# Extraction of Probabilistic Behaviour Models based on Contexts*

Lucio Mauro Duarte
Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre/RS – Brazil
lmduarte@inf.ufrgs.br

Paulo Henrique Mendes Maia
Group of Software Engineering and Distributed Systems – State University of Ceará (UECE)
Fortaleza/CE – Brazil
pauloh.maia@uece.br

Ana Carolina Sanchotene Silva
Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre/RS – Brazil
ana.sanchotene@inf.ufrgs.br

## ABSTRACT

Model extraction allows the automatic construction of behaviour models from an available implementation, which can be fed into existing analysis tools. Even though these models are usually analysed using qualitative properties, many interesting and relevant properties of current systems are related to quantitative aspects, such as the probability of reaching a certain state or how many times a certain task is expected to be executed. In this work, we extend an existing model extraction approach to include probabilistic information. The original approach creates Labelled Transition Systems (LTS) from Java code based on execution traces. The traces are processed by a tool that identifies contexts, which represent abstract states of the system, considering static and dynamic information, producing context traces. We use these context traces to calculate transition probabilities and generate models in the input language of a probabilistic model checker. We evaluate our approach in case studies and demonstrate that, by using context traces rather than simple traces, we produce more accurate models, thereby with probabilistic information closer to the real behaviour of programs, based on their observed traces. We also show how to build models of programs with single and multiple components.

## CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; *Software verification*; • **Theory of computation →** *Formal languages and automata theory*;

## KEYWORDS

Probabilistic models, Model extraction, Quantitative analysis.

**ACM Reference Format:**
Lucio Mauro Duarte, Paulo Henrique Mendes Maia, and Ana Carolina Sanchotene Silva. 2018. Extraction of Probabilistic Behaviour Models based on Contexts. In *MiSE'18: MiSE'18:IEEE/ACM 10th International Workshop on Modelling in Software Engineering , May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3193954.3193963

*Produces the permission block, and copyright information


*MiSE'18, May 27, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5735-7/18/05…$15.00
https://doi.org/10.1145/3193954.3193963

## 1 INTRODUCTION

Although models play a valuable part in software development, they are not always trivial to build and maintain. *Model extraction* [14] is the process of producing a model from an existing implementation and can be used to facilitate the creation and evolution of models. It eliminates the difficult and error-prone task of manually building a model and enables model-based analyses even when previous models do not exist. Moreover, changes can be easily introduced by recreating the model from an updated version of the program.

Most model extraction approaches have focused only on producing models for the analysis of qualitative properties, however, the development of new, more complex types of systems brings the need of evaluating properties that are not always connected to or can be represented solely as binary decisions, such as reliability, dependability, and robustness. Moreover, with the creation of new devices and the increase use of embedded and autonomous systems, questions related to performance have also become relevant. These aspects can be modelled using *probabilistic information* and analysed in available tools, such as PRISM [16], where models are described as derivations of Markov chains [4] and *quantitative properties* can be expressed using a probabilistic temporal logic. The effort to create probabilistic models is similar to that of building common behaviour models, with the extra task of annotating the model with the right probabilities in the right places. This leads to two important questions: (i) how to obtain probabilities closer to the real behaviour of the system; and (ii) how to correctly insert them in the model so as to allow an accurate analysis.

Considering question (i), it is usual to base the probabilities either on the knowledge/experience of the developer or on the results of monitoring the system, with the use of log files or operational profiles, in which case there must exist an executable version of the system. As for question (ii), the most common approach is to annotate transitions between states of the model, indicating the probability of a change of states.

We claim, however, that even when we use an accurate information source to compute probabilities for annotations, and have a model that captures precisely the non-probabilistic behaviour of the system, it may not be possible, in some circumstances, to annotate it with probabilities that lead to an accurate probabilistic model. These circumstances appear when the model to be annotated presents states that denote an environment choice that can be made more than one time, such as in a cycle, but whenever that choice occurs, the possible options (outbound actions) are taken with different probabilities.

To tackle this problem, one solution is to use the concept of *context*, which is a combination of control-flow information and values of program variables and represents an abstract state of the

system. Unlike other trace-based approaches (e.g., [20] and [24]), by using context information, the models are guaranteed to be correct (at a certain level of abstraction) with respect to the programs they represent [9]. Therefore, we can be sure the right probabilities go into the right places, which is not always the case with algorithms that combine traces based on inference methods, considering only the names and sequences of events (e.g., the algorithm implemented in the LoTuS tool [3] and the approach presented in [11]).

In this paper, we extend the original context-based approach presented in [9] to extract probabilistic models. The original process creates models, represented as Labelled Transition Systems (LTS) [15], based on execution traces containing context information extracted from Java code. The process is supported by the LTSE tool and generates models in the input format of the LTSA tool [18]. The proposed extension augments the model with the probability of occurrence of each transition obtained from the analysis of the same context traces used to build the model. This means we can easily associate each probability with a transition between contexts. Hence, our model not only contains only valid behaviours but also our probability annotation method is guided by the observed contexts. The generated probabilistic model can be exported to PRISM to verify quantitative properties of interest, such as if the probability of some observed events are under an acceptable threshold or the likelihood of something undesirable occurring.

