Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor’s Thesis Nr. 134

# Enriched Tool Support for Probabilistic Specification Mining (ProSpecMi)

Sven Maier

| | |
|---|---|
| **Course of Study:** | Softwaretechnik |
| **Examiner:** | Prof. Dr. Lars Grunske |
| **Supervisor:** | Antonio Filieri |
| **Commenced:** | 2014/05/05 |
| **Completed:** | 2014/11/04 |
| **CR-Classification:** | D.2.5 |

# Abstract

Specification Mining describes the process of creating a specification from a (probably unknown) program using sample executions. Most of the current specification miners are deterministic. This thesis aims to create a probabilistic specification miner. Therefor, a specification miner with three different probabililistic approaches has been implemented and added to the LearnLib-Framework. The implementation has been validated by letting the specification miner rebuild a predefined specification to compare the template and the result, by running a hypothesis-test to compare the used approaches to calculate the probabilities against another and by letting it mine the usage of a real API from $n$ tests and validate them with $m$ more tests.

# Zusammenfassung

Specification Mining beschreibt den Vorgang, eine probabilistische Spezifikation eines (möglicherweise unbekannten) Programms unter Verwendung von Beispielausführungen zu erstellen. Die meisten der aktuellen Specification Miners sind deterministisch. Ziel dieser Ausarbeitung ist ein probabilistischer Specification Miner. Dafür wurde ein Specification Miner mit drei verschiedenen probabilistischen Ansätzen implementiert und dem LearnLib-Framework hinzugefügt. Die Implementierung wurde validiert, indem der Specification Miner eine vorgegebene Spezification nachgebaut hat, um die Vorlage mit dem Ergebnis zu vergleichen, durch einen Hypothesen-Test, der die benutzten Ansätze zur Berechnung der Wahrscheinlichkeiten untereinander verglicht und indem der Specification Miner eine Spezifikation für die Benutzung einer realen API aus $n$ tests erstellt, die mit $m$ weiteren Tests validiert wird.

# Contents

Chapter 1

# Introduction

Section 1.1 deals with the motivation of this bachelors thesis, section 1.2 presends the goals this thesis has and section 1.3 shows the structure of this document.

## 1.1 Motivation

Current software projects are becomming bigger and bigger over the time. What seems to be a good thing on the one hand side, as the software can do much more complex things that would have been impossible only a few years ago, is a bad thing for the internal complexity of a software project, as it also is dramatically increasing with the size of a project. The overall expense for both maintanance and extension of a given project largely depends on the quality and extend of the formal specification [Goues and Weimer, 2011]. Especially for bigger project, a problem occurs: given that writing a complete specification is both a time-consuming and an expensive job [Weimer and Necula, 2005] and a lot of projects have last-minute changes that don't make it to the final specification [Raffelt et al., 2009], specifications are only too often incomplete. An incomplete specification often is one of the reasons for the occurence of bugs [Kremenek et al., 2006].
Many companies are forced to use unspecified black-box legacy systems where none of the current programming team really knows how the program works, where re-implementing the program would be too expensive and the programmers who wrote the code don't work at the company anymore. For such legacy software systems, both maintanance and extension is almost impossible. Thus, at the occurence of unwanted behavior by the software caused by bugs can hardly ever can be fixed on such systems.
The research field of specification mining aims to automate the process of creating a formal specification for a whole program, a certain api or several objects [Chen and Roşu, 2008]. A specification miner is a program that estimates a fitting specification from an existing program by using special tests or sample executions to gain execution traces. The result is, in most cases, a deterministic automaton [Ammons et al., 2002; Goues and Weimer, 2009; Livshits et al., 2009; Chen and Roşu, 2008; Kremenek et al., 2006; Mao et al., 2011; Dallmeier et al., 2006] or a set of certain rules [Engler et al., 2001], in other words invariants for the program.
The deterministic automaton has transitions that lead from one state to an other state. Those transitions can be infered from traces that were taken by sample executions or by

special tests.
Most of the current, deterministic specification miners are based on the assumption that not every execution leads to a successful state, but that most executions do. Thus, once a fitting (sometimes even probabilistic) specification has been found, a pruning step removes those transitions that were rarely used during the execution, as they are probably faulty, and then determinizes it by removing any counter or probabilistic value from the transitions.
A calculated specification can be used as a tool, let's say, a plugin for eclipse, that automatically checks if a programmer used an API in the right way. To check for the validity of the usage, the program could use a mined specification. Due to the removal of rarely used edges, a deterministic specification would lead to false negatives if the specification miner pruned a path to an otherwise accepting state. In that case, the tool would show an error eventhough the api was used in the right manner.
A probabilistic specifiation is an automaton-based specification that uses a probabilistic automaton instead of a deterministic one. During the creation of a probabilistic specification, the miner would not remove transitions based on the fact that they hardly ever occured during the execution.
Every transition in a probabilistic automaton is labeled with a floating point value, indicating the likeliness of a certain transition to be taken once the automaton reached the source state. In a probabilistic automaton, a rarely used (but possibly still correct) transition wouldn't be pruned, but merely be assigned with low transition probability. Hence, an eclipse plugin wouldn't throw an error, but maybe a warning, indicating that it *could* be wrong, as the probability of this edge is very low. Ignoring those warnings, however, would lead to false positives, as the transition actually *can* lead to an error. Using those warnings with care would make debugging a bug that came from false usage of an api a lot easier.
This thesis targets at the creation of a program that can create such a probabilistic specification from sample executions, namely a probabilistic specification miner. The probabilistic specification miner is capable of creating a probabilistic specification for an Application Programming Interface (API) or a given set of classes, using traces gathered from sample executions. Those executions are performed by executing the Unit-Tests. Those tests are taken from programs that actually use the desired API already. Using them, a probabilistic automata is created that specifies the order in which the functions are to be called. The actual program is written as an extension for the existing machine-learning framework called LearnLib [Raffelt et al., 2009].
LearnLib is a framework that offers many algorithms needed for automata learning purposes. It was originally developped by the german *Dortmund University of Technology* as a closed-source project [Howar et al., 2014c], before it was reimplemented as open-source java-project under the terms of LGPLv3 [Howar et al., 2014b].

## 1.2 Goals

The major goal of this thesis is it to write a probabilistic specification miner and include it to the *LearnLib*-Framework. This goal can be split into several sub-goals.

### 1.2.1 Execute Unit-Tests from within Java

Any specification miner needs traces. To collect those traces, a program has to be executed in the way it was meant to be. Due to the fact that programs usually don't come with an operational profile, the *sample executions* for the specification miner from this thesis are comming from Unit-Tests. Thus, the first goal is to manage the execution of Unit-Tests from the specification miner.

### 1.2.2 Collect Traces

During the execution of the Unit-Tests from the specification miner, the mined API has to leave execution traces for every function that has been called. The second goal is it to collect the traces from those classes that were chosen to create the probabilistic specification for. Using the bytecode-manipulation framework JAVASSIST [Chiba, 2000], this goal includes embedding tracing code to every function the classes that have to be observed offer.

### 1.2.3 Build a DFA from the gathered traces

Once the traces have been collected in the order they occured during the execution of the Unit-tests, the next goal is it to use the traces in order to create a Deterministic Finite Automaton (DFA). The DFA is calculated using the algorithm by Angulin [Angulin, 1987] that is already implemented in the LearnLib-Framework. The Automata has nameless states. At this time, the transitions are labeled only with an input parameter, indicating what function is to be executed when the automaton takes this transition.

### 1.2.4 Calculate the probability for every transition

The automaton returned by Angulin is purely deterministic. The transitions already exist, but there is no probabilistic value indicating how likely any transition is to be taken. Hence this goal aims at the calculation of the transition probabilities for all transitions. To achieve this goal, the traces gathered from the sample executions are taken one more time to traverse the automaton and update a frequency counter for the edges. Using those measured value, the probability then is calculated using one of three different methods. The methods themselve are described in section 4.1.3.

### 1.2.5 Integrate the algorithm in LearnLib

Once the probabilistic specification miner has been created, it has to be integrated to the open-source learning framework LearnLib [Raffelt et al., 2009].

## 1.3 Document Structure

The remainder of this document is structured as follows:
Chapter 2 offers an overview of foundations and technologies relevant to this thesis. Chapter 3 introduces the research questions, the implementation requirements and the qualitative research question relevant to this thesis. Chapter 4 presents the Sherlock-approach used in this thesis. The evaluation of this approach is written in chapter 5. Chapter 6 includes a brief discussion about the qualitative research questions from chapter 3. Chapter 7 gives an overview on what was done in this thesis and how this work can be extended in future work.

Chapter 2

# Foundations and Technologies

This chapter points out the foundations and technologies that are required to understand this thesis. Section 2.1 introduces the concept of automata learning, Section 2.2 presents the foundations of Specification Mining alongside with several different approaches to mine a specification from a program. Section 2.3 introduces the LearnLib-Framework that offers many automata–learning algorithms. The AutomataLib-Framework, that is used by LearnLib to hold their automata-objects, is introduced in section 2.4. Section 2.5 contains information about a Java bytecode manipulating framework called Javassist that was used by the actual specification miner.

## 2.1 Automata Learning

Automata learning, also called *regular extrapolation* or *regular inference*, describes the process of building an automata that represends a piece of software or a protocol automatically [Raffelt et al., 2009]. This usually happens by creating a deterministic finite automaton (DFA), where the language $\Sigma^*$ that the automaton represends matches the observations of an unknown automaton.
Automata learning can be used to automatically create a behavorial model from the input, so programmers can use model-based verification techniques to verify the system. This model typically is represented as a deterministic finite automaton (DFA) that matches the behavior of the observed system and tells the programmer more about the internal structure.

## 2.2 Specification Mining

Specification Mining is an automated process, where a machine, typically a piece of software, creates a formal specification from a given code. Cook and Wolf [1996] categorize it as reverse engineering method, where the goal is to get a high-level specification like a graph or an automaton from a low-level specification like the source code. This makes it easier for engineers to understand the system [Cook and Wolf, 1996]. According to Weimer and Necula [2005], a specification miner takes a program as input in order to create a specification, typically a finite state machine, that specifies a set of interesting events.

### 2.2.1 Cook and Wolf (1996)

Cook and Wolf [1996] published a paper back in 1996 with one of the earliest approaches for Specification Mining. Under the name of *Process Discovery*, the authors presented three different methods to get the specification from the program. Two of them included the usage of already existing algorithms, namely RNet and KTail [Bierman and Feldman, 1972], that were extended by the authors. The third method, Markov, was designed by Cook and Wolf [1996] to create a specification. The authors were trying to merely create a description of the patterns that can be derived from execution traces and not a complete specification that covers every aspect of the underlying program.
To create a specification, the tracer first creates a set of socalled *event streams*. An event stream is a set of traces from the program. The Markov-Algorithm then traverses the event streams. It uses frequency-counters in order to create a socalled *event-sequence probability table*, a lookup-table that holds the ratio that, after an input $i$ has been read from the event stream, the input $j$ will follow. From that table, they create an *event graph* by assigning a vertex to any event type and connect two vertices $v$ and $w$ with an edge if the event $w$ happens after $v$ and the probability from the event-sequence probability table is over a user-defined threshold. A splitting step then ensures that, given the rule of transitivity, there are no sequences in the graph that would lead to an illegal sequence. Finally, the event graph gets converted to a graph where the node from the new graph is labeled with a unique transition from the event graph.
Figure 2.1 shows an example specification derived from the *MARKOV*-algorithm [Cook

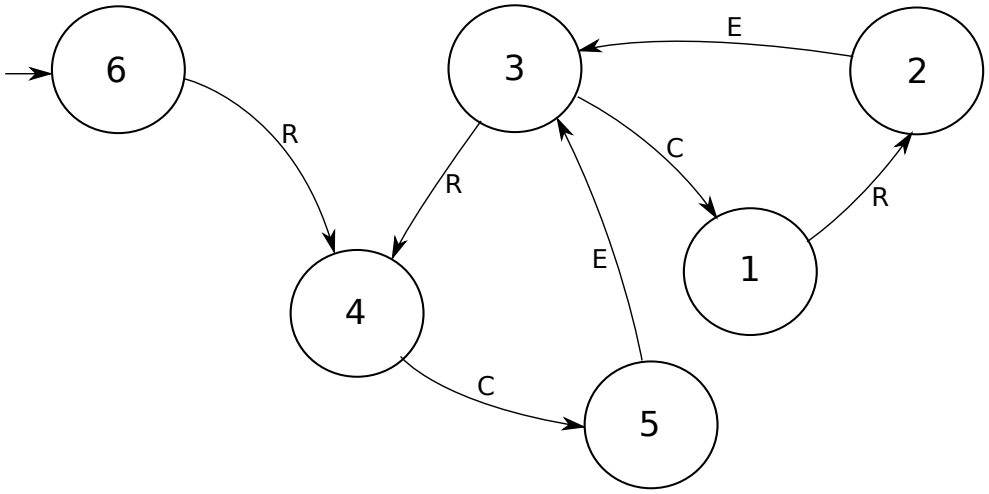


**Figure 2.1.** Graph created by the *MARKOV* algorithm from the event-stream *RCERCECRECRERCECRECRERCECRERCERCECRECR* as seen in [Cook and Wolf, 1996].

and Wolf, 1996]. The traces that created the graph are a concatenation of the two substrings

*RCE* and *CRE*. Thus, the graph starts with an *R* from state 6 and uses it to create *RCE* one time and ends in state 3. From there, it can either go to state 1 and create *CRE* or to state 4 to create *RCE*.
The described method by Cook and Wolf [1996] works well and as one of the earliest approaches on specification mining, it is quite sophisticated. It does prune rarely used edges by a user-defined threshold, what makes the accuracy of the specification dependent on the users choice of the threshold. Given that this thesis aims to create a probabilistic specification instead of a deterministic one, the pruning step at the end would be removed and thus, a probabilistic specification would be created.

### 2.2.2 Ammons, Bodík and Laurus (2002)

Ammons et al. [2002] created a specification miner that was especially designed to model interactions between the program and abstract datatypes (ADT) or application programming interfaces (API) [Ammons et al., 2002]. To achieve this goal, their algorithm first collects traces from the program executions – either by rewriting and recompiling the APIs source code or by applying changes directly to the binary file. Those traces then are marked with flow dependencies and types to figure out those traces that belong to a common scenario. A scenario extracter then uses those marked traces and user-specified scenario-seeds to extract specific interaction scenarios. Using those scenarios, an automata learning algorithm creates a probabilistic, non-deterministic finite automata (NFA) that accepts only the language from the scenarios. In their paper, Ammons et al. [2002] used the PFSA-Learner [Raman et al., 1998] that uses the k-tails algorithm. The algorithm builds a retrieval tree using the input strings and generates, from every state, a set of strings with length *k*. If two states are likely to generate the same set of strings, they are merged. The algorithm terminates, when there are no more states to be merged. After the PFSA-learner is done, the algorithm by Ammons et al. [2002] removes those edges whose probability is below a certain threshold. From the remaining edges, the probability gets removed, so the automaton is non-deterministic instead of probabilistic.
Figure 2.2 shows an example set of the traces that was collected during an execution of a program that uses the Socket-API in C. Using their specification miner, they derived the NFA. Using the automaton, it is clear that, after accept, a user can read and write on the sockets as long as he wants, but after the usage the sockets both have to be closed. It can also be seen, that the scenario extractor used in their approach realizes that, eventhough in the traces it seems that *read(fd=y)* always comes right before *write(fd=y)*, it didn't put a path of *read(fd=y)→write(fd=y)* followed by an edge back before *read*. They put a loop in the specification that also allows runs where the user only writes to a socket before closing it or write before he reads.
The approach in this thesis specifies at the usage of an API by using a probabilistic specification. Thus, the approach described by Ammons et al. [2002] would not have been a bad choice. To get a *probabilistic* specification instead of a *non-deterministic* one, all that would have to do is to remove the corer. What would remain is a tracer to collect traces from

```
1  socket(domain = 2, type = 1, proto = 0, return = 7)
2  bind(so = 7, addr = 0x400120, addr_len = 6, return = 0)
3  listen(so = 7, backlog = 5, return = 0)
4  accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 8)
5  read(fd = 8, buf = 0x400320, len = 255, return = 12)
6  write(fd = 8, buf = 0x400320, len = 12, return = 12)
7  read(fd = 8, buf = 0x400320, len = 255, return = 7)
8  write(fd = 8, buf = 0x400320, len = 7, return = 7)
9  close(fd = 8, return = 0)
10 accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 10)
11 read(fd = 10, buf = 0x400320, len = 255, return = 13)
12 write(fd = 10, buf = 0x400320, len = 13, return = 13)
13 close(fd = 10, return = 0)
14 close(fd = 7, return = 0)
```

