Inference of Behavioral Models that Support Program Analysis

PhD Dissertation by:
Mauro Santoro

Advisors:
Prof. Mauro Pezzè
Dott. Leonardo Mariani

Supervisor of the Ph.D. Program:
Prof. Stefania Bandini

Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Dottorato di Ricerca in Informatica

XXIII edition

Contents

List of Figures

List of Tables

Listings

Introduction

The use of models to study the behavior of systems is common to all fields: from wind-tunnel to Navier-Stokes equations to circuit diagrams to finite models of buildings, engineers in all disciplines construct and analyze models [68]. A behavioral model formalizes and abstracts the view of a system and gives insight about the behavior of the system being developed.

In the software field, behavioral models can support software engineering tasks. In particular, models that represent the behavior of the program during its execution can be used to reason about questions like: "*What did happen during program executions?*, *How program should have behave?*, *What will happen afterwards?*". Relevant uses of behavioral models are included in all the main analysis and testing activities: models are used in program comprehension to complement the information available in specifications, are used in testing to ease test case generation, used as oracles to verify the correctness of the executions, and are used as failure detection to automatically identify anomalous behaviors.

Unfortunately, it is extremely effort demanding to produce and maintain behavioral models. Fortunately, when behavioral models are not part of specifications, automated approaches can automatically derive behavioral models from programs. The degree of completeness and soundness of the generated models depends from the kind of inferred model and the quality of the data available for the inference. When model inference techniques do not work well or the data available for the inference are poor, the many testing and analysis techniques based on these models will necessarily provide poor results.

This PhD thesis concentrates on the problem of inferring Finite State Automata (the model that is likely most used to describe the behavior of software systems) that describe the behavior of programs and components

and can be useful as support for testing and analysis activities. The thesis first empirically investigates the limitations and the capabilities of state of the art techniques, and then defines two complimentary approaches that can produce accurate behavioral models from software systems.

More in detail, this thesis contributes to the state of the art by:

- *Empirically studying* the effectiveness of techniques for the inference of FSAs when a variable amount of information (from scarce to good) is available for the inference.
- *Empirically comparing* the effectiveness of techniques for the inference of FSAs and Extended FSAs.
- *Proposing a white-box technique* that infers FSAs from service-based applications by starting from a complete model and then refining the model by incrementally removing inconsistencies.
- *Proposing a black-box technique* that infers FSAs by starting from a partial model and then incrementally producing additional information to increase the completeness of the model.

A proper technique between the white-box and black-box approaches can be selected depending from both the availability of the source code and the quality aspect (completeness or soundness) that is more relevant

The thesis is organized as follows:

Chapter1

This chapter discusses the key role played by behavioral models to support software testing and analysis. The chapter first presents a classification of the techniques according to their role in the software life cycle. Then it surveys the main behavioral model-based testing and analysis techniques. It finally discusses the open issues about the inference of high-quality behavioral models.

Chapter2

This chapter surveys the most effective and well-known techniques that can infer finite state behavioral models. The chapter first presents techniques that produce models of sequences of events. Then it introduces

a technique to derive models from data values, and discusses techniques that extract FSA annotated with constraints and data-flow information. It concludes by surviving techniques that generate models that represent the ordering of the events.

Chapter3

This chapter presents an empirical comparative study between techniques that infer Finite State Automata and techniques that infer Finite State Automata annotated with constraints and data-flow information. It investigates the effectiveness of these techniques when applied to traces with different levels of sparseness, produced by different software systems. It terminates by discussing complementarities, strengths and weaknesses of the approaches providing a roadmap for developing better model-inference solutions.

Chapter4

This chapter presents a static analysis technique that extracts models of the service interaction protocol. It also presents a novel refinement strategy to eliminate infeasible behaviors that reduce the usability of statically derived models. It terminates by providing some empirical data from the experience of the technique with an application that interacts with the eBay Web Services.

Chapter5

This chapter presents a black-box technique that incrementally improves the completeness of dynamically inferred behavioral models. It first describes how the technique uses machine learning to explore the execution space. Then, it terminates by showing, with the help of a case study, that the inferred model is precise and incrementally more complete, according to the degree of exploration that is achieved.

Chapter 1

Testing and Analysis by Means of Inferred Behavioral Models

The extensive adoption of test and analysis techniques can improve the quality of the developed software but traditional methodologies are not always applicable. For example, many verification and validation techniques require source code or specifications to be successfully applied, but these requirements limit their applicability when systems are provided without source code or with incomplete specifications.

Novel testing and analysis techniques, based on the synthesis of behavioral models from program executions, have been recently defined. These techniques take advantage of the automatically generated models in a number of ways: to derive test cases, to recognize failures, to debug failures, etc.

In the following we emphasize the key role played by behavioral models to support software testing and analysis.

1.1 The use of behavioral models: a classification

Models of the program behavior can effectively support software test and analysis activities. Behavioral models can be used to verify protocols [16],

to detect anomalies [41] [71], to generate test cases [43], to capture unexpected event sequences [87], to verify program properties [26] [65] and to check the compatibility between software components both statically and dynamically [59] [62].

We classified techniques into 3 groups according to their role in the software life cycle:

- **program comprehension:** Understanding the behavior of a program requires the understanding of its specifications. Unfortunately, most programs do not come with precise and complete specifications, and even when high quality specifications are produced they tend to become obsolete while the software evolves. As a consequence, several works studied the problem of documenting the program behavior. Mining behavioral models from program executions is an effective way to automatically obtain specifications. Mining specifications from programs is known as program comprehension.

- **testing:** Specifications play a critical role in testing. For instance, specifications are the main source of information for test case generation, implementation of oracles and test suite maintenance. Unfortunately, specifications are often incomplete or outdated, and the lack of suitable specifications can harm the effectiveness of testing and analysis of software systems. The automatic inference of behavioral models can reduce the risks related to lack of specifications. In particular, the behavioral models automatically extracted from programs can play the same role of specifications. The benefits of model inference in testing results in both an increased effectiveness and a reduced effort of test cases design and execution.

- **anomaly detection:** Anomaly detection techniques can identify anomalous behaviors by looking for behaviors that differ from the ones observed during successful executions. The analysis of the behavioral deviations guide developers in the identification of faults and reduce the time between patches. The strength of this solutions is strictly related to the availability of models of the correct behaviors. Although models of correct behaviors are rarely available, they can be easily automatically derived.

.

The following sections discusses the main program comprehension, testing and anomaly detection techniques based on inferred behavioral models.

1.2 Program comprehension

When specifications are not consistent and complete, behavioral models directly inferred from programs can help developers to avoid misinterpreting the functionalities.

In the following we discuss techniques for specification mining of properties that programs likeley satisfy during executions. The term specification normally refers to a description that indicates the behavior expected from the system. Specifications are written before the system is implemented. Here, consistently with the literature in specification mining, we use specification to refer to behavioral properties automatically extracted from program artifacts during development and maintenance phases. The inferred properties can be used like specifications.

Dynamic and static analysis techniques can infer program specifications in the form of interaction protocols and contracts. Interaction protocols describe sequence of messages that two components can exchange. Contracts represent constraints on the values that can be assigned to variables.

To the best of our knowledge, the first paper that can be labelled as program comprehension technique, can be tracked back to as early as 1972 [20] when finite state machines was used to synthesize execution traces. The technique describes a method for deriving a behavioral model that specifies the legal finite set of possible input-output pairs. Since then, this type of software analysis has grown resulting in several interesting contributions. Ammons et al. [16] discover temporal and data dependencies relationships that the program satisfies when it interacts with an application programming interface (API) or abstract data-type (ADT). To infer the model, a specification miner monitors the interactions of a running program and uses the recorded data to derive a general rule about how the program interacts with the API or ADT. Ernst et al. also proposed automatic deduction of formal specifications [35]. Daikon works by learning likely invariants from dynamic traces and produces contracts that hold at specific program points. Lorenzoli et al. and Lo et al. derive different kinds

of finite state automata [55] [53]. These techniques produce compact models that summarize the observed behaviors. The technique implemented by Yang et al. in the tool Perracotta [87] addresses the challenge of mining specifications from large programs. Perracotta infers temporal API rules by using an analysis technique for detecting dominant behaviors from imperfect traces.

Good examples of recent work in the static analysis area are the proposals of Shoham et al., who derive models of the usage protocol for security analysis [76], and of Wasylkowski et al. who statically learns how objects can be used in general from actual usages that occur in the application [80]. The latter technique discovers the interaction patterns that can occur within single methods with an intra-procedural analysis, but does not support inter-procedural analysis, thus it cannot discover interactions that derive from the execution of multiple methods. Caso et al. generate models of operation contracts [32], i.e., from operations specified with pre and post conditions. Dallmeier et al. derive object usage specifications [29] while Bertolino et al. generate models of operation dependencies [19]. Dataflow models abstract from many details about the program structure and only represent the definitions and uses of variables across a program [38]. Static slicing algorithms build program slices, i.e., sets of instructions, that represent the program statements that can affect a given statement [81]. Mariani et al. [60] present SEIM, a static analysis technique that derives accurate models of the interactions between applications and the Web Services integrated in them.

Finally, the technique implemented in the Dysy [27] tool, proposes an hybrid approach, by means combining dynamic and static analysis, to infer contracts with greater precision than Daikon.

1.3 Testing

The potential benefits of automatic specification mining, and the necessity to overcame the lack of specifications, has encouraged novel forms of testing which lay directly on the use of inferred behavioral models.

In the following we summarize some representative works, in the field of testing, to demonstrate the effectiveness of inferred behavioral models as replacement of specifications.

If both the source code and the specification are not available, e.g., systems that integrate COTS components, we need to infer the information necessary to effectively apply testing techniques from the programs. Both Wu, Pan and Chen [82] and Mariani, Pezzè and Willmor [61] automatically derive a model of interactions that can be used for selecting test cases. The former technique derives the model statically, while the latter derives the model dynamically. Wu, Pan and Chen use a model, called Component Interaction Graph (CIG), that captures interactions and dependencies among components. A CIG consists of a graph where nodes can represent both interfaces and events, and edges can represent both control-flow and data-dependencies among interfaces and events. Wu, Pan and Chen defined a family of adequacy testing criteria that can be formulated in terms of entities in the CIG. They do not provide a technique for generating test suites that satisfy the adequacy criteria, but conducted early experimental studies to identify the cost-effectiveness of the different criteria in the family. Mariani, Pezzè and Willmor developed a technique for the dynamic construction of a test suite by recording the stimuli that generate relevant interactions. Relevance of an interaction is defined in terms of the inferred model. In particular, if an interaction increases coverage of entities of the inferred model, it is selected as test case. Registration of relevant interactions and inference of the model can be incrementally performed at the same time.

When at least a deficient test suite is available, the generation of test suites can benefits of the synergy between the pre-existing set of tests and mined behavioral specifications. Substra [88] is a framework to generate test cases based on inferred constraints on components interfaces. Constraints are inferred on the bases of information on shared states between method calls and define-use relationships between parameter values of method calls. During execution of test cases, Substra uses Daikon to collect object state and infers call-sequence constraints. A call-sequence constraint is for example a define-use constraint that tells that a variable can only be used after its definition. Then Substra uses the generated constraints to guide an automatic generation of integration tests. The result is a sequences of method-call with randomly generated input values. Harder et al. [43] developed a test case selection technique for augmenting, minimizing, and generating test suites in cooperation with an automatically

inferred behavioral model. The used model is expressed in the form of an operational abstraction, that is a property of the program's run-time operation inferred with Daikon. The generation process of the test suite starts with an empty test suite and an empty operational abstraction that are augmented, run-by-run of the available test cases, when available test cases change the operational abstraction. Test cases that do not serve for operational abstraction are retained, under the assumption that an operational abstraction generated from a larger test suite is better. When n candidate cases have been consecutively considered and rejected, the process is complete. The value chosen for n is a trade-off between the running time of the generation process and the desired quality of the test suite.

Developers who want to test their own code can use Agitator [21]. It is a tool that automatically generates initial tests, infers operational abstraction like observations, lets developers to promote these observations to assertions, and generates more tests to violate the inferred and confirmed observations. When developers are satisfied by the inferred models, they can create tests cases that express the behavior contained into the assertions. Also Xie et al. proposed a technique that take advantage of of behavioral models to produce unit tests [85]. The technique start to execute an existing unit-test suite and monitor the program using Daikon, which was modified to generate design-by-contract (DbC) annotations. The DbC annotations represent conditions at entry and exit of methods in the program. Additional unit tests are created to try to violate DbC invariants. Finally, developers examine the generated tests and define the unit-test suite.

Behavioral models can also be used to add assertions into a test suite so that the augmented test suite has an improved capability of guarding against faults. The Orstra approach [83] augments a set of automatically generated test inputs with assertions useful to detect regression faults. Orstra works by first running the given test suite to collect the return values and receiver object states after the execution of each method under test. Then, based on the collected information, it synthesizes constraints that are injected into the existing test cases as new assertions that will be checked in the future regression testing sessions. Other works addressed this same problem with different strategies [43] [21].

When the quality of a test suite is measured in term of the amount of code elements executed by test cases with respect to the total program code,

it is important to identify and discard the infeasible program elements to be excluded from the computation of the structural coverage. To cope with this obstacle, Baluda et al. [17] exploited the use of a model, inferred from the control flow graph of the program, to define a new generation of structural testing techniques and to compute accurate structural coverage measurements. The model is a labelled rooted graph where nodes represent abstract states and are annotated with predicates over the program variables, while edges are annotated with the corresponding statements. The role of the models is to distinguish executable paths from infeasible paths: it incrementally guides the construction of new test cases that increase code coverage, and discovers infeasible code elements that can be therefore excluded from the coverage count.

Another way to improve the quality of unit tests is proposed by the Abstra tool [84]. The tool helps developers to inspect unit test execution results by inferring a set of object state machines (OSM). An OSM is an abstract view of the tested software where each state represents an object state of the tested classes and each transition represents a method call. The OSM can help testers in the identification of the untested areas of the software under test.

1.4 Anomaly detection

Anomaly detection techniques use models of the application behavior to identify anomalies in failing executions. To overcome the lack of complete specifications these techniques use models inferred by monitoring correct executions to detect the likely causes of the failures. Anomaly detection techniques output the identified anomalies to developers. The analysis of the deviations from standard behavior guides developers in the localization of the fault.

In the following we summarize some representative anomaly detection techniques.

Anomaly detection techniques vary for the kind of models that is inferred and the type of data that is used for the inference. Diduce [42] and Carrot [69] infer boolean expressions that describe potential data invariants by applying a predefined set of rules to data values collected during program executions. For example, these techniques can detect that the in-

teger value returned by a method is always positive. Other approaches instead do not consider data values but focus on method calls to identify anomalies. In [30], sequences of method invocations are collected to derive, for every class of the system, the input and output protocols, that is the sequence of invocations that the classes can accept and the sequences of invocations that the classes can execute. Pachika [31] and Behavior Capture and Test [59] instead focus on both data and invocation sequences. Pachika infers automata that model the object usage protocol: states are identified by inferring invariants over object attributes, and transitions correspond to methods that caused the transition from one state to another. Faults are identified by comparing models of faulty executions with models of correct executions. Behavior Capture and Test (BCT) instead infer Finite State Automata that generalize the sequences of interactions between components, and data invariants on the parameters exchanged in different invocations. The technique is used to identify integration problems.

Lee et al. in [49] derive association rules that, given a sequence of k events, predict the event at position $k + 1$. Violations of the rules are treated as anomalies. Association rules do not model the order of the observed events. For this reason they cannot be applied to detect faults characterized by the wrong order of executions of the operations. Warrander et al. in [79] infer Hidden Markov Models (HMMs) that generalize the sequences of events observed at run-time. Each state of the model represents a different sequence of events of length k (k is a parameter defined by developers), while each transition describes the probability to reach a given state from another state. The inferred HMMs are then used at run-time to identify anomalous events that lead to unusual state transitions. The main limitation of HMMs is constituted by the time needed to build the model which can take several days for complex executions. Other techniques identify anomalies not as deviations from expected behaviors but by matching the actual execution with models of known faulty behaviors. These models are built through supervised learning algorithms that use information collected during both correct and faulty executions. Chen et al. for example use decision trees to identify failures in large internet sites [90]. The authors derive decision trees that can detect if a failure is occurring at run-time. This technique permits only to identify problems that already occurred and does not help in case of failures never experienced before. Since fault localization usually regards problems not diagnosed in the past, the

applicability of supervised learning approaches for fault localization is limited.

1.5 Discussion

The many research works in testing and analysis, based on the inference of behavioral models, demonstrate the relevance of this research direction. The need of inferring behavioral models is emphasized by the new software development methodologies (i.e reuse of COTS components), the diffusion of software with scarce documentation and by the scalability of the novel model inference techniques. However, testing and analysis technique based on inferred models have to tolerate some model imprecision and incompleteness that might be otherwise responsible for poor testing and analysis results.

In our classifications, we showed that models can be derived in two main ways: from the source code or from program executions. Both approaches suffer of some limitations. Techniques that statically identify and discover behavioral models exhaustively analyze the program code, thus cannot always distinguish feasible from infeasible executions, and consequently the resulting models can include infeasible behaviors. Techniques that derive models from executions analyze only the observed behaviors, thus do not include infeasible behaviors but consequently the inferred models may miss some feasible behaviors that have been never executed. A promising research direction is hybrid approaches which combine static and dynamic analyses to effectively explore program behaviors, by mitigating the disadvantages of either techniques. These solutions are extremely effective, but also extremely expensive because they require the integration of complex techniques as theorem provers and constraint solvers [18] [40]. Moreover, these techniques are often limited by presence of complex (non-linear) expressions and complex language constructs (aliases, pointers and polymorphism) in programs.

It is clear that, nevertheless the high efficacy of behavioral models to support software testing and analysis, there is a need to develop new techniques to discover more thorough and more precise behavioral models.

Chapter 2

Inference of Behavioral Models

Manually specifying and maintaining software behavioral models is expensive, error prone and require specific skills that are not always available in development teams. To reduce the effort required to generate models, many model-based test and analysis techniques use models that are automatically generated [22] [59] [70].

Several kind of behavioral models can be automatically generated. The main representatives vary from finite state automata [20], to grammar inference [1], to describe relations between events, from patterns of events [50] [73], temporal logic rules [51] [87] to graph transformations [39], to describe temporal relation between events, and data-flows properties [36], to describe program states.

A kind of model that is both commonly used to represent the program behavior and largely supported by automatic model generation techniques is finite state automata. Finite state automata can be easily inferred from program traces. The inferred finite state automata can also integrate data-flow information to represent how data values affect the operations executed by programs.

This chapter surveys the most effective and well-known techniques that can infer finite state automata and have been used to support software engineering tasks. In particular, we first present the kTail [20] and kBehavior techniques [59] that produce models of sequences of events, then we describe how the Daikon [35] technique derive models from data values, and

finally we discuss the gkTail [55] and KLFA [57] techniques that, taking advantage of kTail, kBehavior and Daikon, extract FSA annotated with information about attribute values to capture both the relations among events and among attributes. We conclude the chapter by surviving techniques that generate models that represent the ordering of the events.

2.1 Finite state automata

The problem of inferring an FSA that accepts a language that contains a given set of samples is well known and has been extensively studied. There exist several techniques whose applicability depends both on the nature of the samples and the knowledge about the FSA to be inferred. Some techniques requires only positive samples, that is samples that belong to the language to be inferred [23] [53], while other techniques require both positive and negative samples, that is both samples that belong and samples that do not belong to the language to be inferred [66]. Other techniques take advantage of additional information like *teachers* that answer to membership queries, that is they assume to know if any specific sample does or does not belong to the language to be inferred [22].

In the following we describe techniques that generate FSA from a set of positive samples. We restrict the survey to the case of positive samples because in applications to software engineering tasks it is extremely difficult to obtain a significative number of negative samples. We also presents a running example that is used to illustrate the various inference processes discussed in this section.

2.1.1 A running example

This Section presents the example that we use to illustrate the techniques that generate FSA from positive samples.

Let us consider the pseudo-code in Listing 2.1, which represents a sample `Purchase` component that manages requests issued from users. The code represents the confirmation of the purchase for all the products in the cart. The service first retrieves both the content of the cart and the user details, and then tries to execute the transaction with a bank. If the transaction fails, the method generates an error message, otherwise it modifies the content of the database (if quantities are low, products are immediately

```
1  public void performPurchase() {
2
3          ...
4
5          orderedItems=0;
6          cartItems = cart.getCart();
7          user = userMng.getUser();
8          result = transManager.doBankTrans( cart, user.getBankAccount(), myBank
               );
9
10         if (!result.pass) errorWindow(result);
11
12         for(i=0; i<cartItems.getLength(); i++) {
13                 qtOK = storage.removeQt(cartItems[i].id, cartItems[i].qt);
14
15                 if (!qtOK.pass) {
16                         dt=orderMng.addProduct(cartItems[i].id, qtOK.qt);
17                         timeTable.items[orderedItems++] = dt;
18                 }
19         }
20
21         returnOrder(cart, timeTable, user)
22
23         ...
24
25 }
26
27 private void returnOrder (CartItems cart, TimeTable tt, User usr) {
28
29         for(i=0; i<cartItems.getLength(); i++) {
30                 itemDetails[i] = catalog.getItemDetail(cart[i].id);
31         }
32
33         generateHTMLPage(itemDetails, tt, usr);
34
35         ...
36
37 }
```

Listing 2.1: *A sample source code of a* `Purchase` *component that manages the purchase of items in a cart.*

ordered from producers). Finally, the service visualizes all the information about the purchase.

