementations of some common design patterns. The preliminary results are presented in Subsection 4.1, and indicate that the size of the EFSMs generated with our technique does not depend on the size of the software application, but rather on the size of interaction patterns within software components.

We then analyzed the obtained EFSMs to identify those that model behaviors that cannot be captured by both FSMs generated with classic algorithms and constraints generated with Daikon, independently from the FSMs. The results presented in Subsection 4.1 indicate that EFSM models are often more accurate than both FSMs and constraints generated independently, especially for models of a non-trivial size.

To evaluate the impact of the merging criteria presented in Section 3 we compared the EFSM models computed with the different merging criteria on some sample cases. The preliminary results are discussed in Subsection 4.2 and indicate that stronger merging criteria deal better than equivalence in presence of complex conditional loops.

4.1 Comparing Inference Engines

Given a software system, our technique generates several EFSM models of component interactions, one for each method of the monitored components. Thus, the size of the generated EFSMs should not depend on the size of the application, but on the complexity of the interaction patterns among components. To validate this observation, we computed EFSM models for a set of sample open source applications of different size: Jedit [23], Makagiga [32], Jabref [21], Squirrel SQL Client [24], and jcvs [22].

For each application, we considered from 7 to 12 components. We designed test cases for the core functionalities with the category partition method [25], and we recorded interactions and data exchanged between components while running these test cases. We derived EFSM models with GK-tail with $k = 2$ (values between 2 and 4 are often used in software engineering applications [31, 7, 18]) and equivalence merging.

The results are reported in Table 1, and confirm that the size of the generated EFSM models does not depend on the size of the application, and thus we do not expect major scalability problems when applying the technique to large software systems.

![Figure 9: The EFSM generated from the initial EFSM shown in Figure 6 with $k = 2$ and weak subsumption.](image)

<table>
<thead>
<tr>
<th>Application</th>
<th>Appl. size (LOC)</th>
<th>largest EFSM size (g of states)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jcvs</td>
<td>36067</td>
<td>95</td>
</tr>
<tr>
<td>Makagiga</td>
<td>58502</td>
<td>203</td>
</tr>
<tr>
<td>Jabref</td>
<td>62921</td>
<td>26</td>
</tr>
<tr>
<td>Jedit</td>
<td>102574</td>
<td>73</td>
</tr>
<tr>
<td>Squirrel</td>
<td>139027</td>
<td>109</td>
</tr>
</tbody>
</table>

Table 1: Size of EFSMs for different applications.

Annotated edges carry more information than both classic FSMs and constraints on data values computed without referring to the states of the application. To verify the impact of this hypothesis on application models, we identified the EFSM models that include behaviors that cannot be represented as a combination of FSMs and constraints on data values.

Daikon generates constraints that hold independently from the history of the execution. EFSM models that include at least two transitions labeled with the invocation of the same method but associated with different constraints capture details that cannot be represented with a combination of FSMs and constraints on data values computed independently. For example, if an EFSM includes two transitions associated with method $m$, one with constraint $c_1$ and the other with constraint $c_2$, Daikon can infer $c_1$, $c_2$ or $c_1 \lor c_2$, but cannot distinguish the states in which the two constraints hold.

We automatically analyzed the EFSM models generated for the open source applications mentioned earlier in this section and for five design patterns to detect these cases. For our experiments, we implemented commonly used design patterns that include interactions depending from both parameters and state values: Builder, Factory, Chain of Responsibility, State, and Strategy.

As expected, EFSMs are seldom relevant in practical cases. Only 1 out of 59 EFSM models with less than 5 states includes interplays between invocations and data values that cannot be captured by independent FSMs and constraints. On the
contrary, EFSMs are extremely useful when the complexity of the interactions grows. As reported in Column Interplay of Table 2, the percentage of EFSM models that represent interactions not captured otherwise ranges between 14% for Squirrel and 100% for Jabref and jcvs, among the analyzed open source applications. The EFSM models generated for the implementation of the 5 design patterns carry always more information than the corresponding independent models.