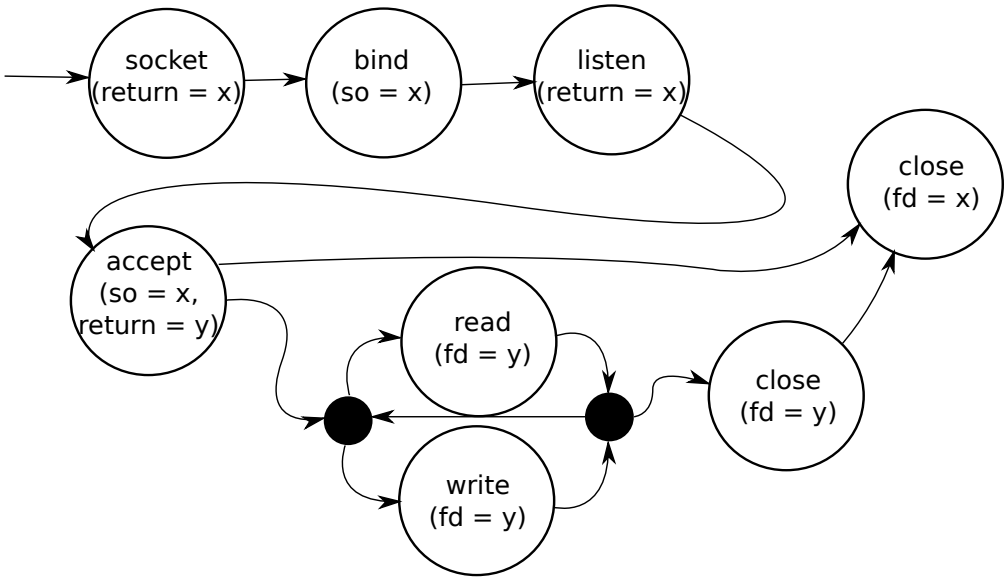


**Figure 2.2.** Example for traces from the Socket-API in C alongside with the resulting NFA one gets after applying the learning algorithm described by Ammons et al. [2002]. Example from [Ammons et al., 2002]

executed methods, a flow dependence annotator, a scenario extractor and a PFSA-Learner. The major reason for not taking this approach is that in this approach, the probabilistic automaton that represends the specification would have to be *deterministic* if the probabilistic factor would be taken away. This means, that the automaton would have to have exactly *one* initial state and *one* transition for any state $s$ and input $f$. Hence there were two

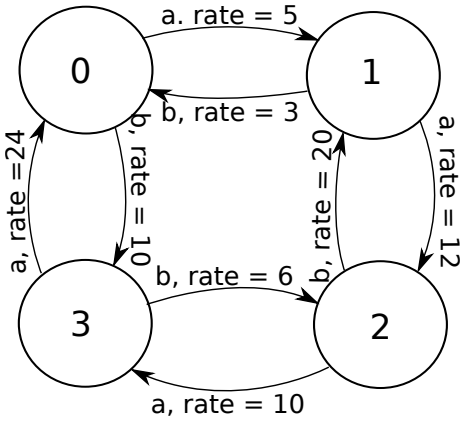


**Figure 2.3.** Markov-Chain as seen in [Sen et al., 2004] to demonstrate the algorithm. The *rate* has the value of the occurence count.

possible approaches: either take this approach and use the algorithm by Rabin and Scott to determinize the resulting automaton, or use a different approach. Determinizing the approach would result in states that contain numerous functions, thus for the simplicity of traversing the automaton, this thesis takes another approach.

### 2.2.3 Sen, Viswanathan and Agha (2004)

The approach by Sen et al. [2004] aims at the creation of Continuous-time Markov Chains (CTMCs) [Sen et al., 2004]. From a set of traces gained from sample executions, their algorithm creates a prefix tree. Under the assumption of a lexicographic order, the algorithm then searches this prefix tree for states that are equivalent in order to merge them. For every new state observed, the algorithm checks its compatibility to all states that have been previously observed. After two states $s_i$ and $s_j$ have been merged, the corresponding prefix tree might be non-deterministic, so a function determinizes it. This is possible because the two states $s_i$ and $s_j$ are equivalent. Thus, two states $s_k$ and $s_l$, with $s_k \neq s_l$, both successors of $s_i$ or $s_j$, are by definition equivalent as well. Hence, they get merged to one new state. Once this is done, the probability and rate of the whole tree gets recalculated.

An example for a learned continuous-time markov chain can be seen on figure 2.3. Its edges are labeled with *a* and *b* for the input function and a *rate*-value indicating how often the path was taken. In the states 0 and 2, the probability of *b* is twice as high than the one of *a*. But still, they differ in their actual rate.

The described algorithm works and the result can be used for model-checking. The actual problem is just, that the web page for the reference implementation, that should be availlable on *http://osl.cs.uiuc.edu/ksen/vesta/*, has been unreachable during the time this

thesis was written.

### 2.2.4 Weimer and Necula (2005)

Based on the idea that most bugs come from violations agains API specification, Weimer and Necula [2005] tried to automatically create several temporal specifications in order to attenuate this kind of bug in software [Weimer and Necula, 2005]. Quite simmilar to the work of Engler et al. [2001], they tried to create a state machine that specifies event-pairs $< a, b >$. The functions $a$ and $b$ are an event-pair, if, once $a$ has been called, $b$ has to be called eventually for the program to work correctly.
Violations against those rules often lead to errors in Java. Especially in a *try-catch*-block without *finally* statement, where after $a$ an error prevents $b$ from being executed, violations against those rules can happen.
The presented algorithm uses a filering technique with certain rules that have to be followed by the code. The authors defined four rules for any event-pair $< a, b >$:

- ex: $b$ is at least once in a cleanup code, where $E_{ab}$, the amount of error traces where after $a$, $b$ is called, is positive.
- oe: $E_a$ has to be positive. This means, that $a$ has been called at least once in an error trace (a trace that ended with an error), where $b$ was not.
- sp: $a$ and $b$ must belong to the same package.
- df: All parameters and return values from $b$ have also to occur in $a$.

The result is a set of *candidate specifications*. Every candidate specification has to be ranked based on how likely they satisfy the criteria. For any two functions $a$ and $b$, the rank $r_{ab}$ is calculated by $\frac{N_{ab}}{N_{ab}+N_a}$ with $N_{ab}$ being the amount of occurences where $a$ is followed by $b$ and $N_a$ the amount of calls for function $a$ without $b$ following.
Figure 2.4 shows an observation table with 8 different candidate specifications. The first

| Event $a$ | Event $b$ | Real | $N_a$ | $N_{ab}$ | $E_a$ | $E_{ab}$ | Filters | rank |
|---|---|---|---|---|---|---|---|---|
| SF.openSession | S.close | ✓ | 3 | 100 | 1348 | 1040 | ex oe sp df | 0.971 |
| S.beginTransaction | S.close | ? | 2 | 56 | 1037 | 501 | ex oe sp df | 0.966 |
| S.beginTransaction | T.commit | ✓ | 2 | 56 | 565 | 973 | ex oe sp df | 0.966 |
| S.flush | S.close | × | 9 | 39 | 200 | 473 | ex oe sp df | 0.812 |
| T.commit | S.close | ? | 1 | 57 | 474 | 504 | ex oe sp - - | 0.983 |
| S.beginTransaction | s.save | × | 4 | 54 | 37 | 1501 | - - oe sp df | 0.931 |
| SF.openSession | T.commit | ? | 47 | 56 | 1415 | 973 | ex oe sp - - | 0.544 |
| SF.openSession | println | × | 82 | 21 | 2121 | 267 | ex oe - - df | 0.204 |

**Figure 2.4.** Observations from hibernate2 and the session class that is between the program and hibernate2 made by Weimer and Necula [2005].

to columns display the event pairs $a$ and $b$ and on the third column, Weimer and Necula

[2005] denoted if this candidate specification actually holds in the real program or not. A questionmark denotes that it *could* be a valid event pair. The next column, $N_a$, denotes the amount of failure-free execcutions of $a$ without $b$ following, followed by $N_{ab}$, the amount of failure-free executions that have $a$ being folowed by $b$ eventually. The same rule goes for paths that lead to an error in $E_a$ with $a$ being executed without $b$ and $E_{ab}$ for $a$ being followed by $b$. The next column displays the applied filters the event-pair matches. The filters were described earlier. The last column displays the rank. The maximum number a rank can take is 1. The higher the rank, the more likely this candidate specification is valid. The described approach by Weimer and Necula [2005] can actually help to prevent bugs that come from false usage of an API, but only those that apply to the scheme of two functions, where the second function has to be called after the first one in order to prevent errors. Eventhough this was one of the targets for this thesis, the candidate specifications don't tell a programmer exactly how he has to use the given API. It only gives us a set of rules, while lacking the *tutorial* factor that comes with an automaton. For example, the approach by Ammons et al. [2002], displayed in figure 2.2, displays exactly *how* the Socket-API has to be used in a way, that probably even programmers who haven't used it before would understand, what the candidate specifications infered here would not.

### 2.2.5 Kremenek, Twohey, Back, Ng and Engler (2006)

The work of Kremenek et al. [2006] was it to try to use factor graphs in order to infer a specification especially for the right usage of pointers [Kremenek et al., 2006]. Their algorithm first assigns annotations to specific functions that deal with pointers. The two annotations are *ro* (returns ownership) for allocators and *co* (claims ownership) for deallocators. From the source code, a set of possible annotations for the functions is created. For all those annotations, there is also a logical counterpart, namely $\neg ro$ and $\neg co$. A function that handles a pointer has to be either $ro/\neg ro$ depending on if it allocates the pointer space or not, or $co\neg co$ depending on if it releases the pointer. Thus, for $n$ functions dealing with the same pointer, there are $2^n$ different specifications. For every assignment, a user can define factors, namely *prior*-beliefs, in terms of non-negative values indicating how likely they think these specifications are. Using those prior-beliefs, the probabilities for all specifications are calculated, indicating how likely it is that it is a valid one. A static analysis tool then extracts *behavorial signatures* from the source code. The algorithm uses them to perform a *behavorial test*. Based on the idea that correct signatures *should* lead to a lower amount of errors, the beliefs that a behavorial signature *may* apply is being checked if they match with any of the annotations.
As for the visualisation, the authors introduced *Annotation Factor Graphs*, an extended version of the factor graphs where annotations are visualized for the functions. They also visualize how one annotation can influence others from the same data flow.

An example for an Annotation Factor Graph derived from a C-code can be seen on

```
FILE * fp1 = fopen( "myfile.txt", "r" );
FILE * fp2 = fdopen( fd, "w" );
fread( buffer, n, 1, fp1 );
fwrite( buffer, n, 1, fp2 );
fclose( fp1 );
fclose( fp2 );
```

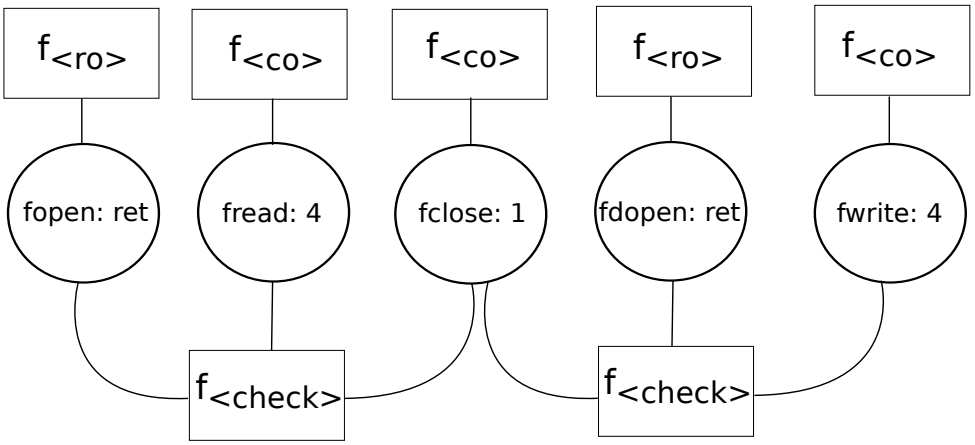


**Figure 2.5.** C code for opening a file alongside with an Annotation Factor Graph for the role of the pointers. As seen in [Kremenek et al., 2006].

figure 2.5. The C-code opens two files, one of them with reading rights (fp1) and one of them with writing rights (fp2). It then reads and writes on the right pointers and closes the files.
The bipartite Annotation Factor Graph has circular nodes for variable nodes that are internally mapped to annotation variables [Kremenek et al., 2006]. Rectangular nodes represend factor nodes, that can be mapped to the factors that were assigned earlier. An edge from a factor node to a variable node indicates that the variable is used as in input for the factor. Complete paths within the graph from a variable node $v$ to a variable node $w$ indicate that $v$ can influence $w$ and vice verca. For example, both fopen and fdopen have a path to at fclose, indicating that there is a correlation.
For the approach described in this thesis, the Annotation Factor Graph isn't of much use. Java doesn't use pointers like C does, and a basic check for the usage of uninitialized objects is already included to the Eclipse–Environment. A data-flow oriented test would probably give us the same results. Also, very much like the approach by Weimer and Necula [2005], this specification doesn't give us an automaton that can be used as a tutorial, but only points out a set of rules, pointing out what pointer has what role.

### 2.2.6 Dallmeier, Lindig, Wasylkowski and Zeller (2006)

The approach by Dallmeier et al. [2006] aimed at the creation of a state machine from java code. The machine should display the behavior of the observed object. To achieve this goal, a state has to be previously defined by the user [Dallmeier et al., 2006]. The methods for the observed class are defined as either inspectors or mutators. Inspectors return the gathered information from the current state of the class, while mutators may change the state. The decision if a function is a mutator or a inspector is made by static analysis procedures. From any program run, abstract versions from all the inspectors of the corresponding class can serve as part of the state. Once the state is defined, the algorithm forces the program to update it before and after any mutator–method is executed. That way, it is made clear in what way the mutator changes the state. In their prototype, Dallmeier et al. [2006] used the Javassist-Framework described in section 2.5 to force a method to update the state before and after the actual method is called. After the method call, the algorithm adds a transition to the automata that goes from the previous state to the current one. Eventhough the authors did not clearly mention it in their article, on some objects the prototype also uses an occurence counter to indicate how often a state has been changed. From the gathered traces, their program creates a non-deterministic finite automata (NFA) for every class observed.

An example DFA for the Java *Vector* class can be seen on figure 2.6. For the state, the

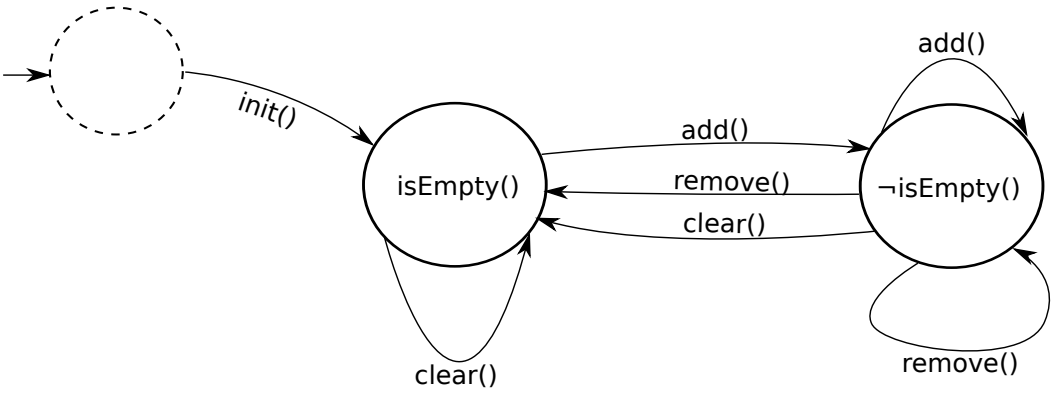


**Figure 2.6.** Example specification of the object behavior for the Java Vector class made by [Dallmeier et al., 2006]

authors chose the inspector *isEmpty()*, a function indicating that there are no elements stored within the vector object. After the initialisation of the object, that is displayed as the edge going away from the empty circle, the vector is, obviously, empty. After adding an object, it changes states to $\neg$*isEmpty()*. From there, the function *remove()* will either change the state to *isEmpty()*, or let the function remain at the current state, depending on how many elements are currently stored in the vector object. No matter what state the NFA is

in, the method *clear()* always returns to *isEmpty()*.
The derived NFA can again be quite useful for anyone trying to infer a specific set of rules for the observed class or who wants to get a closer look to the general behavior of a certain object. For the approach described in this thesis, however, the states are not quite useful to determine the correct usage of the class. The automaton in figure 2.6 tells us, that *clear()* makes the vector empty and that a programmer can't call *remove()* after the initialisation of the object. Also, the approach delivers us a non-deterministic automaton, what makes traversing harder. There is no accepting state, thus a programmer can't tell if a given set of chronologically ordered traces would be likely or valid using our automaton, even after using the occurence count to make the NFA a probabilistic automaton.