The execution of the service includes several interactions with the components `Cart`, `UserMng`, `TransManager`, `Storage`, `OrderMng` and `Catalog`. If we monitor run-time interactions between components, a possible trace file is shown in Table 2.1.

2.1.2 kTail

kTail is a technique that generates a FSA from a set of positive sample in two step [20].

It first builds an initial FSA, called *prefix tree automaton*, that accepts all the individual samples by creating a branch for each trace. Eventually, two branches can share the prefix. In the second phase, kTail merges two states if they accept the same *k-future*, i.e., the two states accept the same

Execution 1				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→removeQT**
→removeQT	**→getItemDetail**	**→getItemDetail**	**→getItemDetail**	
Execution 2				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→removeQT**
→addProduct	**→removeQT**	**→removeQT**	**→removeQT**	**→getItemDetail**
→getItemDetail	**→getItemDetail**	**→getItemDetail**	**→getItemDetail**	
Execution 3				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→addProduct**
→removeQT	**→addProduct**	**→removeQT**	**→getItemDetail**	**→getItemDetail**
→getItemDetail				
Execution 4				
getCart	**→getUser**	**→doBankTrans**		
Execution 5				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→addProduct**
→removeQT	**→addProduct**	**→getItemDetail**	**→getItemDetail**	
Execution 6				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→removeQT**
→removeQT	**→removeQT**	**→removeQT**	**→getItemDetail**	**→getItemDetail**
→getItemDetail	**→getItemDetail**	**→getItemDetail**		
Execution 7				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→addProduct**
→removeQT	**→removeQT**	**→removeQT**	**→getItemDetail**	**→getItemDetail**
→getItemDetail	**→getItemDetail**			
Execution 8				
getCart	**→getUser**	**→doBankTrans**	**→removeQT**	**→removeQT**
→removeQT	**→addProduct**	**→getItemDetail**	**→getItemDetail**	**→getItemDetail**

Table 2.1: *Traces recorded during the execution of the program shown in Listing 2.1.*

set of strings of length k. The second phase can require several iterations to find all the states that must be merged.

The initial FSA that is built in the first step from traces in Table 2.1 is shown in Figure 2.1. Each branch is a different trace and common prefixes are merged. The root is the initial state, while leafs are accepting state. Then, the second phase is executed and states are merged. The output obtained with kTail for k=2 is shown in Figure 2.2.

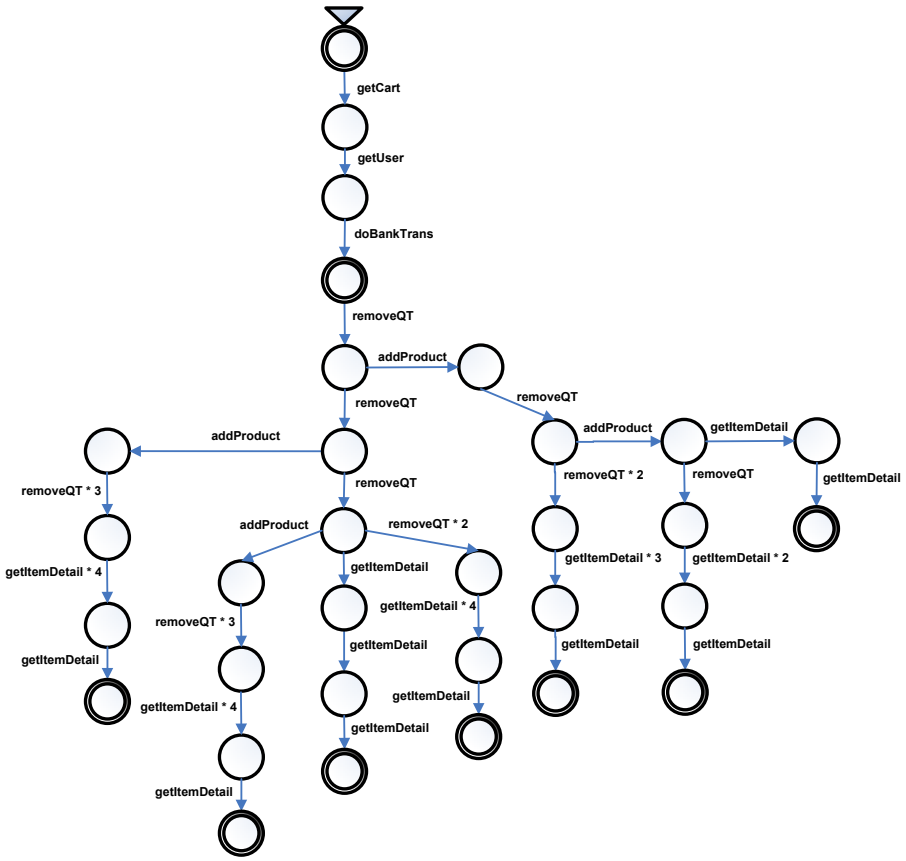


Figure 2.1: *The prefix tree automaton that is built from the trace file shown in Table 2.1. The symbol $*$ indicates that there is a repetition of the same label, and the length of the sequence is specified with the associated number.*

It is possible to notice that the inference engine generated a FSA that accepts more behaviors than the ones directly represented in the traces. For instance, state 10 accepts an infinite repetition of `removeQT`. However, many legal sequences are still rejected by the FSA. For instance, consider

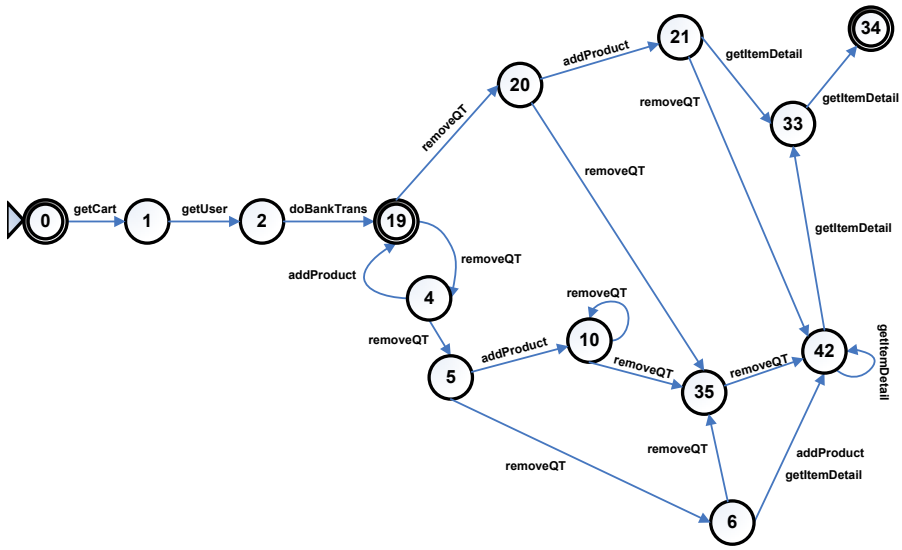


Figure 2.2: *The output obtained by running kTail with $k = 2$ on the trace file shown in Table 2.1.*

any execution that starts with the sequence `getCart,getUser,doBankTrans,removeQT,removeQT,removeQT,removeQT,removeQT,removeQT` or `getCart,getUser,doBankTrans,removeQT,removeQT,removeQT,removeQT,removeQT,addProduct`. kTail succeeds in detecting that `getItemDetail` is possible only at the end of the behavior, but often the number of `getItemDetail` that can be generated does not correspond to the interaction sequences that can be executed. The quality of the result can be improved if the number of behaviors represented in the traces is increased. However, in the real cases it is frequently hard to generate additional executions.

Cook and Wolf's extension

kTail is not suitable to infer end of loops because it creates many redundant nodes [23]. This effect is clear from the automaton in Figure 2.2, where the loop of `getItemDetail` is represented by many redundant nodes (see nodes 42, 33, 34).

Cook and Wolf extended the kTail technique by adding a third phase where all nodes of the FSA are analyzed. In this step, if a node S_i has

outgoing edges $e_1 \ldots e_n$ to states $S_1 \ldots S_n$ with the same label on edges, and there exists a state S_m that can accept a set of symbols subsuming all symbols that can be accepted by the other states $S_1 \ldots S_n$, then all states $S_1 \ldots S_n$ can be merged with S_m. This step merges redundant nodes and improves the quality the model.

If we run the Cook and Wolf's extension with the example trace shown in Table 2.1, we obtain the FSA in Figure 2.3. The inferred FSA over-generalizes the behavior observed in the traces and after the initial sequence `getCart,getUser,doBankTrans` allows any combination of `removeQT`, `addProduct` and `getItemDetail`. Therefore, it fails in capturing that only one `addProduct` is possible after an `removeQT` and that repeated `getItemDetail` are possible only at the end of the execution.

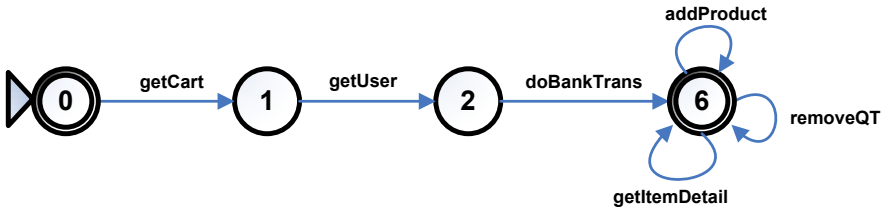


Figure 2.3: *The FSA inferred by the Cook and Wolf's extension on the input traces shown in Table 2.1.*

Reiss and Renieris' technique

Reiss and Renieris proposed another variant of the kTail technique that uses a merging criterion that is weaker than the kTail, that is it merges more states than kTail. [72]. In this case, two states are merged if one state shares some *k-future* with another state.

We used this criterion with the traces in Table 2.1 and we obtained the same FSA of the Cook and Wolf's extension that is shown in Figure 2.3.

2.1.3 kBehavior

kBehavior [59] is a technique that, starting from an empty FSA, incrementally generates a FSA that accepts all the observed samples.

At each iteration, kBehavior extends the current FSA A with a string α by iteratively executing the following steps: identification of the prefix of α already generated by A; identification of a suitable behavioral pattern ρ (a behavioral pattern is a sub-string of α already generated by a sub-machine of A); extension of A by connecting the state reached by generating the prefix with the state of A that generates the ρ so that α is generated up to ρ; if a behavioral pattern cannot be identified, a new branch is added to A

Identification of the prefix

Given a new string α, kBehavior determines the longest prefix of α that is already generated by A. If A generates the whole input string, no extensions are necessary and the algorithm returns. Otherwise, a behavioral pattern must be identified for extending the current FSA. The identification of a behavioral pattern represents the identification of a known sequence of interactions that can be already generated by A and that is part of α.

Identification of the behavioral pattern

The identification of the behavioral pattern consists of decomposing α into three parts: a prefix μ, the behavioral pattern ρ and the tail υ. The behavioral pattern must be already generated by a sub-machine of A. There are two possible search strategies: searching for (1) the decomposition that contains the longest behavioral pattern, or (2) the first decomposition with a behavioral pattern longer than a parameter k_ρ. Detecting the longest behavioral pattern can be time consuming, therefore the selection of a behavioral pattern that is closer to the current state and is longer than k_ρ can be an effective optimization of the search. The parameter k_ρ represents the minimal length of a behavior that can be considered good enough to stop the search.

If the decomposition does not exist, there are no behavioral patterns and it is not possible to assume that part of the observed behavior can be generated by A. In this case a new branch is added to A.

Extension

If the behavioral pattern is identified, A can be extended. The extension takes place by adding a FSA A' from the current state to the initial state of the sub-machine that generates the behavioral pattern. A' must generate

the part of the string that is included between the prefix and the identified behavior, i.e., the sub-string μ.

Figure 2.4 shows a simple example of extension of a FSA that detects behavioral patterns of minimal length 2, by using the first two execution traces of the Figure Table 2.1. Do not consider the transition and states inside gray zones as part of the initial FSA. The input trace is `getCart,getUser,doBankTrans,removeQT,removeQT,addProduct,removeQT,removeQT,removeQT,getItemDetail,getItemDetail,getItemDetail,getItemDetail,getItemDetail`. The first step consists of identifying the already generated prefix, which in this case is `getCart,getUser,doBankTrans,removeQT,removeQT`. Then, a behavioral pattern is searched. In this case, a behavior of length 6 corresponding to the sub-string `removeQT,removeQT,removeQT,getItemDetail,getItemDetail,getItemDetail` is generated by the sub-machine rooted in the state 3. The FSA is then extended by suitably connecting the prefix with the behavioral pattern, therefore the edge in the gray zone 1 is added. Then the technique continues in the same way from state 9 with the remaining part of the string: `getItemDetail`. The FSA is extended with a new edge that corresponds to the branch in the grey area 2.

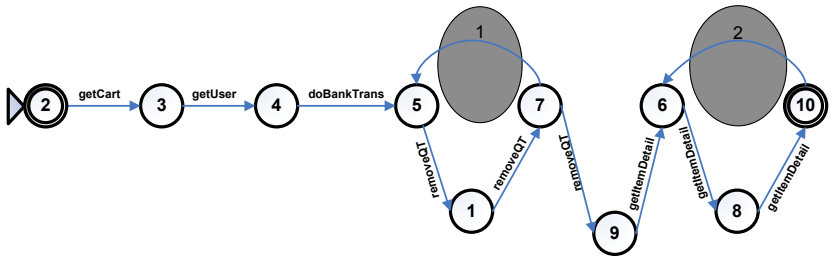


Figure 2.4: *An example of extension of a FSA with kBehavior.*

Finally, if the remaining part of the trace is not empty, i.e., $\upsilon \; ! = \lambda$, the algorithm iteratively continues from the current state with the remaining part of the input string. Otherwise, the computation is completed and the final state can be added to the set of final states.

If we consider the example traces in Table 2.1, and we execute kBehavior, we obtain the FSA shown in Figure 2.5. The inferred FSA correctly captures the initial `addCart,addUser,doBankTrans` sequence and the

sequence of `getItemDetail` that is executed only at the end of the computation. The loop starting from state 6 correctly allows an arbitrary amount of `getItemDetail`. We notice that the set of traces is enough for correctly identifying the loop and the possible behaviors inside the loop. In fact, any sequence of `removeQT` is possible, an `addProduct` is accepted after each `removeQT`, and no more than one `addProduct` can be generated in each state with the exception of the state 7 that can generate a second `addProduct`. We must notice that the automaton inferred by kBehavior both includes all possible behaviors and little over-generalizes the possible sequences with respect to the other techniques.

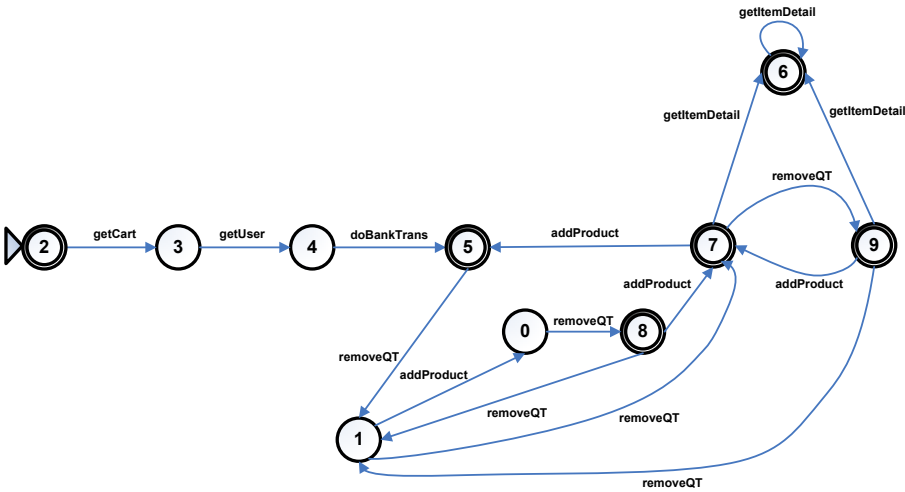


Figure 2.5: *The FSA that is obtained by running the kBehavior technique on the traces shown in Table 2.1.*

2.2 Program invariants

The idea of inferring invariants from program traces has been introduced for the first time by Ernst et al. [35]. Invariants are typically in the form of boolean expressions representing a set of monitored program variables at specific program points, e.g, method entries and exits. The methodology through which program invariants are detected mainly differ for the used resources: there exist techniques that use only information available at run-time, while other consider also the source code. The firsts are tech-

niques that define program invariants from a set of predefined templates by choosing only the invariants with a statistically significant confidence that their occurrence is not incidental [35] [42]. The other techniques instead build invariants by simultaneously considering the run-time values of the variables and invariants derived from conditions in branch statements [27].

In the following we describe the Daikon invariant inference tool developed by Ernst at the MIT. We decided to present it for two reasons: the first because Daikon is the first and most mature representative tool for the dynamic inference of invariants, with the widest use in other researches [21] [25] [34] [37] [48] [86], and the second because it is used by the gkTail techniques, presented in the next session, to infer annotated FSA. We also presents a running example for a better understanding.

2.2.1 A running example

Let consider the `Item` class shown in Listing 2.2 which is part of an e-commerce system. The class implements methods both for the check out (`getTotalCost()`), and to manage stock (`setQuantity()`) of all the products sold by the system.

To collect the information required to infer invariants, we monitor program variables respectively at line 10, 11, 18, 23 and record data values at method entry and exit. Table 2.2 shows examples of values observed for each variable before and after methods execution.

Statement	Variable	Value before exec.	Value after exec.
`Item.23`	\result	-	$\{30, 45, 2, 7, ..., 90, 5, 7, 2\}$
	quantity	$\{2, ..., 3, 7, 3\}$	$\{2, ..., 3, 7, 3\}$
	price	$\{6, 15, ..., 13, 5, 7, 3\}$	$\{6, 15, ..., 13, 5, 7, 3\}$
`Item.18`	this.quantity	$\{4, 2, 7, ..., 4, 8, 1\}$	$\{2, 7, ..., 4, 8, 1\}$
	quantity	$\{2, 7, ..., 4, 8, 1\}$	$\{2, 7, ..., 4, 8, 1\}$
`Item.11`	this.price	$\{6, 15, ..., 13, 5, 7, 3\}$	$\{15, ..., 13, 5, 7, 3\}$
	prc	$\{15, ..., 13, 5, 7, 3\}$	$\{15, ..., 13, 5, 7, 3\}$
`Item.10`	this.quantity	$\{0, 2, ..., 3, 7, 3\}$	$\{2, ..., 3, 7, 3\}$
	qty	$\{2, ..., 3, 7, 3\}$	$\{2, ..., 3, 7, 3\}$

Table 2.2: *Example of variables values recorded during execution of the code shown in Listing 2.2.*

```
1  private class Item {
2
3          private int itemId;
4          private int quantity = 0;
5          private int price = 0;
6
7          public Item(int id, int qty, int prc) {
8
9                  this.itemId = id;
10                 this.quantity = qty;
11                 this.price = prc;
12         }
13
14         ...
15
16         public void setQuantity (int quantity) {
17
18                 this.quantity = quantity;
19         }
20
21         public int getTotalCost () {
22
23                 return quantity * price;
24         }
25
26 }
```

Listing 2.2: *An example of an* `Item` *class of an e-commerce system.*

2.2.2 Daikon

Daikon is a machine learning inference technique that infers likely invariants from execution traces. Daikon discovers likely program invariants by instrumenting the target program to trace variables of interest, executing the instrumented program, and generating the invariants over the instrumented variables. The generated invariants are expressed in the form of boolean expressions that represent properties involving a single variable, that is a constraint that holds over its values, or multiple variables, that is a relationship among the values of the variables.

Daikon works on a set of *(variable, value)* pairs and automatically generates relations that are satisfied by all the recorded pairs. The generated relations are filtered by probability thresholds to exclude incidental relations. Daikon initializes a set of expressions obtained by instantiating a predefined set of operators (the default is about 75 operators but user can extend the list of operators) on the considered variables, and then incrementally analyzes each variable and removes those expressions that are not satisfied by all the recorded *(variable, value)* pairs. For each of the resulting predicates, it computes a statistical index that indicates the probability that expressions are incidentally verified.

If we consider data collected for the class `Item`, at program points reported in Table 2.2, Daikon can infer invariants over each *(variable, value)* pair as shown in Table 2.3.

Statement	Variables	Invariants
`Item.23`	\result	$\backslash result > 0$
	this.quantity	$this.quantity > 0$
	this.price	$this.price > 0$
		$\backslash result = quantity * price$
`Item.18`	this.quantity	$this.quantity > 0$
	this.quantity'	$this.quantity' = quantity$
	quantity	$quantity > 0$
`Item.11`	this.price	$this.price > 0$
	this.price'	$this.price' = prc$
	prc	$prc > 0$
`Item.10`	this.quantity	$this.quantity \geq 0$
	this.quantity'	$this.quantity' = qty$
	qty	$qty > 0$

Table 2.3: *Example of invariants inferred using Daikon.*

For example, the invariants associated with each variables at line 10 indicates the following relations over the data observed:

- $\backslash result > 0$: return value is always greater than 0
- $this.quantity > 0$: variable `this.quantity` is always greather than 0
- $this.price > 0$: variable `this.price` is always greather than 0
- $\backslash result = quantity * price$: the return value is always equals to $this.quantity * this.price$

The relation between multiple variables is represented as a boolean expression obtained as a conjunction of terms, where each term is a single inferred property. So, the invariants associated with `Item.10` is:

$$\backslash result > 0 \wedge quantity > 0 \wedge this.price > 0 \wedge \backslash result = quantity * price$$

2.3 Finite state automata with annotations

FSAs can model sequences of events but they do not capture other important behavioral elements, like conditions and relations between values. To

build accurate behavioral models that integrate different aspects, for example event order and data flow information, classic inference techniques have been extended to generate annotated FSA [58]. There exist techniques that annotate FSA by adding information into the state, that is constraints that specify the concrete state of the program [29], techniques that annotate the transition with data about its probability, that is the likelihood of a transition to occur [70] [16], and other techniques that add information to the transition, that is constraints on the values that are assigned to the attributes associated with the events modeled by the transitions.

