Softw Syst Model
DOI 10.1007/s10270-015-0466-0

REGULAR PAPER

Using contexts to extract models from code

Lucio Mauro Duarte[1] · **Jeff Kramer**[2] · **Sebastian Uchitel**[2]

Received: 15 June 2014 / Revised: 3 March 2015 / Accepted: 13 April 2015

Abstract Behaviour models facilitate the understanding and analysis of software systems by providing an abstract view of their behaviours and also by enabling the use of validation and verification techniques to detect errors. However, depending on the size and complexity of these systems, constructing models may not be a trivial task, even for experienced developers. Model extraction techniques can automatically obtain models from existing code, thus reducing the effort and expertise required of engineers and helping avoid errors often present in manually constructed models. Existing approaches for model extraction often fail to produce faithful models, either because they only consider static information, which may include infeasible behaviours, or because they are based only on dynamic information, thus relying on observed executions, which usually results in incomplete models. This paper describes a model extraction approach based on the concept of contexts, which are abstractions of concrete states of a program, combining static and dynamic information. Contexts merge some of the advantages of using either type of information and, by their combination, can overcome some of their problems. The approach is partially implemented by a tool called LTS Extractor, which translates information collected from execution traces produced by instrumented Java code to labelled transition systems (LTS), which can be analysed in an existing verification tool. Results from case studies are presented and discussed, showing that, considering a certain level of abstraction and a set of execution traces, the produced models are correct descriptions of the programs from which they were extracted. Thus, they can be used for a variety of analyses, such as program understanding, validation, verification, and evolution.

Keywords Behaviour models · Model extraction · Model analysis

Communicated by Dr. Juergen Dingel.

This work was sponsored by CAPES and CNPq.

✉ Lucio Mauro Duarte
lmduarte@inf.ufrgs.br

Jeff Kramer
jk@doc.ic.ac.uk

Sebastian Uchitel
su2@doc.ic.ac.uk

[1] Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Av. Bento Gonçalves, 9500, Porto Alegre, RS 91501-970, Brazil

[2] Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK

1 Introduction

Behaviour models are abstractions that provide a restricted view of the behaviour of systems. They can be used for program documentation and comprehension, as artefacts to be presented to stakeholders for validation, or as a basis for automated validation and verification techniques, such as model-based testing [58] and model checking [13]. There are many tools that are able to carry out these analyses based on a model, such as GraphWalker,[1] Spin [32], LTSA [45], ModelJUnit [57], and JTorX [5]. However, constructing these models normally requires some effort and expertise and is often a non-trivial task, even for experienced designers [33]. Furthermore, after producing a corresponding implementation, one cannot always guarantee its correctness with respect to the specification, as properties preserved in the model may

[1] http://graphwalker.org.

Published online: 03 May 2015

not have been carried over to the program [27]. Hence, there is a need for (semi-)automatic techniques that could help construct the model as well as keep conformance with the most recent version of the corresponding implementation. Moreover, because building a model requires some effort and expertise, many systems are implemented without any type of model to document their behaviour, which might cause problems for future maintenance and evolution. Therefore, it would also be necessary to have techniques and tools that could generate a model from an implementation so as to enable all model-based analyses, taking advantage of the existing tool support, and to guarantee that the model faithfully represents the behaviour of the corresponding program.

Model extraction [35] is a process that generates an abstraction of a system based on an existing implementation, allowing models to be constructed and used even in situations where there is no model or models are out of date. It can also be used for reverse engineering legacy code, thus providing information about the code behaviour and facilitating software maintenance and evolution. The process defines a mapping from information obtained from the code to a behaviour model description, which means that the developer might not need to know either the programming or the modelling language. It also favours conformance between model and code, because if the code is modified, then a new execution of the same extraction process would generate an updated model. However, any model extraction process has to deal with the *model construction problem* [15], which corresponds to finding a mapping from the concrete program to the abstract behaviour model that produces a *faithful* representation of the program behaviour. A faithful model should describe the complete behaviour of the code and include only feasible behaviours, so that an analysis on model would correspond, in a given level of abstraction, to an analysis on the program itself.

Motivating example As an example of this problem, consider the piece of code of a simple editor presented in Fig. 1. It uses two attributes: one to control whether a document is currently open (`isOpen`) and another to check whether this document has been saved after modifications (`isSaved`). There are two relevant properties for this program: (P1) only opened documents can be edited, printed, saved, or closed; (P2) it is only possible to save modified documents that have not been saved yet. This code preserves both properties, and therefore, a faithful model of this program would have to preserve them too.

There have been proposed many approaches to try to overcome the model construction problem, such as [2,8,11,14,30,34,43,47,60], that we could use to create a model of our code. These techniques can be divided into two groups: *static approaches*, which build models based on information from the source or compiled code, and *dynamic approaches*, which infer models from samples of execution. To compare these groups, Figs. 2 and 3 show models based on the static approach and the dynamic approach, respectively. We use the formalism of *Labelled Transition Systems* (*LTS*) [38] to present our example models because most of the model extraction approaches deal with similar formalisms, based on finite state machines, and thus, it is possible to see a result close to what these approaches would produce. Moreover, using the same formalism makes it easier to compare the models. Transitions are labelled with the method names, representing actions of the program, so that a transition labelled with method *m* represents that action *m* has been executed. State 0 is the initial state. In these models, we ignore method `readCmd` as it only represents the way the program reads the inputs. We leave the command in line 25 to be discussed later, as part of our approach description.

Techniques based on static information usually produce over-approximations of the system behaviour, such as the

Fig. 1 Editor code

```
1  public class Editor {
2   private boolean isOpen;
3   private boolean isSaved;

4   public Editor () {
5    isOpen=false;
6    isSaved=true;
7    int cmd=-1;
8    String name=null;

9    while(cmd!=4){
10    cmd=readCmd();
11    switch (cmd) {
12     case 0: if(!isOpen)
13               name=open();
14             break;
15     case 1: if(isOpen)
16               edit(name);
17            break;
18     case 2: if(isOpen)
19               print(name);
20             break;
21     case 3: if(!isSaved)
22               save(name);
23             break;
24     case 4: exit();
25     default: #action:"incorrectCmd";
26    }}
27  }
...
28  void exit (String n) {
29   if (!isSaved) {
30    int opt=readCmd();
31    if (opt==0)
32     save(n);
33   }
34  if (isOpen) close(n);
35  }
36 }
```

one in Fig. 2, which guarantees completeness but may lead to the inclusion of infeasible behaviours. For example, the model considers the sequences allowed by the control flow but, because it does not correctly represent the order between the actions of the system, it allows `edit` to be executed even before a document has been opened (loop transition in state 0), which violates property P1, even though it is preserved by the code. On the other hand, techniques that use samples of execution build models based only on feasible behaviours (i.e. observed executions), but there is no guarantee of completeness, because it might require observing all possible executions. However, they can also have a correctness problem due to the generalisation of the system behaviour based on these samples. The model in Fig. 3 was built based on trace ⟨`open, edit, save, print, edit, edit, print, save, print, edit, exit, save, close`⟩. The basic idea of the model is that each action has the effect of leading the system to a specific state, so that every execution of an action takes the system back to the corresponding state. The model correctly allows the behaviour observed in the trace; however, it also allows infeasible behaviours, such as saving a document that has not been previously edited, as action `save` from state 4 to state 5 can happen after a previous `save` from state 4 to state 5 and subsequent `print` from state 5 to state 4. This occurs because the model cannot distinguish when an action happens in different situations (for instance, when `save` occurs before closing a document and when it happens after the document has been modified). In this case, the behaviour present in the model violates property P2. Hence, the model in Fig. 2 would mislead us into thinking that there was no determined order for executing a set of actions due to the absence of information about control predicates based on program variables, while the model in Fig. 3 would include a feasible behaviour but, because there is no information on how this behaviour was produced by the code, it would mistakenly infer a model that contains invalid sequences of actions. Nimmer and Ernst [52] proposed a hybrid approach that aims to use static and dynamic information in combination to overcome their limitations. Nevertheless, they do not actually combine the types of information, but use static information to confirm behaviours that are dynamically inferred. Therefore, such confirmation is still limited by the capacity of the static checker of determining whether some behaviour is feasible or not, which has been seen to be a problem.