### 2.2.7 Chen and Roşu (2008)

Chen and Roşu [2008] presented an approach for mining a state-based deterministic finite automaton, where the automaton has parameters [Chen and Roşu, 2008]. As an invariant, the program that their algorithm uses to mine the specification has to be correct. This means, that all the traces must lead to accepting states. Their goal was it to infer safety properties from the probabilistic finite state automata. The authors describe the parameters as "free parameters, that need to be instantiated at runtime". For their approach, they are used as generalized objects, interfaces or abstract classes, that could have several implementations.
To create the specification, the algorithm takes the traces collected from program execution and slices them. Slicing means, that the program analyses the trace and binds the predefined parameters to them if the objects they represend were used in this trace. Then, to attenuate noise, the algorithm removes redundant traces. Traces are redundant, if they contain parameters they don't need.
A miner than takes those traces and first runs a PFSA-Learner on them. The PFSA-learner the authors used uses the sk-string algorithm [Raman and Patrick, 1997]. This algorithm creates a prefix tree that accepts the input strings and counts the occurence of every arc. It then merges equivalent states to create a NFA. The resulting automaton from the PFSA-Learner may accepts strings that shouldn't be allowed, thus a refiner expands the automaton in a way that for every incomming edge of a state, a new state is created, so every state has only one incomming edge. Then, using the traces gathered earlier, the algorithm traverses the extended automaton and marks all edges that have been used. Once this is done, unused edges are removed. Then, the extended automata gets compressed by the automaton refining algorithm called $\mathbb{R}$. That algorithm merges states that are identical.
Figure 2.7 shows an example automaton for the two Java-Classes *Collection* and *Iterator*. A collection is represented by the parameter *c*, an iterator by *i*. The states are labeled with numbers. The bigger number indicates what state from the automaton this was before it was expanded by the algorithm – initially, there were only four states. The index-number is only added to keep the new states appart.

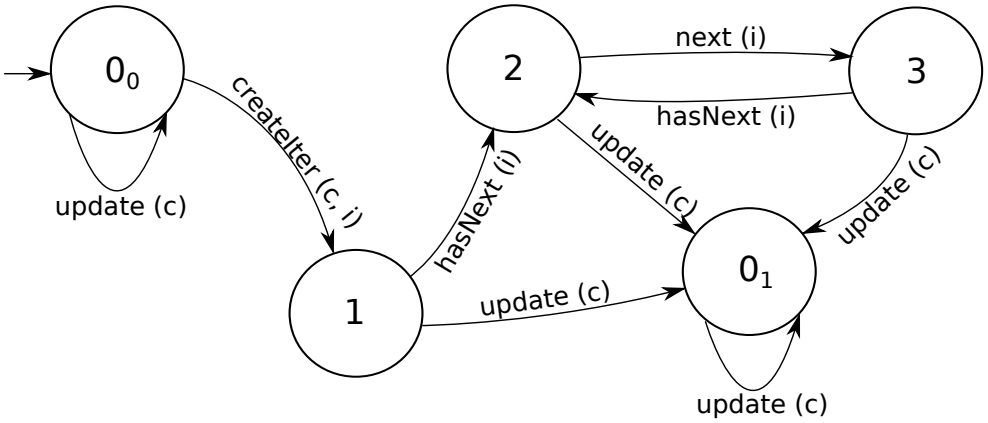


**Figure 2.7.** Example for an automaton created for a Collection *c* and an Iterator *i*. The parameter on a transition means that the representing object is affected with this change. As seen in [Chen and Roşu, 2008].

The parameters that were bound to the functions are added in braces. For example, *update(c)* only has a *c* as parameter, as it doesn't make any changes to the iterator but only to the collection. *createIter(c,i)* uses both the collection to create the iterator from and the iterator to store the elements from the collection inside.
What can be seen from the automaton is, that once the iterator has been created from the collection by calling the method *createIter*, it can only perform the reading functions *hasNext* and *next* on the iterator. The moment an update happens to the collection, the automaton moves to a trash state with no outgoing edges. This makes sense, as the iterator is a read-only representation of the objects from the collection. After any update of the collection (by adding or removing anything), the iterator would be outdated.
The resulting specification is, aside from the fact, that it doesn't contain any probabilistic value, very simmilar to the one that is aimed to be created at this thesis. Instead of *update(c)*, the specification from the SHERLOCK-Approach would contain *Collection.update* as a transition. Very much alike the approach by Ammons et al. [2002], this approach creates a probabilistic automaton using a PFSA-Learner and then determinizes it. The automaton created by this approach has no accepting state, only an initial one. Thus, it can't be told if a given set of traces would lead to an accepting state or not, This is probably due to the fact that every state is accepting, since the invariant here tells us that the program code that is used to mine the specification has to be bug-free. Thus, as long as there is an outgoing state for the current input, the usage would have to be correct.

### 2.2.8 Livshits, Nori, Rajamani and Banerjee (2009)

Merlin, the specification miner created by Livshits et al. [2009], aims at the creation of a formal specification that displays the information flow between the observed methods [Livshits et al., 2009].
Once the algorithm has the execution traces, a propagation graph is built. A propagation graph is a graph that displays the information flow in a program. The methods of a program are displayed as nodes, an edge indicates the flow of information between those two methods. As described by Livshits [2006], nodes in a propagation graph are *sources*, *sanitizers*, *sinks* or *regular* nodes. Merlin classifies the used methods to one of the node-types mentioned above. The propagation graph then gets translated to a *factor graph* [Yedidia et al., 2003]. A factor graph is a bipartite graph that has a node for every variable and every function. An edge between a function node and a variable node indicates that the function the node represends uses the variable. Using the factor graph, the algorithm creates a *probabilistic set of constraints* by applying a probabilistic inference on the factor graph.
Figure 2.8 displays a test code with many information flowing between the functions and the infered factor graph. Quite simmilar to the Annotation Factor Graph used by Kremenek et al. [2006], the factor graph here consists of circular and rectangular nodes. Circular nodes are variable nodes for the variables and rectangular nodes represend functions/methods of the program. An edge between a function node and a variable node indicates the usage of the variable that is represented by the variable node in the function from the function node. Quite simmilar to the approach of Kremenek et al. [2006], the result of the algorithm is a factor graph. The factor graph displays the information flow between methods. This can be helpful to apply model checking techniques to the program, for the approach described in this thesis, however, this is not really helpful – as it tries to create probabilistic specification in form of a probabilistic automaton, that can be used by programmers to get familiar with different APIs and classes.

### 2.2.9 Mao, Chen, Jaeger, Nielsen, Larsen and Nielsen (2011)

Mao et al. [2011] presented an algorithm to learn a probabilistic automata. Using this automata, one can infer linear temporal logic properties and perform probabilistic linear temporal logic (PLTL) model checking on the learned specification [Mao et al., 2011].
The algorithm starts by building a deterministic labeled markov chain (DLMC) from the execution traces by applying a modified version of the Algeria-Algorithm [Carrasco and Oncina, 1994]. Their version of the algorithm, *Aalgeria*, is basically the Algeria-algorithm as described by Carrasco and Oncina [1994] but with a compatibility-criterion by Angulin. The algorithm first creates a *frequency prefix tree acceptor* (FPTA) from the Dataset. A frequency prefix tree acceptor is a tree where every state represends a string (i.e. a Word, $\Sigma^*$, assuming that the program to be an automaton) from the prefix set. The states are labeled with the number of strings with that prefix and the amount of occurences of this state.

```
public void TestMethod1() {
    string a = ReadData1();
    string b = Prop1(a);
    string c = Cleanse(b);
    WriteData(c);
}
```

```
public void TestMethod2() {
    string d = ReadData2();
    string e = Prop2(d);
    string f = Cleanse(e);
    WriteData(f);
}
```

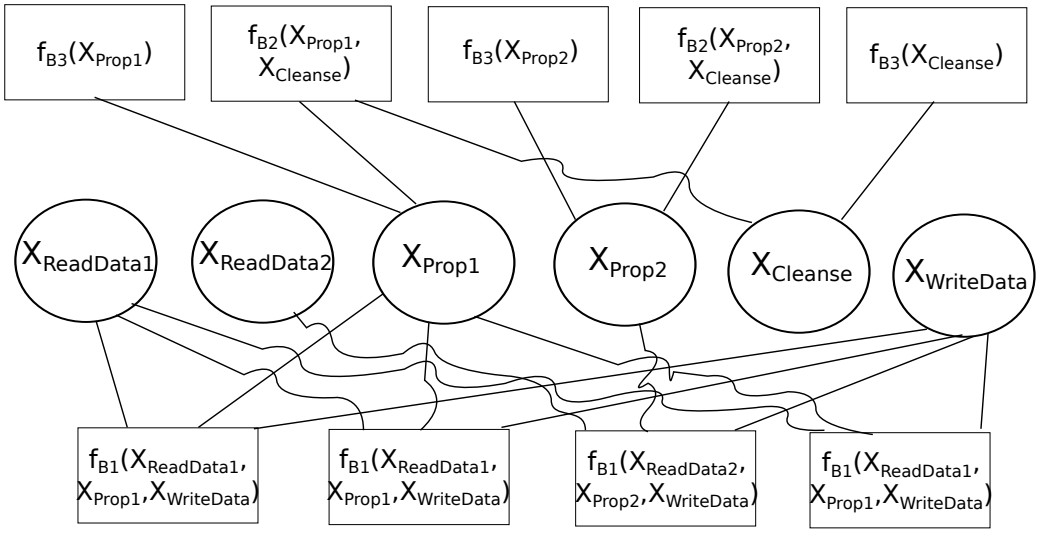


**Figure 2.8.** Example for a code and an infered factor graph using the described method. As seen in [Livshits et al., 2009].

Using that tree, the algorithm creates a second FPTA. This FPTA is created by merging states from the first FPTA. It does so by ordering the states into RED and BLUE states. RED states will be included in the final version of the second FPTA, while BLUE states have to be tested for compatibility with any of the RED states. If two states, $q_b \in$BLUE and $q_r \in$ RED are compatible (according to Angulins criteria), they are merged. The merge function of the algorithm first redirects the incoming transition of $q_b$, let's say from $q_e$, directly to $q_r$. Therefor, the frequency of $q_e \rightarrow q_r$ gets set to the frequency of $q_e \rightarrow q_b$ and then the frequency of $q_e \rightarrow q_b$ gets set to 0. This step then gets repeated for all the following states, where the frequencies of the successors of $q_r$ get added to the ones of $q_b$ and after that, the frequencies of $q_b$'s successors get set to 0. The result is a deterministic probabilistic finite automaton, that is used to create the desired Deterministic Linked Markov Chain (DLMC). This happens by normalisation of the transition frequencies.

On figure 2.9, a DLMC for the simulation of a gambling game is shown [Mao et al., 2011]. The game works as follows: a player rolls a six-sided dice twice and then check. If he has a total count of 7 or 11 points, he wins. If he has a count of 2, 3 or 12 points, he looses. If he

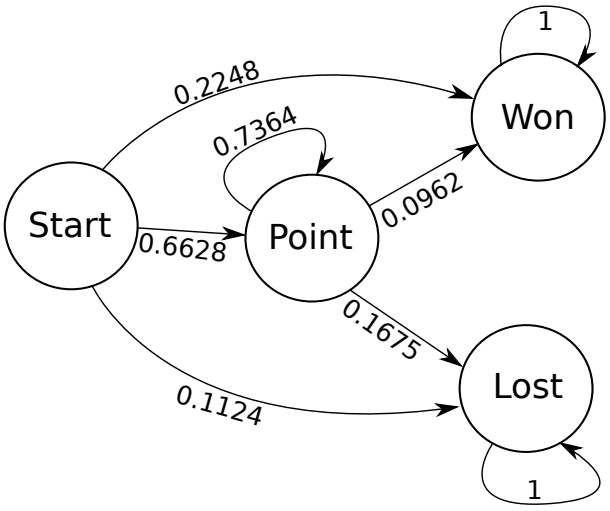


**Figure 2.9.** An example for a DLMC for a gambling game where a player rolls two dices with 6 sides each and have to get a total count of 7 or 11 points. If he has 2, 3 or 12 points, he loose. On all the other cases, he rolls again. if he get a total count of 7 with the second roll, he looses if he gets the rolls from the first roll, he wins. The player rolls again until he either wins or looses. As seen in [Mao et al., 2011]

has any other number, he rolls again until he either has the points from our first roll again, meaning that he wins, or he gets a total of 7 points, what makes him loose.
*Start* is the initial state where the player haven't been rolling any of the dices. A transition indicates that he was rolling both dices again. *Won* is the state where he won the game, *Lost* is the one where he didn't. *Point* indicates that he has to reroll the dice to get our initial points. The example was made by Mao et al. [2011] by using 65 observation sequences. As can be seen is, that the sum of the probability for all the outgoing edges of any state always equals 1.
The resulting specification has some similarities to the one that this thesis tries to infer. While they use certain *states* to define their DLMC, like *Win* or *Point* from figure 2.9, and uses transitions to mark the probability of a change, the approach presented in this thesis aims to predict the next called function in any state with the probabilistic value.

## 2.3 LearnLib

LearnLib [Raffelt et al., 2009] is a framework with implementations of several automata learning algorithms. It is developed by the Dortmunt University of Technology, Germany.

### 2.3.1 Description

LearnLib started as a closed-source Java library [Howar et al., 2014c]. This version was supported by the european union and featured implementations of learning algorithms such as Observation packs algorithm and a graphical modelling tool called *LearnLib studio* [Howar et al., 2014b].
The current version of LearnLib is completely re-implemented [Howar et al., 2014b]. The new version is open-source under the licence of LGPLv3 [Howar et al., 2014a]. It features other algorithms than the original, closed-source version, such as Random-Walk and Wp-Methods.
As data-structures for the used automata, LearnLib uses the AutomataLib–Framework. This framework is described in section 2.4.
LearnLib has a very modular structure. It uses Maven for its dependency- and plugin-management. The overall project consists of many single projects that are being held together by the learnlib-parent. This particular project specifies the overall project info for every new module. Every project that is part of LearnLib inherrits from this project by mentioning it as the parent in maven configuration file, *pom.xml*. Also, the parent projects *pom.xml* holds all the other projects as modules.

## 2.4 AutomataLib

AutomataLib is a Java-framework developed by Malte Isberner from the Dortmund University of Technology [Isberner, 2014]. It offers models for storing graphs, automata and transition systems.

### 2.4.1 Description

AutomataLib was originally designed as a framework especially for the usage of LearnLib [Isberner, 2014]. It can also be used as a framework on its own, to offer classes for automata and graphs. Like LearnLib, it is completely open-source and written under LGPLv3.

## 2.5 Javassist

Javassist is a Java-bytecode manipulating framework developped since 1999 by Shigeru Chiba [Chiba, 2000]. Using Javassist, one can, during runtime, dynamically create new classes and fields. It is also possible to manipulate an existing class and insert own code to the beginning or the end of existing methods.

### 2.5.1 Description

Javassist stands for *Java programming assistant*. It offers a framework that extends the Java reflection API in a way it can deal with Java bytecode manipulation at runtime [Chiba, 2000]. Due to the fact that it uses an own compiler, that runs on every existing java architecture, it can be used on every operating system with a working Java environment [Chiba, 2000]. It can manipulate java classes on both compile-time and load-time.
Javassist uses an own compiler to insert the Java-code to the given class file, hence a user using this API does not have to have any knowledge of Java bytecode itself. During the implementation phase with Javassist, the user enters normal Java code, that Javassist then compiles to Java bytecode. The compiled byte-code can be used as an own class or to extend methods. On a method extension, the compiled byte-code gets inserted to the previously compiled method.