In the following sections, we discuss two techniques that can derive FSA annotated with data-flow information. The first is gkTail, which derives constraints that specify the concrete values that can be assigned to attributes. The second is kLFA, which labels transitions to represent the repeated occurrence of the attribute values across events, regardless of their concrete values. We firstly presents a running example that is used to illustrate the inference processes of annotated FSA of gkTail and kLFA

2.3.1 A running example

This Section presents the example that we use to illustrate the gkTail and kLFA approaches.

Let consider the code in Listing 2.3 that represent the component `ReserveFlight` that interacts with several Web Services to reserve flights for a number of people. In particular, it implements the logic to reserve the cheapest flight that allows a party of people to fly together from the start to the final destination. If there are no flights that allow the party of people to flight together from the start to the final destination, the application reserves different flights for different subsets of people. The application interacts with other components responsible for communicating with the Web Services required to finalize the job.

An interaction trace is a sequence of method invocations represented with the name of the invoked method, the values of the parameters and the return values, if any. For instance, the method `findBestSolution` returns an object of type `SingleSolution`, which includes a field `totalSeatsAvailable`. So, when executing `findBestSolution`, the trace will contain the pair ⟨`returnValue.totalSeatsAvailable`, 6⟩, which indicates that the field `totalSeatsAvailable` of the object returned by the

```
1  public class Booking    {
2
3          List<Flight> allInOneFlight = new ArrayList<>();
4          List<Flight> splittedInMultipleFlights = new ArrayList<>();
5
6          public void bookFlight( int persons, String from, String to, Date
               departure, Date back){
7
8                  Iterator<Airline> it = CompaniesRegistry.INSTANCE.
                       getCompanbiesIterator();
9
10                 if ( !it.hasNext() ){
11                         ErrorLogger.configurationError();
12                 }
13
14                 while ( it.hasNext() ) {
15                         Airline airline = it.next();
16                         processAirline(airLine);
17                 }
18
19                 if ( allInOneFlight.size() > 0 ) {
20                         SingleSolution solution = findBestSolution(
                               allInOneFlight);
21                         reservationMaker.book(solution);
22                 } else {
23                         CompositeSolution solution = findCompositeSolution(
                               splittedInMultipleFlights);
24                         reservationMaker.book(solution);
25                 }
26
27         }
28
29         private void processAirline ( AirLine airLine,int persons, String from
               , String to, Date departure, Date back ){
30
31                 List<Flight> flights = airLine.getAvailableFlights(persons,
                       from,to,departure,back);
32                 Iterator<Flight> it = flights.iterator();
33                 while ( it.hasNext() ) {
34                         Flight flight = it.next();
35
36                         if ( flight.getAvailableSeats() >= persons ){
37                                 allInOneFlight.add(flight);
38                         } else {
39                                 splittedInMultipleFlights.add(flight);
40                         }
41                 }
42
43         }
44
45 }
```

Listing 2.3: *An excerpt of the* `ReserveFlight` *component.*

Execution 1				
bookFlight persons=7 from=MXP to=NYC depDate= 03/18/10 retDate= 04/02/10	→**getCompaniesIterator** return=Iterator	→**hasNext** return=true	→**hasNext** return=true	→**next** return.name =“KLM”
→**getAvailableFlights** persons=7	→**iterator** return=Iterator	→**hasNext** return=true	→**next** return.availSeats=8 return.flightNo=KL1017	
→**getAvailableSeats** return=8	→**add** object.availSeats=8 object.flightNo=KL1017	→**hasNext** return=false	→**hasNext** return=false	→**size** return=1
→**findBestSolution** allInOne=List return.seats=7	→**book** solution.seats=7			
Execution 2				
bookFlight persons=4 from=BGY to=JFK depDate= 03/22/10 retDate= 03/31/10	→**getCompaniesIterator** return=Iterator	→**hasNext** return=true	→**hasNext** return=true	→**next** return.name =“Ryanair”
→**getAvailableFlights** persons=4	→**iterator** return=Iterator	→**hasNext** return=true	→**next** return.availSeats=9 return.flightNo=KL1027	
→**getAvailableSeats** return=9	→**add** object.availSeats=9 object.flightNo=KL1027	→**hasNext** return=false	→**hasNext** return=false	→**size** return=1
→**findBestSolution** allInOne=List return.seats=4	→**book** solution.seats=4			

Table 2.4: *1 of 2 - Traces of the execution of method* `bookFlight`.

method `findBestSolution` has value 6.

Tables 2.4, 2.5 show four traces recorded during the execution of method `bookFlight`. Values of the attributes associated with the invoked methods are reported below the name of the method.

2.3.2 gkTail

gkTail [55] is a technique that automatically generates Extended Finite State Automata (EFSA), that is automata augmented with constraints on transitions. gkTail derives EFSA from a set of interaction traces (positive samples) that include information about both the ordering of the events

Execution 3				
bookFlight persons=10 from=BGY to=JFK depDate= 03/22/10 retDate= 03/31/10	→**getCompaniesIterator** return=Iterator	→**hasNext** return=true	→**hasNext** return=true	→**next** return.name =“KLM”
→**getAvailableFlights** persons=10	→**iterator** return=Iterator	→**hasNext** return=true	→**next** return.availSeats=4 return.flightNo=KL1022	
→**getAvailableSeats** return=4	→**add** object.availSeats=4 object.flightNo=KL1022	→**hasNext** return=true	→**next** return.availSeats=7 return.flightNo=KL1028	
→**getAvailableSeats** return=7	→**add** object.availSeats=7 object.flightNo=KL1028	→**hasNext** return=true	→**next** return.availSeats=5 return.flightNo=KL1058	
→**getAvailableSeats** return=5	→**add** object.availSeats=5 object.flightNo=KL1058	→**hasNext** return=false	→**hasNext** return=false	
→**size** return=0	→**findCompositeSolution** allInOne=List return.seats=10		→**book** solution.seats=10	
Execution 4				
bookFlight persons=6 from=BDS to=CIA depDate= 03/16/10 retDate= 03/20/10	→**getCompaniesIterator** return=Iterator	→**hasNext** return=true	→**configurationError**	

Table 2.5: *2 of 2 - Traces of the execution of method* `bookFlight`.

and the values of the attributes associated with the events, such as the ones shown in Tables 2.4, 2.5. gkTail processes traces in four steps. In the first step, gkTail identifies similar traces, namely traces with the same sequences of method invocations and possibly different values of the parameters, and merges sets of similar traces into traces where method invocations are annotated with sets of attribute values. In the second step, gkTail derives constraints that represent the set of attribute values associated with the same method invocation. In the third step, gkTail creates an initial EFSA from interaction traces annotated with constraints. In the fourth step, gkTail iteratively merges states that can accept similar sequences of method calls.

Merging similar traces

In the first step, gkTail processes a sequence of interaction traces. Each interaction trace is a sequence of inter-component method invocations. Tables 2.4, 2.5 show four examples of interaction traces collected from the execution of the running example.

When the monitored component executes similar tasks, we obtain similar traces, namely traces that share the same sequence of method invocation and differ only for the values of the parameters and return values. For example, the first and second traces in Table 2.4 are similar. To produce models that capture the general nature of the interactions, gkTail identifies and merges sets of similar traces and produces traces where each method is associated with a set of parameter values. The set of values associated with a method in the merged traces corresponds to the parameter values associated to the same method in the original traces. For example, merging the first and second trace in Table 2.4 produces a trace whose first element is a call to the method `bookFlight` associated to a set that includes the following two items: { `persons=7 from=MXP to=NYC depDate=03/18/10 retDate=04/02/10, persons=4 from=BGY to=JFK depDate=03/22 /10 retDate=03/31/10` }.

Tables 2.6, 2.7 show the merged traces obtained from the interaction traces shown in Tables 2.4, 2.5.

Execution 1-2

bookFlight {persons=7 from=MXP to=NYC depDate=03/18/10 retDate=04/02/10, persons=4 from=BGY to=JFK depDate=03/22/10 toDate=03/31/10}	→**getCompaniesIterator** {return=Iterator, return=Iterator}		→**hasNext** {return= true, return= true}	→**hasNext** {return= true, return= true}	→**next** {return.name =“KLM”, return.name =“Ryanair”}
→**getAvailableFlights** {persons=7, persons=4}	→**iterator** {return= Iterator, return= Iterator}	→**hasNext** {return= true, return= true}	→**next** {return.availSeats=8 return.flightNo=KL1017, return.availSeats=9 return.flightNo=KL1027}		
→**getAvailableSeats** {return=8, return=9}	→**add** {object.availSeats=8 object.flightNo=KL1017, object.availSeats=9 object.flightNo=KL1027}		→**hasNext** {return= false, return= false}	→**hasNext** {return= false, return= false}	→**size** {return=1, return=1}
→**findBestSolution** {allInOne=List return.seats=7, allInOne=List return.seats=4}	→**book** {solution.seats=7, solution.seats=4}				

Table 2.6: *1 of 2 - The set of merged traces obtained from the traces in Table 2.4.*

Execution 3				
bookFlight {persons=10 from=BGY to=JFK depDate= 03/22/10 retDate= 03/31/10}	→**getCompaniesIterator** {return=Iterator}	→**hasNext** {return= true}	→**hasNext** {return= true}	→**next** {return.name =“KLM”}
→**getAvailableFlights** {persons=10}	→**iterator** {return=Iterator}	→**hasNext** {return= true}	→**next** {return.availSeats=4 return.flightNo= KL1022}	
→**getAvailableSeats** {return=4}	→**add** {object.availSeats=4 object.flightNo=KL1022}	→**hasNext** {return= true}	→**next** {return.availSeats=7 return.flightNo= KL1028}	
→**getAvailableSeats** {return=7}	→**add** {object.availSeats=7 object.flightNo=KL1028}	→**hasNext** {return= true}	→**next** {return.availSeats=5 return.flightNo= KL1058}	
→**getAvailableSeats** {return=5}	→**add** {object.availSeats=5 object.flightNo=KL1058}	→**hasNext** {return= false}	→**hasNext** {return= false}	
→**size** {return=0}	→**findCompositeSolution** {allInOne=List return.seats=10}		→**book** {solution.seats=10}	
Execution 4				
bookFlight {persons=6 from=BDS to=CIA depDate= 03/16/10 retDate= 03/20/10}	→**getCompaniesIterator** {return=Iterator}	→**hasNext** {return= true}	→**configurationError**	

Table 2.7: *2 of 2 - The set of merged traces obtained from the traces in Table 2.5.*

Generating constraints

Transition constraints are predicates that specify the values that can be assigned to the attributes associated with the events. In the case of the running example, constraints represent the values that can be assigned to parameters and return variables. gkTail generates predicates from interaction traces using Daikon. gkTail uses Daikon to transform the set of merged traces into a set of traces annotated with constraints.

Daikon can infer the predicate $x \geq 0$ from the values associated with the event `book` as shown in Figure 2.6.

Figure 2.6: *An example predicate generated by Daikon.*

Initializing EFSA

In the third step, gkTail builds an initial EFSA by simply creating a tree where each branch of the tree accepts a different merged trace. The initial EFSA is refined in the fourth and last step. Figure 2.7 shows the initial EFSA built from the traces in Tables 2.6, 2.7.

Merging equivalent states in EFSA

In the previous steps, gkTail generalizes the values of the attributes and produces transition constraints. In the last step, gkTail generalizes the ordering of the events and produces compact EFSA. The initial EFSA produced in the third step accepts only sequences of method calls that correspond to the input traces. However in general, the finite set of input traces is a sample of the infinitely many behaviors of a component. By generalizing the ordering of the events in the initial EFSA, gkTail extends the model of the program behavior including a possibly infinite set of sequences of

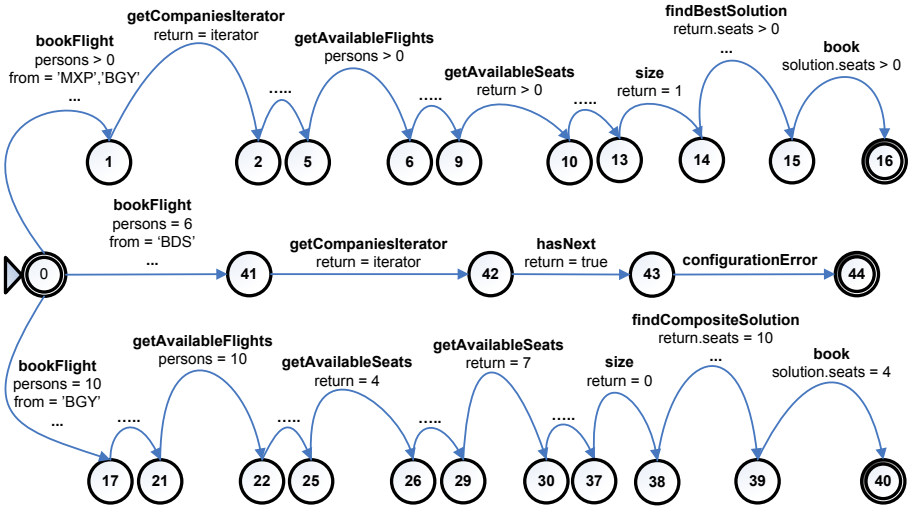


Figure 2.7: *The initial EFSA obtained from the traces in Tables 2.6, 2.7.*

method calls. gkTail generalizes the ordering of the events with a heuristic inspired from the heuristics proposed in the kTail technique that iteratively merges likely equivalent states.

The heuristics adopted by gkTail suggests to merge states that accept equivalent sets of behaviors up to a maximum length k (*k-future(s)*). The heuristic is based on the observation that the initial version of the model may include multiple representations of a same logical state, and merging states with the same future can expand and generalize the set of behaviors accepted by the model, likely increasing the model accuracy as well.

gkTail modifies the initial EFSA by iteratively merging the states with an equivalent *k-future*. gkTail merges states according to three equivalence criteria: *equivalence*, *weak subsumption*, and *strong subsumption*.

Two states are *equivalent* if the sequences of events in their *k-future(s)* are the same, and the predicates associated with each pair of corresponding events are equivalent. Figure 2.8 shows two states that are 2-equivalent in the running example.

A state s_1 *weakly subsumes* a state s_2 if the sequences of event in the *k-future* of s_1 and s_2 are the same, and the constraints in the *k-future* of s_1 are more general than the corresponding constraints in the *k-future* of s_2. Given a pair of corresponding events and their associated constraints P_1 and P_2, P_1 is more general than P_2 if whenever P_1 holds P_2 holds as well.

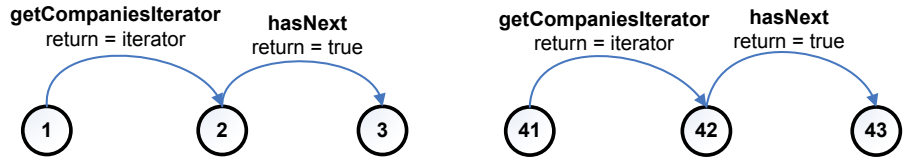


Figure 2.8: *Two 2-equivalent states.*

Figure 2.9 shows an example of a state (4) that weakly subsumes another state (20) for $k = 2$.

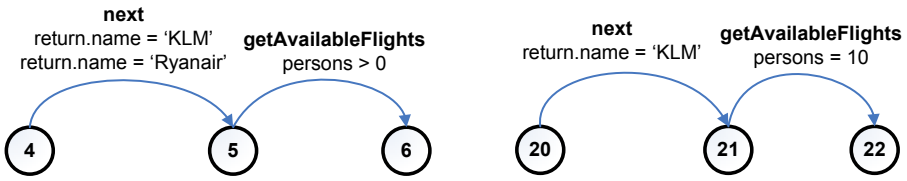


Figure 2.9: *State* 4 *weakly subsumes state* 20.

A state s_1 *strongly subsumes* a state s_2, if the sequences of event in the *k-future* of s_1 includes the sequences of events in the *k-future* of s_2, and the predicates in the *k-future* of s_1 are more general than the corresponding predicates in the *k-future* of s_2. Figure 2.10 shows an example of a state (4) that strongly subsumes another state (20) for $k = 2$.

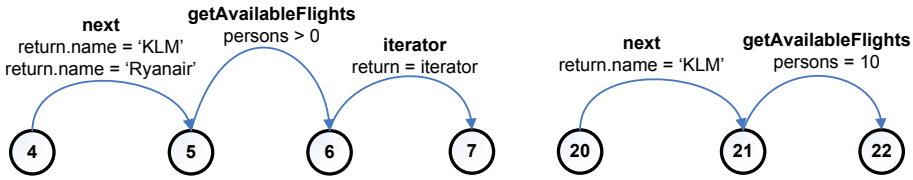


Figure 2.10: *State* 4 *strongly subsumes state* 20.

Given a merging criterion and a value for k, gkTail iteratively merges pairs of equivalent states until there are no equivalent states according to the given equivalence criterion. gkTail merges two states s and s' by removing the state s, adding to s' all incoming and outgoing transitions of s, and replacing the specific predicates with the more general predicates on the transitions (if strong or weak subsumption is applied). If a merg-

ing step produces an EFSA with two or more transitions that share input and output state, gkTail merges these transitions into a unique transition annotated with the OR disjunction of the predicates of the original transitions. Finally, if any of the merged states is a final state, the merged state is final. Figure 2.11 shows the EFSA obtained from the initial EFSA shown in Figure 2.7 with $k = 2$ and weak subsumption.

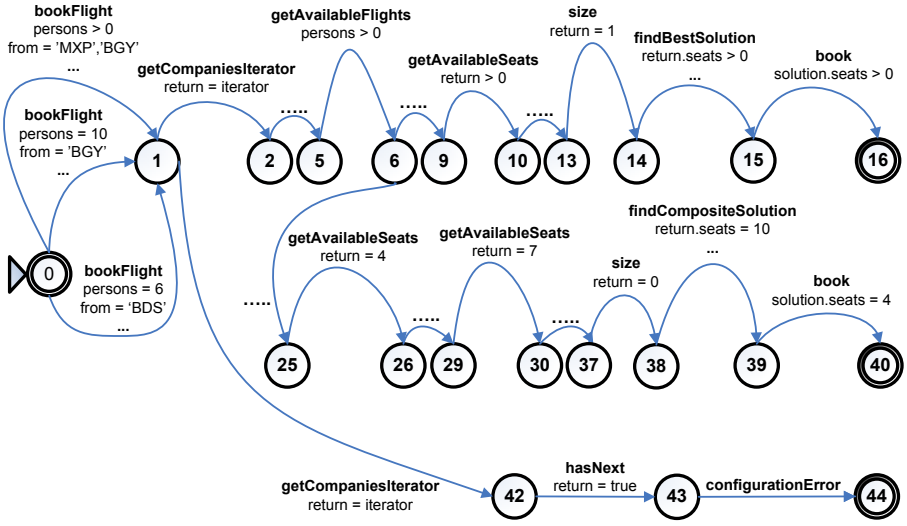


Figure 2.11: *The EFSA produced by gkTail from the interaction traces in Tables 2.4, 2.5 by using weak subsumption and $k = 2$ as merging criterion.*

gkTail correctly generalized some of the observed behaviors. For instance, the transition between states 37 and 38 shows that the component under analysis looks for composite solutions only if no normal solutions exist. On the other hand, the constrains associated to some transitions have limited validity with respect to the system. For instance, the model indicates that an invocations to `bookflight` method is accepted only for some specific values of the departure and destination airports, while the system behavior is more general. This is a typical effect of monitoring a limited set of executions of the component under analysis. This effect can be reduced by generating the model from a more thorough set of executions.

2.3.3 kLFA

KLFA is a technique that derives FSAs annotated with data-flow information [57]. While gkTail focuses on the values that are assigned to attributes, kLFA focuses on the patterns of occurrence of values across events within the same trace (we call these recurrences data-flow patterns). kLFA represents data-flow patterns by replacing the monitored events (both the event names and their attribute values) with new labels that do not include attribute values but incorporate information about the occurrence of the attribute values within the labels, as illustrated in the example in Figure 2.12. KLFA includes rewriting strategies that can identify different data flow patterns: *global ordering*, *relative to instantiation* and *relative to access* rewriting strategy.

The KLFA inference process consists of two phases: data preprocessing and model generation. In the data preprocessing phase, KLFA rewrites traces. In the model generation phase, KLFA infers a FSA that incorporates data-flow information from the preprocessed traces.

Preprocessing data

kLFA rewrites the events in the traces in three steps.

In the first step, kLFA identifies clusters of related attributes, that is attributes that refer to homogeneous types. This step avoids identifying data-flow patterns that incorrectly relate heterogeneous quantities. For instance, it may make sense to relate occurrences of values that represent distances, but it does not make sense to relate occurrences of values that represent distances with values that represent names of persons. For example, one of the data clusters that KLFA automatically identifies from the execution traces in Tables 2.4, 2.5 is composed of the following attributes: attribute `persons` of event `bookFlight`, attribute `persons` of event `getAvailableFlights`, attribute `return.seats` of event `findBestSolution`, attribute `return.seats` of event `findCompositeSolution`, and attribute `solution.seats` of event `book`.