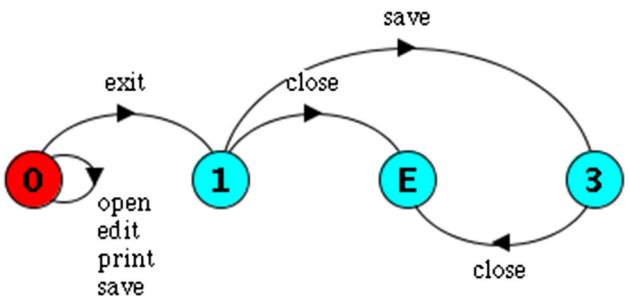

Fig. 2 Editor model using static information

Contexts We propose a hybrid approach for model extraction that effectively combines static and dynamic information, so that it is possible to identify how a given trace was produced in the code and support the safe merging of multiple traces. Our model extraction process is based on the concept of *contexts* [20], which represent abstract states of a system. Each context describes a combination of an execution point in the system control flow graph, a set of values of selected program variables (system state), and a call stack, representing the stack of method calls waiting for a return. Contexts are identified based on recorded execution traces containing the necessary information. Hence, contexts combine the control flow information, used in approaches based on static information, with execution traces from which approaches based on dynamic information infer models, adding information about the program variables and unfinished method calls/executions. This enables the construction of models from sequential and concurrent systems that describe only feasible behaviours based on observed samples. Moreover, because contexts also take into account the situation of unfinished method calls/executions, our models can describe

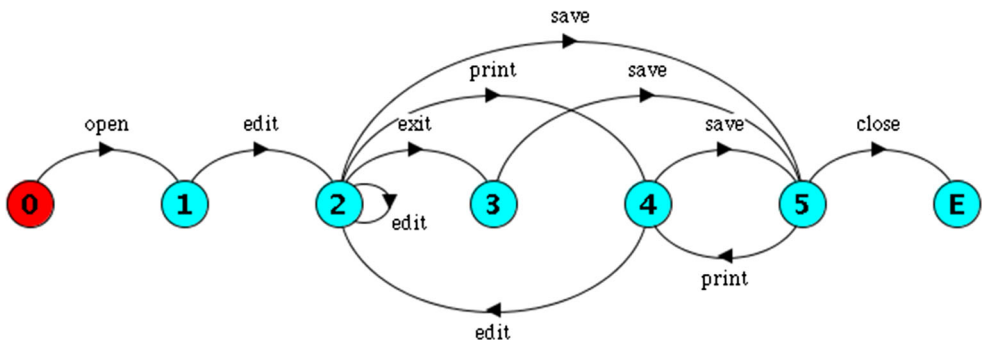

Fig. 3 Editor model using dynamic information

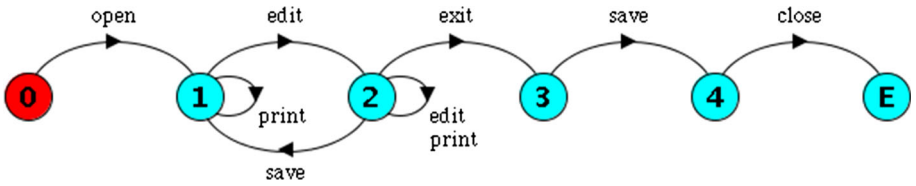

Fig. 4 Editor model using contexts

blocking situations, which are common in concurrent systems.

Figure 4 shows the model we generated applying our approach and using the same trace used to create the model in Fig. 3 (enhanced with information about the control flow and program variables). It is clear that it correctly describes the order between actions, preserving the properties of the Editor program. Even though it considers only one observed sample of execution, the combination of static and dynamic information with the addition of values of program variables (in this case, variables `isOpen` and `isSaved`) allowed the identification of abstract states (contexts) and which set of actions is enabled in each one of them. Therefore, it is possible to distinguish action `save` happening after an edit (state 2 to 1) and after an exit occurring after a previous `edit` (state 3 to state 4). Moreover, additional behaviours were included in the model, such as the loops with action `print` in states 1 and 2, even though the original trace did not include any sequence of two or more consecutive occurrences of this action. This is a result of having contexts that help us know exactly where multiple traces intersect and, therefore, how to merge them without introducing invalid behaviours. Therefore, contexts make it possible the generalisation over samples of execution to create a single model that includes all the observed behaviours and no invalid sequences of actions. Thus, this model could also be easily enhanced with the later addition of traces without affecting its correctness.

Contributions Contexts have been successfully used to extract models from sequential and concurrent systems [18], and the analyses carried out using our models in a number of case studies [19] have demonstrated that they are correct abstractions of the systems they describe, at a certain level of abstraction and according to a set of observed behaviours (traces). Hence, the extracted models are useful for many analysis purposes. Initial models can be refined by reducing their level of abstraction. This is particularly important when there are some known properties of the program that should be preserved by the model. Moreover, new behaviours can be incrementally included in the model in order to improve completeness, without the need of rebuilding the whole model. The approach is partially supported by a tool called LTS Extractor (LTSE) [18], and the generated models can readily serve as inputs to the LTSA tool [45], which supports visualisation, model execution, and verification of temporal properties.

The main contributions of this work towards a process for extracting faithful models from existing implementations are as follows:

- The *concept of a context*, which combines static and dynamic information from the existing code to build behaviour models that correctly represent the behaviour of the code at a certain level of abstraction;
- The creation of a *hybrid model extraction process* that produces models from *sequential and concurrent systems* (including the representation of mechanisms of synchronisation) based on contexts that can serve as inputs to an existing tool, where these models can be visualised and analysed. This process allows the possibility of *tailoring the model to a specific purpose* by adjusting its level of abstraction and selecting the set of behaviours to achieve a certain coverage criterion;
- The approach enables the *incremental construction* of behaviour models, allowing the developer to start with a very restricted set of behaviours (such as in our example) and, gradually, add new behaviours, without rebuilding the entire model;
- The *LTSE tool*, which partially automates the process based on execution traces containing the necessary information for the identification of contexts.

Structure This article is organised as follows. The next section presents the formalism we adopt and discusses model faithfulness. Section 3 introduces the concept of a context and describes how it can be used to extract models. Section 4 presents in more detail our approach for model extraction, discussing all the steps involved, how to refine models, how to apply our approach to concurrent systems, and existing tool support. Section 5 discusses the formal foundations of our approach. In Sect. 6, we describe some case studies that demonstrate practical results of our context-based approach, and we discuss possible threats to validity. Section 7 presents some of the related work and how our approach advances the research on model extraction. And, finally, Sect. 8 contains the conclusions and possible future work.

2 Background

An easy and intuitive way of describing *behaviours* is to represent them as sequences of *actions* that the system can

execute, where an action normally represents the execution of a method. One well-known formalism for describing models using this action-based approach is *Labelled Transition Systems* (LTS) [38]. LTS models have well-defined mathematical properties [45] and, consequently, can be used to reason about sequential, concurrent, and distributed systems. An LTS can be formally defined as follows:

Definition 1 (*Labelled Transition System*) A LTS $M = (S, s_i, \Sigma, T)$ is a model where:

- S is a finite set of states,
- $s_i \in S$ represents the initial state,
- Σ is an alphabet (set of action names), and
- $T \subseteq S \times \Sigma \times S$ is a transition relation.