### 2.5.2 Usage

Javassist controlls the byte-code manipulation in a class called *ClassPool* [Chiba, 2000–2012]. The code from listing 2.1 creates a ClassPool object called *pool*. In order to manipulate a

```
1   ClassPool pool = ClassPool.getDefault();
```

**Listing 2.1.** Create a ClassPool Object

class at load-time (when it's loaded for the first time), Javassist creates a compile-time class (*CtClass*)-Object. The ClassPool offers such CtClass Objects. Using the code displayed in listing 2.2, the ClassPool returns a CtClass object for the given class name.

```
1   CtClass ctClass = pool.get(className);
```

**Listing 2.2.** Get the CtObject for a certain class

The CtClass-Object contains contains all the information of the original class file, but can still be manipulated. Once the manipulation is done, the changes can be applied to the original class object by applying the code displayed in listing 2.3. Using the code shown

```
1   Class newClass = ctClass.toClass();
```

**Listing 2.3.** Change the reflected CtClass to a Class-Object

in listing 2.4, it is also possible to export the class and write it to a file. That way, one can reuse a class file once created on future runs without the need to compile it again.

```
1           ctClass.writeFile();
```

**Listing 2.4.** Export a class to the hard drive.

Using the class pool, it is also possible to create a new class, fill it with methods, make it inherrit another class and insert certain fields. The code from listing 2.5 produces a new class named “Test”, inserts the method “doSomething()” and an integer for the state.

```
1           // Create class "Test".
2           CtClass testClass = pool.makeClass("Test");
3
4           // Update its superclass.
5           testClass.setSuperclass("Superclass");
6
7           // Create a new integer field and insert it to our class.
8           CtField newField = CtField.make("private int state = 0;", testClass);
9           testClass.addField(newField);
10
11          // Create a new method and insert it into our class.
12          CtMethod newMethod = CtNewMethod.make("public int doSomething()"
13          +"{state = state + 2; return state;}", testClass);
14          testClass.addMethod(newMethod);
15
16          // Now, we finalize it and get the class file.
17          Class newClass = testClass.toClass();
```

**Listing 2.5.** Create a test class, insert Method doSomething(), make it a subclass of Superclass and give it an integer.

In order to manipulate any method of an existing class, probably the easiest method in Javassist is it to implement the *Translator*-Interface that Javassist offers and insert it to the ClassLoader. The methods of the Translator-Interface can be seen on listing 2.6.

```
1           public void start(ClassPool pool)
2                   throws NotFoundException, CannotCompileException;
3           public void onLoad(ClassPool pool, String classname)
4                   throws NotFoundException, CannotCompileException;
```

**Listing 2.6.** Methods of the Translator Interface

The function *start* is executed when the translator starts for the first time. The interesting

function for our purpose is onLoad. This function is executed when the class specified in *className* is being load from the class path for the first time. A specified class can

```
1   CtClass currentClass = pool.get(className);
```

**Listing 2.7.** Get the class specified in className from the classpool specified in pool

be load using the code from listing 2.7. Once the CtClass object is obtained, it can be manipulated. In this case, neither the *writeFile()*-method, nor *toClass()* has to be called, as the class automatically gets returned after the *onLoad* function.
The compile-time methds (CtMethod) for the loaded class can be obtained by code simmilar to Javas built-in reflection, by calling the code shown in listing 2.8. For any CtMethod object

```
1   CtMethod[]  methods = currentClass.getDeclaredMethods();
```

**Listing 2.8.** Get the CtMethods of a CtClass file called currentClass

representing an actual method from a Java class, the code can be inserted either before the actual method body is being executed or after it. Therefor, a String is needed that contains actual compilable Java-code. The example shown in listing 2.9 manipulates a method in a way that a simple text output denotes the beginning and the end of a function. Note that

```
1   final String premethod = "{System.out.println('Method␣begins');}";
2   final String postMethod = "{System.out.println('Method␣ends')}";
3   currentMethod.insertBefore(preMethod);
4   currentMethod.insertAfter(preMethod);
```

**Listing 2.9.** Manipulate a method called currentMethod so the program knows that it begins or ends

both methods, insertBefore() and insertAfter(), can throw a *CannotCompileException* in case the code specified in the String parameter can't be compiled.

Chapter 3

# Approach Section

This thesis covers a topic that has yet to be fully researched. Thus, many questions still remain unanswered. This chapter introduces the *research questions* (RQ), *qualitative research questions* (QRQ) and the *implementation requirements* (IR) related to this thesis. The implementation requirements are analysed in chapter 4, the research questions in chapter 5 and the qualitative research questions in chapter 6.
This chapter also contains the experimental setup for this thesis describing the experiments done for the evaluation of the specification miner.

## 3.1 Description

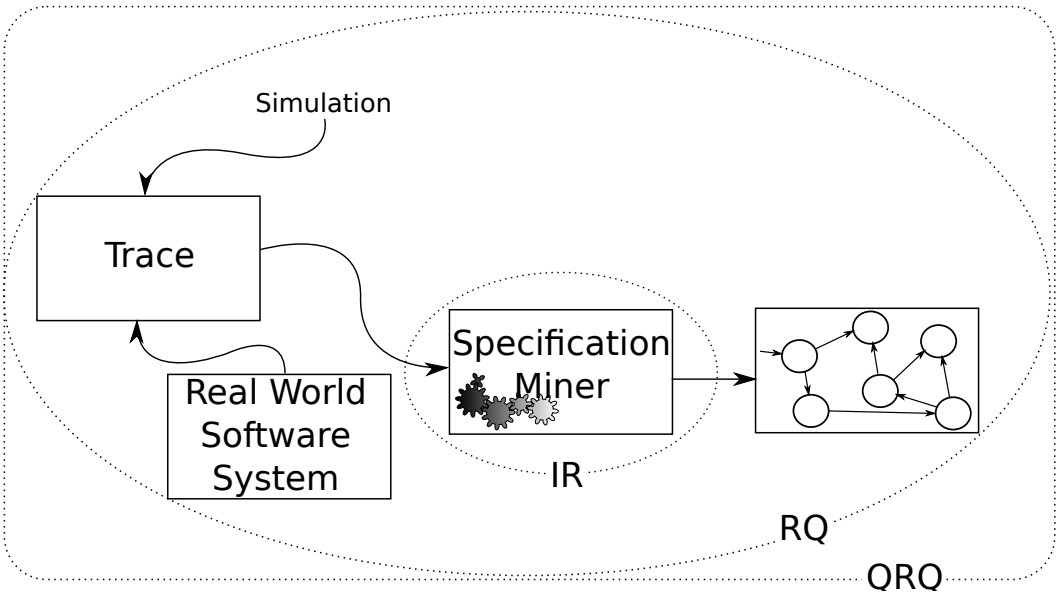


**Figure 3.1.** Image that describes the architecture of a specification miner and shows the connection between the architecture and research questions / implementation requirements.

This section covers both the research questions and the implementation requirements.

Figure 3.1 shows both the architecture of a specification miner and the connection between the project and the implementation requirements and research questions.
The overall project works as follows: from simulations or from actual real world software systems, the program gets the traces from program executions. Using the traces, a Spcification Miner creates a formal specification. What comes out is a probabilistic automaton that can also be drawn as a graph.
The implementation requirements are requirements to the specification miner that describe and specify how the *product* has to be implemented. It therefor only covers the source code of the specification miner itself. The research questions on the other hand cover the whole project. They are open questions that research has not yet fully answered. The qualitative research question covers the whole project, alongside with the research question.

## 3.2 Implementation Requirements

Most of the specification miners work by forcing the program to leave a trace at runtime. Those traces later get analysed. Given that the may source code of the program that has to be analysed may not be known, in order to change and recompile the program manually on our own, the following implementation requirement comes up for many specification miners:

**IR1:** Functions of specific classes have to be forced to leave a trace at runtime.

Most important of all is, that leaving traces must not, under any circumstances, change the general behavior of the program.

A tracing method that works is to use an own class loader for the program that has to be observed. The class loader should manipulate the program in a way that it leaves traces when it is called. The problem here, is that any java program (especially when it uses the probabilistic specification miner described in this thesis as an API) initially starts with the generic Java-Classloader. After the creation of a new class loader, that works for the tracing purposes, it can easily replace the generic class loader and run methods using the new classoader. The problem then only is, that any method, any class and any object loaded with the new class loader, even when it didn't change anything from the actual class file, can not be handed over to an object, method or class that was loaded with the generic class loader. This leads to the next implementation requirement:

**IR2:** Information between two functions that are using a different class loader have to be exchanged.

Naturally, it would be desirable to return a probabilistic specification object back to the function that uses the specification miner. That means, that the specification miner has

to return an object that was loaded using the generic Java class loader. Therefor, the probabilistic specification has to be created using the generic Java class loader to avoid a *ClassCast-Exception*. The probabilistic specification, however, can not be created without having the traces. But the traces were created using an alternate class loader. This means, that it isn't possible to just hand them over to the function from the generic class loader. Thus, a way of communication between functions that are using different class loaders has to be found.

Once the specification miner created a probabilistic specification, the information has to be stored somehow. Naturally, the person who ordered the program to create the probabilistic specification planns to use it afterwards. This leads to the following implementation requirement:

**IR3:** Given the specific data of a probabilistic specification, we have to store the probabilistic automaton efficiently and have fast access to each node and edge.

This requirement is a soft requirement, but it is anything but trivial. A probabilistic automaton can have quite a lot of nodes and edges, If they are to be stored in a way that it needs linear time to access any node and edge, let's say, by holding them in a list, iterating over a complete probabilistic automaton of $m$ nodes and $n$ edges would result in a complexity level of $\mathcal{O}((m \cdot n)^2)$, meaning that it would take a lot of time. Even a logarithmic time would take us $\mathcal{O}((m \cdot n) \log(m \cdot n))$. That's why it is desirable to store the information in a way that it needs a constant access time to every node and edge. That way, iterating over the whole automaton can be done in $\mathcal{O}(m \cdot n)$

## 3.3 Research Questions

Once the specification miner calculated an automata that is designed as a mealy automaton, meaning that the edges form *words* over our input alphabet $\Sigma$ (with every $s \in \Sigma$ being a function name in our case), the edges have to be labeled with certain *costs*. The costs are not costs in a traditional way, they are a probabilistic real value telling us for every edge connecting two states $v$ and $w$ how likely it is that, once we're in state $v$, we will use the transition to state $w$ in the next round. Those values have to be calculated using the traces that were used earlier to create the automaton alongside with a certain formula. The probabilistic values have to be a real value between 0 and 1 and for any node in our specification the invariant has to hold, that the sum of probabilities of all outgoing edges has to be 1.0, meaning that any path has to be taken. Finding a formula that assigns accurate values to the edges leads us to the second research question:

**RQ1:** Given an automata and a set of traces leading to accepting words, how can the probability be calculated that any transition is to be taken?

Naturally, there are several ways to achieve this. This point covers the question of which way is decirable for our purpose.

The probabilistic specification should also be as accurate as possible. Therefor, it is good to know if the specification the algorithm described in this thesis actually creates a specification that describes the behavior of the API the specification miner had to mine. Given that the traces are just comming from a *sample* of executions, there can always be *false positives* and *false negatives*. A *false positive* in our context would mean, that there is a collection of function calls that leads to an accepting state in our automaton, but that would lead to an error if it would actually be applied to the real program in that order. Simmilarly, a *false negative* leads to a non-accepting state in our automaton, but wouldn't throw any compile errors or lead to bugs the coded version.
Once there is an algorithm that can mine probabilistic specifications, the following research question comes up:

**RQ2:** Given an algorithm that creates a probabilistic specification from sample execution traces, how can its functionality be verified?

## 3.4 Qualitative Research Questions

The overall target of any thesis is it to create something that helps either making existing tasks easier/better, or making entirely new tasks that were not possible before.
This thesis aims at automatically creating a probabilistic specification for any API. To give this topic a practical relevance, the final research question comes up:

**QRQ1:** How can a probabilistic specification of an API be used to improve software?

## 3.5 Experimental Setup

As for the evaluation, this thesis uses the metrics described by Lo and Khoo [2006], QUARK, to evaluate the automaton-based specification. The miner from the SHERLOCK-Approach offers three different methods to calculate the transition probabilities, namely a Lightweight Adaptive Filter, the Ratio and the Keep Alive Models with Implementation-Approach. All of them offer different mechanisms to infer transition probabilities for a given deterministic automaton together with traces from the language the automaton represends.

The Lightweight Adaptive Filter works in rounds. On every round, the filter measures the transitions taken from the automaton and then uses the latest measurement to update the estimation matrix. The filter adapts to changes from the previous measurements by constantly changing the weight that is put on the latest measurement. That way, it successfully attenuates noise.
The Ratio simply increases a counter for any edge taken. At the end, the probability of every edge connecting the states $s$ and $t$ is calculated by dividing the measured counter by the amount of times the automaton was at state $s$.
The Keep Alive Models with Implementations (KAMI) Approach takes an initial, Dirichlet-distributed transition matrix and puts a weight on the measurement of the latest round. Using that weight, it adjusts the measurement. Unlike the Lightweight Adaptive Filter, the weight that is calculated here is only based on the amount of times a transition has been taken.
A more detailed description of the three probabilities can be found in section 4.1.3
For the three probability approaches, let $L$ be the Lightweight Adaptive Filter, $R$ the Ratio and $K$ the Keep Alive Models with Implementation Approach, $L, R, K \in P$ with $P$ being the set of Probability Functions implemented in this specification miner.
For any probability function $x \in P$, let $M(x)$ be the metrics that evaluates the automaton $A(x)$ that was created using the probability function $x$.
For any of recreated automata $Y$, the input automaton $X$ is used to measure both the *trace similarity* and the *probability similarity*.
The *trace similarity*, $X(Y)$, holds the per centage of accepting traces that can be created from automaton $X$ and that are accepted by $Y$. $Y(X)$ holds the conter-value, meaning the per centage of accepting traces that can generated with automaton $Y$ and that also lead to an accepting state in $X$ [Lo and Khoo, 2006].
Probably the most interesting value for the validation of the probability function from the QUARK-Framework is the *probability similarity* $PS(X, Y)$. To calculate it for the input automaton $X$ and any of the recreated automata $Y$, Lo and Khoo [2006] first define the *Co-emission* of the two automata as $P_{CE}(X, Y) = \sum_{s \in L(X \cap Y)}(P_X(s) \cdot P_Y(s))$. It takes the all the sentences that can be created with both automata $X$ and $Y$ and calculates the product of the probabilities for each automaton to create this sentence. After that, it sums up over all the sentences that both automata can create.
Using the *Co-Emission*, the authors then define the *probability similarity* as

$$PS(X, Y) = \frac{2 \cdot P_{CE}(X, Y)}{(P_{CE}(X, X) + P_{CE}(Y, Y))}$$

The comparisson of the probabilities uses those three metrics. For all three probabilities, a given Specification has to be *recreated* by running the miner with an interpreter that pretends to be a program from the given specification. For the original specification $X$ and the recreated specification $Y$ the metrics are applied and analysed.

Using the same technique, an hypothesis-test is performed afterwards. From 1.000 runs, every metrics $M$ is saved as a sample for the hypothesis test. For any automaton $Y$ recreating a given automaton $X$, the metrics $M$ is defined as

$$M(X,Y) = X(Y) \cdot Y(X) \cdot PS(X,Y)$$

This means, the amount of words accepted by one automaton but that are declined by the other counts as much as the probability simmilarity. The evaluation from section 5.1 showed that, when using enough runs from the input automaton, one can force both values $X(Y)$ and $Y(X)$ to be 1; this updates the used metrics to

$$M(X,Y) = PS(X,Y)$$

An hypothesis test as described by Arcuri et al. [2012] now tests the assumption that, no matter what probability we use, the resulting metrics is always the same.
This leads to the following hypothesisses:

$$\begin{array}{l|l} H_0^{K,L} : M(KAMI) = M(LAF) & H_1^{K,L} : M(KAMI) \neq M(LAF) \\ H_0^{L,R} : M(LAF) = M(RATIO) & H_1^{L,R} : M(LAF) \neq M(RATIO) \\ H_0^{R,K} : M(RATIO) = M(KAMI) & H_1^{R,K} : M(RATIO) \neq M(KAMI) \end{array}$$

For the macro-evaluation, the automata $X$ and $Y$ are mined from real projects, $Y$ with all of those projects used for $X$ and some more. Given that, during the macro evaluation, all the automata $A(L)$, $A(R)$ and $A(K)$ are created using the same traces, it is clear that the $X(Y)$ and the $Y(X)$ values for any automata $X$ and $Y$ created using the approach described in this thesis is 100%. Thus, the value for $M(z)$ for any $z \in P$ is defined as

$$M(z) = PS(X(z), Y(z))$$

This means, for any of the used probability, the Probability Similarity after Lo and Khoo [2006] is measured to see, how much the probability has changed using the probability.
To compare the used probabilities, a small analysis-step is performed. There, the goal is it to see how much the probability simmilarity changes on otherwise completely equal probabilistic specifications $X$ and $Y$ from the usage of a certain probability.
Let $x, y \in P$ be two different probabilities. Then

$$M(x,y) = abs(PS(X(x), Y(x)) - PS(X(y) - Y(y))$$

is the metric that measures the change in their probability simmilarity.

Chapter 4

# Approach

This chapter describes how the goals mentioned in section 1.2 and the implementation requirements from section 3 were handled. Also, it introduces the approach from this thesis, called Sherlock. Sherlock stands as an akronym for **S**pecification-Mining **H**andy **E**nrichment for the **R**e-implemented **L**earnLib **O**pen-Source **C**omputational-learning **K**it.

## 4.1 Description

This section describes the Sherlock-approach used in this thesis to create a probabilistic specification.