In the second step, each cluster with homogeneous attributes is rewritten according to three rewriting strategies implemented by kLFA, thus producing three versions of each data cluster (*global ordering*, *relative to instantiation* and *relative to access* rewriting strategies).

The *global ordering rewriting strategy* replaces all occurrences of the same concrete value with a number. kLFA incrementally introduces numbers according to the order of appearance of new values. Thus, the first concrete value that occurs in a data cluster is rewritten with a 1, the second concrete value is rewritten with a 2 if never observed before, otherwise the same number is consistently used, and so on for all attribute values within a data cluster. The numbers represent the re-occurrence by abstracting from concrete values. The new event labels are obtained by concatenating the event names with the numbers produced by the global rewriting strategy.

Since the attributes of the events `bookFlight`, `getAvailableFlights`, `findBestSolution`, `findCompositeSolution` and `book` belong to the same data cluster, the global rewriting strategy will replace all attributes with the same symbolic value. Table 2.8 shows the symbolic values used to replace the concrete values associated with these attributes: column # indicates the position of the event in the original trace in Tables 2.4, 2.5, column *Events* reports the name of the event, column *Attributes* indicates the name of the rewritten attribute, while columns *Actual Values* and *GO* show the value associated to the attribute in the trace and the symbolic value derived by applying the *global rewriting strategy*. Table 2.8 shows that in all the four considered executions the *global ordering rewriting strategy* generates the same symbolic values for all the values in the data cluster thus suitably identifying their re-occurrence.

The *relative to instantiation rewriting strategy* aims to explicitly represent the re-occurrence of the generation and use of values rather than the re-occurrence of the same concrete values, to obtain a compact representation of produce-consume behavioral patterns. The relative to instantiation rewriting strategy rewrites values following the generation and use of the new values. Each time a new value occurs in a trace, it is rewritten with 0. If an existing value occurs in the traces, the value is replaced with a number that indicates the number of new values that have been introduced from its first occurrence plus 1.

Let us consider the sequence of events `next`, `getSeatsAvailable`, and `add` that occurs three times in the third trace in Table 2.5. The three occurrences of the sequence share a common data-flow pattern that indicates that the number of seats available remains constant within a cycle but changes among different cycles. Table 2.9 shows the attribute values

#	Events	Attributes	Actual Values	GO
Execution 1				
1	bookFlight	persons	7	1
6	getAvailableFlights	persons	7	1
15	findBestSolution	return.seats	7	1
16	book	solution.seats	7	1
Execution 2				
1	bookFlight	persons	4	1
6	getAvailableFlights	persons	4	1
24	findBestSolution	return.seats	4	1
16	book	solution.seats	4	1
Execution 3				
1	bookFlight	persons	10	1
6	getAvailableFlights	persons	10	1
23	findCompositeSolution	return.seats	10	1
16	book	solution.seats	10	1
Execution 4				
1	bookFlight	persons	6	1

Table 2.8: *Common patterns that spans over different executions.*

within these sequences (we omit the attribute values that do not belong to the same data-flow cluster to keep the table small). The column # shows the position of the event in the trace 3 in Table 2.5. The column *Events* reports the event names. The column *Attributes* indicates the names of attributes. The columns *Actual Values* and *GO* specify the attribute values and the corresponding symbols generated by the global ordering rewriting strategy. The last column *RI* reports the symbols generated by the relative to instantiation rewriting strategy.

The *relative to access rewriting strategy* replaces the first occurrence of a concrete value with 0, and the subsequent occurrences with a number that indicates the number of events observed from its last occurrence. Column *RA* in Table 2.10 shows the values produced by the relative to access rewriting strategy. We can observe that these values capture well the patterns occurring in these traces.

In the third step, kLFA heuristically identifies the best rewritten version of each cluster, among the three available alternatives. kLFA may select different rewriting strategies for different data clusters in the same system. The choice of a strategy mainly depends on the nature of the ob-

#	Events	Attributes	Actual Values	RI
8	hasNext			
9	next	return.availSeats	4	0
10	getAvailableSeats	return	4	1
11	add	flight.availSeats	4	1
12	hasNext			
13	next	return.availSeats	7	0
14	getAvailableSeats	return	7	1
15	add	flight.availSeats	7	1
16	hasNext			
17	next	return.availSeats	5	0
18	getAvailableSeats	return	5	1
19	add	flight.availSeats	5	1
20	hasNext			

Table 2.9: *A pattern of data reuse.*

#	Events	Attributes	Actual Values	RA
1	hasNext			
2	next	return.availSeats	4	0
3	getSeatsAvailable	return	4	1
4	add	flight.availSeats	4	1
5	hasNext			
6	next	return.availSeats	7	0
7	getSeatsAvailable	return	7	1
8	add	flight.availSeats	7	1
9	hasNext			
10	next	return.availSeats	4	4
11	getSeatsAvailable	return	4	1
12	add	flight.availSeats	4	1
13	hasNext			

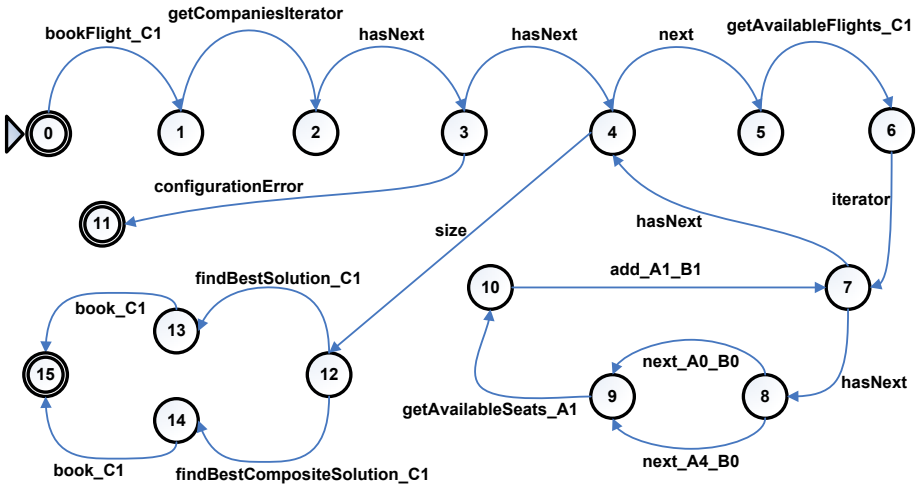
Table 2.10: *Capturing patterns with repeated values.*

served behaviors and on the collected data. KLFA automatically identifies the best rewriting strategy for each data cluster, based on the observation that the effectiveness of a rewriting strategy depends on the ability of capturing the regularity of the data flows. kLFA measures such regularity as the number of symbols used by a rewriting strategy to rewrite a data cluster: the smaller the number of symbols used to rewrite the concrete values, the better the rewriting strategy capture the regularity of the data flow.

Generating models

The rewritten traces are traces of event names where the attributes are implicitly represented as part of the event labels. Thus, we can generate models with classic engines to infer automata from traces.

KLFA uses the KBehavior incremental inference engine to infer automata. At each step, KBehavior reads a trace and updates the current FSA according to the content of the trace. The updated FSA generates all the traces that have been analyzed.



Legend

We added a letter that identifies the cluster before the number that rewrites a concrete value to increase the readability of the model. The strategies used to rewrite each cluster are:

A *Relative to Access* rewriting strategy

B *Relative to Instantiation* rewriting strategy

C *Global Ordering* rewriting strategy

Figure 2.12: *A kLFA model of the behavior of method* `bookFlight`.

Figure 2.12 shows the automaton that kLFA derives from the traces recorded during the execution of the running example. The model includes only the attributes reported in the traces of Tables 2.4, 2.5.

The automaton in Figure 2.12 suitably represents data-flow information. For example, kLFA used the *relative to instantiation strategy* to rewrite the second attribute (`return.fligtNo`) of the two transitions from state 8 to state 9. The resulting automaton indicates that new values are always introduced at that point of the execution (due to the presence of label `B0`), thus a new flight number is produced every time the loop through states 7, 8, 9 and 10 is covered.

Similarly, kLFA used the *relative to access strategy* to rewrite the first attribute (`return.availSeats`) associated with the transitions from state 8 to state 9. The symbolic values show that `return.availSeats` can be associated with either a new value (symbolic value equals to $A0$) or a value observed two iterations before (symbolic value equals to $A4$).

The transition from state 10 to state 7 shows that the two attributes of event `add`, `object.flightNo` and `object.availSeats`, are always equal to attributes `return.fligthNo` and `return.availSeats` associated with the event `next` (`A1` and `B1` denote the reuse of concrete values).

Finally, transitions from state 0 to state 1, from state 12 to state 13, and from state 13 to state 15 show an example of attributes rewritten with *Global Ordering*. Symbols associated with these values show that `persons`, `return.seats`, and `solution.seats` have the same values across all executions.

2.4 Models of the ordering of the events

We already discussed techniques that generate FSAs, invariants, and annotated FSAs from execution traces. Here we conclude this chapter by briefly discussing another kind techniques that generate behavioral models representing the ordering of the events.

Some mining techniques derive models different from FSA to represent information about the ordering of events. Some techniques simply derive a visual representation of the execution traces without mining any extra information that is not already in the traces. These technique are useful to

simplify the manual inspection of execution traces, but do not derive compact representations of the (general) program behaviors. For instance, techniques presented in [45] [63] can represent execution traces as sequence diagrams.

Other techniques analyze execution traces to derive frequent patterns of events [73] [50] [33]. Frequent patterns can be useful to discover anomalous behaviors, but their usage is restricted to anomalies that impact the frequent events. On the contrary, FSAs represent the entire behavior of a software program and can include all the events recorded in execution traces. Thus, FSAs can discover anomalies impacting any behavior of a software program, regardless the frequency of these behaviors. The cost of this wider scope of the model is a stronger dependency of the quality of the model on the completeness of the samples used to infer the model.

Other techniques mine temporal rules that capture a set of dependencies between events [51] [87]. These models focus on the relations between key events, rather then the exact ordering of the single events. Temporal rules have an interesting complementarity with respect to FSA: temporal rules can suitably represent relations between events occurring at arbitrary points of an execution, while FSAs can well represent relations between non-consecutive events only by suitably representing the relations between the intermediate events, which is typically harder to achieve. This complementarity has been exploited in [53] [78] to derive FSAs that satisfy inferred relations between events that occur at arbitrary points within traces.

Yet other techniques mine algebraic specifications and graph transformations from program traces. They rely on a ground mathematical background and can compactly represent a large number of behaviors [47] [39], but so far, they have been applied only to rather simple software components (single classes and containers), while FSAs have been shown to be useful with a broader set of applications.

Chapter 3

An Empirical Assessment of FSA Inference Techniques

If we want to study the effectiveness of behavioral inference techniques, with specific emphasis on finite state automata, we have to understand which kinds of models can be more effectively inferred. In fact, there exists at least two kinds of automata: simple automata and annotated automata. Annotated FSAs are automata that integrate information about the data-flow to also represent how data values affect the operations executed by programs.

The integration of information about operation sequences and data values into a unique model is indeed conceptually useful to accurately represent the behavior of a program. However, even if these models can represent behaviors that simple FSAs cannot represent, it is yet unclear whether extended models are generally more accurate than simple FSAs or not.

In this chapter, we present an empirical comparative study between techniques that infer FSAs and techniques that infer extended FSAs [54]. The goal is to understand the most promising research direction, in term of the kind of automata, for testing and analysis techniques based on inferred models. To compare the two classes of techniques, we use, as case studies, the techniques described in the previous chapter, that is kBehavior [59], kTail [20], gkTail [55] and KLFA [57]. In particular, we want to study the

trade-off between these two classes of techniques with specific reference to models extracted from software systems. The empirical study evaluates the performance of these techniques while varying the set of available traces from sparse, which is an extremely common case when traces are collected by testing software systems, to dense, which only happens for thoroughly tested systems. We compare the techniques that infer FSAs and extended FSAs according to the quality of the inferred models and the time required to obtain them.

3.1 Goals of the assessment

To better show the differences between using simple FSA and annotated FSA, let us consider the FSA in Figure 3.1 which represents how an application uses a file. The real usage of the file is abstracted on several aspects. The choice between reading only or reading and writing values is presented as a non-deterministic choice, while in reality it is determined by the opening mode. This lack of information can result in imprecise model-based analysis. For instance, such FSA, when used for analysis, cannot detect that an application opens a file with read mode and then illegally attempts to write values into the file.

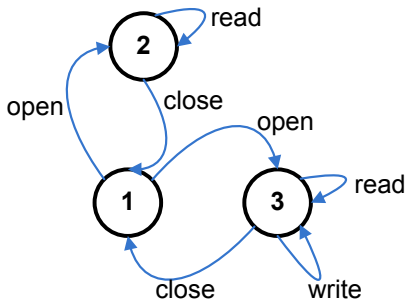


Figure 3.1: *A simple FSA that describes the usage of a file. This king of FSA can be inferred by both kTail and kBehavior.*

FSAs can be extended with the additional information required to represent constrained behaviors like the one exemplified above. For example, a more accurate model for the file reading/writing scenario is shown in Figure 3.2. Although this extension is indeed useful on a conceptual point of view, there exist no comparative and quantitative empirical study that

confirms the presence of a greater accuracy for extended models compared to simple FSAs. In addition, the inference of extended FSAs is more challenging than the inference of FSAs. An extended FSA, due to its capability to represent complex behaviors that cannot be represented with a simple FSA, may be poorly learned and cause inaccuracies in the representation of simple behaviors that are well captured by simple FSAs. The goal of our empirical validation is to understand and quantify these trade-offs.

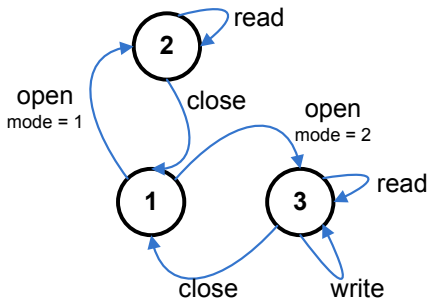


Figure 3.2: *An extended FSA that describes how a file is used taking into account how the file is opened. This king of FSA can be inferred by gkTail.*

In particular, our empirical comparison investigates the following three research questions:

R1 Do inference techniques producing extended FSAs generate models that can better identify legal behaviors as compared to those producing simple FSAs?

R2 Do inference techniques producing extended FSAs generate models that can better reject illegal behaviors as compared to those producing simple FSAs?

R3 What is the performance difference between the generation and the checking of extended FSAs as compared to those of simple FSAs?

These research questions are extremely important when these models are used to support automated software analysis, but are not necessarily critical research questions for every possible domain. For instance if the models must be manually inspected, human readability can be more relevant than model accuracy. We refer interested readers to [24] for a survey

of dynamic analysis techniques that can be used to support program comprehension.

3.2 Empirical setup

To answer each research questions, kTail, kBehavior, gkTail and KLFA are used to learn 7 models extracted from 7 real software systems: JFreeChart [6], Lucane [8], Thingamablog [13], Jeti [5], Columba [2], Open Hospital [9], and Rapid Miner [10]. We concentrate on models that represent the method invocations that can be generated when executing a method of the program. We choose such scale of models, opposed to models that represent the entire execution flow of a program, for two reasons: (1) many testing and analysis techniques use model inference to produce models of that scale [28] [16] [72] [55], thus it is an important scale in the practice; (2) we can manually produce the ideal models from the source code. A representation of the ideal models that must be learned is necessary to measure the quality of the inferred models. Considering huge models that represent the entire behavior of an application would introduce the issue of deriving the ideal models, in addition of being a scale that is seldom used in the practice.

Since all the inference techniques considered in this study must have a fair chance to be effective, we identified program models complex enough to include: behaviors that can only be represented with FSAs produced by KLFA, behaviors that can only be represented with EFSAs produced by gkTail, and several behaviors that can be represented with simple FSAs. To avoid collecting data biased by a single application, instead of using several models obtained from a same application, we only used one model per application ending up with 7 models.

To obtain a representation of the ideal models that should be inferred, we carefully inspected the applications' code, hand-draw models that represent the execution flow of methods that invoke other methods, and double-checked the correctness of the models with respect to the corresponding code. To choose the methods from which we extracted the execution flow, we looked for a complex method that satisfies the fairness principle mentioned above.

We represented the ideal models extracted from the source code as EFSAs according to the following semantics:

Application	N. States	N. Trans.	N. Const.	Method Name
JFreeChart	11	26	38	ChartFactory.createPieChart
Lucane	11	16	26	MessageHandler.run
ThingamaBlog	10	18	18	ParagraphComboHandler.actionPreformed
Jeti	12	18	38	Jeti.actionPerformed
Columba	13	18	14	FetchNewMessagesCommand.execute
OpenHospital	23	32	23	PatientBillEdit.getJButtonSave
Rapid Miner	19	32	16	DBScan.generateClusterModel

Table 3.1: *Data about the reference EFSAs.*

- *transitions labels* are method signatures and represent method invocations.
- *transition constraints* are boolean expressions that represent constraints on the values that can be assigned to program variables and parameters. For example, a constraint `file.status == 0` associated with the transition label `open(file, mode)` indicates that only closed files are accepted as a parameter.
- *parameter names* have global semantics, that is if the same variable name reoccurs in different signatures and constraints, its value in the same execution must always be the same. This semantics allows the generation of traces that are both coherent with the behavior of the programs and potentially useful to KLFA to identify recurrences of concrete values. If a parameter with the same name must be assigned with different values in different transitions, we simply change the name of the parameter in the different transitions to preserve the global semantics.

For the rest of the chapter, we refer to these ideal EFSAs as the *reference EFSAs*. Detailed data about the EFSAs used for the empirical assessment is presented in Table 3.1. Column *Application* indicates the application that has been used to obtain the reference model. Columns *Num States*, *Num Transitions* and *Num Constraints* specify the number of states, transitions and constraints in the reference model, respectively. Column *Method Name* indicates the name of the method that can produce the sequence of method calls represented in the corresponding model. An example of reference EFSA, derived from Lucane, is shown in Figure 3.3; Table 3.2, 3.3 show the corresponding mappings.

Edge	Method	Constraints
t1	m1	{}
t2	m1	{}
t3	m1	{c1,c2,c3}
t4	m1	{c1,c2,c3}
t5	m2	{c4}
t6	m3	{!c1,!c2,!c3,!c4}
t7	m4	{c5}
t8	m5	{c6}
t9	m6	{c7}
t10	m5	{!c7}
t11	m7	{c5, c7, c8}
t12	m8	{c6, c9}
t13	m9	{c8, c9, !c10}
t14	m9	{c8, c9, c10}
t15	m10	{}
t16	m11	{}

Table 3.2: *Mapping from transition to method and constraint ids.*

Identifier	Definition
m1	run()
m2	handleServerMessage()
m3	handleServiceMessage()
m4	getName()
m5	getApplication()
m6	ServiceManager.getInstance().getService()
m7	getUser()
m9	Store.getServiceStore().getService()
m10	isAuthorizedService()
m11	sendAck()
m12	process()
c1	!isAlreadyConnected
c2	!isAuthenticationMessage
c3	!isServerInfoMessage
c4	message.getApplication().equals("Server")
c5	userName
c6	serviceName
c7	s
c8	user
c9	service
c10	isAuthorizedService

Table 3.3: *Mapping from ids to actual methods and constraints.*

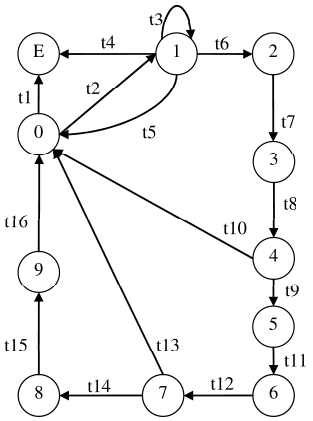


Figure 3.3: *The reference EFSAs extracted from Lucane.*

The effectiveness of the techniques experienced in our evaluation is influenced by two main factors: the value of the parameter k, which determines how much the inference techniques generalize the behavior represented by traces, and the completeness of the set of traces used to infer the models. In this study, the primary goal is to evaluate the effectiveness of the inference techniques when varying the completeness of the available traces. Intuitively it corresponds to measuring the effectiveness of the techniques according to the thoroughness of the test suites available for executing the program under analysis. We do not intend to study the sensitivity of the techniques to the choice of the parameter k, especially because there exist a number of studies that already show that small values of k, usually 2 or 3, are good choices when the inferred models are used to support software engineering tasks [72] [23] [59]. In line with these results, we run the inference techniques with a value of k equals to 2.

To study the effectiveness of the technique according to different levels of completeness of the traces, we inferred the models from traces that satisfy three coverage criteria: state coverage, transition coverage, 2-transition coverage. Intuitively *state coverage* corresponds to a sparse set of executions that do not exercise all the method invocations, i.e., transitions, but partially sample the behavior of the program. *Transition coverage* corresponds to a good set of executions that sample all methods invocations, but does not invoke the methods in all the possible execution contexts, e.g., loops are not necessarily executed multiple times. 2*-transition Coverage*

corresponds to a thorough test suite that samples each method invocation at least twice increasing the chance to execute method invocations in different contexts. We do not consider stronger coverage criteria because they would represent unrealistic scenarios with fairly complete sets of executions that are extremely hard to obtain in the practice.