Transitions are labelled with the names of actions from the alphabet that trigger a change from the origin state to the destination state. Therefore, given two states $s_0, s_1 \in S$ and an action $a \in \Sigma$, then a transition $s_0 \xrightarrow{a} s_1$ means that it is possible to go from state s_0 to state s_1 through the execution of action a. A *behaviour* of an LTS M is then a finite sequence of actions $\pi = \langle a_1 \ldots a_n \rangle$ such that $a_1, \ldots, a_n \in \Sigma$. The set $L(M) = \{\pi_1, \pi_2, \ldots\}$ of all behaviours of M is called its *language*. For a state $s \in S$, $E(s) = \{a \in \Sigma | \exists s' \in S \cdot (s, a, s') \in T\}$ represents the finite set of actions *enabled* in s. A *path* $\lambda = \langle s_1, a_1, s_2, a_2, s_3, \ldots \rangle$ is a sequence of alternating states $s_1, s_2, s_3, \ldots \in S$ and actions $a_1, a_2, \ldots \in \Sigma$ labelling transitions connecting these states, such that, for $i \geq 1$, for every transition $t = (s_i, a, s_{i+1})$ composing λ, $t \in T$. A path always starts and—if finite—ends with a state. We use $\Lambda(M)$ to denote the set of all paths of M.

2.1 Model faithfulness

In order to have confidence on the results of any analysis on a behaviour model extracted from an existing implementation, it is necessary to guarantee that it faithfully describes the system behaviour. In this work, we define *faithfulness* as a relation between a behaviour model and the behaviour of the implementation the model represents. An LTS model M is a *faithful* representation of the behaviour of an implementation Imp iff all and only feasible behaviours of Imp are present in M (at a certain level of abstraction). Considering properties of a program, M faithfully represents Imp if, for any property $Prop$, the satisfaction/violation of $Prop$ by M implies that Imp also satisfies/violates $Prop$. This means that, ideally, $L(M) = L(Imp)$, where $L(M)$ is the language described by M (i.e. the set of all behaviours described in M) and $L(Imp)$ represents the language of Imp (i.e. the set of all feasible behaviours of Imp). Hence, when building a behaviour model, the objective is to achieve a faithful abstraction of the implementation it represents, so that any analysis on the model would correspond, at a certain level of abstraction, to an analysis on the actual program.

Because the level of faithfulness essentially depends on the quantity and quality of information used to build the model, we consider the faithfulness of a model in terms of its completeness and correctness:

Definition 2 (*Completeness*) M is *complete* with respect to Imp iff $L(Imp) \subseteq L(M)$.

Definition 3 (*Correctness*) M is *correct* with respect to Imp iff $L(M) \subseteq L(Imp)$.

Therefore, both completeness and correctness are related to language containment. *Completeness* refers to the inclusion of all feasible behaviours of the program in the model, whereas *correctness* means that the model contains only the feasible behaviours of the program. If the model is not complete, then feasible behaviours of the program are missing, which means that properties that hold in the model might be violated by the program. If the model is not correct, then it includes at least one infeasible behaviour that violates a property not violated by the actual program. A faithful model guarantees, therefore, that the set of behaviours it describes is the exact set of feasible behaviours of the program. Nevertheless, as this set of feasible behaviours might be too large or even infinite, and depending on the purpose of the model, it may be reasonable to reduce the requirement of completeness and correctness to the minimum necessary to achieve a certain goal. For instance, if the model will be used to check whether the code preserves some specific property, then the model should, at least, be complete and correct with respect to this property. If, on one hand, this approach might prevent the construction of a model of the whole behaviour of the system, on the other hand, it allows engineers to build separate models to analyse different portions of the system independently, producing models that can be more easily visualised and handled by analysis tools. As will be discussed in Sect. 4.5, producing individual models also help create models of concurrent systems, allowing the separate analysis of each component and, then, the analysis of their composition.

3 Context information

An abstract state of a program, defined as a *context*, can be seen as a combination of a *control component*, which indicates the current execution point, and a *data component*, representing the current values of program variables. In this work, we consider a control component obtained based on the control flow graph (CFG) of the implementation of a system.

Definition 4 (*Control flow graph*) Let Imp be a program. Then, its *control flow graph* is defined as $CFG_{Imp} = (Q, q_i, Act, \Delta)$, where

- Q is a finite set of control components of Imp, where each control component $q \in Q$ is a pair (bc, cp), with bc representing a block of code (statement or method body) and cp describing the logical test associated with bc (i.e. its *control predicate*);
- $q_i = (bc_i, true)$ where, $q_i \in Q$ and bc_i is the initial block of code;
- Act is the set of actions (method calls or any other events of interest) of Imp; and
- $\Delta \subseteq Q \times Act \times Q$ is a transition relation.

The *data component* represents the values of a set of program variables (system state). Let P_{Imp} be the finite set of variables of Imp and $val(x)$ a function that provides the current value of a given expression x, where expressions can be single variables or composed expressions involving, for instance, arithmetic and logic operators. A finite set of values $v = \{val(p_1), \ldots, val(p_n)\}$ represents one possible valuation of variables $p_1, \ldots, p_n \in P_{Imp}$. The possibly infinite set $V(P_{Imp}) = \{v_1, v_2, \ldots\}$ is composed of all possible valuations of variables of Imp, such that $v_1 = \emptyset$ represents the beginning of the execution, when the values are yet unknown. The finite set $V(P) \subseteq V(P_{Imp})$ represents all possible valuations of variables $p_1, \ldots, p_n \in P$, such that $P \subseteq P_{Imp}$. The program variables considered in this work include the state of the call stack, which contains the names of methods that have been initiated but have not yet terminated. Thus, there might be different contexts depending on whether a certain method execution is pending or not.

The data component adds, to each context, the valuation of the system state at each point of the control flow, so that we can distinguish different situations in which a certain part of the code can be executed. Thus, by the combination of the control component and the data component, it is possible to identify different states of execution for the same point in the control flow. Depending on this combination, different actions may be enabled to execute next. Hence, we can define contexts as follows:

Definition 5 *Context.* Given a program Imp, a context $C = (bc, cp, val(cp), v, cs)$ is the combination, at a certain point of the execution of Imp, of the control component represented by the block of code bc, described by a unique identifier (block ID), its control predicate cp, and the value $val(cp)$ of cp, and the data component represented by the current valuation $v \in V(P)$ of variables in $P \subseteq P_{Imp}$ and the state of the call stack cs, describing the stack of method calls awaiting for a return.

The combination of a control component (control flow information) with a data component (state information) to identify a context is denominated *context information*. According to our definition of contexts, the execution of a system can then be seen as a sequence of contexts, with sequences of actions happening in between them. An execution starts in an *initial context*, where no control predicates have yet been evaluated, and the initial values of variables have not been assigned. As the execution continues, changes in context occur, indicating that at least one of the components of context information has been modified. For instance, given the code in Fig. 1, at the beginning of the execution, the program is in the initial context $C_0 = (0, -, true, \{\}, \langle\rangle)$, which is assigned the block ID 0, and has no associated control predicate (we then use $true$ as a default value for its control predicate) and no current value for the system state. As the execution proceeds, it reaches the block of code in line 9, which determines a new context $C_1 = (1, (cmd! = 4), true, \{false, true\}, \langle\rangle)$, where the block ID is 1; the control predicate is $(cmd! = 4)$, which is evaluated as true (variable cmd is initialised with -1); the values of the attributes `isOpen` and `isSaved` are $false$ and $true$, respectively; and no method is in execution. Hence, the system has moved from context C_0 to C_1, which means that any action happening at this point executes in context C_1 (i.e. it is enabled by C_1). Reaching line 11 determines a new context defined by the switch statement, defined as $C_3 = (2, (cmd), v, \{false, true\}, \langle\rangle)$, where v corresponds to the value read in line 10, and the system state remains the same. Hence, the system now has moved from context C_2 to C_3. Note that, as the value v read as input changes, it determines different contexts. This means that the same block of code can represent multiple contexts depending on the context information. Also note that the execution might reach the same block of code multiple times, which may cause the system to go back to previous context (if the context information is the same) or identify a new context (if at least one value of the context information has changed). Moreover, contexts help determine conditions for a certain action to happen. For instance, the occurrence of the action corresponding to method `open`, in line 13, requires, at least, that the test in line 9 be evaluated as $true$, the control predicate evaluated in line 11 be equal to 0, and the control predicate in line 12 be also evaluated as $true$. As this last control predicate needs the program variable `isOpen` to be $false$ for it to be $true$, then the referred action can only happen in a context where all the previous requirements are fulfilled, and the value of attribute `isOpen` is $false$. Therefore, contexts not only show when an action happens under different circumstances, which may influence its result and the next set of enabled actions, but also determine whether an action can be executed or not.