Because of contexts, our models are more accurate and a better representation of the probabilistic behaviour described in the observed executions than other existing approaches. In addition, the extraction process is mainly automatic (apart from trace generation) and supports compositional construction, i.e., models of different components can be created individually and then composed using tool support. Furthermore, models can be iteratively built, with the gradual inclusion of more behaviours without affecting correctness and improving completeness. We evaluate our approach with 3 case studies and demonstrate we can produce more accurate models than using simple traces. We also show how our models can be used in PRISM to verify quantitative properties of programs of single or multiple components.

The paper is structured as follows: Section 2 presents some concepts related to model extraction and probabilistic models; Section 3 describes our approach in detail; Section 4 presents our experimental results and a discussion about them; Section 5 describes some comments about related work; and Section 6 contains our conclusions and possible future work.

# 2 BACKGROUND

## 2.1 Probabilistic Models

Probabilistic models add probability information to traditional behaviour models. They can be built and analysed in available tools (e.g., [16]). The usual formalism for probability analysis are *Markov chains* [4], which consist of a set of states connected by transitions labelled with *transition probabilities*. Changes of state are based on the probabilities associated to each transition, therefore providing a quantitative representation of choice. Markov chains can be analysed in tools such as PRISM, where model checking, cost/reward estimations, and simulations can be carried out. Other tools, such as LTSA-PCA [22] and LoTuS [3] allow the description of a *Probabilistic LTS (PLTS)*, which is an LTS that also contains a probability value for each transition. In the case of LTSA-PCA, it serves only as a modelling tool, where the model can be described using an extension of the FSP language [18]. Models are then exported to PRISM for analysis. In LoTuS, however, it is not only possible to create a PLTS using either a GUI or a probabilistic extension of FSP, but also run probabilistic analyses. Moreover, LoTuS can also export a PLTS to PRISM and allows plugins to be used to extend its features. It is important to mention that, in both cases, when the model is exported, the action names labelling transitions are lost, as Markov chains contain only transition probabilities.

As mentioned before, creating a probabilistic model involves constructing the model and assigning probabilities to its transitions. In order to generate accurate results, reliable estimates and measurements of probabilities are needed to annotate behaviour models [19]. Hence, not only should the probabilities be accurate in properly reflecting reality, but also the model being used. Usually, the probabilities assigned to transitions come either from the knowledge/experience of a developer or from the results of monitoring the system, with the use of log files or operational profiles. For instance, LoTuS includes an option to calculate probabilities from a set of traces: it counts the number of times a certain action $a$ ($t(a)$) appears in the traces and the number of times $a$ precedes an action $b$ ($t(ab)$); then the probability of having executed $a$ and then execute $b$ is $p_{ab} = t(ab)/t(a)$. This process is repeated for all actions and all their successor actions in the traces and the resulting probabilities are assigned to the corresponding transitions. The process is highly dependent on the traces and assumes that every action name represents the exactly same event, which is not necessarily true. For instance, if an action name represents a method call, this method call might happen in different parts of the code and in different circumstances.

## 2.2 Probabilistic model checking

One usual application for probabilistic models is *Probabilistic model checking*, which is a formal verification technique for the analysis of systems that exhibit stochastic behaviour. It involves the construction of a precise mathematical model of a real-life system to be analysed, the formal specification of one or more properties of the system, and the analysis of these properties based on an exhaustive exploration of the constructed model [17].

Properties to be analysed by probabilistic model checking are typically specified using temporal logics such as CSL [2] or PCTL [12], a probabilistic extension of the classical temporal logic CTL. These properties relate not just to the functional correctness of a system, as is the case with non-probabilistic verification techniques, but also to quantitative measures, such as performance and reliability, for example: "the probability that a message will be delivered is at least 0.75", or "the probability of an alarm failure is at most 0.01".

## 2.3 Model Extraction

Several model extraction techniques have been proposed in the literature [14] [6] [1] [20] [24]. They follow either a *static approach*, which creates the model directly from the source code of the system, by the analysis of its *control flow graph* (CFG), or a *dynamic*

*approach*, which works on traces of the system, generalising behaviours from the observed behaviours through some inference process. The approach extended in this work [9] follows a *hybrid idea*, where static and dynamic information are used together to build models. The goal is to use static information to guide the process of combining multiple traces to derive a single model, which includes only valid behaviour at a given level of abstraction. The advantage of this approach is that it bases the decision of connecting traces on a combination of control flow information and values of selected program variables, called *context*.

Contexts encapsulate a *control component*, which indicates the current execution point of the system, and a *data component*, representing the state of the system in terms of values of program variables. More formally, given a program *Prog*, a context $C = (bc, val(cp), v)$ represents the combination, at a certain point of the execution of *Prog*, of the current block of code *bc* being executed, the value $val(cp)$ of its control predicate *cp* (the condition associated to the block of code) and the current set of values $v$ of attributes of *Prog*. Therefore, model extraction based on contexts uses static information to identify sequences of actions, for example, that can be repeated due to the existence of a repetition statement, and also identify traces that share some actions, thus detecting alternative paths not present in the single traces. This characteristic allows not only the inference of additional valid behaviours, but also the possibility of a later inclusion of new trace information to an existing model, thereby supporting model evolution. For more details of this approach, please refer to [9].