### 4.1.1 Collect traces

SHERLOCK first has to collect traces from the executions of Unit-Tests. To achieve this, Sherlock uses the JAVASSIST-Framework [Chiba, 2000] that manipulates every function from the observed classes.
The framework is used to insert tracing code to any method of every class that has to be observed as follows: before the byte-code of the actual method body, JAVASSIST inserts code that forces the function to leave a trace that contains both the class- and the function name. There are currently two different functions to choose from, one that leaves a trace in an internal list and one that writes traces to a temporary file. The reason for that will be explained later.
Due to Implementation Requirement 1, Sherlock is using the manipulated classes to run the Unit-Tests. Those tests then use the manipulated classes and hence leave a trace for every called method. This closes **IR 1** from section 3.2, as we now *can* force methods of any class to leave a trace whenever it is called.
Sherlock also uses the Javassist-Framework to plant an *init*-function at the beginning of every unit test class, so the specification miner knows when a new test starts. Without this code, for two testing functions $s$ and $t$, where $s$ is executed right before $t$, Sherlock would insert a transition from the last observed function that $s$ uses to the first observed function from $t$.

### 4.1.2 Creating an automaton

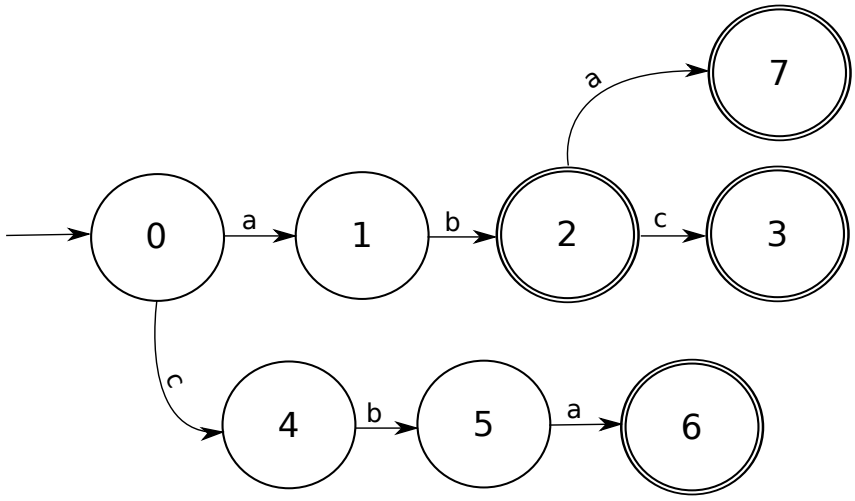


**Figure 4.1.** Example for an initial DFA created by three traces $t_1 = a \rightarrow b \rightarrow c$, $t_2 = a \rightarrow b$, $t_3 = c \rightarrow b \rightarrow a$ and $t_4 = a \rightarrow b \rightarrow a$.

Sherlock uses those traces to create a deterministic finite automaton that uses new states for any new path that hasn't been taken before. For the transitions, the function names are used. For example, for four sets of traces, $t_1$, $t_2$, $t_3$, and $t_4$, with $t_1 = a \rightarrow b \rightarrow c$, $t_2 = a \rightarrow b$, $t_3 = c \rightarrow b \rightarrow a$ and $t_4 = a \rightarrow b \rightarrow a$, the resulting automaton is displayed in figure 4.1. Every new trace gets an own path and the last state of a set of traces is made an accepting state.
The Sherlock-implementation uses the CompactDFA-class provided by the AutomataLib for this initial automaton.
Sherlock then uses Angulins $\mathcal{L}^\star$-Algorithm [Angulin, 1987] provided by the LearnLib-Framework to merge equivalent states.
The result is a deterministic finite automaton. This automaton then gets to be translated to a probabilistic specification object from the ProbabilisticSpecification-Class. It represends the automaton internally as a graph by assigning an integer value to every state and then using arrays for offsets, targets, input and the probability for every edge.
Every edge of the probabilistic specification is labeled with both the input value, namely the function name that belongs to this transition, and a floating point value indicating the probability that, once the automaton reaches the source state, this transition will be taken.

### 4.1.3 Calculate the probability

At this point, the probabilities for any transition is at 0.0. To calculate them, the Sherlock-implementation offers three different mechanisms: a *Lightweight Adaptive Filter*, the *Ratio* and the *Keep Alive Models with Implementations*.

#### Lightweight Adaptive Filter

The Lightweight Adaptive Filter described by [Filieri et al., 2014] was designed to automatically adapt to changes from periodic measurements when measuring a signal. This approach is based on the assumption that our program represends the *original* signal, and with the traces, the filter is trying to recreate its transition probabilities.
The described algorithm uses several rounds for the measurement. After every round, an estimation is updated by using the latest measurements using adaptive filtering mechanisms.
For the probabilistic specification with $n$ states, the filter creates a $n \times n$ Matrix $m_{n\times n}$ to use as occurence count for the current round. To fill them, Sherlock reuses the traces it collected earlier. To do that, it traverse the graph according to the traces and raises the counters on our matrix for any transition it takes. Due to inflicted code in the testing methods, Sherlock knows when a test is done, hence it knows when a round is over. After any round, the latest measurement data is used to update our estimation accordingly.
Once the final estimation matrix $m_{n\times n}$ has been calculated, for any edge connecting to states $i$ and $j$, $i, j \in \mathbb{N}$ and $i, j \leqslant n$ Sherlock takes the probability of $m_{ij}$ from our matrix and assign it to the probability of the current edge.

#### Ratio

The ratio uses the statistics of the measurements as probabilistic factor. It works like the lightweight adaptive filter, but with only one round.
Simmilar to the approach on the lightweight adaptive filter, for counting purposes Sherlock creates a $n \times n$ Matrix $m_{n\times n}$ to use it as an occurence counter. Also, an array with $n$ entries, $a_n$, works as state counter. Whenever the automaton reaches state $i$ and takes the transition to state $j$, Sherlock increases both the state counter for $a_i$ and the occurence counter $m_{ij}$.
Once a new round beginns, Sherlock resets the current state to the initial state and uses the remaining traces to continue our count.
Once Sherlock is done with all our traces, it assigns for any edge connecting the states $i$ and $j$ the probability $p(e) = \frac{m_{ij}}{a_i}$.

**Keep Alive Models with Implementations**

The Keep Alive Models with Implementations Approach, short KAMI, was described by Epifani et al. [2009] and Filieri et al. [2012]. It was one of the earliest approaches for infering probabilities of a discrete time markov chain (DTMC) that uses parameter adaptation for the measurement [Filieri et al., 2014].
KAMI uses a Bayesan-based inference for the measurements [Filieri et al., 2012]. For $d$ sets of traces, all starting from a common state $s_{init}$, Sherlock measures the amount of times the automaton took the edges in the $h$-th run from a state $i$ to a state $j$ in $N_{ij}^{(h)}$ [Epifani et al., 2009]. The goal is to get is an estimation matrix $M_{n\times n}$ that holds on $M_{ij}$ the probability that the edge leading to state $j$ is taken, once the automaton is in state $i$. Initially, the matrix $M$ has a *Dirichlet distribution*. With any of the $d$ sets of traces, we update the estimation matrix $M_{ij}$ to $M'_{ij}$ with the following formulas [Filieri et al., 2012]:

$$\alpha' = \alpha + N_i$$
$$M'_{ij} = \frac{\alpha}{\alpha'} \cdot M_{ij} + \frac{N_{ij}}{\alpha'}$$

$\alpha$ and $\alpha'$ serve as weight, indicating how much trust is put in the latest measurement. $N_i$ is the amount of times the automaton has been at state $i$ so far, which can be calculated by $N_i = \sum_j N_{ij}$ or counted separately.
As stated out by Filieri et al. [2012], those updates are fast to calculate. After every step, the value of $\alpha'$ gets assigned to $\alpha$ and the posterior Matrix $M'$ is the new prior distribution $M$. One can show that, if $M$ is a *Dirichlet distribution*, the Matrix $M'$ that is calculated using those updates is a *Dirichlet distribution* as well [Filieri et al., 2012].

### 4.1.4 Remove redundant information

Figure 4.2 shows a probabilistic specification that was created using the Sherlock-approach until this point. The automaton represends a program that offers the functions $a$, $b$ and $c$. The specification still has a lot of redundant information. For example, the transition from state 1 to state 4 has a probability of 0%.
That's why Sherlock removes redundant information from the automaton. During the calculation of the probability, it traverses the automaton using the collected traces. While doing so, it marks every edge used in the automaton. Sherlock then removes all unmarked edges, as they have not been used. Note that, eventhough in figure 4.2 they are drawn as one edge each, some edges contain of multiple edges that were just summarized. For example, the edge that goes from state 0 to state 1 would actually have to be drawn as two separate edges, both starting at state 0 and ending at state 1, both with the probability of 99.999964%, but one of them labeled with $b$ as input and one of them with $c$. Hence, marking used edges and removing unmarked ones can lead to smaller inputs for the transitions.

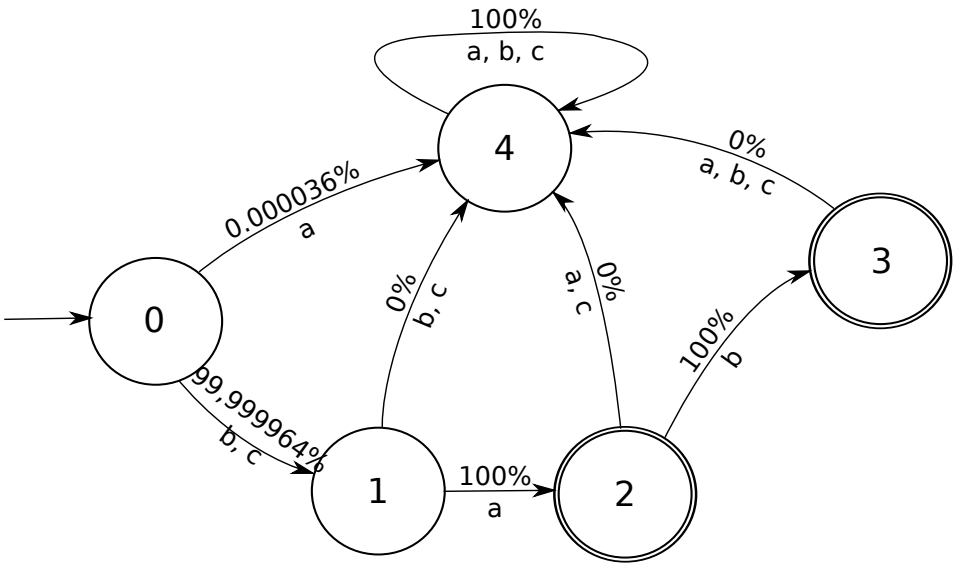


**Figure 4.2.** A probabilistic specification made using the described methods. Note that there i a lot of redundant information left.

Once the edges have been removed, Sherlock checks for the incoming transitions of every state. Any state except for the initial state has to have at least one incomming edge. If it does not, Sherlock removes both the state and all its outgoing transitions. Given the fact that due to the removal of the outgoing edges, other states could become useless, Sherlock checks for states to remove until there is one whole loop where no new state is marked as useless.

The pruned version of the probabilistic specification from figure 4.2 can be seen at figure 4.3. Unnessecary transition have been removed, hence for all of the remaining transitions the probability now is bigger than 0.

While the automaton from figure 4.2 had to be stored using 5 states and 15 edges, the pruned version from figure 4.3 has 5 states and only 8 edges, without losing any information compared to the original one from figure 4.2.

### 4.1.5 Store the Specification in the ProbabilisticSpecification-Object

After Sherlock successfully calculated and pruned the probabilistic specification, it has to store it in the object – according to **IR 3**, it has to be stored in a way that we can access information fast. The probabilistic specification object serves as a class that can be extended if needed. The Sherlock-Approach uses extensions for the different probability objects.

The overall ProbabilisticSpecification Object can store the probabilistic automaton using

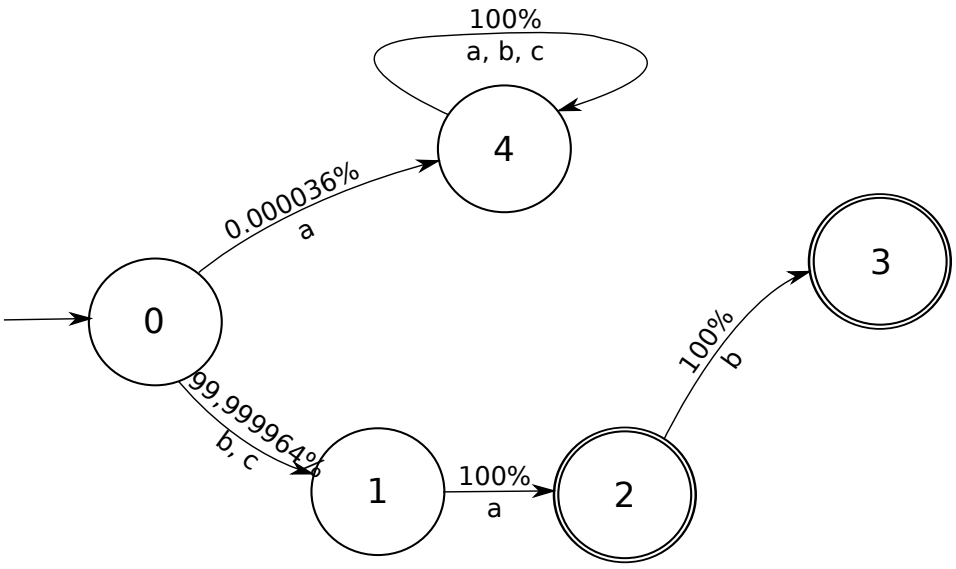


**Figure 4.3.** The probabilistic specification from figure 4.2, after Sherlock removed the redundant information.

several arrays. Any of the $n$ states from the automaton is represented by the integers $1 \dots n$. The state has a state type that is stored in an array called *stateTypes*. For any of the $n$ states, the $i$-th entry denotes the state type of the $i$-th state by using an enum-type. For the $m$ edges, there are three arrays of $m$ entries. Any edge has a target, so the *target*-array points at the state this edge leads to. The edge also has a probability, thus a *probability*-Array holds the floating point valued probability for any of the $m$ edges. The *functionName* Array holds the input function this edge represends.
To assign the $m$ edges to the $n$ states, an *offset*-array with $n+1$ inputs holds on the $i$-th input the index of the first edge that has the $i$-th state as a source. The indices of the edges that belong to state $i$ are all $j \in \mathbb{N}$ with $offset[i] \leqslant j < offset[i+1]$. All edges of index $j$ have state $i$ as a source.
This method offers a constant access time for any edge. To iterate over the full automaton takes a complexity-level of $\mathcal{O}(m)$. This closes **IR 3** from section 3.2, as the probabilistic specification class now offers fast access to any node and edge.

## 4.2 Usage of Javassist

The Java-bytecode manipulation framework Javassist [Chiba, 2000] has been used in this project to force functions to leave a trace. To achieve this, a translator inserts the new code to the class files whenever the desired class is loaded. From the input parameters, Sherlock

has a list of regular expressions for classes it has to manipulate. Whenever the manipulated ClassLoader loads a new class, it checks first if the class one of the classes Sherlock wants to observe. If it is, Sherlock creates a *prefunction*, a String containing Java-code that calls the tracer. The *AbstractTranslator* forces any class that extends it to implement a function that returns java code that the Javassist-Compiler can compile for the prefunction.
The newly created translator that inserts the tracing code into the functions only works with a new ClassLoader that differs from the default Java-classloader. As described in **IR 2**, any internally used trace now can't be handed back to the generic Java class loader and thus, can't be used to create and return a probabilistic specification to the function that is using the generic class loader. To overcome this problem, the final version of Sherlock contains two different translators, both of them extending the abstract class *AbstractTranslator*.
The first translator is called *ListTranslator*. The ListTranslator uses an internal list of Strings to keep track of the traces. That way, the traces never leave the main storage and thus this is the more efficient translator. This also means, that Sherlock can't hand the traces back to the generic Java class loader the parent function probably uses. Thus, the probabilistic specification only is created internally. The only possible thing to do is to export it to the hard drive at a location specified by the user.
In case the programmer wants to do more with the probabilistic specification than just exporting it, Sherlock offers the function *ProbabilisticSpecification createProbabilisticSpecification(...)*. It creates a ProbabilisticSpecification object according to the parameters and then returns it, so the programmer can use it. To overcome the ClassLoader-Problem mentioned above, this function uses the second translator: a *FileTranslator*. Like the ListTranslator, it manipulates class files, but the inserted code forces a BufferedWriter to write an index number representing the current trace together with a mapping table to a file in the temp-folder (on Unix-Systems */tmp*) and then, once the traces are completely collected, to read those file using the generic class loader Java uses. Using it, Sherlock can create a probabilistic specification and return it to the user. Naturally, this approach takes slightly more time, as file operations usually take more time than operations completely in the RAM. Also, for bigger sets of traces, Sherlock is bounded by the amount of free space left in the temp folder. This might seem trivial, but can be a real problem.
Using those two translators, Sherlock offers two different ways to enable a communication between functions that use a different class loader. Thus, it closes **IR 2** from section 3.2.
For the evaluation, it is also nessecary to know when a new test function from the Unit-tests starts. Not only does the LAF-Probabilistic described above work in *rounds*, where one round equals one test-function, the init function is also needed to prevent transitions that weren't in the actual code, but that would occur when test-function $t$ is executed right after test-function $s$. Sherlock would create a transition from $s$'s last observed function to the first observed function executed by $t$. Hence, the translators also put code at the beginning of every test-function, so the method tells the tracer that a new round starts now. The translator then also leaves a special trace indicating the beginning of a new round.