4 Model extraction based on contexts

Our ultimate goal was to produce models that faithfully represent the behaviour of existing systems and could be used for several types of analysis. As previously discussed, faithfulness is an ideal requirement, but it is usually unattainable due to the complexity and size of current systems. The use of contexts, which are identified in execution traces, guarantees that the model will contain only valid behaviours according to a certain level of abstraction (we shall more formally discuss this in Sect. 5), ensuring correctness. However, as commented in Sect. 2.1, completeness might not be easy—or even possible—to achieve. For this reason, completeness should be evaluated in terms of certain coverage criteria, depending on the objective of the model. We, therefore, focus on producing models that are faithful abstractions of the behaviour of the systems they represent in the sense that they are correct with respect to a level of abstraction and complete with respect to some coverage criteria. The level of abstraction and the type of coverage to be used to produce the model will depend on the purpose of having a model. Thus, we propose an approach that is flexible enough to allow users to adjust these parameters according to their needs.

As our current focus is on Java programs, we work on systems divided into classes, and the variables that determine the system state (in addition to the call stack) are the attributes of these classes. Our model extraction approach begins with the instrumentation of the code of the classes for which we need a model to include annotations to collect the necessary information. The execution of the annotated code produces traces from which we extract context information and the actions that happened in each context. This information is then used to create a model of the system using the *Finite State Processes* (FSP) process algebra [45], which can serve as input to the LTS Analyser (LTSA) [45], where it can be visualised and analysed. A general view of this process is presented in Fig. 5, where ellipses represent processing phases and boxes represent inputs/outputs of these processes. Arrows show the sequence of information processing, and the large block delimited by a dashed line on the lower part of the figure represents the part of the process automated by our tool (presented in Sect. 4.6), called LTSE. Each part of the approach is described in more detail next. We refer to the code in Fig. 1 to exemplify results from each phase.

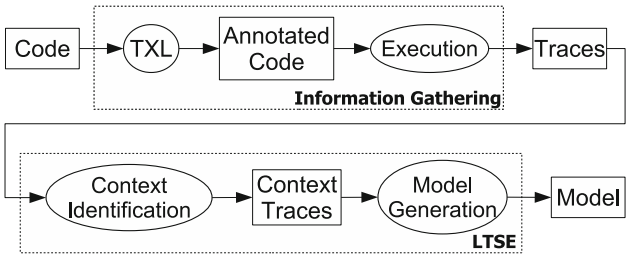

Fig. 5 General view of the model extraction process

4.1 Information gathering

We have been using the TXL engine [16] and a Java grammar specified using the TXL language[2] to automatically create an annotated version of Java codes based on TXL rules already developed.[3] Each rule describes a transformation in the code according to the identified patterns, such as control flow statements, method bodies, and call sites. The application of the rules introduces the necessary annotations for each pattern. It is important to mention that we use the TXL language and engine to simplify the annotation process, but any other technique or tool could be used to instrument the code, as long as the instrumentation follows the same patterns described in the annotation rules.

Annotation format The annotations can be of three different types. An *enter-context annotation* (labels `SEL_ENTER` for selection statements, `REP_ENTER` for repetition statements, `CALL_ENTER` for method calls, and `MET_ENTER` for method bodies) is an annotation regarding the beginning of a control structure (selection or repetition structure, method call, or method body); *exit-context annotations* (labels `SEL_END`, `REP_END`, `CALL_END`, and `MET_END`) represent, on the other hand, the end of a control structure; and an *action annotation* (label `ACTION`) indicates the occurrence of an action. Each enter-context annotation regarding a repetition or selection statement contains the label that identifies the type of the control structure; the control predicate associated with this structure; the value of this predicate; the name of the annotated class; an *object ID (OID)*, which is a Java identifier associated with each object; a list of attribute names and their respective values; and a *block ID (BID)*, representing an identification of the specific block of code, which can be interpreted as an abstraction of the program counter. Annotations representing method calls or method executions include all the information of the other enter-context annotations except for the control predicate and its value as there is no control predicate associated with a method call or method body. Annotations of type exit-context contain only the label, the control predicate, the name of the class, the OID, and the BID. They do not require the information about values because they are only used to mark the end of a context. All action annotations contain only the label `ACTION`, the name of the action, the name of the class, the OID, and the BID.

[2] Available from http://www.txl.ca/.

[3] The complete description of the annotation rules and their corresponding TXL code can be found in http://www.inf.ufrgs.br/~lmduarte/doku.php?id=ltse.

Besides the automatically identified actions (method calls and method executions), we allow users to define their own actions. *User-defined actions* can be introduced in any part of the class body using the format:

```
#action:"<name>";
```

where `name` is the name used to identify the action. They are important in situations where, for example, reaching a given point in the code, other than a method body or a call site, has some particular meaning, such as the completion of a task (e.g. a set of methods that should be executed in order to realise some specific computation), a certain variable assignment, or a particular situation. If the TXL engine is used, they are automatically converted into action annotations when the code is instrumented. As an example, we have introduced a user-defined action, named `incorrectCmd` in line 25 of the code in Fig. 1 to identify the situation where the user enters an invalid option. Note that without this action, there would be no record of this specific situation, which means that the model would not describe this possibility.

In the same way that we allow a user to define new actions, we provide support for the definition of additional variables. We call these *user-defined attributes*, which represent expressions over the values of the original attributes. User-defined attributes are, therefore, used to provide a simple form of data abstraction, which is particularly useful for model refinement, thus reducing the possible number of contexts (see Sect. 4.4). They should be used when the interest is not on the concrete value of an attribute but rather on some predicate over this value. For instance, consider an attribute `temp` that represents the temperature of a boiler. To understand the boiler control behaviour, it might be more important to know whether or not the temperature of the boiler is above some threshold `limit` than knowing the exact current temperature. Hence, a user-defined attribute could be used to monitor whether the limit temperature is reached or not. This could be defined by the annotation:

```
#attribute:"tempOK" = (temp <= limit);
```

where `tempOK` is the name of the attribute and `(temp <= limit)` is the expression used to define its value. Note that the expression of a user-defined attribute can only refer to values of other attributes or constants (in this case, attribute `temp` and constant `limit`). Also notice that, unlike user-defined actions, user-defined attributes can only be defined in the area of attribute definitions, so that they can be processed along with the other attributes and incorporated into the annotations that record the system state. Not only the use of these attributes helps focus on what is relevant to know about the values of attributes, but also it reduces the state space of the model to be constructed. Instead of dealing with a possibly infinite range of values (e.g. all valid temperatures), it makes it possible to work with a binary set of possibilities (e.g. above or not the threshold).

Trace generation Traces can be generated by randomly executing the annotated code. However, as the resulting model will reflect the quality and the quantity of the traces, the observed traces should be selected according to the specific purpose of the model. If the focus is on specific behaviours, the traces to be observed should include these behaviours. Our experience has shown that the best way to produce traces is applying a test suite based on some coverage criteria. The creation of test cases allows the selection of which behaviours will be observed and, consequently, included in the model (besides the possibility of detecting some existing error when applying the tests). Moreover, the application of coverage criteria restricts the evaluation of the completeness to the defined criteria. It is also important to note that, due to the incremental aspect of our approach, an initial set of traces can be used to create a model that can then be improved by the later addition of new traces.

Part of a trace of the editor code is shown in Fig. 6. It describes that a document was opened (lines 1–10), an invalid option was entered (lines 11–15), and then, the next input was the command to edit the document (lines 16–23). Lines 24–33 represent the execution of option print, followed by a document save (lines 34–42). Note that, for simplification, annotations produced by the method `readCmd` are not included.

Line 1 presents an example of an enter-context annotation, where a repetition structure has been executed; the control predicate evaluated was `(cmd!=4)`, which was *true*; the class that produced the annotation was `Editor`; the OID was 31505416; the values of attributes `isOpen` and `isSaved` were *false* and *true*, respectively; and the BID of this structure was 18. This annotation corresponds to the while statement in line 9 of the code in Fig. 1. The annotation in line 2 was produced by the execution of the switch statement in line 11 of the code, the annotation in line 3 was produced by the if statement in line 12 of the code, and so on. Line 13 shows the action annotation produced by the execution of our user-defined action included in line 25 of Fig. 1.