## 3 EXTRACTION APPROACH

This section presents our approach for probabilistic model extraction. We use, as a running example, a simple text editor originally presented in [9]. Part of the source code is presented in Figure 1. This editor allows basic actions on a text document and uses two program variables to control whether a document is currently open (*isOpen*) and whether the document has been saved (*isSaved*).

An overview of the approach is presented in Figure 2. The input is a set of traces collected from the target program. These traces, obligatory containing *context information* (control-flow information and values of attributes), are processed by the LTSE tool, which converts them into *context traces*, which are traces that represent sequences of contexts and actions that happened in between them. Based on these context traces, the LTSE generates a model combining all the observed executions, creating a transition system, which is saved in a text format (MDL file). This text description contains information about identified contexts and transitions between contexts. Using this information, our tool (LTSEProb) computes transition probabilities and produces a *Discrete-Time Markov Chain (DTMC)*[4] in the input language of PRISM. This DTMC represents the probabilistic behaviour observed in the initial set of traces. Next, we present each step in more detail.

### 3.1 Trace Analysis

The first phase of the extraction process is the analysis of collected traces. Trace generation is carried out exactly as in the original approach: the source code is annotated following the rules described

```
public class Editor {
 private boolean isOpen;
 private boolean isSaved;

 public Editor () {
  isOpen=false;
  isSaved=true;
  int cmd=-1;
  String name=null;

  while(cmd!=4){
   cmd=readCmd();
   switch (cmd) {
    case 0:
     if(!isOpen)
      name=open();
     break;
    case 1:
     if(isOpen)
      edit(name);
     break;
    case 2:
     if(isOpen)
      print(name);
     break;
    case 3:
     if(!isSaved)
      save(name);
     break;
    case 4: exit();
    ...
   }}
  }
  ...
 void exit (String n) {
   if (!isSaved) {
    int opt=readCmd();
    if (opt==0)
     save(n);
    }
    if (isOpen)
     close(n);
  }
 }
```

**Figure 1: Source code of the running example.**

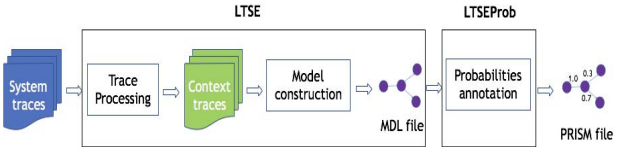


**Figure 2: Overview of the approach.**

in [7] and then executed to produce traces[1]. We do not apply a specific technique to generate traces, hence they can come from log files, monitoring the system, operational profiles, or as a result of executing a test suite. Since our focus is not on models designed for real-time analysis, the possible overhead caused by the annotation in the code is not a problem, as it does not affect the transition probabilities. Moreover, we do not need access to the source code ourselves, but only to the produced traces with the necessary context information.

Given a set of traces, the LTSE tool analyses them and produces a *context table (CT)* and a set of *context traces*. Each row of the CT stores information about an identified context, assigning a unique number to it, called *context ID (CID)*. Context traces are simplified representations of the original traces where information about contexts is translated to its corresponding CID and information about actions are mapped to action names. Therefore, a context trace is a sequence involving CIDs and action names. Two consecutive CIDs $c_1$ and $c_2$ indicate a transition from the context corresponding to $c_1$

[1]Note that the traces can come from any program in any language as long as they are presented in the expected format and contain the necessary information.

to the context corresponding to $c_2$. A sequence $\langle c_1\ a\ c_2 \rangle$, where $a$ is an action name, represents a transition from context $c_1$ to context $c_2$ with the execution of action $a$. The LTSE maps the CT and the set of context traces to an LTS model in the input language of the LTSA tool [18]. In this mapping, contexts from the CT represent states in the LTS, sequences of contexts in the context traces are transitions, and action names between contexts describe transition labels. One of the outputs of the LTSE is an MDL file, which is the textual description of the model. In this file, each state (context) of the model is described presenting its CID and its outgoing transitions (CID of the destination state and associated action name[2]).

In the extension proposed in this work, we add to the description of the MDL file the information about the number of occurrences of contexts and their outgoing transitions, following a similar idea to that applied in the LoTuS tool [3] and other approaches (e.g., [11] and [5]). The important difference is that these approaches calculate probabilities based only on the occurrences of sequences of action names (transitions between states), whereas we produce probabilities guided by sequences of contexts. This means that we can precisely map traces to the code structure to understand how they were produced. Hence, we can also identify in which part of the code an action is executed and what are the conditions to reach that spot. Given a set of context traces $T$, we process each trace $t \in T$ and obtain:

- $co(c, t)$: the number of occurrences of a context $c$ in $t$;
- $to(c_1, act, c_2, t)$: the number of occurrences of a transition with an action $act$ from a context $c_1$ to a context $c_2$ in $t$.

The probability $Pr(c_1, act, c_2)$ of executing a transition from a context $c_1$ to a context $c_2$ with action $act$, considering the set of context traces $T$, is given by the following formula, where $size(T)$ is the number of traces in $T$:

$$Pr(c_1, act, c_2) = \frac{\sum_{i=1}^{size(T)} to(c_1, act, c_2, t_i)}{\sum_{i=1}^{size(T)} co(c, t_i)} \quad (1)$$

Hence the probability of a transition $Pr(c_1, act, c_2)$ is the ratio of the total number of appearances of sequence $\langle c_1\ act\ c_2 \rangle$ in the set of traces to the total number of appearances of context $c_1$.