We produce traces from a reference model by randomly traversing the EFSA from the initial state to a final state. If a final state has outgoing transitions, we randomly choose if ending the trace or continuing producing a longer trace. When traversing a transition that has one or more parameters, we randomly generate a value that satisfies the constraints associated with the transition. We continue generating traces until the selected criterion is satisfied. We produce traces based on a random strategy to avoid obtaining empirical data biased by the strategy used to cover models.

A final remark is about the constraints considered in the reference EFSAs. In the specific case of gkTail, we do not go into the issue of considering the inference of constraints that include operators that cannot be represented by gkTail. We are not interested in exploring the complexity of the constraints that gkTail cannot identify, but rather to evaluate its effectiveness to produce the right constraints, when there is a chance to do that. Thus, the reference EFSAs only include constraints that can potentially be inferred by gkTail.

In the following paragraphs, we describe the empirical process that we follow to answer the three research questions.

3.2.1 RQ1: Do inference techniques producing extended FSAs generate models that can better identify legal behaviors as compared to those producing simple FSAs?

To answer the first research question, we compute the recall (also known as true positive rate) of the models inferred with kTail, kBehavior, gkTail and KLFA, for all the coverage levels and reference models considered in our empirical assessment. Recall is a common measure used in information retrieval [56]. In our setting it measures the ability of the model in identifying correct behaviors. The intention here is to verify whether considering models more complex than simple FSAs result in an increment or a loss in recall.

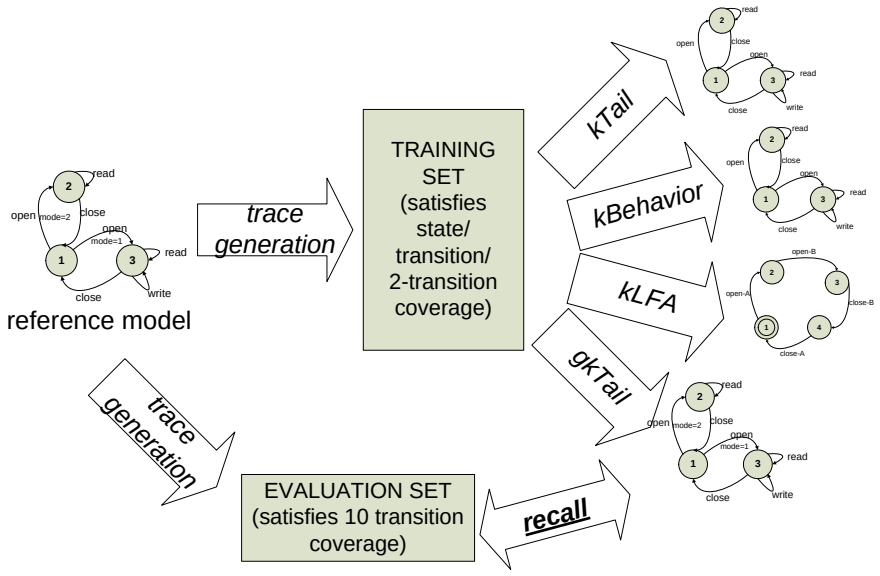


Figure 3.4: *The empirical process for the computation of the recall.*

To produce this quantification, we followed the process shown in Figure 3.4. We first generated three training sets from the reference models. The training sets satisfy the coverage criteria that have been previously described. We applied the four model-generation techniques considered in this study to infer models from the training sets. To compute the recall of the inferred models we generated a new set of traces, namely the evaluation traces, from the reference models. To deeply compare the reference and inferred models, the evaluation traces satisfy 10-transition Coverage, i.e., traces cover each transition at least 10 times. We finally computed the recall of the inferred models as the fraction of the evaluation traces that are correctly accepted by the inferred models. The more traces are accepted, the higher the recall of the inferred model is.

3.2.2 RQ2: Do inference techniques producing extended FSAs generate models that can better reject illegal behaviors as compared to those producing simple FSAs?

In order to answer this research question, we measure how many illegal behaviors are erroneously included into the models inferred by kTail, kBe-

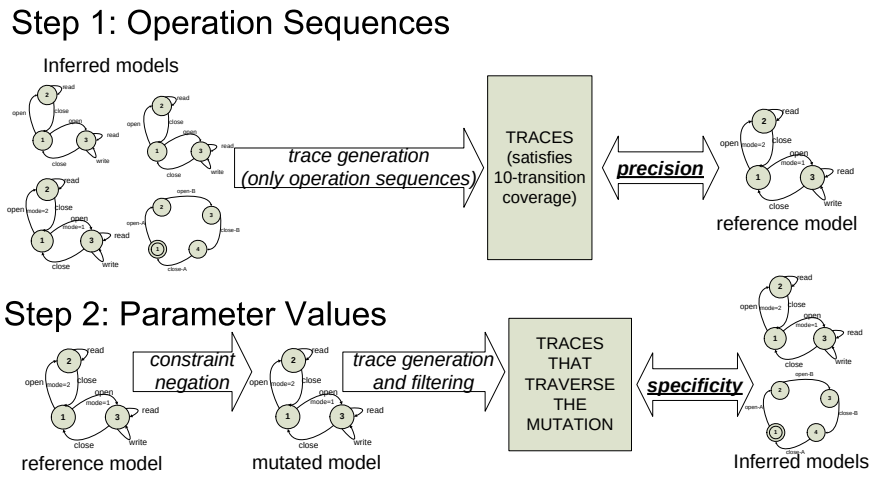


Figure 3.5: *The empirical process for the computation of precision and specificity.*

havior, gkTail and KLFA. In our context, a behavior can be illegal for any of the two following reasons: it includes an illegal sequence of operations, or it includes an illegal parameter value. Since only extended models can detect illegal parameter values, we evaluate the quality of the models with respect to these two classes of illegal behaviors separately. In this way, we can specifically compare and measure how much the capability of rejecting illegal behaviors due to the presence of illegal parameter values, which is a unique capability of extended FSAs, impacts on the capability to reject illegal behaviors due to the presence of illegal sequences of operations.

Figure 3.5 shows the process that we followed to study this research question. To measure how many illegal sequences of operations are erroneously included into inferred models we compute precision [56]. The precision quantifies the percentage of illegal behaviors that have been (erroneously) incorporated into the inferred models. A low precision indicates that many illegal behaviors are present in the inferred model, thus compromising its rejection capability. A high precision indicates that few illegal behaviors are present in the model, thus the model has an excellent rejection capability. Precision is computed by generating traces from the inferred models and checking the fraction of those traces that are accepted by the reference model.

To obtain a precise value of the precision, we generate traces until covering all transitions in the inferred model 10 times. Thus, the generated

traces cover each single operation in multiple contexts of use. When generating traces from extended models, we only generate traces with event sequences ignoring the parameter values, which are studied separately.

To study the capability to reject illegal behaviors due to illegal parameter values we measure specificity. Specificity (also known as true negative rate) [56] is obtained by producing illegal traces from the reference model and checking the fraction of traces that are correctly rejected by the inferred models. In this case, we did not use precision because the models inferred by KLFA do not specify the values that can be assigned to parameters, but they only specify how those values reoccur across events. Trying to produce traces (that include parameter values) from such models would result in traces with no-sense values, e.g., parameters with numeric values even if a string is expected by the reference model. The measure of the precision for KLFA would thus be biased.

To produce the traces with illegal parameter values, we mutated the reference automata by producing automata that exactly match the original ones with the exception of a randomly selected constraint that is negated. We then generate traces from the automata with the requirement to cover each transition 10 times. Finally, we filter those traces that do not traverse the mutated constraint, and we measure the fraction of traces correctly rejected by the inferred automata. We perform this evaluation only for gkTail and KLFA because we know in advance that kTail and kBehavior cannot reject any trace based on the parameter values only.

3.2.3 RQ3: What is the performance difference between the generation and the checking of extended FSAs as compared to those of simple FSAs?

To answer this research question, we measured both the time spent by the techniques to perform the inference and the time required to check traces. These measures give hints about the possible uses of the techniques. For instance, techniques that require a long time to generate models can be used when the set of the traces is quite stable over time, and cannot be effectively used when this set frequently changes. Similarly, techniques that require a substantially long time to check traces can be used in a human-driven environment, e.g., to support most of the known model-based anal-

ysis techniques, but might not be suitable to be used in a deployed systems as they can compromise performance. Note that the reported inference time includes all the operations necessary to obtain the models from the traces. For example, in the case of KLFA, this includes the time needed to run the rewriting strategy.

3.3 Toolset

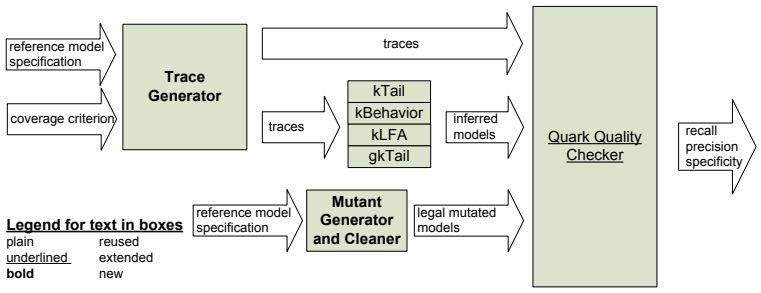


Figure 3.6: *The toolset that supported our empirical validation.*

To support the empirical validation, we extended existing tools and implemented new ones. In particular, we developed a trace generation tool, which implements the coverage criteria described in section 2, we developed a tool to mutate EFSAs and we extended the QUARK [52] platform. QUARK is a tool that can compare an inferred FSA with a reference automaton to compute some goodness measures. We extended the QUARK tool to support such analysis for the models inferred by gkTail and KLFA. In particular, we added the capability to compute recall, precision and specificity, for EFSAs, like the ones generated by gkTail, and FSAs with data flow information incorporated into labels, like ones generated by KLFA. Finally, the four inference techniques have been integrated in our toolset as black box components. Figure 3.6 shows the toolset.

3.4 Empirical results

Recall

To answer the first research question we computed the recall of the inferred models for a total of 21 cases per technique, obtained by experimenting the three coverage levels (state, transition, 2-transition) for each of the 7 case studies.

Application	kTail			kBehavior		
	state	**trans**	**2-trans**	**state**	**trans**	**2-trans**
JFreeChart	0.31	0.54	0.82	0.71	1	1
Lucane	0.79	0.87	0.92	0.92	0.98	0.97
ThingamaBlog	0.92	0.98	0.99	0.94	1	1
Jeti	0.8	0.83	0.95	0.95	0.9	0.98
Columba	0.68	0.88	0.85	1	1	1
OpenHospital	1	1	1	1	1	1
Rapid Miner	0.76	0.85	0.93	0.92	0.98	0.99
Average	**0.76**	**0.85**	**0.93**	**0.92**	**0.98**	**0.99**

Table 3.4: *1 of 2 - Empirical data about recall.*

Application	gkTail			KLFA		
	state	**trans**	**2-trans**	**state**	**trans**	**2-trans**
JFreeChart	0.26	0.4	0.61	0.3	0.59	0.7
Lucane	0.79	0.87	0.91	0.81	0.83	0.88
ThingamaBlog	0.91	0.97	0.99	0.88	0.99	0.99
Jeti	0.76	0.87	0.9	0.8	0.68	0.82
Columba	0.68	0.88	0.85	0.46	0.76	0.54
OpenHospital	0.95	1	1	0.9	0.92	0.97
Rapid Miner	0.74	0.85	0.89	0.84	0.97	0.91
Average	**0.74**	**0.85**	**0.89**	**0.71**	**0.82**	**0.83**

Table 3.5: *2 of 2 - Empirical data about recall.*

Tables 3.4, 3.5 shows the empirical data about recall. Figure 3.7 graphically shows the value of the recall for each technique, grouped by the coverage level. Each box associated with a technique spans from the the minimum to the maximum value of the recall observed among the 7 case studies, while the solid line indicates the average value of the recall computed among the 7 case studies. Gray boxes show the recall for techniques that do not work with data-flow information (i.e., kTail and kBehavior), while white boxes show the recall for techniques that infer data-flow information (i.e., gkTail and KLFA).

We can immediately notice from the plot that considering data-flow information causes a loss of recall. In case of gkTail compared to kTail the

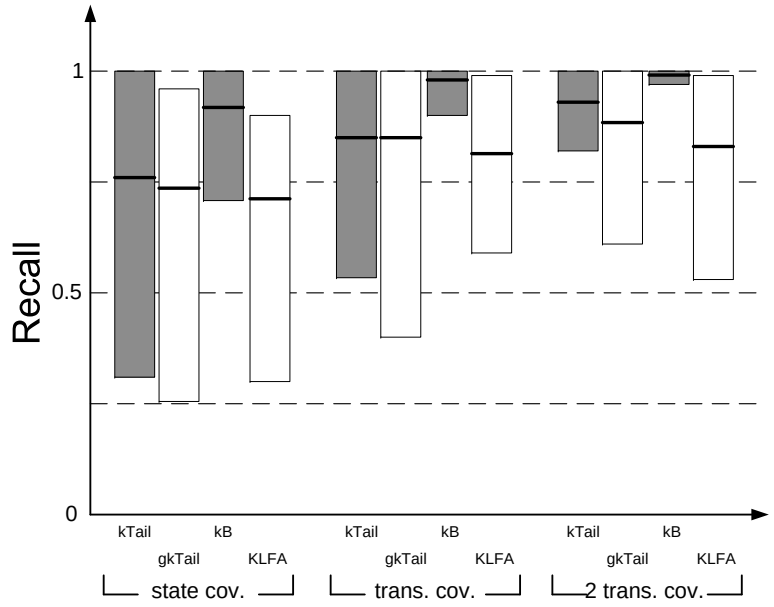


Figure 3.7: *Recall for different coverage levels.*

loss is moderate (0.03 in average), while it is relevant between KLFA and kBehavior (0.18 in average). We have a similar loss on the size of the boxes, which represents how varying the recall is. The recall of gkTail is slightly more variable than the recall of kTail, while the recall of KLFA varies a lot compared with the recall of kBehavior, which is quite close to its average value. These results indicate that it is harder to correctly learn how data values reoccur across events, as KLFA does, than learning constraints on data values, as gkTail does.

Even if gkTail performs slightly worse than kTail according to recall, the difference of recall between gkTail and kBehavior is relevant (0.15 in average). The small relative distance between kTail and gkTail might let us suppose that kBehavior could be ideally extended with heuristics similar to the ones used by gkTail to effectively work with constraints without losing much recall. Unfortunately there is no obvious way to adapt the heuristic incorporated in gkTail to kBehavior.

If we look at the trends of the techniques when the level of coverage increases, we have another relevant information. The recall and the stability of the results improve faster for the techniques that do not consider data-flow information (boxes are smaller and closer the top of the diagram). The

slow improvement of gkTail and KLFA is due to the amount of extra information needed by those techniques to produce accurate models: covering twice each transition is not always sufficient to discover enough data-flow information to significantly improve the model.

We now discuss the recall of the techniques in absolute terms. When traces only provide state coverage, kBehavior is the only technique with a high average recall. The results are more encouraging for all the techniques - especially for kTail and kBehavior - when traces provide transition coverage (average values are above 0.75 and 3 out of 4 techniques have boxes entirely above 0.5), and are definitely good for 2-transition coverage. These results suggest that FSAs with good recall can be effectively extracted even from a sparse set of executions, while extended FSAs require thoroughly tested software to derive models with good recall. An important research direction is thus the definition of techniques that can produce test cases that well exercise the program under analysis, as early investigated in [28].

Finally, a minor observation coming from the detailed data reported in Tables 3.4, 3.5. We can expect that inference techniques produce better models when more traces are available, but this is not always true for KLFA. In fact, data can be sometime confounding for the heuristics that extract data-flow relations. Consider for instance the empirical data reported for Jeti, Columba and Rapid Miner, where an increased coverage causes the generation of models with smaller recall.

Precision and specificity

To answer the second research question, we measured both the fraction of illegal sequences of operations that can be erroneously accepted by the inferred models and the fraction of operation sequences that include illegal parameter values that are correctly rejected by the inferred models. The former quantification is given by the precision and has been computed for all the techniques. The latter quantification is given by specificity and has been only computed for the techniques that can infer extended FSAs.

Tables 3.6, 3.7 shows the empirical data about precision. Figure 3.8 graphically shows the value of the precision for each technique, grouped by the coverage level. Boxes and solid lines have the same semantic as those in Figure 3.7.

Application	kTail			kBehavior		
	state	**trans**	**2-trans**	**state**	**trans**	**2-trans**
JFreeChart	1	1	1	0.41	0.23	0.49
Lucane	1	1	0.99	1	0.97	0.47
ThingamaBlog	1	1	1	0.29	0.19	0.96
Jeti	1	1	1	0.86	1	1
Columba	1	1	1	0.83	1	0.58
OpenHospital	1	1	1	0.69	0.07	0.1
Rapid Miner	0.92	0.94	0.86	0.38	0.18	0.1
Average	**0.99**	**0.99**	**0.98**	**0.64**	**0.52**	**0.53**

Table 3.6: *1 of 2 - Precision (limited to operation sequences).*

Application	gkTail			KLFA		
	state	**trans**	**2-trans**	**state**	**trans**	**2-trans**
JFreeChart	1	1	0.67	1	0.06	0.09
Lucane	1	1	1	1	1	1
ThingamaBlog	1	1	1	0.96	0.1	0.89
Jeti	1	1	1	0.96	1	1
Columba	1	1	1	1	0.52	0.66
OpenHospital	1	1	1	0.54	0.68	0
Rapid Miner	1	1	1	0.07	0.19	0
Average	**1**	**1**	**0.95**	**0.79**	**0.51**	**0.52**

Table 3.7: *2 of 2 - Precision (limited to operation sequences).*

The empirical data about precision show that extending techniques for the inference of FSAs with the capability of handling data-flow information positively affects the precision of the inferred model when the traces partially cover the reference model (compare gkTail with kTail for the state and transition coverage, and KLFA with kBehavior for state coverage), while it negatively affects precision when the set of available traces well cover the reference model (compare gkTail with kTail for the 2-transition coverage, and KLFA with kBehavior for transition and 2-transition coverage). In other words, techniques that infer simple FSAs can over-generalize the observed behaviors even when a limited number of traces is available, but they can also effectively handle an increasing number of traces. Techniques that infer extended models require more traces before producing over-generalized models, but after a while precision decreases quite fast. Overall, all the techniques show a worsening precision moving from transition coverage to 2-transition coverage. This can sound surprising, but many traces (e.g., traces that satisfy 2-transition coverage) can be difficult to handle compared to a good and limited set of traces (e.g., traces that satisfy transition coverage).

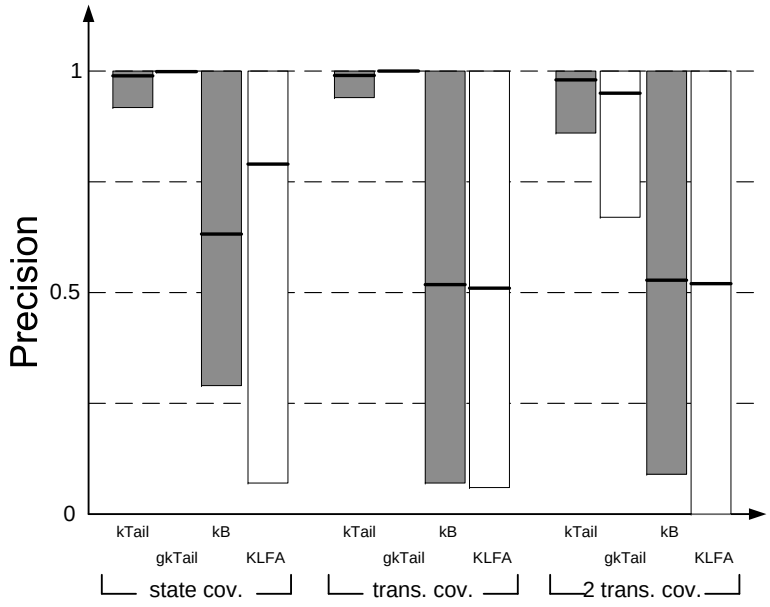


Figure 3.8: *Precision for different coverage levels.*

If we compare gkTail with KLFA, we can notice a significant difference on the precision (0.37 in average). This confirms the results already obtained with recall: learning how data values reoccur across events is harder than learning constraints.

If we look at the results in absolute terms, we can notice two main facts. First, gkTail and kTail have better precision than the other techniques. This suggests that when precision is important compared to other aspects (e.g., recall), one of these two algorithms should be selected, depending on the eventual necessity to have data-flow information incorporated in the inferred model. Second, the heuristic used to generalize the behavior about operation sequences strongly affects precision when parameter values are also handled. In fact, kTail and gkTail (which use a heuristic based on iterative state merging) have similar precision, kBehavior and KLFA (which use a heuristic based on incremental merging of sequences of operations) have similar precision as well, but the two classes of approaches show a very different precision. This is likely a consequence of defining mechanisms to handle data-flow information as an extension of existing heuristics that reason on the sequences of operations. It would be relevant to investigate the possibility to extend techniques that infer data-flow relations with the

capability to consider operation sequences, to build extended FSA. An early step into this direction is given by [29], which uses relations on attribute values to identify the states of the inferred FSA. It would be interesting to investigate how this approach can be further extended to derive extended FSAs.