4.2 Context identification

After producing the traces, the next step is to parse each one of them to identify which contexts have occurred. In this parsing, annotations present in the trace are processed in the order they appear, which provides the sequence of contexts. For each annotation, we collect the necessary context information. All context information collected from the traces is recorded in a *context table* (CT), which serves as a memory of all contexts already identified. Each entry of the CT corresponds to a different enter-context annotation found in a trace.

Fig. 6 Example of a recorded execution trace

```
1  REP_ENTER:(cmd != 4)#true#Editor=31505416#{isOpen=false^isSaved=true}#18;
2  SEL_ENTER:(0)#0#Editor=31505416#{isOpen=false^isSaved=true}#17;
3  SEL_ENTER:(! isOpen)#true#Editor=31505416#{isOpen=false^isSaved=true}#11;
4  CALL_ENTER:open#Editor=31505416#{isOpen=false^isSaved=true}#4;
5  MET_ENTER:open#Editor=31505416#{isOpen=false^isSaved=true}#19;
6  MET_END:open#Editor=31505416#19;
7  CALL_END:open#Editor=31505416#4;
8  SEL_END:(! isOpen)#Editor=31505416#11;
9  SEL_END:(0)#Editor=31505416#17;
10 REP_END:(cmd != 4)#Editor=31505416#18;
11 REP_ENTER:(cmd != 4)#true#Editor=31505416#{isOpen=true^isSaved=true}#18;
12 SEL_ENTER:(cmd)#5#Editor=31505416#{isOpen=true^isSaved=true}#17;
13 ACTION:incorrectCmd#Editor=31505416;
14 SEL_END:(cmd)#Editor=31505416#17;
15 REP_END:(cmd != 4)#Editor=31505416#18;
16 REP_ENTER:(cmd != 4)#true#Editor=31505416#{isOpen=true^isSaved=true}#18;
17 SEL_ENTER:(1)#1#Editor=31505416#{isOpen=true^isSaved=true}#17;
18 CALL_ENTER:edit#Editor=31505416#{isOpen=true^isSaved=true}#5;
19 MET_ENTER:edit#Editor=31505416#{isOpen=true^isSaved=true}#20;
20 MET_END:edit#Editor=31505416#20;
21 CALL_END:edit#Editor=31505416#5;
22 SEL_END:(1)#Editor=31505416#17;
23 REP_END:(cmd != 4)#Editor=31505416#18;
24 REP_ENTER:(cmd != 4)#true#Editor=31505416#{isOpen=true^isSaved=false}#18;
25 SEL_ENTER:(2)#2#Editor=31505416#{isOpen=true^isSaved=false}#17;
26 SEL_ENTER:(isOpen)#true#Editor=31505416#{isOpen=true^isSaved=false}#12;
27 CALL_ENTER:print#Editor=31505416#{isOpen=true^isSaved=false}#6;
28 MET_ENTER:print#Editor=31505416#{isOpen=true^isSaved=false}#21;
29 MET_END:print#Editor=31505416#21;
30 CALL_END:print#Editor=31505416#6;
31 SEL_END:(isOpen)#Editor=31505416#12;
32 SEL_END:(2)#Editor=31505416#17;
33 REP_END:(cmd != 4)#Editor=31505416#18;
34 REP_ENTER:(cmd != 4)#true#Editor=31505416#{isOpen=true^isSaved=false}#18;
35 SEL_ENTER:(3)#3#Editor=31505416#{isOpen=true^isSaved=false}#17;
36 SEL_ENTER:(! isSaved)#true#Editor=31505416#{isOpen=true^isSaved=false}#13;
37 CALL_ENTER:save#Editor=31505416#{isOpen=true^isSaved=false}#7;
38 MET_ENTER:save#Editor=31505416#{isOpen=true^isSaved=false}#22;
39 MET_END:save#Editor=31505416#22;
40 CALL_END:save#Editor=31505416#7;
41 SEL_END:(! isSaved)#Editor=31505416#13;
42 SEL_END:(3)#Editor=31505416#17;
43 REP_END:(cmd != 4)#Editor=31505416#18;
...
```

By 'different', we mean that, considering the context information, they can be distinguished (i.e. they differ in some part of the control or data component of the context, as described in Sect. 3). During the parsing of the trace, every time an enter-context annotation is found, and its context information is collected and compared to each existing entry of the context table. If none of the entries of the CT contains the same context information collected from the annotation, then a new context has been found, and a new entry is created to store its information. This new entry is assigned a new *context ID* (CID), which is a unique sequential numeric identifier. If, however, an existing entry contains the same context information obtained from the current annotation, then we have found an already known context, and the CT remains the same.

The call stack information is controlled based on annotations regarding methods. The enter-annotations `CALL_ENTER` and `MET_ENTER` cause the corresponding method name to be included in call stack, whereas the exit-annotations `CALL_END` and `MET_END` have the opposite effect. The idea is that, an enter-context annotation generated by a method *m* indicates that this method started at that point. If, before the corresponding exit-context annotation, another enter-context annotation is found, then this new context corresponds to the current values of the control component, the values of the attributes, and the call stack where method *m* is still executing, awaiting for a return.

In parallel with the construction of the CT, we build a set of *context traces*. Basically, a context trace is the original trace, but with every enter-context annotation replaced by the corresponding CID from the CT, all exit-context annotations removed,[4] and every action annotation replaced by the name of the action. As methods are the basic actions, every enter-

[4] They serve for purposes that will be discussed in Sect. 4.5.

annotation regarding methods also produces an action name in the context trace. More formally:

Definition 6 (*Context trace*) Given a program $Prog$ with a set of actions Act, a context trace t of $Prog$ is a finite sequence $\langle C_1\alpha_1C_2\alpha_2\ldots\alpha_nC_n\rangle$, where $C_1, C_2, \ldots, C_n$ are CIDs of contexts of $Prog$ and $\alpha_1, \ldots, \alpha_n$ are (possibly empty) finite sequences of actions from Act.

We produce the context traces because they make it easier to later generate the model. As the CIDs compactly represent the contexts and their order describes the sequence of contexts that happened during the generation of the trace, we can produce the models without having to consult the CT. Therefore, whereas the CT stores information about all contexts identified in the original traces, context traces present the *ordering* of occurrences of contexts and sequences of actions in each trace. This means that, analysing their contents, we can identify the contexts the system went through to be able to execute a certain sequence of actions. Moreover, it is possible to know in which context each execution of an action happened and, therefore, detect executions of the same action in different contexts.

Algorithm 1 describes the procedure applied to create the CT and generate context traces. The inputs are a finite set of traces and a finite set of attribute names, used to determine which values of attributes will be used to distinguish contexts. The algorithm produces a CT containing the information from all contexts identified in the traces and a set of contexts traces, one for each original trace. We use variable t_a to represent the type of an annotation (enter-context,

Algorithm 1 Algorithm that builds a CT and generates context traces.

function BUILDCT($Traces, P$)
Inputs:
Traces: finite non-empty set of traces,
P: finite set of attributes
Outputs:
CTraces: a finite non-empty set of context traces,
CT: a context table containing information from all contexts from *Traces*