## 3.2 Simple traces x context traces

If we compare this probability calculation process with that of approaches based on simple traces (i.e., without context information), it is possible to clearly see the difference. For example, consider two simple traces from the editor program:

```
open edit print save exit close
open edit edit print exit save close
```

Calculation based on sequences of action names would result in a probability of 0.75 of action `edit` being followed by a `print`, which could indicate a causal relationship between these two actions. However, if we look back at the code in Figure 1, we can see that action `print` is enabled as long as a document is open. Another example of possible incorrect results from simple traces is the two consecutive occurrences of action `edit`: it could have occurred in a loop, indicating that there could be more consecutive occurrences, or it could represent two consecutive call sites in the code. Nevertheless, looking at the code, it is easy to see that the latter option is not possible. As for the former, it seems true, because the call to method `edit` occurs in the loop where an input is read and the option associated to the received value is executed. And what could we say about action `save`? Does it happen before or after `exit`? Or always after `print`?

When traces with context information are collected, annotations are included in the code to mark the occurrence of control-flow structures, such as selection and repetition statements, method call sites, and method bodies. Context information also regards the values of attributes. A partial example of a trace collected from the editor is presented below, where labels represent the beginning/end of a control structure (`SEL_ENTER`/`SEL_END` for selection, `REP_ENTER`/`REP_END` for repetition, `CALL_ENTER`/`CALL_END` for method calls, and `MET_ENTER`/`MET_END` for method bodies):

```
1  REP_ENTER:(cmd!=4)#true#{isOpen=false^isSaved=true^}#18;
2  CALL_ENTER:readCmd#{isOpen=false^isSaved=true^}#0;
3  CALL_END:readCmd#0;
4  SEL_ENTER:(0)#0#{isOpen=false^isSaved=true^}#17;
5  SEL_ENTER:(!isOpen)#true#{isOpen=false^isSaved=true^}#14;
6  CALL_ENTER:open#{isOpen=false^isSaved=true^}#4;
7  MET_ENTER:open#{isOpen=false^isSaved=true^}#19;
8  MET_END:open#19;
9  CALL_END:open#4;
10 SEL_END:(!isOpen)#14;
11 SEL_END:(0)#17;
12 REP_END:(cmd!= )#18;
13 REP_ENTER:(cmd!=4)#true#{isOpe=true^isSaved =true^}#18;
```

For each annotation representing the beginning of a structure, besides a label identifying its type, it is collected the predicate (guard) associated to the structure and its value (if selection or repetition), name of the method (if method call or method body), values of attributes, and the block ID, which is a unique number identifying the specific block of code. Annotations corresponding to the beginning of a structure determine a context, whereas, annotations marking the end of structures are used to determine the scope of the corresponding context. In this trace we see all the path from the start of the execution to the occurrence of action `open`. Note how it starts with the loop test (line 1), when no document is open and `isSaved` is true. There is a call to read the next input (lines 2 and 3), then the value read (0) is tested in a selection structure (line 4) and the corresponding behaviour is executed (lines 5-10). Line 7 marks not only the context involving the body of method `open` but also indicates the occurrence of the corresponding action. It is interesting to verify that lines 1 and 13 both refer to the same block of code (block ID 18), but their context information is different, as the value of attribute `isOpen` has changed. This means that we reached the same part of the code where we started, but the following behaviour may be different because we are executing in a new context. Considering this, the translation of traces equivalent to the two simple traces presented before, but with context information, results in context traces partially presented below:

```
#0..#7 open #8..#15 edit #16..#50 print #16..#22 save
#8..#53 exit #54..#34 close #35

#0..#7 open #8..#15 edit #16..#57 edit #16..#50 print
#16..#25 exit #26..#31 save #32..#34 close #35
```

[2] In cases where there is no action name associated to the transition, we use a predefined action name `null`, which plays the role of a silent transition.

Numbers preceded by "#" are CIDs and ".." represents the omission of intermediate CIDs. Hence, context traces are similar to simple traces, but action names are put into contexts. For instance, in the first trace, action `print` occurs in a sequence of contexts initiated by context 16 and, after executing this action, the program goes back to the same context 16. This means that this action occurred inside a loop and we can safely take this to the model, even though the traces do not show any consecutive occurrences of `print`. Moreover, the two consecutive occurrences of `edit` in the second trace, although happening in the same call site inside a loop, occur in different contexts (the first in sequence initiated in context 8 and the second in sequence initiated in context 16). Using all this information, if we were to connect these two traces, we could merge common sequences of contexts without introducing invalid behaviours, because contexts tell us when we really are in the same "state". Furthermore, when joining traces, we might introduce new valid behaviours not observed in traces, such as a loop with action `print`. Next, we describe how we use the context traces to create a model and produce the probability values.