## 4.3 Internal Class Structure

The projects contains of several classes. Figure 4.4 contains the packets used in this project and their classes.

### 4.3.1 Packets

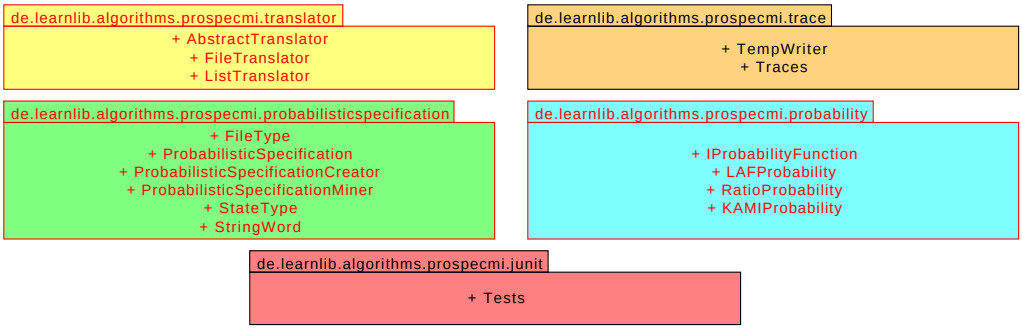


**Figure 4.4.** Packet diagram for the project.

The following subsections contain the packets names and describe their functionality.

#### de.learnlib.algorithms.prospecmi.junit

The packet de.learnlib.algorithms.prospecmi.junit contains the file that is needed to run the JUnit tests with our modified class loader. The file contains a main method, where the additional arguments array hols the names of the JUNIT-files Sherlock has to execute.

#### de.learnlib.prospecmi.probability

This packet contains all the classes needed for the calculation of the probability for our probabilistic specification. The *IProbabilityFunction* serves as an interface for the probability functions. It holds both the *calculateProbability()*-function that calculates the probability for the given probabilistic specification and the *getUsedEdges()*-Function. During the calculation of the probabilities, Sherlock traverses the automaton using the traces. While doing so, it marks those edges that were used. The function *getUsedEdges()* returns an array that holds a boolean-type value for every edge indicating if it was used during the calculation of the probability or not. The three classes *LAFProbability*, *RatioProbability* and *KAMIProbability* all contain different implementations of the interface. The approach of the probabilities are described in section 4.1.3.

**de.learnlb.algorithms.prospecmi.probabilisticspecification**

The packet holds classes for everything Sherlock needs to create a probabilistic specification. Any instance of the *ProbabilisticSpecification*-class holds a probabilistic specification as a finite-state probabilistic automaton. The *ProbabilisticSpecificationCreator* serves as an own class that creates a probabilistic specification once the traces from the *JUnit*-Package have been collected. The *ProbabilisticSpecificationMiner* is the class that any user calls when they want to create the probabilistic specification. It doesn't have to be instantiated, as the functions inside are static. From there, the specification miner initializes the translator and the classloader and calls the creator to create the *ProbabilisticSpecification*-Object.
The *StateType*-class holds an enumeration for the type of any state, telling us for example if the current state is an accepting state or an initial state. The StringWord is needed by the AutomataLib framework. It extends the net.automatalib.words.Word-interface-class for the usage Strings. This is needed for the input alphabet $\Sigma$ of AutomataLibs automaton. The *FileType*-enum holds information on accepted file types our export function understands.

**de.learnlib.algorithms.prospecmi.trace**

This packet holds all the classes needed for collecting traces. As described in section 4.2, Sherlock contains two different ways to temporarly store the traces. The *TempWriter* holds the BufferedWriter needed to write to the temp file with our traces. The Traces class holds a List for the same purpose.

**de.learnlib.algorithms.prospecmi.translator**

The translator-packet holds anything needed for translators. In it are the abstract class any translator has to implement, namely *AbstractTranslator*. In there it is defined what classes are changed. The code that gets plugged in to the methods has to be implemented by the other classes. Sherlock offers two implementations of the class, a *FileTranslator* and a *ListTranslator*.

## 4.4 Embedding it to LearnLib

The LearnLib has, as described in Section 2.3, a very modular structure. Embedding the new *module* into the LearnLib framework was really easy. First, the project had to be a maven project. This was already given for the project. Maven projects have a configuration file called *pom.xml*. This file configures the build path for the whole project as well as general settings like the used character encoding. In here, the *parent* project had to be set to Learnlib-Parent, so general settings get inherited.
The source code of the LearnLib comes from a Github-Repository. The root folder contains the pom.xml for the LearnLib-parent. In there, a folder for every project included to the LearnLib-Framewrk exists. They are all mentioned as *Modules* in the LearnLib-Parents

pom.xml file. The learnlib-algorithms project has some subprojects itself, like the $L^\star$-Algorithm by Angulin [Angulin, 1987] that was used for this project.
The probabilistic specification miner is a subproject of learnlib-algorithms. To embed it there, the whole projects folder had to be copied to the *learnlib/algorithms* folder and the project had to be mentioned as a module in the *learnlib/algorithms/pom.xml* so it gets loaded automatically when someone imports the learnlib-parents pom.xml.

## 4.5 Conclusion

The Sherlock-Approach had to fullfil three *Implementation Requirements* (IR):

- ▻ Force methods of classes to leave a trace whenever they are called.
- ▻ Exchange information between functions that use a different class loader.
- ▻ Store the probabilistic specification in a way that it offers fast access to nodes and edges.

The first IR has been achieved by using the JAVASSIST-Framework. This bytecode-manipulation framework is capable of manipulating classes from precompiled *.class*-Files. By inserting code to the beginning of every function that has to be observed, the function calls another function telling it that it is about to be executed before running the actual method body.
For the second IR, Sherlock offers a way of communication for the two functions with different class loaders by letting them both access the same file. That way, function $a$ that uses class loader $A$ can write to the file, that an other function $b$ from class loader $B \neq A$ then reads.
To achieve the third IR and allow fast access to the data stored in the probabilistic specification, Sherlock uses an offset array approach to store both edges and states efficiently – accessing any information stored in a java array has a time efficiency of $\mathcal{O}(1)$.

Chapter 5

# Evaluation

This chapter holds the evaluation of the Sherlock approach described in chapter 4. Section 5.1 holds the description and the results of the the comparison performed for the probabilities, while section 5.2 holds the description and the results of the hypothesis-test that has been performed for the micro-validation. Section 5.3 holds the description for and the results of the macro-validation. Section 5.4 show the threads to validity.

## 5.1 Comparison

This section contains the comparison of probabilities performed for the Sherlock-Approach. Section 5.1.1 describes how the comparison was done and section 5.1.2 presends the results.

### 5.1.1 Description

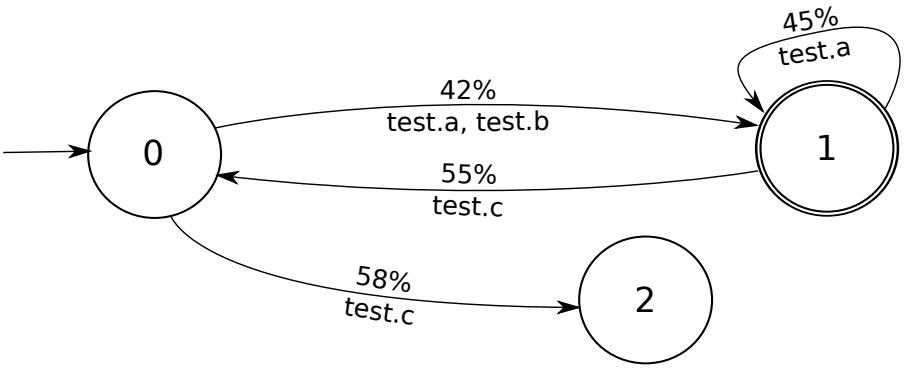


**Figure 5.1.** An example graph representing a probabilistic automaton. Every transition is marked with a per centage value, indicating how likely this transition is to be taken once the automaton has reached the source state, alongside with an input indicating what functions can be called if this tranisition is taken.

In order to validate the functionality of the probabilistic specification miner, the following evaluation procedure has been done:

First, a probabilistic specification was created. The classes/api the probabilistic specification specifies only have methods without any specific body. The unit-tests used to create the probabilistic specification consist of random method calls. That way, it is ensured that the methods are not called in a specific order. Once the probabilistic specification has been created using any of the three probabilities described in section 4.1.3, the created automaton is used to simulate the underlying program while leaving traces for any simulated method call. Those traces then are used to create new probabilistic specifications for all of the availlable probabilities. If the program works like it should, the resulting specification should look similar to the one that was created earlier.
As described in section 3.5, the QUARK-Framework offers three different metrics: $X(Y)$, $Y(X)$ and $PS(X,Y)$ [Lo and Khoo, 2006]. $X(Y)$ stands for the per centage of sets that would lead to an accepting state in autoomaton $X$, that also lead to an accepting state in $Y$, $Y(X)$ is the same metrics vice verca. $PS(X,Y)$ offers a comparison of the transition probabilities of two probabilistic automata $X$ and $Y$.

To run the evaluation from the program, there are two different tests that have to be run separately. The first one is to create and export the sample probabilistic specification from the random tests described above. The results then are exported as a *.csv*-file in the temp folder. Due to restrictions of the Javassist–Framework, the second part has to be executed on its own, in a new java environment and can not be executed on the same execution as the other test class. The second execution first reads the probabilistic specification that has been exported in the first run and then uses the Javassist–Framework to dynamically create a code that uses the probabilistic specification to simulate an execution under equal conditions using random variables to simulate the transition probabilities. The class itself is made entirely using Javassist, thus it is an executable class instead of a Unit-test. Instead of executing the functions directly, the function only leaves traces using the FileTranslator written for the Javassist–Framework. Those traces then are treated as if they were from an execution with the FileTranslator, thus Sherlock creates a probabilistic automaton with the desired probability using the temp-file.
Figure 5.1 shows an example for a probabilistic automaton. There, the transition that goes from state 1 to state 2 has two input functions that can be taken. In the testing environment, they are assumed to be equally distributed.
The resulting run function for the probabilistic specification from figure 5.1, that simulates one transition and leaves the trace as return value, is shown in listing 5.1, while listing 5.2 shows the other functions of the class.
The code works as follows: the program holds the current state locally for the object in our *state* integer. The *run()*-Method now simulates one transition. Therefor, the program first gets two double-valued random variables. The first random variable is used to decide the next state. To do that, it first figures out what state the automaton currently is in and then compares the random variable with the probabilities of the outgoing transitions to take one. Note that the program compares against the cumulated outgoing frequencies of the other

states, so the last comparison (if the state *has* outgoing states) always holds a 1 – otherwise the automaton might end up not doing anything at all.That way, eventhough Javas random function gives us an equally distributed value between 0.0 and 1.0, the simulator can still prefer those transitions that have a higher probability. After finding the next transition, the simulator updates the state for the next round. A transition can have more than one function to take us to the desired state. As mentioned before, it is assumed that they are equally distributed. The second random value to decides which of the functions from the current transition is *taken*. Since this is just a simulation and not a real execution, the function isn't *executed* – it merely returns their names as String. An interpreter then can take those returned values and treat them as if they were actually coming from the tracer.

The interpreter class then creates an object from the test class using the constructor from line 29 of listing 5.2. It then defines two variables, *rounds* and *depth*, to configure how exactly the test is to be simulated. Given two parameters of type integer, *rounds* and *depth*, the code from listing 5.3 creates the traces.

The integer-value *rounds* specifies how many rounds are simulated. Every round starts at the initial state and ends either in an accepting state or in a *trash* state, from where the automaton can't reach any accepting state anymore. If the automaton ends in an accepting state, the round is treated as if they were actual traces from a sample execution. If it ends in a *trash*-state, the traces from the last round are dumped.
In every round, the *depth*-parameter is taken as the maximum amount of transitions for the minimal depth. This means, the simulator simulates at least $i$ rounds for any $i \in [1, depth]$. After the $i$ rounds, the simulation is continued until the automaton reaches either an accepting- or a trash-state. That way, the interpreter can also collect paths $\Pi = p_1, p_2, p_3, \ldots, p_n$ with $p_j$ being an accepting state for $j < n$, instead of stopping once the automaton has reached an accepting state.
A simulation round now works as follows: the interpreter holds a traceList, where it collects the Strings that represend the function names are collected. The program then runs $i$ rounds of the simulation, where $i$ is bounded by the *depth* variable, and then continues until the automaton is in an accepting or failing state. If the automaton is in an accepting state, the interpreter writes all the elements stored in the TraceList to the temp file, as if they were from a real execution that uses the FileTranslator for Javassist. After that, it removes all the contents of the traceList and reset the simulation, so the next round starts at the initial state.
The function *ProbabilisticSpecificationCreator.createProSpecWithTmpFile(...)* then works as if the traces were collected from actual Unit-Tests and creates a probabilistic specification out of them.
The resulting specifications then are to be evaluated as described above.

```
1  public String run(){
2    double r = rand.nextDouble();
3    double r2 = rand.nextDouble();
4    if(state==0){
5      if(r<=0.42){
6        state=1;
7        if(r2 <= 1d/2d){
8          return "test.a";
9        }else if(r2 <= 2d/2d){
10         return "test.b";
11       }
12     }else if(r<=1.0){
13       state=2;
14       if(r2 <= 1d/1d){
15         return "test.c";
16       }
17     }
18   }else if(state==1){
19     if(r<=0.45){
20       state=1;
21       if(r2 <= 1d/1d){
22         return "test.a";
23       }
24     }else if(r<=1.0){
25       state=0;
26       if(r2 <= 1d/1d){
27         return "test.c";
28       }
29     }
30   }else if(state==2){
31     return "";
32   }
33   state=3;
34   return "error";
35 }
```

**Listing 5.1.** Java code for the test classes run code that simulates the next round of the automaton from fig. 5.1.

### 5.1.2 Results

Figure 5.2 shows a probabilistic specification that was mined from a testing program. The program had the functions *a* and *b*, that were called in a different order. The mined

```
1  private int state;
2  private final java.util.Random rand;
3
4  private void write(java.util.List list){
5      for (int i = 0; i < list.size(); i++){
6          de.learnlib.algorithms.prospecmi.trace.TempWriter.instance.write(
7            (java.lang.String)list.get(i));
8        }
9
10       de.learnlib.algorithms.prospecmi.trace.TempWriter.instance.testDone();
11 }
12
13 public void newRound(){
14   this.state=0;
15 }
16
17 public void close(){
18   de.learnlib.algorithms.prospecmi.trace.TempWriter.instance.close();
19 }
20
21 public boolean done(){
22   return state == 1 || false;
23 }
24
25 public boolean failed(){
26   return state == 2 || false;
27 }
28
29 public Test() {
30   state = 0;
31   rand = new java.util.Random();
32 }
```

**Listing 5.2.** Other needed functions of the testing class, alongside with the run functin from 5.1.

specification is good for testing purposes, as there are many transitions with a very low probability. Note that the transition from state 5 to state 1 only has a probability of 0.01%, the same value that the transition from state 1 to state 5 has.
State 4 has no outgoing edges to any states other than itself. Given that the testing framework only uses traces that end in an accepting state, none of them ever visited state 4 in an accepting run. This means, that all the specifications are created without knowing about that state. Hence one can not expect to find it in any of the specifications that were

```
1  Test test = new Test();
2  java.util.List traceList = new java.util.ArrayList();
3  for (int i = 0; i < rounds; i++) {
4    for (int j = 0; j < depth; j++) {
5      for (int k = j; k >= 0; k--) {
6        traceList.add(test.run());
7      }
8      while (!test.done() && !test.failed()) {
9        traceList.add(test.run());
10     }
11     if(test.done()) {
12       test.write(traceList);
13     }
14     traceList.clear();
15     test.newRound();
16   }
17 }
18 test.close();
19 ProbabilisticSpecification ps = ProbabilisticSpecificationCreator
20   .createProSpecWithTmpFile(0.01, 0.99, 0.002, 100);
```

**Listing 5.3.** Simulation of the interpreter for given integer-type variables *rounds* and *depth*

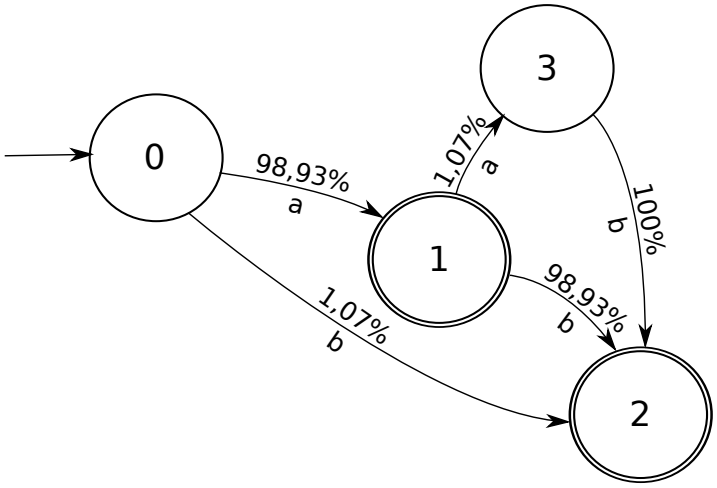


**Figure 5.2.** Mined probabilistic specification for a program with the functions *a* and *b* that other functions try to copy using the interpreter described in section 5.1.1.

created using the simulation. This also means, that the states from the *new* specifications will have other numbers, what makes a comparison a little harder.