$CT = \emptyset$ // *Initialises the CT*
$nextID = 0$
$CTraces = \emptyset$ // *Initialises the set of context traces*
$c_{initial} = (nextID, INITIAL, -1, true, \{\}, empty)$ // *Creates initial context*
$nextID = nextID + 1$
$CT = CT \cup \{c_{initial}\}$ // *Adds the initial context to the CT*
for all $t \in Traces$ **do**
  $cs = empty$ // *Initialises the call stack*
  Create new context trace *ctrace*
  for all annotations $an = (t_a, bid_a, cp_a, val_a, v_a, act_a) \in t$ **do**
    if t_a is an enter-context annotation **then**
      if $\exists c = (cid_c, cp_c, bid_c, val_c, v_c, cs_c) \in CT$ s.t. $bid_c == bid_a \wedge cp_c == cp_a \wedge$
        $val_c == val_a \wedge (v_c \cap P) == (v_a \cap P) \wedge cs == cs_c$ **then**
          $append(ctrace, cid_c)$ // *Context found and added to context trace*
      else // *New context found*
          $newID = nextID$
          $nextID = nextID + 1$
          $newc = (newID, cp_c, bid_c, val_c, v_c, cs \cap P)$
          $CT = CT \cup \{newc\}$ // *Adds new context to the table*
          $append(ctrace, newID)$ // *Adds new context to the context trace*
      end if
      if t_a is related to a method **then**
          $push(cp_c, cs)$ // *Adds the method name to the call stack*
          $append(ctrace, cp_c)$ // *Adds the method name to the context trace*
      end if
    else
      if t_a is an exit-context annotation $\wedge$ t_a is related to a method **then**
          $pop(cs)$ // *Removes the top method name from the call stack*
      else
        if t_a is action annotation **then**
            $append(ctrace, act_a)$ // *Adds the action name to the context trace*
        end if
      end if
    end if
  end for
  $CTraces = CTraces \cup \{ctrace\}$ // *Adds context trace to the set*
end for
return $CTraces, CT$
end function

exit-context, or action) and bid_a, cp_a, and val_a to represent, respectively, the BID, the control predicate, and the value of the control predicate recorded in t_a. Variable v_a represents the valuation of the system state present in the annotation, and act_a describes the name of the corresponding action, when applicable. All variables with index c refer to a context c recorded in the context table. Hence, variable cs_c refers to the call stack of context c.

The algorithm initiates with an empty CT, with the initial CID set to 0, and the set of context traces is empty (lines 9–11). The CT is initialised with the initial context, updating the CID counter (lines 12–14). The loop in lines 15–45 represents the processing of each trace. For each trace, the call stack is reset (line 16), a new context trace is created (line 17), and all its annotations are parsed (lines 18–43). Enter-context annotations are processed by comparing their information with contexts already stored in the CT (lines 20–21) to check whether it is a known context. Note that the attribute comparison is restricted to the set of attributes P, received as input. This is necessary because, even though the value of every single attribute is always recorded in the annotations, just a subset of them might actually be used to build the model. By increasing the cardinality of this subset with the addition of other attributes, we modify the level of abstraction of the model, which is the basic idea of our refinement process described in Sect. 4.4. After the comparison, if the context is already in the CT, its CID is added to the context trace (line 22); otherwise, a new entry is created in the CT containing the information from the annotation, and the new CID is added to the context trace (lines 24–28). If the annotation is related to a method, then the method name is added to the call stack and to the context trace (lines 30–32), since the name of the method is used as predicate for contexts representing a method call or method execution. If the annotation is of the type exit-context and is related to a method, then the first method name in the call stack is removed (lines 35–36). If the annotation is an action annotation, then the name of the action is added to the context trace (lines 38–39). When all annotations of the current trace have been processed, the generated context trace is added to the set that will be the algorithm output (line 44). Once all traces from the set have been processed, the resulting set is returned, along with the produced CT (line 46).

Table 1 shows part of the CT generated for the Editor program, based on the trace presented in Fig. 6. No predicate is associated with the initial context, and we use the word INITIAL to represent it. Method names with the prefix `call` identify method calls, whereas method names without this prefix refer to the execution of the corresponding method body. The fourth column of the table presents the evaluation of the control predicate, and the fifth and sixth columns contain the system state (respectively, the valuation of the selected attributes and the state of the call stack). The last column is not part of the CT, but has been added to help understand the mapping from annotations to CT entries. It contains the line numbers of the corresponding annotations in Fig. 6. Hence, the entry with CID 1 contains the information from the annotation in line 1 of the trace, the entry with CID 2 stores the information collected from the annotation in line 2, and so on. Note that the entry with CID 6 corresponds to the annotations in lines 11 and 16 of Fig. 6, which means that the trace shows the same context been reached twice. The entry with CID 0 corresponds to the initial context and is not associated with any annotation, as it is automatically introduced during the creation of the CT.

To better understand the importance of the data component in the context information, consider, for instance, the contexts with CIDs 1, 6, and 11 (derived from the annotations in lines 2, 11, and 24, respectively, of the trace presented in Fig. 6). They all have the same control component (i.e. the same BID, the same control predicate, and the same value of the control predicate), which would indicate that they represent the same context. However, when we analyse the data component, we see that they have different values for the attributes. Hence, from the control perspective, they are the same context but, when we add the data component, they represent three different contexts, and for this reason, there is one entry in the CT for each one of them.

Part of the context trace created based on the trace shown in Fig. 6 can be seen in Fig. 7, where the context trace can be read from top to bottom. CIDs are preceded by the symbol # to differentiate them from action names. CID `#0` of the context trace corresponds to the initial context (first entry of the CT in Table 1). The CID `#1` was created based on the first annotation of the trace presented in Fig. 6. Similarly, the second and the third lines of the trace were translated, respectively, into the CIDs `#2` and `#3` of the context trace. The fourth and fifth lines of the trace were generated when the program reached the call to method `open`. The enter-context annotation generated the context identified as `#4`. Because every method call also represents an action of the system, the same annotation originates the action name `call.open` as well. CID `#5` represents the entry point of method `open`. As happened with the method call, this annotation also creates a new context and includes an action name (`open`). CID `#6` represents the annotation in line 11 of the original trace, describing the situation where the main loop of the program has been reached again, but the values of the attributes have changed, thereby creating a new context. Note that the occurrence of action `incorrectCmd`, our user-defined action recorded in the annotation in line 13 of the trace in Fig. 6, causes the action name to be added to the context trace. Because this action signals an incorrect input value, the next enter-context annotation (line 16 of Fig. 6) contains the same context information as the one in line 11. This means that they represent

Table 1 Example of context table

CID	Predicate	BID	Value	Attributes	Stack	Annotation lines
0	INITIAL	−1	T	{}	⟨⟩	–
1	(cmd != 4)	18	T	{false, true}	⟨⟩	1
2	(0)	17	0	{false, true}	⟨⟩	2
3	(!isOpen)	11	T	{false, true}	⟨⟩	3
4	call.Editor.open	4	T	{false, true}	⟨⟩	4
5	Editor.open	19	T	{false, true}	⟨call.Editor.open⟩	5
6	(cmd != 4)	18	T	{true, true}	⟨⟩	11, 16
7	(cmd)	17	5	{true, true}	⟨⟩	12
8	(1)	17	1	{true, true}	⟨⟩	17
9	call.Editor.edit	5	T	{true, true}	⟨⟩	18
10	Editor.edit	20	T	{true, true}	⟨call.Editor.edit⟩	19
11	(cmd != 4)	18	T	{true, false}	⟨⟩	24, 34
12	(2)	17	2	{true, false}	⟨⟩	25
13	(isOpen)	12	T	{true, false}	⟨⟩	26
14	call.Editor.print	6	T	{true, false}	⟨⟩	27
15	Editor.print	21	T	{true, false}	⟨call.Editor.print⟩	28
16	(3)	17	3	{true, false}	⟨⟩	35
17	(!isSaved)	13	T	{true, false}	⟨⟩	36
18	call.Editor.save	7	T	{true, false}	⟨⟩	37
19	Editor.save	22	T	{true, false}	⟨call.Editor.save⟩	38
...	...	...	...	...	...	...

```
#0
#1
#2
#3
#4
call.open
#5
open
#6
#7
incorrectCmd
#6
#8
#9
call.edit
#10
edit
#6
#11
#12
#13
call.print
#14
print
#6
#15
#16
#17
call.save
#18
save
...
```

Fig. 7 Example of context trace

the same context; thus, CID #6 is again added to the context trace. The rest of the trace was processed following the same procedure.