### 3.3 Model Construction

Our tool interprets the information obtained from the context traces of a program using the contents of the MDL file. As an example, this is part of the MDL file produced by the LTSE tool for the editor program:

```
Q22#4
Q23#null#1
Q37#null#2
Q44#null#1
```

This part of the file describes a state $Q22$ (CID 22), which occurs 4 times in the traces and has 3 transitions with no associated action names (therefore, all labelled with `null`): to state $Q23$ with 1 occurrence, to state $Q37$ with 2 occurrences, and to state $Q44$ with 1 occurrence. Therefore, $Pr(22, null, 23) = Pr(22, null, 44) = 0.25$ and $Pr(22, null, 37) = 0.50$. After calculating the probabilities of all transitions in the model, we then move on to translating the MDL file to the input language of PRISM [16].

As PRISM works with Markov chains, our tool, named *LTSEProb*, ignores the transition labels when producing the probabilistic model, focusing only on states, transitions, and transition probabilities. It creates a DTMC by processing each state individually, listing all its adjacent states and transition probabilities, and mapping this information to a line in the PRISM language. We use a variable to control the state number, which is initialised with a range from 0 to the total number of states (contexts of the CT). Using this variable (`s` in our example), we map each transition in the MDL file to a transition in the format:

```
[] s=o -> p1:(s'=d1) + p2:(s'=d2);
```

where the command indicates that, from state $o$, it is possible to go to state $d1$ with probability $p1$ or to state $d2$ with probability $p2$ (note that the sum of $p1$ and $p2$ must be 1.0). Therefore, the mapping of context 22 of our example to PRISM results is the following line:

```
[] s=22 -> 0.25:(s'=23) + 0.5:(s'=37) + 0.25:(s'=44);
```

The result of the translation is a PRISM file that can be directly used in the tool for all supported analyses. For instance, if we are interested in calculating the probability of reaching state 44 (action `exit`) from state 22 (action `save`), we could do it by checking the property P=?[true U s=44], which would give the result of 0.25.

## 4 RESULTS AND DISCUSSION

In this section we present the experimental results of 3 case studies. The first case study is our running example, the editor application, and we use it to compare our accuracy with that of models produced using simple traces. The second case study is the single `java.io.PipedOutputStream` class, whose source code is part of JDK7. The last case study is the `BoundedBuffer` implementation presented in [13], involving multiple components. The LTSE tool and all the files for the artifacts discussed in this section can be found at http://www.inf.ufrgs.br/~lmduarte/mise2018.

### 4.1 Case Study 1 - Model Evaluation

Although our probability calculation follows the same ideas of previous work (e.g., [11], [5], and LoTuS), the use of contexts enables the safe merging of multiple traces to form a single, general model for a component [9]. This means the assignment of probabilities to transitions of the model can also be correctly carried out, as each context is unique. Thus, our probabilistic models are a more accurate representation of the probabilistic behaviour described in the observed executions, closer to the real probabilistic behaviour of the actual system.

We demonstrate the accuracy of our models by comparing our model for the editor with the one produced by the LoTuS tool for the same program. We use the results from LoTuS in our evaluation because, besides being a tool freely available, it implements features for model construction, probability calculation, and probability assignment based on traces. Moreover, it allows a set of traces to be generated from models to produce test cases and model visualisation. Models are created as PLTS, but can be exported to PRISM. Hence, we can use these features to compare our models produced from the same set of executions based on visual inspection, set of valid traces in the resulting models, and correct description of probabilistic behaviour. We chose the editor program for this comparison exactly because we know the expected behaviour (probabilistic and non-probabilistic), so that we have a way of evaluating accuracy.

Figure 3 presents the PLTS model generated using the LoTuS tool for the editor application based on 7 traces. Figure 4 presents the PLTS model created by LTSEProb using the same 7 traces, but augmented with context information. For a better visualisation, we have removed most of the transitions labelled with `null` and probability 1.0, which do not affect the resulting model in terms of behaviour description.

The visual comparison shows significant differences between the models, including the number of states. For instance, the initial state of the LoTuS model does not allow `exit` as an initial action, which is a valid option according to the code in Figure 1. To support our visual analysis, we also used the TCG plugin of LoTuS [21] to generate test cases from both models. As the test generation algorithm explores paths of the model, we can observe sequences of actions considered valid in each model. Our model generated 20 test cases with valid sequences (sequences of actions the editor program can execute), whereas the LoTuS model generated 21 test