Finally, we provide a comment looking at the detailed data reported in Tables 3.6, 3.7. The precision is not strictly decreasing when the number of available traces increases. Sometime increasing the number of traces causes over-generalization. This is true for kTail (e.g., see data for Rapid Miner), but it is definitely more evident for kBehavior and KLFA. It would be interesting to deeper investigate the relation between the characteristics of the traces and the precision of the inferred models.

Application	**gkTail**			**KLFA**		
	state	**trans**	**2-trans**	**state**	**trans**	**2-trans**
JFreeChart	0.79	1	1	0.92	0.68	0.51
Lucane	1	1	1	0.66	0.63	0.63
ThingamaBlog	0.94	1	1	0.59	0.07	0.01
Jeti	1	1	1	0.55	0.65	0.47
Columba	1	1	1	0.63	0.24	0.55
OpenHospital	1	0.91	0.91	1	1	1
Rapid Miner	1	1	1	0.96	0.42	0.46
Average	**0.96**	**0.99**	**0.99**	**0.76**	**0.53**	**0.52**

Table 3.8: *Specificity (limited to parameter values).*

Table 3.8 shows the empirical data about the specificity of the inferred models, computed with traces that should be rejected because they include illegal parameter values. We only report data about gkTail and KLFA because the specificity is 0 for kTail and kBehavior. Models inferred by gkTail had high-specificity in all the cases, while models inferred by KLFA provided very variable results, from extremely low specificity (0.01) to extremely high specificity (1).

The specificity of models obtained with gkTail indicates that most of the constraints have been correctly identified. On the contrary, KLFA well represented only part of the constraints, with a decreasing specificity when the number of traces increase (as already noticed this is due to the difficulty of properly handling a large amount of information).

Results show that gkTail is more effective than KLFA with parameter values, but we have to add some remarks. As mentioned in Section 2, we

considered reference models so that both gkTail had a chance to infer the constraints present in the model and KLFA had a chance to determine the recurrence of the parameter values. However, while all the constraints can be potentially identified by gkTail, not all the parameter values reoccur multiple times in a model. Thus, KLFA can necessarily identify illegal parameter values only when the illegal values are assigned to parameters that reoccur multiple times.

In general, software executions do not necessarily use parameter values constrained by expressions that can be inferred by gkTail. Illegal values assigned to parameters constrained with expressions that cannot be inferred by Daikon [35] would be hardly detected by gkTail. In the empirical evaluation, we decided to not consider the case of reference models with constraints not supported by Daikon because the validity of the empirical data would fall short. In fact, the set of constraints that can be inferred by gkTail can be easily extended by adding new operators to Daikon, making the obtained empirical data outdated. We think it is a more valuable and durable knowledge to know the specificity that can be expected when the "right" operators are supported by gkTail, leaving the issue of defining a set of operators that is appropriate for the kind of traces that must be analyzed open.

gkTail appears to be more effective than KLFA when applied to our reference models. However, this conclusion cannot be generalized to any domain because gkTail and KLFA focus on complimentary aspects (parameter values versus parameter recurrence), they can detect different kind of anomalous behaviors and different domains can better adapt to one technique than the other, e.g., log file analysis demonstrated to be a domain well suited for KLFA [57].

Inference and checking time

	kTail	**kBehavior**	**gkTail**	**KLFA**
Inference	5 sec	2 sec	38min	3sec
Checking	7 sec	6 sec	11sec	2sec

Table 3.9: *Average time to infer and check models.*

To answer the third research question we computed the average time spent by kTail, kBehavior, gkTail and KLFA to produce the models and

evaluate traces for acceptance (see Table 3.9). The average is computed over the 21 cases (7 case studies analyzed with 3 coverage levels each) considered for each technique. For the evaluation, we used a standard desktop computer (Intel Core Duo, 2GB RAM).

The generation of the models is fast for all the techniques (in average less than 5 seconds to process hundreds of traces), with the exception of gkTail that required an average of 38 minutes to generate the models. The long inference time is due to the many executions of the Daikon learner [35] that is integrated in gkTail and is used to produce the constraints included in the PTA. Thus gkTail can be effectively used only if the traces used to generate the models are quite stable over time, while the other techniques can be used more flexibly.

All the techniques can check traces fast. We have an exceptional time for KLFA, while we have longer times for kBehavior and kTail. This difference is mostly related to KLFA that integrates an optimized version of kBehavior, while the implementation of kTail and the stand-alone version of kBehavior are not optimized. We have reason to believe that these differences would not occur if all the algorithms had been implemented in an optimized way. gkTail requires more time than the other techniques to check traces because it must execute a constraint solver to verify if data values satisfy constraints. We can also observe that checking constraints impacts the performance, but it has no dramatic effect, according to our empirical experience.

3.4.1 Threats to validity

A threat to validity is related to the limited number of case studies analyzed in this empirical evaluation. Even if the results obtained with 7 models cannot be fully generalized, the empirical observations that we collected are confirmed by each single case study. Moreover, the case studies have been carefully selected to be both realistic (they have been extracted from real software systems) and relevant for the comparison (they include an interplay of aspects relevant for all the compared techniques). We thus think that the empirical experience reported in this work provides relevant insights on the inference of extended models from software and highlights important aspects that should be taken into account when using model inference techniques that extract extended models.

Another threat to the validity of the empirical validation is related to the exclusion of some of the configurations that are acceptable for the techniques from the analysis. For instance, we consider one heuristics out of the three that are defined for gkTail and one rewriting strategy out of the three defined for KLFA. Even if the empirical evaluation is partial, we considered comparable configurations for all the techniques. This choice results in a focused evaluation of the effects related to the inclusion of data-flow information into the inferred models. The analysis of other configurations is surely interesting, but it would introduce differences due to the choice of weaker or stronger generalization criteria hiding the effect related to the introduction of data-flow information into models.

Similarly, we run the empirical experience for a single value of the parameter k, supported by all the techniques. Even if the value of k affects the inference, we relied on the numerous application of model inference to software engineering tasks to run the comparison with a meaningful value. Studying the impact of the choice of k, to the results even if interesting, is outside the scope of this work. We plan to empirically analyze this aspect in the future.

3.5 Discussion

The main conclusions that can be derived from our empirical work are:
Including data-flow information in the inferred models is expensive. In this empirical study, techniques that infer extended FSAs showed a number of shortcomings compared to techniques for the inference of simple FSAs: they always lose recall, they lose precision when many traces are available, and constraints requires a large amount of time to be inferred (this last point is true for gkTail only). Thus, unless your analysis really needs to infer data-flow information, avoid using techniques for inferring extended models. On the other hands, extended FSAs well represent data-flow information (see data about specificity). Thus, if your analysis really needs data-flow information, techniques like gkTail and KLFA can be helpful.

Generally, some "good" traces are better than many "good" traces. We can expect that the more traces we have the better the techniques work. However, this empirical validation showed that it is not true in general,

even if traces do not increment only in number but also in quality (intended as providing additional levels of coverage). Recall tends to strictly increase when the coverage of the reference model increases, but precision decreases from a certain point. Thus, continuously accumulating traces can be harmful for inference techniques. This study suggests that collecting a proper number and type of traces is a relevant aspect, and defining techniques for the design of test suites that properly support model inference is an important challenge. This is especially true for techniques that infer extended models because they perform quite bad both when there are few traces (i.e., bad recall for traces that satisfy state coverage) and many traces (i.e., bad precision for traces that satisfy 2-transition coverage).

Sometimes many "good" traces are better than some "good" traces. The strictly increasing recall is an important element of this study. Several analysis techniques use inferred models to check executions and identify anomalous behaviors. This specific application of inferred models usually requires the generation of few false positives, which are particularly annoying for users, despite the possibility to miss some relevant anomalous executions. Such requirement demands techniques that learn models with high recall (i.e., remembers correct behaviors) even at the cost of a moderate precision (i.e., accepts some bad behaviors). In this empirical study, the scenario that better satisfies this requirement is the case of traces that satisfy 2-transition coverage. Thus, specific domains can demand specific strategies.

Data-flow patterns are harder to be inferred than constraints. KLFA showed to be less effective than gkTail: less recall, less precision, less specificity and less stable results. This result tells us that learning data-flow patterns is difficult and more research is needed into this direction to increase the quality of the inferred models. We also have to say that KLFA has been designed to analyze the log files typically recorded by enterprise applications, which are quite different from the traces we considered in this study. Thus, we cannot reasonably expect different results when KLFA is applied to log file analysis.

Constraints can be conveniently learned only from stable sets of traces. The inference time required by gkTail is several order of magni-

tude larger than the time required by the other techniques. This result indicates that it is not convenient to apply gkTail to sets of traces that frequently change, rather it should be applied to a set of traces that is reasonably stable over time.

The size of the model appears to be relatively important. In this study, techniques obtained better results with the two largest models (Open Hospital and Rapid Miner). This supports the intuitive idea that the size of the reference model is only moderately important, rather the complexity of its structure influences more the results. However, we also noticed that the precision produced by kBehavior and KLFA for the two largest models is the lowest. Thus, this intuitive notion needs additional focused empirical studies to understand if and to what extent the size of the model can influence the quality of the inference.

In summary, even if annotated FSAs can be useful, this empirical investigation shows that their inference is definitely harder than the inference of simple FSAs. Techniques for the inference of annotated FSAs need to be further improved before being applicable in settings where executions partially sample the model to be inferred, which is a common scenario for software systems. For these reasons, our research focused on the definition of strategies to improve the inference of simple FSAs from software (see Chapter 4 and 5), leaving the definition of similar strategies for annotated FSAs for the future.

Chapter 4

White-box Detection of Interaction Models from Service-Based Applications

In this chapter, we present a white-box technique, called SEIM [60], for the inference of precise behavioral models that represent interactions between an application and a third-party service, provided as Web Service.

Accurate models of the interaction protocols, i.e., models of the interactions between applications and the integrated Web services, can facilitate both manual and automatic inspection and analysis, and can support the many existing model-based verification and validation techniques [46].

Extracting interaction protocols from service-based applications consist in describing the set of interactions that can be executed when a system interacts with the integrated services.

SEIM statically derives accurate models of the interactions between applications and Web Services, in the form of Finite State Automata (according to the empirical study, presented in the previous chapter, the inference of simple FSAs should be preferred to the inference of annotated FSAs, unless annotations are of critical relevance for the models). SEIM contributes to the state-of-the-art in two major ways: it proposes a model refinement technique to identify and eliminate many infeasible behaviors

from inferred models, thus alleviating the problem of false positives that reduces the effectiveness of many static analysis approaches [74]; it generates models that distinguish the likely feasible interactions from the interactions with an unknown level of feasibility, thus allowing engineers to distinguish the relevance of the produced information.

4.1 Static extraction of interaction models

SEIM is specifically effective for service-based applications thank to the following aspects that facilitate the static extraction of the interaction models:

Independent states: client applications cannot share their state with Web Services, i.e., executing Web Service operations does not alter the state of the client application.

Easily identifiable services: in client applications, the interactions with Web Services are usually mediated by stubs that are easily automatically identifiable.

Easily distinguishable services: client applications usually instantiate a different type of stub for each used Web Service, thus, the stub type identifies the target Web Service.

The SEIM technique produces an FSA model of the synchronous requests that the application under analysis produces when interacting with a set of Web Services. SEIM works in three main steps, as shown in Figure 4.1.

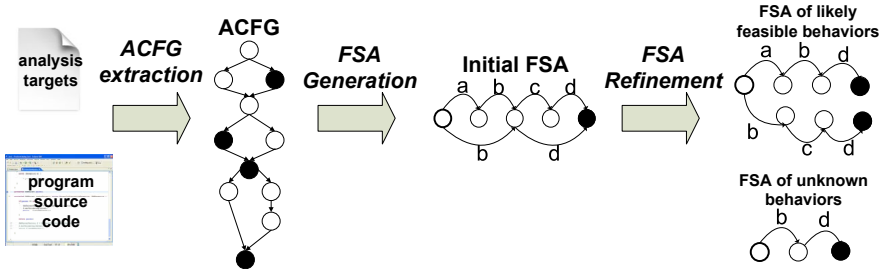


Figure 4.1: *The main steps of the SEIM technique.*

In the first step, SEIM derives an Annotated Control Flow Graph *(ACFG extraction)*. Software engineers indicate the *program to be analyzed* and

the *analysis targets*, which restrict the set of service operations that can be included in the inferred model. This step produces an ACFG, that is an inter-procedural control flow graph of the program under analysis, where the nodes that represent service invocations are annotated with the name of the invoked operations. This model includes all the service invocations and the information about the possible execution flow, and it is the basis of our analysis.

In the second step, SEIM generates the initial FSA (*FSA generation*). It first reduces the ACFG by removing all the nodes and the transitions that are not relevant for the analysis. In particular, it preserves only the nodes and transitions that are related to either calls to service operations or to the application control flow. It then transforms the reduced ACFG into an equivalent FSA.

In the third step, SEIM prunes infeasible sequences of calls, and generates two FSAs that distinguish likely feasible sequences of calls from sequences of calls whose feasibility is not known (*FSA refinement*). One of the FSA produced by SEIM accepts only likely feasible sequences of calls, while the other FSA accepts sequences of calls whose feasibility is not known. SEIM marks a sequence of calls as a sequence with unknown feasibility when it cannot determine if there exists a concrete execution that corresponds to that sequence.

In the following, we first present a running example of an application that integrates eBay Web Services to provide searching and trading operations [3]. Then, we present the analysis of the running example to show how SEIM can extract a precise representation of the interactions between the application and the eBay Web Services.

4.1.1 A running example

This Section presents an excerpt of two methods of a Java application, that provide searching and trading operations for eBay Web Services.

The two methods that we analyze are `findBestExpiringItem` and `getItemByBid`. The method `findBestExpiringItem` searches an eBay auction and finds the product with the lowest cost among the ones that satisfy the following requirements: they match the description passed as parameter, have an actual cost below a given threshold, and are offered

by sellers with positive feedbacks. The method `getItemByBid` adds the product passed as argument to the user watchlist, and makes an offer for the item to beat the best offer. Listing 4.1 shows an excerpt of the methods

4.1.2 ACFG extraction

In the ACGF extraction step, SEIM produces an annotated control flow graph of the target system. The inputs of this step are the component to be analyzed and the analysis targets that include both a list of *stub types* and a *usage protocol*.

SEIM interprets interactions with *stubs* as interactions with the Web Services, i.e., an invocation of a method implemented by a stub represents an invocation of a Web Service operation. The *usage protocol* is given as a FSA, where transitions are labeled with the names of methods implemented by the interface of the component under analysis.

In our running example, the stub types are `EBayAPIInterface` and `ShoppingInterface`, the interface methods that are analyzed are `findBestExpiringItem` and `getItemByBid`, and the usage protocol is specified by the FSA shown in Figure 4.2 (software engineers are welcome to specify other usages of interest. They may for example look for invocations executed within a loop or for the independent execution of the two methods under analysis).

SEIM generates the ACFG by first generating the Inter-Procedural Control Flow Graph (ICFG) of each interface method, and then combining the ICFGs according to the usage protocol. The generation of the ACFG differs from the standard generation of an ICFG in the way the methods implemented by the stub objects are analyzed. When an invocation of a method implemented by a stub object is considered, SEIM does not generate the ICFG corresponding the method, but simply annotates the node that corresponds to the method invocation with the name of the invoked method. SEIM does not add the ICFG of the method to the ICFG under construction either. Note that the construction of the ACFG requires the analysis of all the methods invoked from the interface methods. We implemented the ACFG extraction step for Java programs by extending the SOOT toolkit [11]. The ACFG extracted from the running example is too large to be shown in the paper. Figure 4.4 shows a reduced ACFG.

SEIM combines the ACGFs of the single methods by simply replacing

```
1  public static String findBestExpiringItem(String productDescr, Double price) {
2      Long positiveFeedback = null;
3      SimpleItemType bestItem = null;
4      String itemToBuy = "";
5          ...
6
7          FindItemsAdvancedResponseType items;
8      items = findItemsAdvanced(productDescr, price);
9      ...
10
11     for (SimpleItemType item : items.getItem()) {
12
13           if (item.isReserveMet()) {
14                  SimpleItemType detailedItem = getSingleItem(item.getItemID())
                        ;
15              GetUserProfileResponseType sellerInfo = getUserProfile(
                     detailedItem.getSeller().getUserID());
16
17              if (positiveFeedback == null) {
18                 bestItem = detailedItem;
19                 positiveFeedback = sellerinfo.getFeedbackHistory().
                        getUniquePositiveFeedbackCount();
20              } else {
21
22                 if (sellerinfo.getFeedbackHistory().
                        getUniquePositiveFeedbackCount() > positiveFeedback) {
23                    bestItem = detailedItem;
24                    positiveFeedback = sellerinfo.getFeedbackHistory().
                           getUniquePositiveFeedbackCount();
25                 }
26              }
27           }
28        }
29     }
30
31     if (bestItem!=null) {
32         bestItem = getSingleItem(bestItem.getItemID());
33
34         if (bestItem.getListingStatus() == ListingStatusCodeType.ACTIVE) {
35                ...
36             itemToBuy = bestItem.getItemID();
37         }
38     }
39
40     return itemToBuy;
41 }
42
43 public void getItemByBid() {
44         ...
45
46         if (isInWatchList != true) {
47            addToWatchList(itemToBuy);
48         }
49
50         ItemType biddenItem;
51         biddenItem = getItem(itemToBuy);
52
53         if (isInWatchList != true) {
54
55            if(biddenItem.getSellingStatus().getMinimumToBid().getValue() <=
                    maxBid) {
56               placeOffer(itemToBuy, maxBid);
57            } else {
58               removeFromWatchList(itemToBuy);
59            }
60
61         } else {
62
63            if(!(biddenItem.getSellingStatus().getHighBidder().getUserID().
                    equals(getUser()))){
64               placeOffer(itemToBuy, maxBid);
65            } else {
66               ...
67            }
68         }
69         ...
70 }
```

Listing 4.1: *An excerpt of* `findBestExpiringItem` *and* `getItemByBid`.

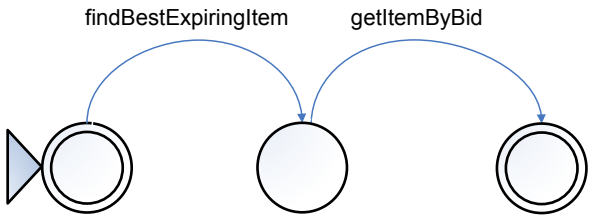


Figure 4.2: *Usage protocol for the interface methods analyzed in the running example.*

the transitions in the usage protocol given as input with the ACFG of the methods that correspond to the label of the transitions. In particular, given a transition t from a state s_1 to a state s_2 labeled with an interface method m, and the ACFG $acfg(m)$ extracted from the analysis of the method m, SEIM removes the transition t, merges s_1 with the entry node of $acfg(m)$, and merges s_2 with the exit node of $acfg(m)$.

4.1.3 FSA generation

In this step, SEIM transforms the ACFG produced in the former step into an FSA that represents the interactions between the component under analysis and the target services.

SEIM starts by eliminating data that are irrelevant for the analysis. The only data in the ACFG that are relevant for identifying sequences of requests are invocation and control nodes. Invocation nodes represent the invocation of Web service operations and are annotated with the operation names. Control nodes represent non-sequential execution flows, and are the nodes with more than 1 incoming edge or more than 1 outgoing edge. SEIM removes from the ACFG all nodes except invocation and control nodes. Figures 4.3 and 4.4, show the reduced ACFG, respectively for the methods `findBestExpiringItem` and `getItemByBid`.

After eliminating irrelevant nodes, SEIM transforms the ACFG into an equivalent FSA. An ACFG is a Moore machine [64], i.e., an automaton whose outputs depend only on the state. We can produce an equivalent FSA, i.e., an FSA that accepts the same language accepted by the ACFG, in few trivial steps. We first add a new final state to the ACFG. We then connect all former final states of the ACFG to the newly added final state

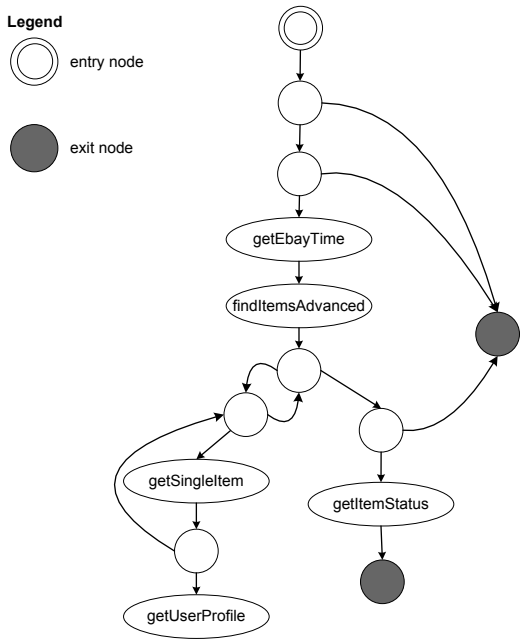


Figure 4.3: *The reduced ACFG for the* `findBestExpiringItem` *method.*

with transitions, and we label all the transitions with the label of the source node, if not empty, or with the empty label ϵ otherwise. We finally transform the FSA in a deterministic one. The obtained FSA represents the interactions between the component under analysis and the Web Services when the component under analysis is executed according to the usage protocol passed as parameter to SEIM. Figure 4.5 shows the FSA obtained for the running example.