<table>
<thead>
<tr>
<th>Application</th>
<th>EFSM</th>
<th>Interplay</th>
<th>Relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squirrel</td>
<td>21</td>
<td>3 (14%)</td>
<td>2 (66%)</td>
</tr>
<tr>
<td>Jabref</td>
<td>3</td>
<td>3 (100%)</td>
<td>2 (67%)</td>
</tr>
<tr>
<td>Jedit</td>
<td>5</td>
<td>3 (60%)</td>
<td>2 (67%)</td>
</tr>
<tr>
<td>MakaGiga</td>
<td>32</td>
<td>23 (72%)</td>
<td>15 (65%)</td>
</tr>
<tr>
<td>jcvs</td>
<td>1</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Design Pattern</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Builder</td>
<td>1</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>Factory</td>
<td>1</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>Chain of Resp.</td>
<td>1</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>State</td>
<td>1</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>Strategy</td>
<td>1</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
</tr>
</tbody>
</table>

Legend:

Column EFSM indicates the number of EFSMs with more than 4 states, column Interplay indicates the amount of EFSMs that represent behaviors that cannot be captured by computing FSMs and constraints on data values independently, and column Relevant indicates the amount of Interplay EFSMs that are relevant with respect to the use of the modeled application.

Table 2: Modeling power of EFSMs

Since annotations associated with dynamic models can represent incidental constraints, some interplays in EFSM models can result in noisy and useless descriptions. To investigate the nature of the annotations, we manually inspected the implementations of all the methods associated with an interplay between data values and interactions that can fully be modeled only with EFSMs. We discriminated useful models, that are models where annotations represent a specific implementation detail or a particular use of the system, from incidental annotations, that are annotations that do not correspond to any relevant information. Column Relevant indicates the percentage of EFSMs that specify semantically relevant interactions among the Interplay ones, that is all the EFSM models of the design patterns, and about 66% of EFSM models for the analyzed applications, with only one exception (jcvs).

4.2 Comparing Merging Criteria

In the previous section, we used the equivalence merging criterion to generate EFSM models. This criterion looks for equivalence between conditions and paths. Thus, it is effective when methods well represent the behavioral space of the system. This happens when test cases cover well all possible executions, as in the cases considered in the former section.

In presence of partial test suites, that is test suites that do not sample the execution space systematically, both conditions and paths, and consequently the model that can be automatically generated may represent only part of the complete execution space. If for example, a given method can be called with a value of parameter \( x > 0 \), while an exhaustive test suite that includes all boundary cases as well as a good sample of normal cases may generate the complete condition, a test suite that samples the execution space only partially may result in a partial condition, for instance \( x > 5 \). Similarly, a complete test suite likely identifies all methods that can be invoked in a given state, while a partial one can identify only a subset.

In presence of a partial sample of the execution space, reductions based on equivalence conditions may not work well. For example, the equivalence merging criterion would not consider \( x > 0 \) equivalent to \( x > 5 \), nor it would consider two states with inclusive sets of exiting paths to be equivalent, thus resulting in a limited reduction of the model. This effect is particularly evident when the set of paths can be summarized in complex conditional loops that can hardly be recognized in presence of partial traces, shown later in this section. In these cases, merging criteria based on inclusion relations, like strong and weak subsumption, can be more effective than equivalence in producing a compact model.

In this section, we investigate the effectiveness of subsumption criteria in producing compact albeit meaningful models. To confirm our hypotheses, we designed experiments with test suites that explore the execution space unevenly. The first set of experiments considers test cases that explore well the whole set of paths and the conditions of a subset of the paths, but sample badly the conditions of the remaining execution paths. The second set of experiments consider test cases that do not sample well both execution paths and conditions.