4.3 Model generation

As previously stated, we use context information to generate LTS models. They describe the actions that trigger state transitions, no matter what the state represents. Hence, they require only the information about the valid sequences of actions. Because we use contexts, which comprise control flow and data information, thus giving a meaning to states, we need an intermediate structure that can deal with both actions and states (i.e. contexts) to guarantee that only valid behaviours are included in the model. This structure helps us understand the valid sequences of actions based on the contexts and, this way, guarantee that only behaviours that respect the correct transitions between contexts are included in the model. Once these valid sequences have been determined, we can ignore the meaning of the states, thus producing an LTS.

The formalism we have adopted is labelled Kripke structure (LKS), described as follows, based on the definition presented in [10]:

Definition 7 *Labelled Kripke Structure.* A *Labelled Kripke Structure* (LKS) $K = (S, s_i, P, \Gamma, \Sigma, T)$ is a model where:

- S is a finite set of abstract states,
- $s_i \in S$ represents the initial state,
- P is a finite set of attributes used to label states in S,
- $\Gamma{:}S \rightarrow N^P$ is a state-labelling function, where N is the sum of the ranges of all attributes in P,
- Σ is a finite set of actions, i.e. an *alphabet*, and
- $T \subseteq S \times \Sigma^+ \times S$ is a transition relation.

Our definition slightly differs from that presented in [10] in that we use attributes instead of propositions. Nevertheless, the only difference is that attributes are not always boolean variables and, consequently, may have a wider range of values than that of propositions. Another difference is that we use a singleton set of initial states, which is compatible with the LTS that will be ultimately generated (see Definition 1). One last small difference is that we allow a sequence of actions to label transitions. This represents the possibility of more than one action occurring in between two consecutive contexts. The use of sequences of actions does not affect the LKS definition, because a transition is still a connection between two states, which is labelled with actions from the alphabet. In our case, this label can be compound, comprising all actions that can happen in between two states.

4.3.1 LKS construction

Algorithm 2 describes the generation of an LKS model from a set of context traces, using an alphabet and a set of attributes as parameters. In our example of the editor, we used the alphabet Σ = {`open`,`edit`,`print`,`save`,`close`}, `incorrectCmd`}, the set of attributes P = {`isOpen`}, `isSaved`}, and the context trace presented in Fig. 7, which was created by the execution of Algorithm 1 using the trace in Fig. 6 and the same set P of attributes .

Lines 9–12 initialise all the necessary variables, which include the variables that control the initial state of the model, the current state, and the previous state. These two last variables determine how to build a transition (from `previousState` to `currentState`. The loop from line 13 to line 43 describes how each context trace is processed. Each component of the context trace is parsed and identified (loop from line 15 to line 35). CIDs represent contexts and are mapped to states of the LKS (line 18), whereas actions are mapped to labels of the model. Whenever a CID is found in the context trace, it is checked whether it is the first context found, so that variables `initialState` and `previousState` are set (lines 19–20). If it is not the first context, it means that there has been a context found before, and variable `previousState` contains its CID. As a consequence, a new transition should be added, connecting the previous state to the current state. The transition between these states is labelled with the sequence of actions identified in between them when parsing the context trace (lines 31–33) or with the empty sequence if no action was found (lines 22–23). After creating the new transition (line 25), the sequence of actions is reset (line 26), and the new transition is added to the set (line 27). Lines 36–38 describe the addition of a state called $FINAL$, which is used to represent the termination of the execution and is added when the first context trace has been completely parsed. The state corresponding to the last identified context is connected to $FINAL$, and an infinite loop of an artificial action $\_EXIT$ is added to it (lines 39–42 of Algorithm 2). Although there are systems that do not terminate, we cannot really tell when the termination was successful or not. Hence, we use state $FINAL$ to indicate how far we were able to identify that the execution could go. The loop labelled with action $\_EXIT$ (line 41) prevents that deadlock alarms be generated in case the model is used for verification. The user can, later on, manually modify the model to eliminate the $FINAL$ state, if necessary. The resulting set of states and transitions is used to build the LKS, considering the provided alphabet and the set of attributes (line 44).

4.3.2 Final model generation

Finally, the LKS model is translated into a *Finite State Process* (FSP) [45] description. FSP is a process algebra for describing LTS models that provides an action prefix operator (`->`) e a choice operator (`|`). In FSP, components of a system are described in terms of processes, where each *process* represents the execution of a sequential program. Following the semantics of LTS, the behaviour of a process is represented as a sequence of actions. FSP also allows the definition of subprocesses, which describe partial behaviours of processes.

We translate an LKS model $K = (S, s_i, P, \Gamma, \Sigma, T)$ built by Algorithm 2 into an FSP description in the following manner:

- A process definition $Proc(K)$ is created to represent the behaviour of K;
- For each state $s \in S$, a subprocess $SubProc(s)$ is included in $Proc(K)$;
- For each transition $(s, \alpha, s') \in T$, where $\alpha = \langle a_1 \ldots a_n \rangle$ and $a_1, \ldots, a_n \in \Sigma \cup \varepsilon$, the behaviour ($a_1$ `->` ... `->` a_n `->` $SubProc(s')$) is added to $SubProc(s)$. Transitions where $\alpha == \langle\rangle$ are labelled with the empty action `null`, which is our representation of the empty sequence ε in the generated FSP description;
- Alternative behaviours of a subprocess are defined using the choice operator.

Figure 8 presents the FSP description generated for our example of the editor. Subprocess $Q0$ represents the initial state. It can be seen that our models may have non-

Algorithm 2 Algorithm that creates an LKS model from a set of context traces.

```
1: function CREATELKS(CTraces, P, Σ)
2: Inputs:
3: CTraces: finite non-empty set of context traces,
4: P: finite set of attributes,
5: Σ: finite non-empty alphabet
6: Outputs:
7: K: an LKS model
8:
9:     State initialState, previousState, currentState
10:    Set of states S = ∅
11:    Set of transitions T = ∅
12:    Sequence of actions α = ⟨⟩
13:    for all context traces ctrace ∈ CTraces do
14:        initialState = previousState = currentState = −1 // Initialises control variables
15:        for all entries e ∈ ctraces do
16:            if e is CID then
17:                currentState = e                    // Sets CID as current state
18:                S = S ∪ {e}                         // Adds state to the set
19:                if initialState == −1 then          // If it is the initial context...
20:                    initialState = previousState = e // ...sets the necessary variables
21:                else
22:                    if α == ⟨⟩ then                 // If the sequence of actions is empty...
23:                        append(α, ε)                // ...adds the empty action to the sequence
24:                    end if
25:                    Create transition t = (previousState, α, currentState)
26:                    α = ⟨⟩                          // Resets the sequence of actions
27:                    T = T ∪ {t}                     // Adds transition to the set
28:                    previousState = currentState // Updates previous state variable
29:                end if
30:            else
31:                if a ∈ Σ then
32:                    append(α, a)                    // If action is part of the alphabet, adds it to the sequence
33:                end if
34:            end if
35:        end for
36:        if FINAL ∉ S then
37:            S = S ∪ {FINAL}
38:        end if
39:        Create transition t_f = (currentState, α, FINAL)
40:        T = T ∪ {t_f}
41:        Create transition t_e = (FINAL, ⟨_EXIT⟩, FINAL)
42:        T = T ∪ {t_e}
43:    end for
44:    Create model K = (S, initialState, P, Γ, Σ′, T), where Γ : S → N^P, with N representing the sum of the
       ranges of all attributes in P, and Σ′ = Σ ∪ {ε}
45:    return K
46: end function
```

deterministic choices, in particular involving action `null`. This is usually a consequence of the chosen alphabet and of the level of abstraction. If a transition should be labelled with an action that is not part of the alphabet of the model, then it will be turned into a `null` action and will be ignored when processing the context traces. Therefore, if multiple alternative behaviours of a state involve actions in this situation, we end up having the non-determinism caused by action `null`, such as the one in the subprocess $Q6$. With the FSP description of the editor, we used the LTS Analyser (LTSA) [45] to produce the graphical representation of the LTS shown in Fig. 9. Note that, a hiding operation [45] has been applied to action `null` to simplify the model. We also have hidden the actions related to method calls, so that there is only one action to represent the execution of each method. To allow a better visualisation of the model presented here, we also applied a minimisation operation and made the model deterministic using features provided by the LTSA tool.