4.1.4 FSA refinement

The FSA produced by the second step over-approximates the behavior of the target component: it models the possible interactions with the specified Web services along with many infeasible ones. In the third step, SEIM ranks sequences of operations as infeasible, likely feasible, and with unknown feasibility, eliminates the infeasible sequences, and split the FSA into two FSAs that distinguish likely feasible sequences from sequences with unknown feasibility.

In a nutshell, the process works as follow. SEIM generates a finite set

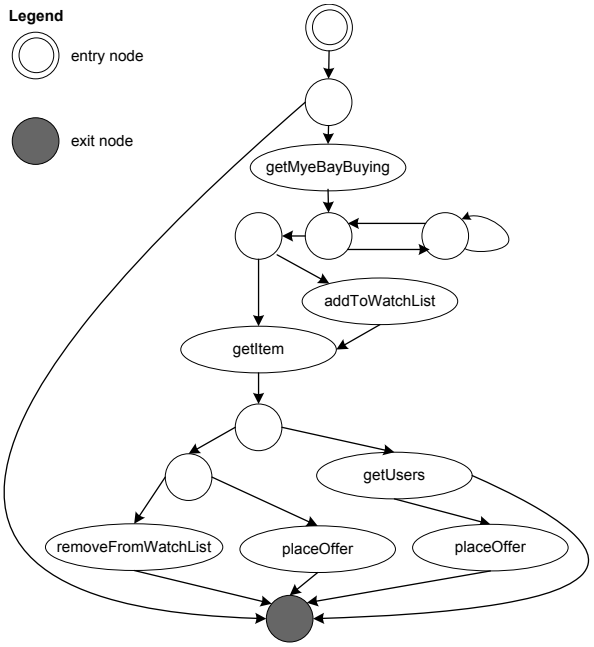


Figure 4.4: *The reduced ACFG for the* `getItemByBid` *method.*

of sequences from the current FSA, passes these sequences to a *concrete execution discovery engine* that provides information about the feasibility of the sequences, and refines the FSA according to the collected information. A concrete execution discovery engine is any technique that can identify concrete executions that satisfy certain properties. In our experiments, we considered the property that consists of covering specific sets of statements an exact number of times and in a given order. This criterion is stronger than statement coverage, but weaker than path coverage, because it ignores the coverage of sub-paths that do not include calls to Web services. The effectiveness of the concrete execution discovery engine is extremely important for the effectiveness of SEIM.

Several analysis techniques can play the role of concrete execution discovery technique, for instance concolic execution [75], symbolic execution [7], and random testing [67]. The current SEIM prototype implementation uses the JPF symbolic executor [7] as concrete execution discovery engine.

In the following we illustrate how SEIM generates a finite set of sequences from the current FSA, how SEIM obtains information about the feasibility of the sequences, how SEIM refines the FSA according to the col-

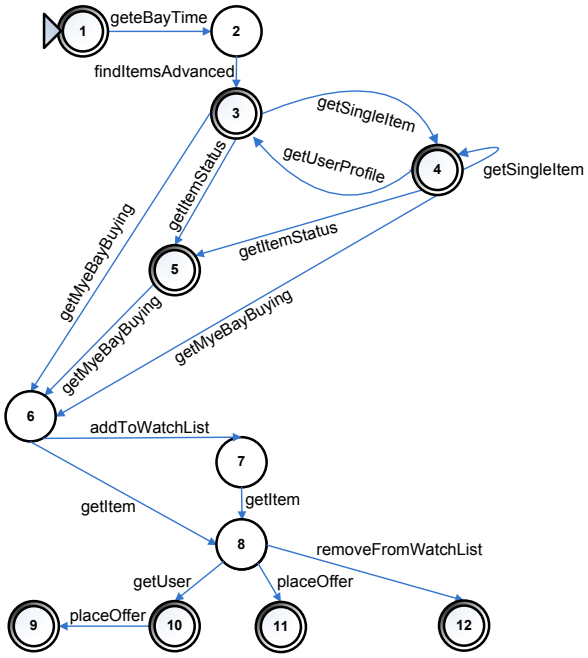


Figure 4.5: *The interaction protocol produced for the running example.*

lected information, and the result obtained for our running example.

Sequence generation

The concrete execution discovery engine (JPF) works on a finite set of behaviors (sequence of calls to Web Service operations) that cover a given set of FSA elements. SEIM is not bounded to a specific coverage criterion, but can use different criteria depending on the goals. In SEIM, the goal is to identify and eliminate as many infeasible behaviors as possible within reasonable execution boundaries. We thus decided to cover not only all the statements and the elementary control flows of the FSA (node and statement coverage), but also a sample of repeated sub-behaviors (loop coverage). In this way, SEIM collects enough information to successively eliminate infinitely many infeasible behaviors from the FSA, when the unbound repetition of some loops in the FSA is infeasible. In the running example, we generated a set of behaviors that covers all loops of the FSA that traverse the same state no more than L times. Our experience with SEIM

indicates that with $L = 3$ SEIM can already identify and eliminate many infeasible behaviors efficiently. The choice of a proper value for L may depend on the application domain and the nature of the code. Tuning the choice of the coverage criterion and the parameter values is part of our ongoing research work. For the running example, SEIM produces 252 call sequences (behaviors to be checked for feasibility) that cover all loops up to a depth of 3 ($L = 3$).

Feasibility analysis

To determine the feasibility of a given sequence of calls, SEIM attempts to generate a concrete execution that traverses the statements that call the Web Services the number of times and in the order indicated by the sequence. In the following, we indicate these statements as call statements for brevity. SEIM uses JPF to explore the set of behaviors extracted from the FSA according to the chosen coverage criterion. Here we illustrate the process by referring to a single sequence of statement calls. Generalizing the process to a finite set of sequences complicates the presentation but does not include complex technical steps. To check for the feasibility of a sequence of call statements, SEIM modifies the program under analysis by introducing a string variable that represents the sequence of call statements traversed by the current execution. SEIM initializes the string to an empty value, and incrementally adds to the string the call statements traversed while executing the program with JPF. JPF drives the symbolic exploration of the execution space by trying to produce a string that matches the target searched sequence. Thus, when the string variable added to the program does not match the prefix of the target sequence, JPF can immediately exclude the corresponding portion of the execution space. For instance, if SEIM looks for a sequence with calls to `m1`, `m2` and `m3`, JPF looks for executions that produce a value `"m1 m2 m3"` for the added string. When the value of the string is inconsistent with the searched value, for example `"m1 m3"`, JPF can exclude the corresponding portion of the execution space without further exploring it.

In summary, the overhead introduced by JPF is limited by three main factors: SEIM invokes JPF only once for the whole set of sequences to be analyzed, limits the exploration of the execution space by binding the number of loop executions to L, and prunes the portions of the space to be explored

according to the given sequences as discussed above.

For each call sequence, the concrete discovery engine, in our case JPF, can produce three results: feasible, infeasible or unknown. SEIM classifies a call sequence as feasible, if JPF finds at least a concrete execution that matches the sequence. SEIM classifies a call sequence as infeasible, if JPF can show that there exists no concrete execution that matches the sequence. SEIM classifies a call sequence as unknown, if JPF does not reach a conclusion. Typically, this happens when JPF does not terminate because it cannot handle some symbolic expressions derived during the exploration of the execution space.

Refinement

SEIM uses the information about infeasible and unknown sequences to generate two FSAs representing these two classes of sequences. Here we illustrate the process of building the two FSAs. We first build two Prefix Tree Acceptors (PTA): `I-PTA` for infeasible sequences, and `U-PTA` for unknown ones, with the classic algorithm described by Bierman and Feldman in [20]. Paths in a PTA represent single executions with no loops, thus simply refining the FSA produced after the second step by removing the behaviors accepted by the `I-PTA` and the `U-PTA` can eliminate at most a finite set of infeasible behaviors. To generalize the finite set of executions and prune a possibly infinite set of executions, SEIM transforms the PTAs in FSAs by using the heuristic of the kTail inference algorithm proposed by Bierman and Feldman [20]. The kTail heuristics merges sets of likely equivalent states that are defined as pairs of states that accept the same behaviors up to length k. This heuristic produces two automata that include loops and other complex behaviors: `IA` from `I-PTA`, and `UA` from `U-PTA`, respectively. Since SEIM explores concrete executions by exploring loops up to length L, it makes sense to assign a similar value to the k parameter of the kTail heuristic. In the running example, we use $k = 4$.

SEIM uses `IA` and `UA` to refine the automaton `A` produced after the second step into two final automata `F-Aut` and `U-Aut`. The automaton `F-Aut` accepts the likely feasible behaviors of the program under analysis, and is defined as $\texttt{F-Aut} = (\texttt{A} \setminus \texttt{IA}) \setminus \texttt{UA}$. The automaton `U-Aut` accepts only the behaviors with unknown feasibility, and is defined as $\texttt{U-Aut} = \texttt{UA} \cap \texttt{A}$. In the formula above $\cap$ represents the intersection between automata, and $\texttt{A} \setminus \texttt{B}$

is defined as $A \cap \overline{B}$, where $\overline{B}$ is the complement operator [44]. The two automata provide a detailed representation of the behaviors of the component under analysis, together with important information about the feasibility of the behaviors.

4.2 Discussion

In our running example, JPF terminates the analysis of all the 252 sequences, thus it does not find sequences with unknown feasibility. This result depends on the ability of JPF to handle all the statements in the component under analysis, but is not necessarily true for an arbitrary program. JPF classifies 198 sequences as infeasible, and 54 as feasible. The large amount of infeasible sequences, which represent behaviors that appear to be feasible from the ACFG, but are not feasible when analyzing the concrete execution conditions in the code, depends on restrictions in the combination of methods. As in many industrial cases, several behaviors that are feasible at the component level depend on the specific combination of the single units. The ACFG simply combines different behaviors according to the static structure of the code and thus cannot reveal the many infeasible combinations. On the contrary, SEIM refines the initial ACFG with an estimation of the feasibility of the call sequences, and can correctly identify many infeasible combinations. Thus, it produces a more accurate model.

For example, one of the execution flows of method `findBestExpiring-Item` returns an empty string that prevents further requests to Web services in method `getItemByBid`. The ACFG contains many infeasible sequences that include a call to method `findBestExpiringItem`, which returns an empty string, followed by calls to method `getItemByBid`, while SEIM prunes all these infeasible sequences.

Figure 4.6 shows the refined model produced for the running example. The readers may notice that the FSA in Figure 4.5 seems to be easier to inspect by software engineers because of its regularity and correspondence to the code. However, the FSA before refinement shown in Figure 4.5 is less precise than the refined FSA in Figure 4.6, and thus the FSA before refinement is much less useful than the refined FSA for verification and validation. To confirm this intuition, we measure the quality of the model extracted by SEIM in the running example in terms of precision and recall

before and after the refinement. We obtained the data with the QUARK framework [52] that can automatically generate traces from an inferred and a reference model, and compute precision and recall with respect to the FSA that exactly matches the behavior of the program. Table 4.1 summarizes the results.

	precision	**recall**
FSA before refinement	0.32	1
FSA after refinement	0.9	0.92

Table 4.1: *Precision and Recall.*

The precision and recall of the FSA built before the refinement step confirm that this FSA is a complete (recall = 1), but imprecise (precision = 0.32) representation of the possible interactions of the component under analysis. The precision and recall of the refined FSA confirm that the SEIM effectively removes most infeasible sequences (precision = 0.9) missing only few correct behaviors (recall = 0.92) The absence of an FSA with unknown behaviors in the case study indicates that the SEIM refinement step has been precise.

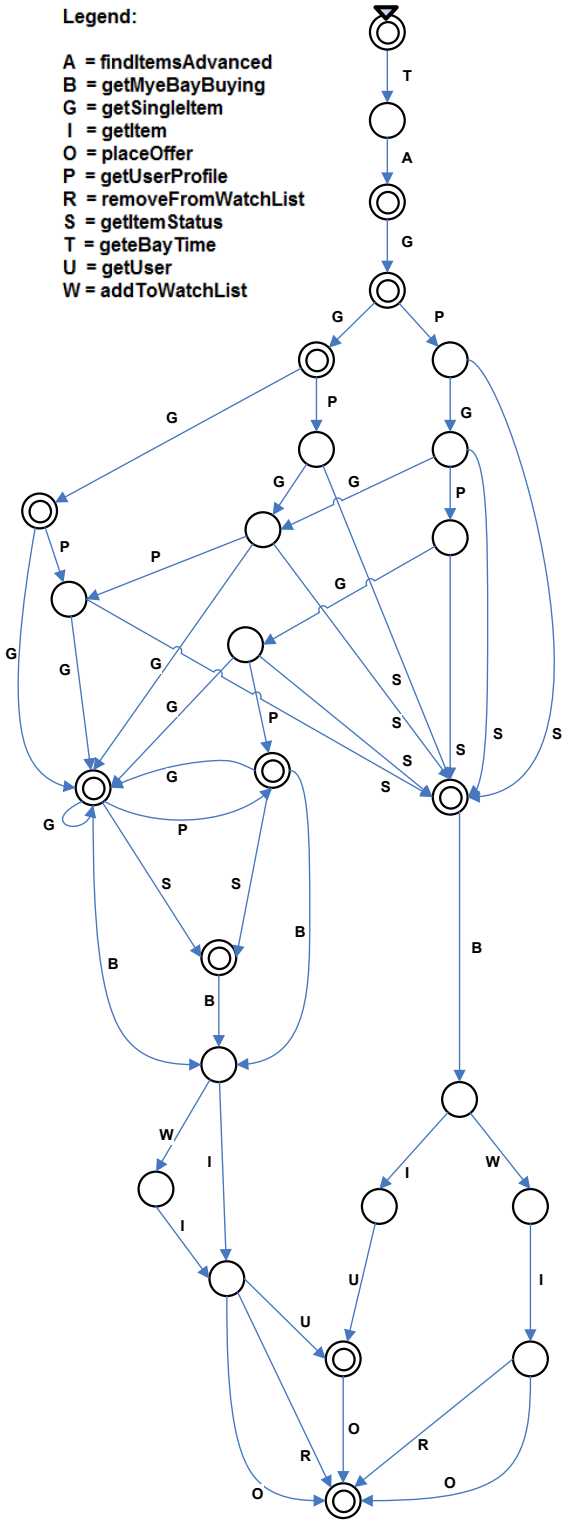


Figure 4.6: *The refined FSA for the running example.*

Chapter 5

Black-box Detection of Interaction Models from GUI-Based Systems

In Chapter 4 we presented SEIM [60], a static analysis technique that extracts interaction models for Web Service applications. SEIM effectively limits models imprecision by identifying and eliminating many infeasible behaviors from the inferred models. SEIM also generates information about the precision of the identified behaviors by distinguishing likely feasible behaviors from interactions whose feasibility is unknown. Figure 5.1 graphically shows the kind of improvement introduced by SEIM.

Static analysis techniques require source code to be successfully applied. Unfortunately this requirement hinders the applicability of these techniques, because systems can be provided without source code. Furthermore static analysis techniques produce scarce results when source code contains complex programming structures (for instance, the use of polymorphism, dynamic class loading, and aliases reduces the effectiveness of the analysis). When systems cannot be effectively analyzed with static analysis, an effective approach is to use information available at run-time. This complementary approach uses only dynamic behaviors without deriving a model that accepts a super-set of the possible behaviors and then refining it. Rather an inference strategy based on dynamic information builds models from behaviors observed at run-time. Such strategy can be completely

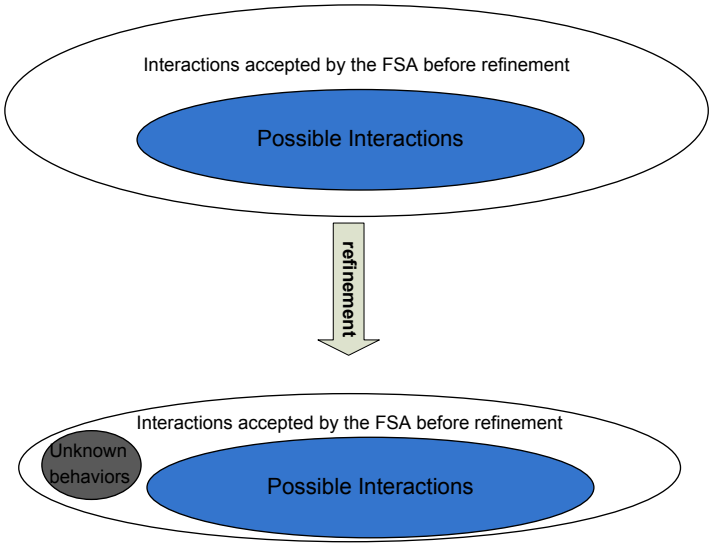


Figure 5.1: *Model refinement with SEIM.*

based on a black-box data.

In this chapter, we present a black-box technique that incrementally produces new executions useful to increase the completeness of the inferred models. The technique uses machine learning [77] to explore the execution space.

5.1 Overview of the technique

Deriving complete models from program executions requires the extensive exploration of the execution space. The key idea of the technique presented in this chapter is the use of machine learning to emulate the activity of the user, automatically understands how an application can be used, and thus extensively sample the execution space.

The process to generate a behavioral model is composed of two phases: the exploration phase, which discovers and produces new executions, and the inference phase, which generates an FSA from traces recorded during the exploration phase.

In the exploration phase, the technique uses reinforcement learning [77] to understand how to interact with the application and incrementally produce new executions. Reinforcement learning includes learning and decision-

making. The former is used to learn how the target application can be used and the latter is used to produce new executions. While new executions are produced, our technique records traces. In this chapter, we illustrate the technique with specific reference to the derivation of a model that represents interactions between an application and Web Services. Thus, traces consist in sequence of invocations of web service operations.

In the second phase, the technique generates behavioral models that summarize interactions with Web Services in the form of finite state automata.

In the following, we first present a running example of an application that integrates the Twitter Web Service. Then, we detail how the technique works. Finally, we present the empirical results obtained from the analysis of the running example, which show that the technique can extract a precise and complete model of the usages of the Tweeter Web Service, by interacting with the GUI of the application.

5.2 Running example

The running example is Twitthere [15], a desktop client Java application that implements a GUI for interacting with the Twitter Web Service. Figure 5.2 shows a screenshot of the GUI implemented by Twitthere. At the startup of the application, no tweets appear in the GUI. Tweets can be added by using the GUI, and thus interacting with the Tweeter Web Service.

The operations of the Twitter Web Service that are used by the Twitthere application are shown in Table 5.1. The first column describes the action that can be executed by using the widgets specified in the second column. The second column indicates the GUI elements that must be used to produce the service invocation indicated in the last column. The numbers in the second column refer to the widgets in Figure 5.2.

5.3 Learning from system interactions

During the exploration phase the technique uses a learner and decision-maker to execute the target system and records behavioral information. Exploration is treated as a reinforcement learning problem. The learner

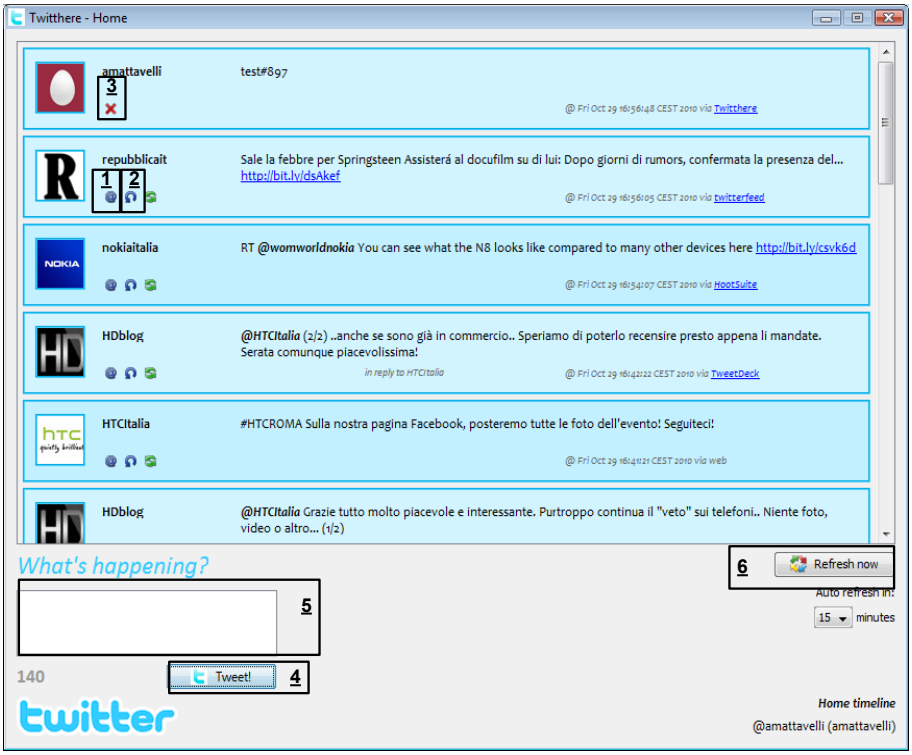


Figure 5.2: *A screenshot of the Twitthere application.*

and decision-maker are both implemented in a single component that it is called agent. The agent interacts interacts with an environment, that in our case is the GUI of the application. A GUI [89] is composed of a set of widgets $W = \{w_1, \ldots, w_n\}$ consisting of executable GUI elements, e.g., buttons, text fields, etc.. Each widget $w_i \in W$ has a set of properties $P^{w_i} = \{p_1^{w_i}, \ldots, p_m^{w_i}\}$, e.g., color, size, font, etc., and each property $p_i^{w_i} \in P^{w_i}$ can assume a set of values $V^{p_i^{w_i}} = \{v_1^{p_i^{w_i}}, \ldots, v_k^{p_i^{w_i}}\}$, e.g., red, bold, 16pt, etc.. Interactions consist in the execution of the widgets, e.g. clicking a botton, filling in a text field, and marking a check box. The reaction of the environment consists in a new screen that is presented to the agent.