### Lightweight Adaptive Filter

**Table 5.1.** Table containing the calculated probabilities for the recreated probabilistic specification from figure 5.2 using the lightweight adptive filter described by Filieri et al. [2014]. Specification 0 is the original one. The first column shows the index of the specification. The second one, $\delta$, holds the depth parameter and the third one, $\rho$, holds the assigned parameter for the rounds. Then the actual values the mined specification had on its edges are displayed. A – symbol indicates that the edge did not exist in the specification, if an edge has 0% then it exists, but has a probability so close to 0, that by rounding it after 2 digits it becomes 0. It is nessecary to mention that on test 7, the transition from state 0 to state 1 was labeled with both *a* and *b*, while on the other specifications – including the original one –, it was only labeled with an *a*.

| | $\delta$ | $\rho$ | $0 \to 1$ | $0 \to 2$ | $1 \to 2$ | $1 \to 3$ | $3 \to 2$ | $X(Y)$ | $Y(X)$ | $PS(X,Y)$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | – | 98,93% | 1,07% | 98,93% | 1,07% | 100% | – | – | – |
| 1 | 1 | 5 | 100% | – | – | – | – | 25% | 100% | 67,37% |
| 2 | 4 | 5 | 98,81% | 1,19% | 100% | – | – | 75% | 100% | 99,99% |
| 3 | 16 | 5 | 100% | 0% | 100% | 0% | 100% | 100% | 100% | 99,97% |
| 4 | 1 | 20 | 100% | – | – | – | – | 25% | 100% | 67,37% |
| 5 | 4 | 20 | 100% | 0% | 100% | 0% | 100% | 100% | 100% | 99,98% |
| 6 | 16 | 20 | 100% | 0% | 100% | 0% | 100% | 100% | 100% | 99,98% |
| 7 | 1 | 400 | 100% | – | – | – | – | 50% | 100% | 50,80% |
| 8 | 4 | 400 | 100% | 0% | 100% | 0% | 100% | 100% | 100% | 99,98% |
| 9 | 16 | 400 | 100% | 0% | 100% | 0% | 100% | 100% | 100% | 99,98% |

Table 5.1 shows the probabilities for the recreated probabilistic specification that was created using the Lightweight Adaptive Filter created by Filieri et al. [2014]. The original specification had 5 states and 6 edges. The variables for recreating it were thus chosen as follows: the depth is chosen in correlation to the amount of states. For $n$ states, the depth was chosen as $n^i$ for $i \in \{0, 1, 2\}$. The amount of rounds was chosen in correlation to the amount of edges and states. In the example from figure 5.2, there were 5 edges, hence the smallest rounds parameter was set to 5. For $m$ edges and $n$ states, the remaining parameters are a product of $(m \cdot n)^i$ for $i \in \{1, 2\}$.
What can be seen from the results is that, when the depth is set to 1, the automaton never visits any states whose path is going through an accepting node and then takes a transition that leads them to a non-accepting one. That is, because the testing algorithm just takes transitions until it reaches an accepting state and when it found an accepting state, it stops. Thus, the only paths taken there are *a* and *b*. The algorithm by Angulin [1987] then merges *a* and *b* together. Hence, many transitions and states are missing.
Tests with other assignments of *depth* have all the transitions, eventhough the probability of them doesn't match the one of the original specification. Assuming that Javas random

variable really offers equally distributed random values, the obvious conclusion here is that the used probabilistic function prefers high probabilities and makes them even higher, while low probabilities from the original specification – like the transition from state 1 to state 4 – are even closer to 0. That is, because the described algorithm by Filieri et al. [2014] treats changes that occur not so often as noise and due to the adaptive nature of both $e$ and $e_{thr}$, those changes don't get to much weight on the estimation.
In most cases, the transition from state 0 to state 2 and the transition from state 1 to state 3 have a per centage of 0%. This is mainly due to the fact that in this paper, I round after two digits – the actual values the filter calculated were at a size of $1^{-120}$.
Watching at the value of *PS(x, Y)*, one can see that, whenever the depth $\delta$ is not 1, the probability simmilarity is above 99%. The value of $Y(X)$ seems to be constantly 100%, meaning that Sherlock doesn't mistakingly *create* new accepting words that wouldn't be accepted in the original specification.

**Ratio/Frequency**

**Table 5.2.** Table containing the calculated probabilities for the recreated probabilistic specification from figure 5.2 using the ratio as probability. Specification 0 is the original one. The first column shows the index of the specification. The second one, $\delta$, holds the depth parameter and the third one, $\rho$, holds the assigned parameter for the rounds. Then the actual values the mined specification had on its edges are displayed. A – symbol indicates that the edge did not exist in the specification, if an edge has 0% then it exists, but has a probability so close to 0, that by rounding it after 2 digits it becomes 0. It is nessecary to mention that on Test 7, the transition from state 0 to state 1 was labeled with both *a* and *b*, while on the other specifications – including the original one –, it was only labeled with an *a*.

| | $\delta$ | $\rho$ | $0 \to 1$ | $0 \to 2$ | $1 \to 2$ | $1 \to 3$ | $3 \to 2$ | $X(Y)$ | $Y(X)$ | $PS(X,Y)$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | – | 98,93% | 1,07% | 98,93% | 1,07% | 100% | – | – | – |
| 1 | 1 | 5 | 100% | – | – | – | – | 25% | 100% | 67,37% |
| 2 | 4 | 5 | 95% | 5% | 100% | – | – | 75% | 100% | 99,89% |
| 3 | 16 | 5 | 97,50% | 2,5% | 93,15% | 6,85% | 100% | 100% | 100% | 99,77% |
| 4 | 1 | 25 | 100% | – | – | – | – | 25% | 100% | 67,37% |
| 5 | 4 | 25 | 100% | – | 100% | – | – | 50% | 100% | 99,97% |
| 6 | 16 | 25 | 98,44% | 1,56% | 98,99% | 1,01% | 100% | 100% | 100% | 99,99% |
| 7 | 1 | 400 | 100% | – | – | – | – | 50% | 100% | 50,80% |
| 8 | 4 | 400 | 99,00% | 1,00% | 99,16% | 0,84% | 100% | 100% | 100% | 99,99% |
| 9 | 16 | 400 | 99,08% | 0,92% | 98,86% | 1,13% | 100% | 100% | 100% | 99,99% |

In table 5.2, the recalculated probabilities from the probabilistic automaton displayed in figure 5.2 with the usage of the ratio as a probabilistic factor can be seen. Simmilar to the results from using the Lightweight Adaptive Filter, the parameters $\delta$ for the *depth* and $\rho$ for the *rounds* have to have a certain value for the specification to cover all edges there are. The values for the rounds and depth were chosen similar to the ones described for the

lightweight adaptive filter.
By using the ratio to get a probabilistic value, the received values are much closer to the original specification than they were after using the LAF. The reason for that is, that the ratio treats every received measurement as equal instead of binding a weight to every round like the LAF did. Hence, edges that had a low probability on our original specification now don't seem to be noise that the filter is trying to attenuate by assigning a higher weight to our past measurements. On every round, the trace increases the occurence counter for both the transition and the state.
Eventhough the absolute difference for the transition probabilities between the *original* specification and the recreated ones are much less, the probability similarity doesn't really differ that much from the one that was calculated for the LAF-Filter.
What can also be seen from the results is, that for this filter, the parameters for depth and rounds have to have a certain minimum, but the accuracy doesn't nessecarly increase with higher values, nor does it decrease.

**Keep Alive Models with Implementations**

**Table 5.3.** Table containing the calculated probabilities for the recreated probabilistic specification from figure 5.2 using the KAMI-approach to calculate the probability. Specification 0 is the original one. The first column shows the index of the specification. The second one, $\delta$, holds the depth parameter and the third one, $\rho$, holds the assigned parameter for the rounds. Then the actual values the mined specification had on its edges are displayed. A – symbol indicates that the edge did not exist in the specification, if an edge has 0% then it exists, but has a probability so close to 0, that by rounding it after 2 digits it becomes 0. It is nessecary to mention that on Test 7, the transition from state 0 to state 1 was labeled with both *a* and *b*, while on the other specifications – including the original one –, it was only labeled with an *a*.

| | $\delta$ | $\rho$ | $0 \to 1$ | $0 \to 2$ | $1 \to 2$ | $1 \to 3$ | $3 \to 2$ | $X(Y)$ | $Y(X)$ | $PS(X,Y)$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | – | – | 98,93% | 1,07% | 98,93% | 1,07% | 100% | – | – | – |
| 1 | 1 | 5 | 100% | – | – | – | – | 25% | 100% | 67,37% |
| 2 | 4 | 5 | 100% | – | 100% | – | – | 50% | 100% | 99,98% |
| 3 | 16 | 5 | 100% | – | 98,65% | 1,35% | 100% | 75% | 100% | 99,99% |
| 4 | 1 | 20 | 100% | – | – | – | – | 25% | 100% | 67,37% |
| 5 | 4 | 20 | 98,73% | 1,27% | 100% | – | – | 75% | 100% | 99,99% |
| 6 | 16 | 20 | 100% | – | 99,33% | 0,67% | 100% | 75% | 100% | 99,99% |
| 7 | 1 | 400 | 100% | – | – | – | – | 50% | 100% | 50,80% |
| 8 | 4 | 400 | 98,62% | 1,38% | 98,73% | 1,27% | 100% | 100% | 100% | 99,99% |
| 9 | 16 | 400 | 98,95% | 1,05% | 98,85% | 1,15% | 100% | 100% | 100% | 99,99% |

Table 5.3 shows the calculated probabilities for the KAMI-Specification. Once again, $\delta$ holds the parameter for the depth, while $\rho$ holds the amount of rounds that were used to mine the original automaton.
Watching only at those specifications with $X(Y) = 100\%$ and $Y(X) = 100\%$, meaning that

$L(X) = L(Y)$, it is clear that the estimations were very accurate and a lot better than the ones calculated from the LAF-Filter.
The calculated values for the probability similarity, $PS(X, Y)$, are not so different from the Ratio. In fact, they don't even differ so much from the values the LAF-Filter gave us.

## 5.2 Micro-Validation

This section contains the hypothesis test as described in section 3.5. Section 5.2.1 contains a description of the hypothesis test. The results are shown in section 5.2.2.

### 5.2.1 Description

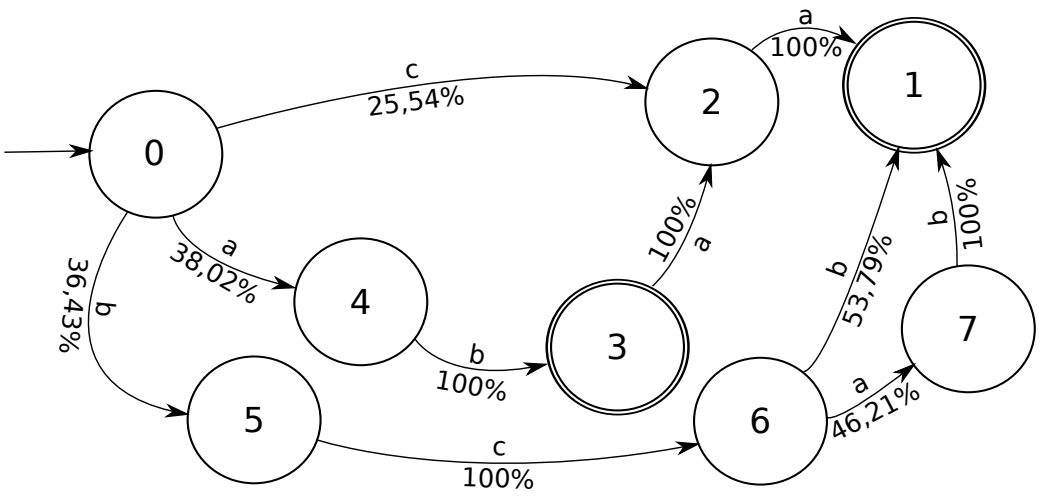


**Figure 5.3.** Template of the probabilistic specification that is recreated by the three probabilities for the hypothesis test.

To perform the Hypothesis-Test, for every probability, 1.000 probabilistic specifications are mined. As template, the specification shown in figure 5.3 is used the same way figure 5.2 was used for the analysis part. For every of the 3.000 specifications, a metric is calculated to measure the *accuracy* of the specification. As described in section 3.5, the used metrics for the hypothesis test that validates a probabilistic $x$ is

$$M(x) = X(Y) \cdot Y(X) \cdot PS(X, Y)$$

The results from the analysis-based Micro-Validation, presented in section 5.1.2, showed that when using depth $\delta = 16$ and rounds $\rho = 400$ – adjusted for the 8 states and 10 edges

to $\delta = 64$ and $\rho = 6400$ – both the values of $X(Y)$ and $Y(X)$ are 100% – 1.0 as a decimal. Thus, using the defined values for $\delta$ and $\rho$, the calculated metrics is

$$M(x) = PS(X, Y)$$

Using this metrics, the hypothesis-test tests against the hypothesisses described in section 3.5.

### 5.2.2 Results

A two-sided hypothesis-test has been performed the following way:
The significance-level $\alpha$ has been set to $\alpha = 0.05$. Based on the assumption of an underlying *Gaussian* distribution – what has only been verified by analysing the values –, this results in $z_\alpha = 1.96$. The expected value is $\mu = n \cdot p$, with $n = 1.000$ and $p = \overline{p}$, the arithmetic mean of the samples. The standard deviation $\sigma$ has been estimated using the algorithm by Knuth [2005] that estimates the variance $\sigma^2$. That same algorithm was also used in the Lightweight Adaptive Filter to calculate the thresholding parameter [Filieri et al., 2014]. A Java-Version of this algorithm is printed in listing 5.4. Using it, the standard deviation can be calculated with $sigma = Math.sqrt(variance(data))$.
The hypothesis-test now takes the 1.000 calculated metrics for every probability. For two

```
1   private double variance(final double[] data) {
2       double n = 0;
3       double mean = 0d;
4       double m2 = 0;
5       double delta;
6
7       for (final double d : data) {
8           n++;
9           delta = d - mean;
10          mean += delta / n;
11          m2 += delta * (d - mean);
12      }
13
14      // Avoid dividing with 0.
15      if (n > 1) {
16          return m2 / (n - 1);
17      } else {
18          return 0;
19      }
20  }
```

**Listing 5.4.** Java-Version of Knuth's algorithm to estimate the variance of a given set of data.

**Table 5.4.** Results of the hypothesis-test.

| **Hypothesis** | $\mu$ | $\sigma$ | $z_{low}$ | $z_{high}$ | $z$ | **Status** |
|---|---|---|---|---|---|---|
| $H_0^{K,L}$ | 999,997167 | $2,625 \cdot 10^{-6}$ | 999,997162 | 999,997172 | 426,605863 | **rejected** |
| $H_0^{L,R}$ | 426,605863 | 0,163769 | 426,284879 | 426,926850 | 999,997167 | **rejected** |
| $H_0^{R,K}$ | 999,997167 | $2,624 \cdot 10^{-6}$ | 999,997162 | 999,997172 | 999,997167 | **failed to rej** |

probabilities $x$ and $y$, the hypothesis-test calculates the expected value $\mu$ and expects a value $z$ with $(\mu - z_\alpha \cdot \sigma) \leqslant z \leqslant (\mu + z_\alpha \cdot \sigma)$. The value of $z$ is calculated by taking the sum of the metrics taken from all 1.000 samples for the probabilistic. If the calculated value is outside the calculated border, the null-hypothesis is rejected, otherwise it is accepted.
The results of the hypothesis-test are shown in table 5.4.
The results show that all null-hypothesisses that contain the Lightweight Adaptive Filter have been rejected. This points out the fact that the LAF calculates the probabilities in a different way than KAMI and RATIO do. This comes as no surprise, given that LAF treats every new sample that occurs the first time as noise, which it tries to attenuate by assigning a very low weight to it compared to earlier measurements. The RATIO on the other hand treats every new sample equal, while KAMI uses a bayessian technique to estimate those transition probabilities that would leave the measured traces, meaning that they also get treated about equally.
The hypothesis-test also showed that KAMI and RATIO had an estimated standard deviation $\sigma$ of $2,62 \cdot 10^{-6}$, meaning that the calculated values all were very much equal, while the LAF had 0,16. This makes out a range of 16% for the calculated LAF probabilities. Also, the mean $\mu$ for the LAF is 426,61. Given the fact that there were 1.000 samples, the average per value has been 0,43 or 43% accuracy, while both LAF and RATIO were very close to 100%.