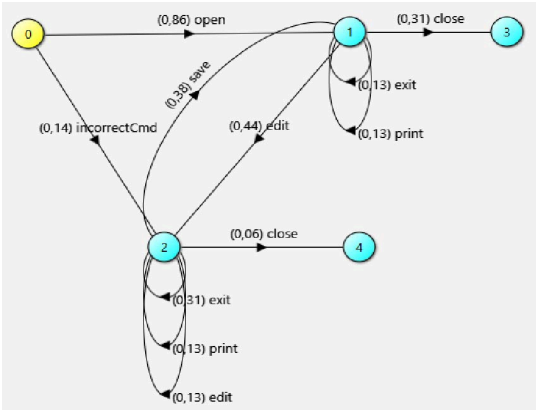


**Figure 3: PLTS model produced by LoTuS.**

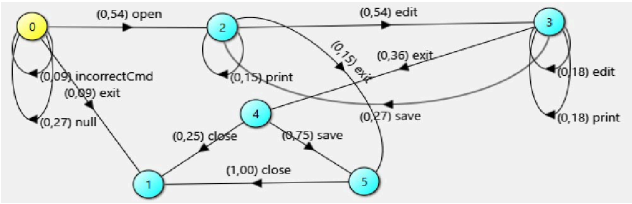


**Figure 4: PLTS model produced by LTSEProb.**

cases out of which only 2 were valid behaviours of the application (⟨*open edit exit close*⟩ and ⟨*open exit close*⟩), which were among the valid sequences of our model. Using the option to determine the probability of each sequence, we analysed the 2 valid sequences of the LoTuS model, which had probabilities 0.007 and 0.033, respectively. The same sequences in our model had probabilities 0.012 and 0.081, respectively. The most likely sequence according to the LotuS model is ⟨*open close*⟩ (0.26), which is not a valid sequence, since there always has to be an *exit*. Incidentally, sequence ⟨*exit*⟩ is the most likely according to our model (0.09) and the second most likely is ⟨*open exit close*⟩ (0.081), which is the correct sequence for opening and closing a document without editing/printing. This, once again, demonstrates the differences between the models, even for valid sequences: as our model contains only valid behaviours, the assignment of probabilities to transitions is more accurate.

## 4.2 Case Study 2 - `java.io.PipedOutputStream`

Our second case study focuses on showing how we can produce a model from a single class code and execute some property analysis. We used the 34 JUnit test cases provided along with the source code to produce traces of the `java.io.PipedOutputStream` class. The use of test cases allows us to produce traces that, according to the developers, represent common scenarios of usage, including those involving unsuccessful executions. It could also let us know before-hand the traces that would be in the model, thus knowing what to expect from the resulting model.

To analyse the model generated by LTSEProb, based on the context traces processed by the LTSE tool, we created 3 PRISM formulas:

| Property | Result |
|---|---|
| `P=? [F exception]` | 0.272 |
| `P=? [F disconnected]` | 0.202 |
| `R=? [F final]` | 9.72 |

**Table 1: Property analysis of the `PipedOutputStream`.**

"exception", which is true in states where an exception occurs, "successful", to define a successful connection/disconnection, and "final" to indicate that the system reached the final state (i.e., termination of the execution). We used the information of the MDL file and context table to identify the states where formulas should be true. Using these formulas, we analysed 3 properties: the probability of reaching an exception state, the probability of executing a successful transmission, and the expected number of steps to reach the end of an execution[3]. The results of these analyses are presented in Table 1, where the corresponding PCTL formulas for each property are presented.

These results show how our models can be readily used in PRISM for quantitative analysis. Moreover, in this particular case, as we based our model construction on traces produced by test cases, developers could use the information from these analyses to improve their test suite. For instance, the result that the probability of a successful connection/disconnection is slightly above 20% indicates that successful scenarios have not quite been explored, whereas above 27% of the cases involved reaching an exception of some type. The low probability values also have to do with test cases that do not actually evaluate complete scenarios (e.g., one trace produced was ⟨*flush connect connected*⟩, which represents an unfinished connection). Because of this, the average number of 9.72 steps for a complete execution might not correspond to the real system. Hence, the developer could use this information to produce new test cases or even mutate existing ones.

## 4.3 Case Study 3 - `BoundedBuffer`

The third case study concentrates on modelling a program with multiple components. The `BoundedBuffer` program has 3 components: a buffer, a producer, and a consumer. We executed the program 12 times using different configurations, such as changing the order in which producer and consumer were initialised and the number of items to be produced/consumed and including a delay for the consumer to start. Following our compositional idea, we produced a separate model for each component (PRISM module) and then manually introduced the necessary synchronisation to create a composed model (transitions guards). This approach allows the analysis of different scenarios, with different number of instances of each component. In this paper, however, we consider only the trivial scenario, with one instance of each component and a buffer capacity of 3 items.

For this program we analysed 3 properties: the probability of the buffer being full, the probability of generating an exception (when the consumer tries to get the next item and the buffer has been halted by the producer), and the long-run expected probability of the

[3]These steps actually represent the number of transitions involving contexts, hence they may not describe exactly the number of actions executed.

| Property | Result |
|---|---|
| `P=? [F full]` | 0.058 |
| `P=? [F exception]` | 0.001 |
| `S=? [empty]` | 0.548 |

**Table 2: Property analysis of the `BoundedBuffer`.**

| Program | Traces | States | Trans. | Choices |
|---|---|---|---|---|
| Editor | 7 | 68 | 80 | 4 |
| PipedOutputStream | 34 | 38 | 55 | 12 |
| Buffer | 12 | 60 | 69 | 7 |
| Consumer | 12 | 9 | 12 | 2 |
| Producer | 12 | 9 | 11 | 2 |
| BoundedBuffer | 12 | 2472 | 7354 | - |

**Table 3: Summary of results from the case studies.**

buffer being empty. The results of these analyses are presented in Table 2, where the corresponding PCTL formulas for each property are presented (where "full", "exception", and "empty" are formulas in PRISM language with obvious meanings).