The structure of an agent-environment interaction is shown in Figure 5.3, where at each time step t, the agent receives the current representation of the GUI-state $s_t \in S$, where S is the set of all possible GUI-states. Given the current GUI-state, the agent selects a GUI-action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions available in state s_t. One time step later the

WS Action	Widget combination	API (twitter4j [14])
mention a tweet	1;5	`Twitter.retweetStatus(statusID)`
reply to a tweet	2;5	`Twitter.updateStatus(text, statusID)`
cancel an own tweet	3	`Twitter.destroyStatus()`
write a tweet	4;5	`Twitter.updateStatus(text)`
refresh	6	`Twitter.getHomeTimeline()`

Table 5.1: *The actions that can be executed with Twitthere, the widgets that must be used to perform these actions and the corresponding Web Service operation that is executed.*

agent receives a numerical reward [77] r_{t+1}, and the GUI reaches a new state s_{t+1}.

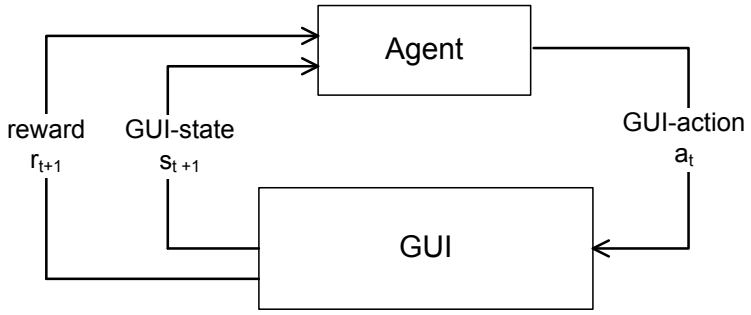


Figure 5.3: *The agent-environment interaction in reinforcement learning.*

In particular, a *GUI-state* $s_t \in S$ is composed of a set of triples and is defined as:

$$\{(w_1, P^{w_1}, S^{P^{w_1}}), \ldots, (w_n, P^{w_n}, S^{P^{w_n}})\}.$$

$S^{P^{w_1}}$ is the set of values of the properties in P^{w_1} for each widget $w_i \in W$. It is defined as $\{v_1^{p_1^{w_i}}, \ldots, v_m^{p_m^{w_i}}\}$. A *GUI-action* is a function $a_t(s_t) = s_{t+1}$ that changes the GUI-state. s_{t+1} is the GUI-state resulting from the execution of action a_t on GUI-state s_t. The reward function computes the *reward* r_{t+1} by mapping each GUI-state transition to a value that varies from 0 to 1 and is defined as:

$$\frac{\#NewElems(W_{t+1}, W_t) + \sum_{w \in W_{t+1} \bigcap W_t} rateChangedProps(w, t+1, t)}{|W_{t+1}|}$$

where

$$\#NewElems(W_{t+1}, W_t) = |W_{t+1} \setminus W_t|,$$

$$RateChangedProperties(w, t+1, t) = \frac{\#Diff(P^W, S_{t+1}^{P^W}, S_t^{P^W})}{|P^W|},$$

and $\#Diff(P^W, S_{t+1}^{P^W}, S_t^{P^W})$ returns the number of properties in P^W that have a different value in the sets $S_{t+1}^{P^W}$ and $S_t^{P^W}$.

The reward estimates how good it is to perform a given GUI-action in a given GUI-state (or how good it is for the agent to be in a given GUI-state). Since the long-term objective of the agent is to maximize the total reward it receives, and since our goal is discovering new ways to execute the system, the return value is higher when the GUI-state resulting from the execution of an action significantly differs from the previous state. Intuitively a high value corresponds to the possibility to access a completely different area of an application, thus giving high chance to discover new executions.

At each time step, the agent has a given probability of selecting each possible GUI-action. This probability results from a policy [77], π_t, where $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$. The policy defines how the agent interacts with GUIs. A policy is a mapping from perceived GUI-states of the environment to a set of probabilities for the actions enabled at that state. The agent alternate the selection of the action with the highest probability to random actions and changes its policy as a result of its experience. The goal is to maximizing the total amount of reward it receives over the long run, that is the optimal policy. The agent finds the optimal policy by computing the action-value function $Q^*(s, a)$ or value function $V^*(s)$ [77] that return the value of taking a GUI-action a in a GUI-state s under a policy π. A policy π is defined to be optimal if it returns values greater than or equal to that of any other policy π', for all GUI-states.

In order to execute the system and record interaction traces, the technique runs steps within episodes. A simple step is a time step as shown in Figure 5.3 (thus a single action is executed in a time step), and episodes end when a maximum number of steps is reached. Each step represents a simple system interaction and an episode produces an interaction trace. All episodes share the same initial GUI-state s_0.

Once traces have been collected, the behavioral model can be inferred with any inference technique that works with positive traces. Here we used the kTail inference technique proposed by Biermann and FeldMan [20].

5.4 Toolset

To implement the technique described in this chapter, we defined the tool showed in Figure 5.4 that integrates IBM Rational Functional Tester [4], a suite for functional and regression testing, and Teachingbox [12], a toolbox for training robots. IBM Functional Tester is used to interact with GUIs of desktop applications independently from the specific library that has been used to implement the application under analysis. Teachingbox [12] includes the reinforcement learning algorithm.

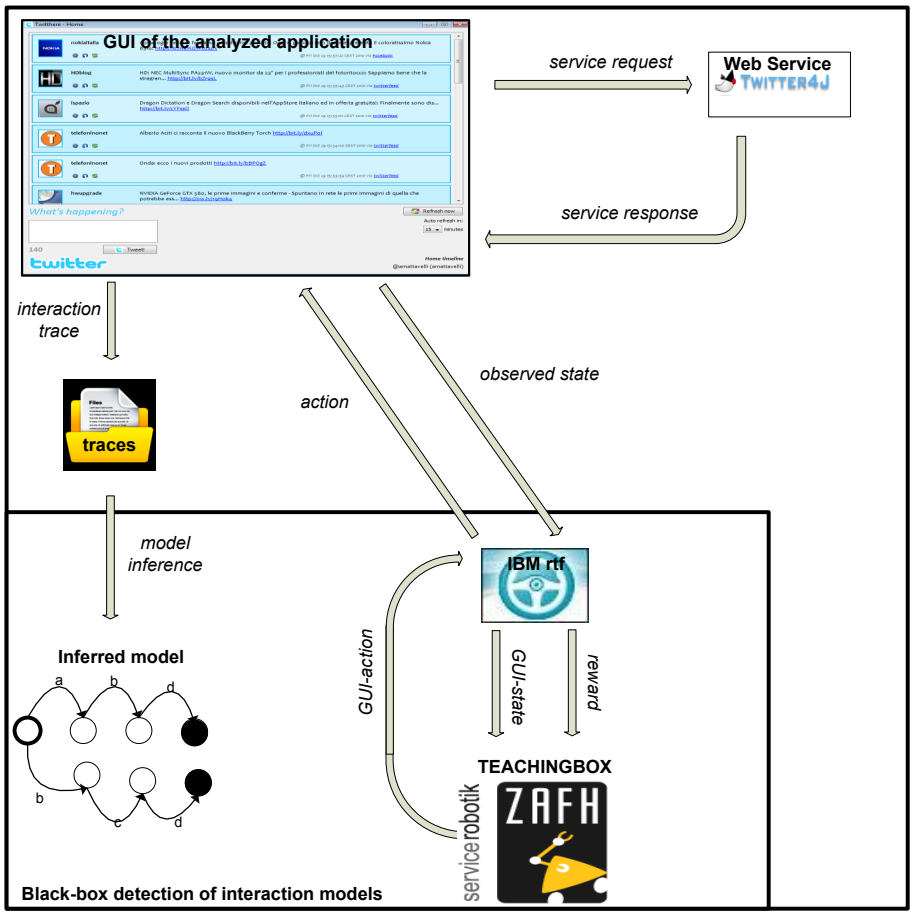


Figure 5.4: *The implemented tool.*

To permit our tool to concretely interact with GUIs of the analyzed application, we defined and implemented the support for set of widgets and corresponding actions. Table 5.2 shows the set of supported widgets. The

set of widgets is sufficient to analyze the most of desktop applications that use GUIs. However it can be easily extended any time a new widget is discovered.

Widget	**Actions**
TextField	setText
TabbedPain	click
TextArea	setText
CheckBox	clickSelect; clickSelect
MenuBar	click
TextPane	setText
RadioButton	clickSelect; clickSelect
Table	multiSelect; doubleClick
MenuItem	click
CheckBoxMenuItem	click
Spinner	click
ComboBox	clickFirst; clickAtHalf; clickLast; click
Tree	halfSelector
List	clickFirst; clickAtHalf; clickLast; click
Menu	click
Slider	click
FormattedTextField	setText
Button	click; drag
Label	click
ScrollBar	scroll

Table 5.2: *The set of widgets and the corresponding methods implemented in our tool.*

5.5 Results from the running example

The goal of the empirical validation is to explore a large portion of the execution space of the program and thus infer a model of the communication between Twitthere and the Twitter Web Service. The hard job is that not all the interactions with Web Service are executed with a single GUI-action, as shown in Table 5.1. For example, a single interaction with the GUI is sufficient producing simple invocations to Web Service, such as GUI-action *cancel an own tweet* or *refresh*, but many other interactions require the execution of a combination of GUI-actions. Therefore, the agent has to learn how to execute the right combinations of GUI-actions.

To record Web Service invocation, we instrumented the application with the java library twitter4j [14].

By running 500 episodes with 10 steps each, the techniques discovered all the GUI-actions and their combination. To run all the episodes we took

2 days by using a standard desktop computer (Intel Core Duo, 2GB RAM). We generated behavioral models from traces. To evaluate the quality of the model extracted, we measure model precision and recall. To obtain values of precision and recall, we use the QUARK framework [52] that can automatically generate traces from an inferred and a reference model, and compute precision and recall with respect to a reference FSA that represents the real the behavior of the program. For the purpose of this validation, we derived the reference FSA manually by inspecting the code of the application.

We obtained a precision of 1 and a recall of 61.55. Values show that the behavioral model well represents correct uses of the system (precision = 1), but does not include all the possible correct Web Service interactions (recall = 61.55). This limitation is caused by the length of episodes. In the reference model all states are final states, so the behaviors accepted by the reference model includes behaviors of length 1 or 2, while the execution of long episodes only prevents the generation of short interaction sequences (in fact the shortest sequence that has been recorded has length 3). However, the short interaction traces are rejected by the inferred model only because of a few missing final states. The interpretation of the results suggests that using episodes of different length, including very short episodes, can strongly improve the completeness of generated models.

Conclusions

Behavioral models play an important role in software development. In testing and analysis behavioral models have been used to complement specifications, to generate test cases, to define oracles, and to identify anomalous behaviors. In many cases behavioral models are not available and must be extracted automatically from the system under analysis. When models are automatically inferred, the effectiveness of testing and analysis techniques strongly depends from the quality of the model, and thus from the quality of the inference.

Since a model that is widely used to represent the behaviors of software programs and that can be easily inferred is Finite State Automata, this PhD thesis investigates the problem of inferring Finite State Automata that support programs analysis and testing. In particular, the thesis first presents an empirical assessment of state of the art inference techniques, then introduces a white-box technique that detects fairly complete models, and finally it describe a black-box technique for the inference of fairly sound models.

The *empirical comparative study* investigates the trade-offs between techniques that infer Finite State Automata and those that infer extended Finite State Automata and discusses complementarities, strengths and weaknesses of the existing techniques providing some useful elements to decide the inference technique to use depending on the kind of information that should be accurately represented into the Finite State Automata. In particular, we evaluate kTail, kBehavior, gkTail and kLFA with a set of case studies extracted from real software systems that include behaviors that can be uniquely captured with extended models.

The *white-box technique* can extract interaction models from program code by pairing a generation step with a refinement step to identify and

eliminate many infeasible elements from the static model. Moreover, the technique generates precise models by distinguishing likely feasible behaviors from behaviors whose feasibility is not known.

The *black-box technique* can generate interaction models from program execution by incrementally producing new executions with a machine learning approach that systematically explores the execution space. The inferred models are precise, because behaviors are dynamically identified, and incrementally more complete, depending from the degree of exploration.

Results obtained from the empirical comparative study show that techniques that infer extended models are too expensive and imprecise to be applicable in many real scenarios. Rather the inference of simple FSAs is a good compromise between the precision of the model and the applicability of the techniques.

Results obtained with SEIM when applied to an application that interacts with the eBay Web Services show that it is possible to generate Finite State Automata that are fairly complete and with good precision, because the technique effectively removed most infeasible sequences missing only few correct behaviors.

Results obtained from the experience of the black-box technique with an application that interacts with the Twitter Web Service show that the behavioral model generated with a black-box approach can represent the real behavior of a system with a fairly sound and complete model.

Even if both techniques can produce good models, the empirical results show that the white-box approach privileges completeness to soundness, while the black-box approach privileges soundness to completeness.

We are currently working to extend the empirical validation with additional cases to produce results that can be better generalized, to integrate the two complementary techniques and use the extracted models in verification, validation and conformance analysis of service-based applications.

Bibliography

[1] see the International Colloquium on Grammatical Inference (ICGI) serie of conferences.

[2] Columba. http://sourceforge.net/projects/columba, visited in 2010.

[3] ebay Web Services. http://developer.ebay.com, visited in 2010.

[4] IBM RFT. http://www-01.ibm.com/software/awdtools/tester/functional, visited in 2010.

[5] Jeti. http://jeti.sourceforge.net, visited in 2010.

[6] Jfreechart. http://www.jfree.org/jfreechart, visited in 2010.

[7] JPF. http://babelfish.arc.nasa.gov/trac/jpf, visited in 2010.

[8] Lucane. http://www.sharewareconnection.com/lucane.htm, visited in 2010.

[9] Open Hospital. http://sourceforge.net/projects/angal, visited in 2010.

[10] Rapid Miner. http://rapid-i.com, visited in 2010.

[11] Soot. http://www.sable.mcgill.ca/soot, visited in 2010.

[12] Teachingbox. http://amser.hs-weingarten.de/en/teachingbox.php, visited in 2010.

[13] Thingamablog. http://www.thingamablog.com, visited in 2010.

[14] Twitter4j. http://twitter4j.org, visited in 2010.

[15] Twitthere. http://twitthere.wordpress.com, visited in 2010.

[16] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[17] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *AST '10: Proceedings of the 5th Workshop on Automation of Software Test*, pages 59–66. ACM, 2010.

[18] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08: Proceedings of the 9th International Symposium on Software Testing and Analysis*, pages 3–14. ACM, 2008.

[19] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *ESEC/FSE '09: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 141–150. ACM, 2009.

[20] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, 1972.

[21] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 4th International Symposium on Software Testing and Analysis*, pages 169–180. ACM, 2006.

[22] O. Cicchello and S. C. Kremer. Inducing grammars from sparse data dets: A survey of algorithms and results. *Journal of Machine Learning Research*, 4:603–632, 2003.

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

[24] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5), 2009.

[25] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 861–864. ACM, 2006.

[26] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–37, 2008.

[27] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 281–290. ACM, 2008.

[28] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[29] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *WODA '06: Proceedings of the 4th International Workshop on Dynamic Systems Snalysis*, pages 17–24. ACM, 2006.

[30] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP'05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 528–550. Springer-Verlag, 2005.

[31] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE '09: Proceedings of the 24th International Conference on Automated Software Engineering*, pages 550–554. IEEE Computer Society, 2009.

[32] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 452–462. IEEE Computer Society, 2009.

[33] M. El-Ramly, E. Stroulia, and P. Sorenson. Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces. In *KDDM '02: Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining*. ACM, 2002.

[34] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, 2005.

[35] 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, 2001.

[36] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[37] D. Evans and M. Peck. Inculcating invariants in introductory courses. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 673–678. ACM, 2006.

[38] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Comput. Surv.*, 8(3), 1976.

[39] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 430–440. IEEE Computer Society, 2009.

[40] P. Godefroid. Compositional dynamic test teneration. In *POPL '07: Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, pages 47–54. ACM, 2007.

[41] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM, 2002.

[42] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM, 2002.

[43] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 60–71. IEEE Computer Society, 2003.

[44] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[45] J. G. Hosking. Visualisation of object oriented program execution. In *VL '96: Proceedings of the 1996 Symposium on Visual Languages*, page 190. IEEE Computer Society, 1996.

[46] R. Hull and J. Su. Tools for composite web services: a short overview. *SIGMOD Record*, 34(2), 2005.

[47] J.Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *ECOOP '03: Proceedings of the 17th European Conference on Object Oriented Programming*, 2003.

[48] N. Kuzmina and R. Gamboa. Extending dynamic constraint detection with polymorphic analysis. In *WODA '07: Proceedings of the 5th International Workshop on Dynamic Analysis*, page 1. IEEE Computer Society, 2007.

[49] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 6–6. USENIX Association, 1998.

[50] D. Lo, S. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD '07: Proceedings of the 13th International Conference on Knowledge Discovery and Data Mining*, pages 460–469. ACM, 2007.

[51] D. Lo, S. Khoo, and C. Liu. Mining temporal rules for software maintenance. *J. Softw. Maint. Evol.*, 20(4):227–247, 2008.

[52] D. Lo and S-C. Khoo. Quark: Empirical assessment of automaton-based specification miners. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 51–60. IEEE Computer Society, 2006.

[53] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the symposium on The foundations of software engineering*, pages 345–354. ACM, 2009.

[54] D. Lo, L. Mariani, and M. Santoro. Learning extended fsa from software: an empirical assessment. In *ICSE '11: Proceedings of the 33st International Conference on Software Engineering*, 2011. submitted.

[55] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[56] C.D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge Press, 2008.

[57] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *ISSRE '08: Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 117–126. IEEE Computer Society, 2008.

[58] L. Mariani, F. Pastore, M. Pezzè, and M. Santoro. Mining finite-state automata with annotations. In *Mining Software Specifications: Methodologies and Applications*, pages – pending –. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2011.

[59] L. Mariani and M. Pezzè. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[60] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Seim: static extraction of interaction models. In *PESOS '10: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, pages 22–28. ACM, 2010.

[61] L. Mariani, M. Pezzè, and D. Willmor. Generation of selft-test components. In *FORTE '04: Proceedings of the 1st International Workshop on Integration of Testing Methodologies*, pages 337–350. Springer, 2004.

[62] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th International Symposium on Foundations of Foftware Engineering*, pages 287–296. ACM, 2003.

[63] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (vet): An interactive plugin-based visualisation tool. In *AUIC '06: Proceedings of the 7th Australasian User interface conference*, pages 153–160. Australian Computer Society, Inc., 2006.

[64] E. F. Moore. *Sequential Machines: Selected Papers*. Addison Wesley, 1964.

[65] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. *Electronic Notes in Theoretical Computer Science*, 55(2):255–276, 2001.

[66] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In N. Pérez de la Blanca, A. Sanfeliu, and E.Vidal, editors, *Pattern Recognition and Image Analysis*, pages 49–61. World Scientific, 1992.

[67] C. Pacheco, S. K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 75–84. IEEE Computer Society, 2007.

[68] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008.

[69] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *CoRR*, cs.SE/0310040, 2003.

[70] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *Proceedings of the Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

[71] O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 302–312. ACM, 2002.

[72] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001.

[73] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *ICPC '06: Proceedings of the 14th International Conference on Program Comprehension*, pages 84–88. IEEE Computer Society, 2006.

[74] M. Santoro. Detecting precise behavioral models. In *ESEC/FSE Doctoral Symposium '09: Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium*, pages 17–20. ACM, 2009.

[75] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV '06: Proceedings of the 18th International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[76] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 174–184. ACM, 2007.

[77] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[78] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *ASE '08: Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society, 2008.

[79] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *SP '99: Proceedings of the Symposium on Security and Privacy*, pages 133–145. IEEE Computer Society, 1999.

[80] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the symposium on The foundations of software engineering*, pages 35–44. ACM, 2007.

[81] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[82] Y. Wu, D. Pan, and M. Chen. Techniques for testing component-based software. pages 222–232, 2001.

[83] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 380–403, 2006.

[84] T. Xie, E. Martin, and H. Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 835–838. ACM, 2006.

[85] T. Xie and D. Notkin. Exploiting synergy between testing and inferred partial specifications. In *WODA '03: Proceedings of the 1st International Workshop on Dynamic Systems Snalysis*, pages 17–20, 2003.

[86] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engg.*, 13(3):345–371, 2006.

[87] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291. ACM, 2006.

[88] H. Yuan and T. Xie. Substra: A framework for automatic generation of integration tests. In *AST '06: Proceedings of the 1st International Workshop on Automation of Software Test*, pages 64–70. ACM, 2006.

[89] X. Yuan and A. T. Memon. Using gui run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 396–405. IEEE Computer Society, 2007.

[90] A. X. Zheng, J. Lloyd, and E. Brewer. Failure diagnosis using decision trees. In *ICAC '04: Proceedings of the 1st International Conference on Autonomic Computing*, pages 36–43. IEEE Computer Society, 2004.