We implemented a shopping cart with methods `buy` and `purchase`. Method `buy` requires a parameter of type `Cart`, and processes the cart by invoking method `purchase` on each item. Method `purchase` works with parameters `itemId` and `itemQuantity`, and returns a Boolean value indicating whether the item has been successfully removed from the warehouse or not.

We executed method `buy` with 120 test cases, 100 of which buy carts with at most 6 items, with a random quantity of each item, while the remaining 20 test cases buy carts with more than 6 items, but all with the same quantity. The first 100 test cases explore systematically a subset of the execution space, while the remaining 20 test cases explore the complement of the execution space only partially.

The results confirm our hypothesis. The equivalence merging criterion leads to the overrestricted EFSM model, shown in Figure 10. It correctly folds the traces corresponding to the purchase of 6 or less items into a single selfloop, but does not identify the similarities between traces corresponding to purchases of more than 6 items, resulting in a large set of long paths. Both weak and strong subsumptions identify all expected similarities and produce the same compact model.

In the above test, method `buy` invokes one method (`purchase`) for each item, and paths differ only for the conditions. This is why weak and strong merging criteria lead to the same model.

We extended method `buy` by adding an invocation of either method `manageAvailableItem` or `manageUnavailableItems`, depending on the result of the invocation of method `purchase`, thus differentiating paths not only for the conditions, but also for the method invocations. We executed 200 test cases to systematically exercise method `buy` with different combinations of carts up to 6 items, and 20 test case to
5. VERIFYING PROGRAMS WITH EFSMS

As suggested by the preliminary results discussed in the previous sections, EFSM models capture more details of software behavior than FSM and constraints computed independently. The additional details captured by EFSM models support accurate testing and analysis. In this section, we present some preliminary data to compare the effectiveness of EFSM with FSM and constraints models.

In a previous work, we proposed a technique for deriving test cases from FSMs and invariant models that selects a set of test cases to cover both all transitions of the FSM models and all boundary values of the invariants [17]. The technique can be easily extended to EFSM models by requiring transition and boundary condition coverage.

To compare the two techniques with respect to testing, we applied these criteria to derive test suites for the implementation of the design patterns used in Section 4.1. Table 3 compares the test suites and the statement coverage obtained with the two classes of models. The suites generated from EFSM models contain test cases that are not required to satisfy criteria based on FSM and invariants models, and provide higher statement coverage. These preliminary data suggests that EFSM models generate more thorough test suites than FSM and invariants models.

<table>
<thead>
<tr>
<th>Design Patt.</th>
<th>TS FSM+Const</th>
<th>TS EFSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Builder</td>
<td>1</td>
<td>88%</td>
</tr>
<tr>
<td>Factory</td>
<td>1</td>
<td>79%</td>
</tr>
<tr>
<td>Chain of resp.</td>
<td>1</td>
<td>65%</td>
</tr>
<tr>
<td>State</td>
<td>1</td>
<td>80%</td>
</tr>
<tr>
<td>Strategy</td>
<td>1</td>
<td>94%</td>
</tr>
</tbody>
</table>

Table 3: Comparing models wrt test case generation.

6. RELATED WORK

GK-tail generates models of dynamic software behavior that capture the interplay between data values and compo-
nent interactions. Dynamic analysis techniques related to our approach are techniques that derive relations over data values, generate models of interaction patterns, or generate models that capture both data and interactions. In the following, we discuss the main research results that belong to these three classes of approaches.

Detecting Relations Over Data Values

Values assigned to variables at specific program points can provide important information to understand and analyze system executions. Several dynamic analysis techniques have been experienced in different application domains to automatically extract information about data value relations.

The most popular approach to detect relations over data values is Daikon that has been originally proposed by Ernst, Cockrell, Grisswold, and Notkin in [10] to discover program invariants, and has been later extended and applied to several problem domains, including discovery of operation contracts [19] and discovery of properties of data returned by data feed systems [30].