The results show that the buffer is rarely full (about 6% of the time) and is likely to stay empty most of the time (about 55% of the time). These values demonstrate that items are quickly consumed as soon as they are stored in the buffer, hence the low probability of reaching full capacity. The observed low probability of producing an exception (0.1%) is also a result of this behaviour of the consumer. Actually, we only observed exceptions in executions where the start of the consumer was delayed, causing the producer to store all the necessary items and halt the buffer at some point before the consumer was able to get all the remaining items.

## 4.4 Discussion

Table 3 presents a summary of the case studies results. We describe, for each program, the number of traces used to produce the model (column Traces), the number of states (column States) and transitions (column Trans) in the resulting model, and the number of choice points (column Choices). Choice points identify parts of the model where there is a probabilistic choice, reflecting selection/repetition structures in the code and the probability of each possible choice according to the observed traces. Note that results of the `BoundedBuffer` involve its 3 components, thus we present the results for each component separately and also the values for the composed model. We do not have the number of choice points because this composed model is internally built by PRISM.

As a general result, our case studies were small programs, but the produced models represent the probabilistic behaviour according to the observed traces. Analyses carried out in the models demonstrate that results correspond to the behaviour of the target codes. Although we cannot visualise the composed model of the `BoundedBuffer`, the individual models accurately describe the behaviour of each component and the synchronisation mechanism implemented in PRISM does the job of putting them together. Hence, as far as we are concerned, we produced accurate values of local probabilities, which serve for calculating global probabilities. We do not, however, provide support for automatic composition, not even for assigning labels to the PRISM transition commands. We intend to pursue this in the future.

The state space of the generated models is, in some occasions, considerably large due to a great number of intermediate (and some times redundant) states that could be omitted without affecting any analysis. In more complex applications, our models may be too large and analyses might take a long time to produce results. Therefore, we are studying how to automatically simplify these models without affecting their properties.

Considering that contexts ensure correctness of the models up to a certain level of abstraction, which we demonstrated to be a better construction abstraction than simple traces (see Section 4.1), the probabilistic results should also reflect a correct representation of probabilistic choices in the traces generated by the applications. Thus, our models are more accurate than those produced by other approaches using only sequences of action names [11] [5]. However, our approach requires access to the source code to achieve this higher level of accuracy and adds some overhead to the program execution with the necessary annotations in the code. Scalability is usually not a problem, as models of individual components can be built separately and then merged in the analysis tool.

Our approach builds models using a set of traces, which means that the models contain the composition of all traces of a particular program. Therefore, the behaviour in one of our models corresponds to the behaviour that can be obtained from combining multiple traces in a single model. Guided by contexts, this a safe state-merging operation, guaranteeing correctness given a certain level of abstraction, which can be adjusted through a refinement process [9]. Completeness, however, depends on the quantity and quality of the traces, but could be achieved by incrementally adding new traces to the model [8]. When we consider probabilistic behaviours, we can also, in a sense, guarantee correctness, considering the accurate calculation of probabilities and assignment to the right transitions in the model. Nevertheless, the values of the probabilities should be as close as possible to those of the actual program. Hence, the model has to be in continuous update, considering more traces. Note that, in contrast to the search for completeness, new traces that are similar to the ones already in the model are extremely useful, as they indicate the frequency of a specific behaviour, enabling the adjustment of the probabilities in the model accordingly.

# 5 RELATED WORK

To the best of our knowledge, known approaches for model extraction of probabilistic models are the ones described in [23], [11], and [5]. In [23], a PEPA (stochastic) model (which corresponds to a *Continuous Time Markov Chain - CTMC*) is created for each function. The extraction occurs by static analysis, based on abstract interpretation, of restricted C code. Using static analysis, it may be possible to guarantee completeness, but the user has to annotate the functions to indicate how they should be interpreted in the model. Rates for each transition are calculated statically and dynamically (by sampling the average time to execute each basic block). In the approach in [11], the authors produce DTMC from users' navigational behaviour in web applications. One DTMC is created for

each class of users, where user classes are defined by designers. They count the number of transitions from each state to each other possible state to determine probabilities. As for the approach in [5], an extended Markov model is used to describe class temporal specifications inferred from a set of traces. They use an online learning algorithm to build the model from the samples and then calculate the ratio between the number of appearances of a transition $(i, j)$ between states (method calls) $i$ and $j$ and the number of times state $i$ shows in the traces. Because of their online approach, they do not need to store the traces and can evolve the specification with new traces. Whereas [5] considers only traces with names of events, [11] uses log files containing more information, such as IP addresses, to identify different users interacting. However, in both cases they only consider the dynamic information of the sequence of events and may introduce some invalid behaviour due to the inference process. Although the authors in [5] mention the possibility of an on-the-fly addition of behaviours, they do not deal with multiple models or compositional approaches.

## 6 CONCLUSIONS AND FUTURE WORK

We presented an approach for the extraction of probabilistic models from code based on a context-based LTS extraction approach. We demonstrated our models are more accurate than those produced without context information. We have developed a tool (LTSEProb) that automates the mapping from outputs from the LTSE tool to inputs of PRISM, enabling all available types of analyses. We presented the results of 3 case studies, with different characteristics, which involved the automatic construction of models by the LTSEProb that can be readily used in PRISM and accurately describe the probabilistic behaviour present in the provided set of samples of execution. Our models can be iteratively built and support compositional construction and analysis.