## 5.3 Macro-Validation

This section presends the Macro-Validation performed to evaluate the Sherlock-Approach. Section 5.3.1 describes the appraoch. The results are presented in section 5.3.2.

### 5.3.1 Description

The Sherlock Approach has been validated with a real example the following way:
First, real projects have been added to the validation project using Maven. Therefor, especially the tests have been of interest. Thus, listing 5.5 imports especially the compiled class files of the tests for the Apache Commons Math3 project. Those tests are then used to calculate a Probabilistic Specification using the Specification Miner described in this thesis

and the testing classes from the imported dependencies.
In this case, the observed API is *Java.util.Random*. Given that it is an internal API, there

```
1  <dependency>
2      <groupId>org.apache.commons</groupId>
3      <artifactId>commons-math3</artifactId>
4      <version>3.3</version>
5      <type>test-jar</type>
6      <scope>compile</scope>
7  </dependency>
```

**Listing 5.5.** Maven-Code that is used to import the compiled Unit-Test files for Apache Math3.

were some extra-steps that had to be done first:
Java first loads the files from its generic JVM [Chiba, 2000–2012]. So in order to avoid that, the whole Java Virtual Machine (JVM) had to be unpacked a new folder. For the Macro-Validation, a 64-Bit Linux machine has been used. The JVM of the used OpenJDK-7 version was stored in */usr/lib/jvm/java-7-openjdk-amd64*. Every jar file that was stored in there was unpacked to a special folder. In there, the class file of *Java.util.Random* has been overwritten using the code from Sherlocks FileTranslator.
For the next step, the content of the *junit.Tests* File has been copied to an own project with a Main function that reads in the names of the test files we have as a dependency and then executes them as JUnit-Tests. The project (that has to have all the Maven-Dependencies of the test classes) then has to be exported as a *.jar*-File. This file then is unpacked in exactly the same folder the JVM was unpacked to earlier. The file of the project that has the main method, let's say *org.test.Main*, then lies in it's own subfolder *org/test/Main.class*. The file then has to be executed using **only** the files stored in the current folder, using the command *java -Xclasspath/p:. org/test/Main* (in case the temp-folder is too small for the trace files, it is recommended to also use *-Djava.io.tmpdir=.*). This project then runs the Unit-Tests and leaves a trace whenever any function calls a method from the *Random*-Class.
A second project now calls *ProbabilisticSpecificationCreator.createKAMIProSpecWithTmpFile()*. The method then uses the proSpec and proSpecMap from the temp folder to calculate a probabilistic specification. After that, the resulting probabilistic specification is exported. The same procedure happens for *RATIO* and *LAF*.
Using a different amount of tests each run for the same probabilistic function, the specification miner exports two probabilistic specifications for each probability: One with traces from all tests and one with fewer tests. Using the same metrics that have also been used in the analysis-based Micro-Validation from section 5.1, the two specifications then get compared. Note that $X(Y)$ and $Y(X)$ won't change from the usage of an other probability, thus they are only mentioned once.

**Table 5.5.** Result of the execution of the Macro-Validation

| **Probability** | $X(Y)$ | $Y(X)$ | $PS(X,Y)$ |
|---|---|---|---|
| LAF | 95,00% | 90,48% | 99,99% |
| KAMI | 95,00% | 90,48% | 99,67% |
| RATIO | 95,00% | 90,48% | 99,66% |

**Table 5.6.** Results of the Analysis from the Macro-Validation.

| Metrics | Difference |
|---|---|
| $M(LAF, RATIO)$ | 0.33 |
| $M(LAF, KAMI)$ | 0.32 |
| $M(KAMI, RATIO)$ | 0.01 |

### 5.3.2 Results

Let $X(z)$ and $Y(z)$ be the two specifications mined for the Macro-Validation from the usage of the *Java.util.Random*-Package and let $z$ be the used probability. $X(z)$ is the smaller probabilistic specification. It was mined using 1.783 test-classes that left 104.746.384 traces. The bigger probabilistic specification, $Y(z)$, was created using 1.877 test-classes, leaving 104.762.624 traces.
Specification $X(Y)$ has 67 states and 116 edges. $Y(X)$ has 75 states and 123 edges.
Table 5.5 shows the result of the Macro-Validation. Specification $X$ was created using 1.783 test-classes, while specification $Y$ was created using 1.877 test-classes. Given that specification $Y$ contains all the test classes that were in specification $X$ before, specification $Y$ gives us 94 test files to validate the specification – a little more than 5%.
Table 5.5 shows that the least changes in the transition-probabilities for those paths that were in both $X$ and $Y$ were made by using the Lightweight Adaptive Filter. This is possibly due to the fact that changes from what was mined before are treated as noise there, while giving those transitions with a previously high probability a bigger weight.
For KAMI and RATIO, the transition probabilities also hardly changed at all.
The calculated value for $X(Y)$ shows, that for some reason, specification $Y$ doesn't seem to accept every word that specification $X$ accepts. Also, specification $Y$ accepts words that specification $X$ does not, which is easily explainable by the fact that specification $Y$ was mined with more testing-methods that specification $X$.
The results of the analysis described in section 3.5 can be seen at table 5.6. The table shows that all the difference of the probability similarity between the old and the new specification is not so big when comparing the RATIO and KAMI, while anything containing the LAF has – due to the fact that the probabilities are almost the same – the biggest difference to the others.

## 5.4 Threads to validity

The experiments from sections 5.1 and 5.2 were conducted using the example specifications – the ones shown in figure 5.2 and 5.3. The results displayed in tables 5.1, 5.2, 5.3 and 5.4 may depend on the used input automaton – results may change when using other specifications as input, maybe even the value of $X(Y)$.
The validation performed in section 5.3 was conducted using several different Unit-Tests from different projects. However, most of the test classes came from Apache Commons Projects (like Apache Commons Math3 or Apache Commons Logging). This may result in projects using the same API in the same way, eventhough there would be other possibilities to use the API correctly. Thus, the result mainly depends on both the API to mine and the Unit-Tests. Changing the preconditions surely will alter the result.

Chapter 6

# Discussion

This chapter offers a brief discussion related to the qualitative research question from chapter 3.

## 6.1 Qualitative Research Question 1

The question asked in section 3.4 was how a probabilistic specification can be used to improve software projects in the future. As stated by Weimer and Necula [2005], a lot of bugs from software projects come from false usage of APIs. The reason that people use apis in a false way is that they often are poorly documented. Thus, a probabilistic specification infered from other – in case bug-free – projects that use the same API can surely help for the programmers to get to know the API-usage a little more.
Current Software Projects rarely come from only one programmer – usually, there are several teams. Given a scenario, where several teams have to use the same API at different states of the development, it is more likely that – especially when the part where a team uses the API isn't very long – the programmer makes an error. In that case, using the own project to infer a probabilistic specification may also give hints about where the API has been used falsely. It is a lot easier for the team that successfully used the API to validate an automaton than to read in on the whole source code, especially when we consider that a false usage of the API would lead to a low transition probability if other parts of the same project used the API correctly.
Raffelt et al. [2009] also stated that most projects suffer from a bad documentation because of last-minute changes, where the deadline doesn't leave to much time for a complete specification. A common result is that, once the project team decides to update the documentation, people probably don't remember all of the changes made since the last specification was written. In that case, there are only a few things a programmer can do to track the changes their project has made since that, like reading the commit messages of their git- or svn repository. A probabilistic specification could help in that case: Given that the project was already used as an API in other projects, the programmers could create a probabilistic specification for both the old version and the new version of the API and compare them. That way, they'd see the new functions that have been added.

Chapter 7

# Conclusions and Future Work

This chapter introduces the conclusions for this thesis and gives thoughts about future work.

## 7.1 Conclusions

In the context of this thesis, a probabilistic specification miner was written and added to the LearnLib-Framework. A probabilistic specification miner does not only return a *deterministic* automaton with paths showing how a certain API has been used by the projects the specification was mined from, it also points out how likely any transition $s \rightarrow t$ is, given that the automaton is currently in state $s$.
The probabilistic specification miner has been described as the Sherlock-Approach and implemented in Java. The implementation first uses the Javassist-Framework [Chiba, 2000] to manipulate the class files that have to be observed in a way, that they leave execution-traces. Those traces then are connected to a deterministic automaton. On this automaton, the $L^*$-Algorithm that was described by Angulin [1987] and implemented in the LearnLib-Framework is applied to merge states wherever it is possible. The resulting automaton still is deterministic. To make it a probabilistic one, Sherlock offers three different methods to infer the transition-probabilities: the Lightweight Adaptive Filter (LAF) described by Filieri et al. [2014], the Keep Alive Models with Implementation Approach as described by Calinescu et al. [2011] and the ratio of every state.
The implemented specification miner has been validated in two different ways.
The first method is called Micro-Validation. To perform this kind of a validation, a predefined probabilistic specification has been given to the specification miner as input. An interpreter then simulated the execution of the underlying program and traversed the specification. Whenever a transition has been taken, the simulator returned the function name the simulated program just took. Those returned values then have been taken as traces for the specification miner to infer one probabilistic specification for every of the three probabilities. A comparison of the original specification and the rebuilt one has been done by using the metrics from the QUARK-Framework described by Lo and Khoo [2006]. The probability simmilarity showed, that for the used example, all three probability types gave a good representation of the original transition probabilities.
The second method is called Macro-Validation. The target of a Macro-Validation is it to

mine a probabilistic specification for any API using a set real free and open-source software. The specification then gets validated by other free and open-source software that has not been used to create the specification in the first place. Applying this technique showed, that specifications can get big very fast. The metrics from the QUARK-Framework also revealed that the Lightweight Adaptive Filter had the least changes in probability for the validated software. KAMI and the Ratio both had a very high probability simmilarity as well and were close together. A hypothesis-test showed that the changes KAMI and the Ratio made were somewhat simmilar, while the LAF returned an other metrics causing the two hypothesis that included the Lightweight Adaptive Filter to decline.

## 7.2 Future Work

The following section presends suggestions for future work.
The current version of the specification miner forces the user to download the free and open-source software himself using maven, jars with source code or by downloading the project directly from the git- mercurial- or svn-repository. A possible future work could automate the repo-mining in a way, that the specification miner automatically searches Github, Bitbucket, the Software Infrastructure Repoistory or any other repository for projects that use the API in their Unit-Tests.
To specifically get projects with the desired API, there are two different ways: the user could cataloguerize the project in a way that any used API is mentioned somewhere, or a code analysis tool could check for imported packages. That way, the specification miner could search directly for projects that use the API without having the programmer search for it manually.
The current probabilistic specification miner only returns (and exports) a probabilistic specification. As of now, it can't be used to validate the usage of certain APIs. A plugin (for example for the eclipse framework) could take a probabilistic specification and then check on a new project if the usage of the API is correct and offer suggestions for the next method to call, ordered by the probability taken from the probabilistic specification.

# Bibliography


[Ammons et al. 2002] G. Ammons, R. Bodík, and J. R. Laurus. Mining specifications. In *Proceedings of the 29th AVM SIGPLAN-SIGACT symposium on Principles of program languages*, pages 4–16. ACM, 2002.

[Angulin 1987] D. Angulin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Arcuri et al. 2012] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2011.121.

[Bierman and Feldman 1972] A. Bierman and J. Feldman. On the synthesis of finite state machines from samples to their behavior. *IEEE Trans. on Computers*, 21(6), 1972.

[Calinescu et al. 2011] R. Calinescu, K. Johnson, and Y. Rafiq. Using observation ageing to improve markovian model learning in qos engineering. In *ICPE'11 – Second Joint WOSP/SIPEW International Conference on Performance Engineering, Karlsruhe, Germany, 2011*, pages 505–510. ACM, 2011.

[Carrasco and Oncina 1994] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium Grammatical Inference and Applications (ICGI '94)*, volume 862. Springer, 1994.

[Chen and Roşu 2008] F. Chen and G. Roşu. Mining Parametric State-Based Specifications from Executions. Technical Report UIUCDCS-R-2008-3000, University of Illinois at Urbana-Champaign, 2008.

[Chiba 2000] S. Chiba. Load-time structural reflection in java. In *ECOOP 2000 – Object-Oriented Programming, LNCS 1850*, pages 313–336. Springer-Verlag, 2000.

[Chiba 2000–2012] S. Chiba. Javassist tutorial, 2000–2012.

[Cook and Wolf 1996] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7: 215–249, mar 1996.

[Dallmeier et al. 2006] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24. ACM, 2006.

[Engler et al. 2001] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code, 2001.

[Epifani et al. 2009] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaption. In *31st International Conference on Software Engineering, ICSE 2009, May 16–24*, pages 111–121. IEEE, May 2009.

[Filieri et al. 2012] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: Continuous assurance of non-functional requirements. *Formal Aspects of Computing*, 24:163–186, 2012.

[Filieri et al. 2014] A. Filieri, L. Grunske, and A. Leva. Lightweight adaptive filtering for efficient learning and updating of probabilistic models. *Automated Software Engineering*, sep 2014.

[Goues and Weimer 2009] C. L. Goues and W. Weimer. Specification mining with few false positives, 2009.

[Goues and Weimer 2011] C. L. Goues and W. Weimer. Measuring code quality to improve specification mining, 2011.

[Howar et al. 2014a] F. Howar, M. Isberner, M. Merten, and B. Steffen. Git repository of learnlib, 2014a. URL `https://github.com/LearnLib/learnlib`.

[Howar et al. 2014b] F. Howar, M. Isberner, M. Merten, and B. Steffen. Homepage of learnlib, 2014b. URL `http://learnlib.de/wordpress/`.

[Howar et al. 2014c] F. Howar, M. Isberner, M. Merten, and B. Steffen. Homepage of the closed-source version of learnlib, 2014c. URL `http://ls5-www.cs.tu-dortmund.de/projects/learnlib/about.php`.

[Isberner 2014] M. Isberner. Git repository of automatalib, 2014. URL `https://github.com/misberner/automatalib`.

[Knuth 2005] D. E. Knuth. The art of computer programming, 2005.

[Kremenek et al. 2006] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 161–176. USENIX Association, 2006.

[Livshits 2006] B. Livshits. Improoving software security with precise static and runtime analysis. Technical report, PhD Thesis, Stanford University, Stanford, California, 2006.

[Livshits et al. 2009] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 2009 ACM SIGPLAN Conferene on Programming Language Design and Implementation*, pages 75–86. ACM, 2009.

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

[Mao et al. 2011] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen. Learning probabilistic automata for model checking. In *Eighth International Conference on Quantitative Evaluation of Systems*, pages 111–120. IEEE Computer Society, 2011.

[Raffelt et al. 2009] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. Learnlib: a framework for extrapolating behavorial models. *International Journal on Software Tools for Technology Transfer (STT)*, 11:393–407, may 2009.

[Raman et al. 1998] A. Raman, P. Andreae, and J. Patrick. A beam search algorithm for pfsa inference. *Patern Analysis and Applications*, 1(2), 1998.

[Raman and Patrick 1997] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *ICML '97*, 1997.

[Sen et al. 2004] K. Sen, M. Viswanathan, and G. Agha. Learning continuous time markov chains from sample executions, 2004.

[Weimer and Necula 2005] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *In TACAS*, pages 461–476, 2005.

[Yedidia et al. 2003] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring Artificial Intelligence in the New Millennium*, pages 239–269, 2003.

**Decleration**

I declare that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

---

place, date, signature