GK-tail uses Daikon to infer constraints from values that annotate the transitions of the FSM. While Daikon produces a model independent from the state of the system, GK-tail produces a model that integrates state information with constraints on data values.

Other approaches to derive models of relations among data values have been studied by Hangal and Lam [12] and Raz et al. [30]. The technique proposed by Hangal and Lam generates lightweight models consisting of masks of bits associated with program points to track bug locations when systems fail. The technique proposed by Raz et al. generates statistical indexes to capture the range of values assigned to single variables and automatically detect anomalous values. Similarly to Daikon, these models do not capture state-based information.

Generating Models of Interaction Patterns

There exist several techniques to automatically generate finite state machines from sets of traces. Some of these inference engines generate models from sets of positive samples only. Most of the techniques working with positive samples only are extensions and variations of the k-Tail algorithm [5, 7, 31]. GK-tail inherits from these techniques the idea of building a simple initial model that is iteratively refined by a state-merging process. However, GK-tail works with traces and models augmented with annotations, while these algorithms work with simple sequences.

Other techniques generate FSA and PFSA by requiring a larger set of information, for example knowledge teachers, ordered samples or negative samples [2, 6, 29, 27]. These techniques are useful when the additional information required for the inference is available. In the case of traces automatically generated from program executions that represent positive samples, additional information is rarely available.

Perracotta derives interaction models in the form of temporal logic propositions that describe legal event sequences [34]. Other techniques derive models of interactions with single objects in the form of algebraic specifications [14, 11]. These techniques can capture specific properties of interactions, but are less effective than classic FSM generation techniques to produce detailed models of interaction sequences.

Generating Integrated Models

EFSMs have been extensively applied in specification of stateful behavior [33], but few approaches that automatically generate EFSMs from program behaviors have been studied so far. Berg et al. proposed a technique to generate EFSMs [4]. The technique supports predicates on Boolean parameters only, and works under strict conditions: it requires the possibility of querying for membership, that is to decide whether an unknown sequence is part of the model, and querying for equivalence, that is to decide equivalence between a partial inferred model and the model that should be inferred. GK-tail works in the more general case, when these strong requirements are not satisfied.

7. CONCLUSIONS

Behavioral models generated from program executions have been extensively applied for testing and analysis of software systems [30, 12, 19, 18, 2, 13, 17, 34, 8, 20]. Inferred models capture either properties of data values, in the form of Boolean expressions that constraint the set of legal values, or properties of interaction patterns, in the form of FSMs that summarize possible interaction sequences, but none of them captures the interplay between data values and interactions, even when considered jointly.

In this paper, we propose GK-tail, a dynamic analysis technique that produces models of the behavior of software systems in the form of EFSMs. These models represent constraints on data values, properties of interaction patterns, as well as their interplay.

The early experience with third-party applications gained with a prototype implementation shows the usefulness and feasibility of the approach. The data collected so far with third-party applications and design patterns indicate that relevant behaviors of software systems depend on the interplays between data and interaction patterns, and can be captured by EFSM models, but not by constraints on data and FSM produced separately. When GK-tail generates models of component interactions, the data collected so far confirm the hypothesis of scalability of the approach, since the complexity of GK-tail does not depend on the size of the analyzed systems, rather than on the complexity of the interactions between components.

EFSM models, as all dynamic models, depend on the quality of the test suites used to produce them. The analysis of the EFSM models derived with the three different inference criteria defined for GK-tail shows the flexibility of GK-tail in generating useful models in presence of test suites of different quality.

We are currently working on a set of experiments on large applications to confirm the preliminary data presented in this paper.

8. ACKNOWLEDGEMENTS

The authors would like to thank Mauro Santoro for his contribution to experiments.

This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157 and by MIUR under the project PRIN 2006-2007 “Mutant hardware/software components for dynamically reconfigurable distributed systems” (COMMUTA).
REFERENCES