As future work, we intend to reduce the size of our models by applying some minimisation algorithm, thus eliminating states not necessary for analyses. Moreover, we intend to apply the state refinement described in [19] to make our models even more accurate and study how to combine our work with statistical testing to produce better probability transitions. We also plan to work with larger systems and on how to best use our models to support test generation, in particular for mutation testing based, for example, on new behaviours inferred from observed executions. Furthermore, we will investigate the possibility of adapting an approach for Graph Grammars extraction [10] to include probabilistic information.

## ACKNOWLEDGMENTS

The authors would like to thank CNPq-Brazil for partially supporting this work.

## REFERENCES

[1] G. Ammons, R. Bodìk, and J. R. Larus. 2002. Mining Specifications. In *ACM POPL*. ACM, New York, NY, USA, 4–16.
[2] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. 1996. Verifying Continuous Time Markov Chains. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK, 269–276.
[3] Davi Monteiro Barbosa, Rômulo Gadelha de Moura Lima, Paulo Henrique Mendes Maia, and Evilásio Costa Junior. 2017. Lotus@Runtime: A Tool for Runtime Monitoring and Verification of Self-adaptive Systems. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '17)*. IEEE Press, Piscataway, NJ, USA, 24–30.
[4] Pierre Bremaud. 2008. *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer.
[5] Deng Chen, Yanduo Zhang, Rongcun Wang, Xun Li, Li Peng, and Wei Wei. 2015. Mining Universal Specification Based on Probabilistic Model. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engine*.
[6] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. 2000. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 439–448.
[7] L. M. Duarte. 2007. *Behaviour Model Extraction using Context Information*. Ph.D. thesis. Imperial College London, University of London.
[8] Lucio Mauro Duarte. 2011. Integrating Software Testing and Model Checking Through Model Extraction. In *14th Brazilian Symposium on Formal Methods (SBMF) : short papers*. 73–78.
[9] Lucio Mauro Duarte, Jeff Kramer, and Sebastian Uchitel. 2017. Using contexts to extract models from code. *Software & Systems Modeling* 16, 2 (2017), 523–557.
[10] Lucio Mauro Duarte and Leila Ribeiro. 2017. Graph Grammar Extraction from Source Code. In *Formal Methods: Foundations and Applications*, Simone Cavalheiro and José Fiadeiro (Eds.). Springer International Publishing, Cham, 52–69.
[11] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining Behavior Models from User-intensive Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 277–287.
[12] H. Hansson and B. Jonsson. 1994. A Logic for Reasoning about Time and Probability. *Formal Aspects of Computing* 6, 5 (1994), 512–535.
[13] Klaus Havelund and Thomas Pressburger. 2000. Model checking Java programs using Java PathFinder. 2, 4 (2000), 366–381. https://doi.org/10.1007/s100090050043
[14] G.J. Holzmann and M.H. Smith. 1999. A Practical Method for Verifying Event-Driven Software. In *International Conference on Software Engineering*. ACM New York, Los Angeles, USA, 597–607. https://doi.org/10.1145/302405.302710
[15] R.M. Keller. 1976. Formal Verification of Parallel Programs. *Commun. ACM* 19, 7 (July 1976), 371–384.
[16] M. Kwiatkowska, G. Norman, and D. Parker. 2001. PRISM: Probabilistic Symbolic Model Checker. In *Proc. of the Tools Session of Aachen 2001 Intl Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, P. Kemper (Ed.). 7–12.
[17] M. Kwiatkowska, G. Norman, and D. Parker. 2005. Probabilistic model checking in practice: Case studies with PRISM. *ACM SIGMETRICS Performance Evaluation Review* 32, 4 (2005), 16–21.
[18] J. Magee and J. Kramer. 2006. *Concurrency: State Models and Java Programming* (2nd ed.). Wiley and Sons, Chichester, England.
[19] Paulo Henrique M. Maia, Jeff Kramer, Sebastian Uchitel, and N. C. Mendonca. 2009. Towards accurate probabilistic models using state refinement. In *ESEC/SIGSOFT FSE 2009*. 281–284.
[20] L. Mariani, M. Pezze, and M. Santoro. 2017. GK-Tail+ An Efficient Approach to Learn Software Models. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1.
[21] Laryssa Lima Muniz, Ubiratan S. C. Netto, and Paulo Henrique M. Maia. 2015. TCG - A Model-based Testing Tool for Functional and Statistical Testing. In *Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 2: ICEIS,*. 404–411. https://doi.org/10.5220/0005398604040411
[22] Pedro Rodrigues, Emil Lupu, and Jeff Kramer. 2015. Compositional Reliability Analysis for Probabilistic Component Automata. In *Proceedings of the Seventh International Workshop on Modeling in Software Engineering (MiSE '15)*. IEEE Press, Piscataway, NJ, USA, 19–24. http://dl.acm.org/citation.cfm?id=2820489.2820494
[23] Michael J.A. Smith. 2007. Stochastic Modelling of Communication Protocols from Source Code. *Electronic Notes in Theoretical Computer Science* 190, 3 (2007), 129–145.
[24] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2013. Inferring Extended Finite State Machine Models from Software Executions. In *20th Working Conference on Reverse Engineering (WCRE)*. 301–310.