Even though it is based on a single trace, this model already shows some correct relations between actions of the editor, according to its implementation (Fig. 1). For example, action `save` can only happen after an occurrence of action `edit`. Moreover, the model also describes that `open` must happen before any occurrence of the other actions. This model also includes behaviours that were not described in the trace, such as the possibility of repeating the command `print` infinitely often once a document has been opened. In fact, the trace did not even include a sequence of two consecutive actions `print`. This additional behaviour was included in the model because the context trace shows that this action

happens in between two occurrences of the same context (indicating a loop) and that this action is always enabled after a file has been opened (when attribute `isOpen` is true). If on one hand all the behaviours in the model are valid behaviours of the real system, on the other hand the model does not include many other valid behaviours. As commented before, after building the model, it is possible to add new behaviours to it using the same CT build for the initial model and, of course, considering the structure of the original model. For instance, consider a new trace generated by the execution of the sequence of inputs that cause the editor to open a document, edit it twice, save it, and then exit. The first annotation of the trace, describing the while statement at the beginning of the execution, would be the same as that of the trace presented in Fig. 6. This means that, when comparing the context information from this annotation with the entries of the CT, a match would be found, including the CID 1 in the context trace. The second annotation, related to the switch statement, would also be the same, thus leading to the addition of the corresponding CID to the context trace. However, a different context would be found when the second call to method `edit` would be identified. It would contain context information similar to the entry identified by CID 9 in the CT (see Table 1), but it would differ on the value of attribute `isSaved`. Hence, a new entry would be created in the CT, and a different CID would be added to the context trace. The new LTS, including all the original behaviours plus the behaviour described in the new trace, is shown in Fig. 10.

```
Editor = Q0,
Q0 = (null -> Q1),
Q1 = (null -> Q2),
Q2 = (null -> Q3),
Q3 = (null -> Q4),
Q4 = (call.open -> Q5),
Q5 = (open -> Q6),
Q6 = (null -> Q7
|null -> Q8
|null -> Q20),
Q7 = (incorrectCmd -> Q6),
Q8 = (null -> Q9),
Q9 = (call.edit -> Q10),
Q10 = (edit -> Q11),
Q11 = (null -> Q12
|null -> Q16),
Q12 = (null -> Q13),
Q13 = (null -> Q14),
Q14 = (call.print -> Q15),
Q15 = (print -> Q11),
Q16 = (null -> Q17),
Q17 = (null -> Q18),
Q18 = (call.save -> Q19),
Q19 = (save -> Q6),
Q20 = (null -> Q21),
Q21 = (call.exit -> Q22),
Q22 = (exit -> Q23),
Q23 = (null -> Q24),
Q24 = (null -> Q25),
Q25 = (call.close -> Q26),
Q26 = (close -> Q27),
Q27 = (null -> FINAL),
FINAL = (_EXIT -> FINAL).
```

Fig. 8 Example of created FSP description

Note that the only difference to the original model is the possibility of executing `edit` after a previous`edit`. As it happened with action `print`, after executing the second `edit`, the system remains in the same context (the value of attribute `isSaved` has already changed to `false`), and for this reason, the model shows that the document can be edited an arbitrary number of times after the first `edit`, keeping the system in the same state.

4.4 Refining models

The level of abstraction of our models can be adjusted by modifying the set of attributes considered in the data component of contexts. Hence, whenever a new attribute is added to this set, the level of abstraction is decreased, and the model gets closer to the implementation of the system. The impact of a refinement is, therefore, that contexts that were identified before as the same (i.e. there was no context information that could distinguish them) may now be seen as distinct. Refinements are necessary to achieve model correctness in situations where the model is so abstract that some important characteristic of the system behaviour is not represented due to the absence of the specific information.

As an example, consider the model in Fig. 11, which is a model extracted from the code in Fig. 1, based on the same trace used to produce the models presented in Sect. 1, and created with a set of attributes containing only the value of

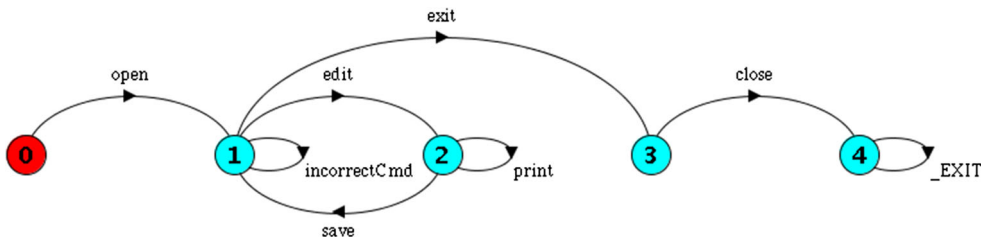

Fig. 9 LTS model generated from the FSP description

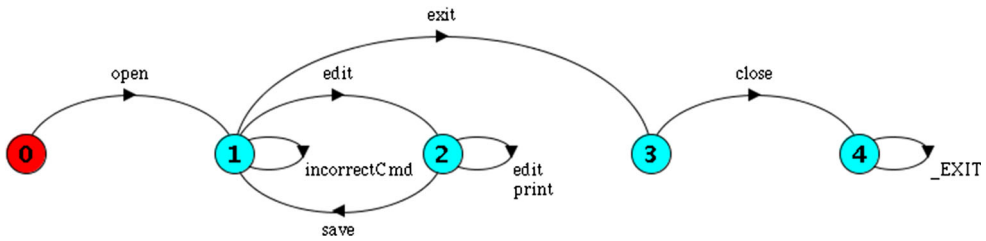

Fig. 10 LTS model with additional trace

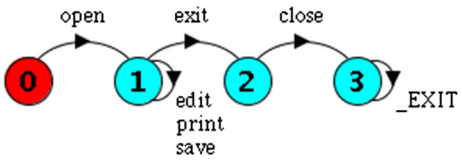

Fig. 11 Editor model with set of attributes $\{isOpen\}$

attribute `isOpen`. It correctly describes the known property that no action can be taken prior to `open`. However, the other known property, which states that `save` can only occur as a response to an `edit`, is violated. In this case, we would need to include the value of attribute `isSaved` to achieve correctness, thus producing the model presented in Fig. 4. Although the refinement idea is quite simple, it is key to the process to identify which attribute should be used to refine a model.

We do not currently provide an automated refinement process, which means that the identification and the inclusion of the necessary attribute has to be done manually. In our experiments, however, we have followed a process similar to the ideas of the CEGAR approach [12]. The requirement to use this process is the existence of a specification of the system. Having a specification, we provide the model and the specification (as a property automaton or formula in LTL [46]) to the LTSA and check whether the model violates the specification. If a counterexample is found, we then use it to identify the necessary attribute according to the following steps:[5]

1. We use the counterexample to traverse our model from the initial state to the state where occurs the specific action that, based on the behaviour prefix, violates the specification;
2. We then execute a backtracking on the specific path of the model where, for each state in the path, we check the control predicate of the corresponding context in the CT;
3. When we find a control predicate that mentions an attribute a that is not in the system state, we add a to the set of attributes for the model refinement.

[5] Considering that it is not a real violation, i.e. it is a false alarm.

The heuristic behind this process is that the invalid path might exist because contexts did not include some information that would prevent it from being executed. This information would most probably be related to some test about the value of an attribute. When the value of this attribute is considered, contexts can correctly identify it as infeasible depending on the current value. In our example, checking the model in Fig. 11 against a property that states that `save` can only occur if `edit` has previously occurred generates the counterexample ⟨`open`,`save`⟩. Looking at the model, we identify that the following path, alternating states and actions, leads to the violation: $Q0 \overset{null}{\rightarrow} Q1 \overset{null}{\rightarrow} Q2 \overset{null}{\rightarrow} Q3 \overset{null}{\rightarrow} Q4 \overset{null}{\rightarrow} Q5 \overset{open}{\rightarrow} Q6 \overset{null}{\rightarrow} Q15 \overset{null}{\rightarrow} Q16 \overset{null}{\rightarrow} Q17 \overset{null}{\rightarrow} Q18 \overset{save}{\rightarrow} Q6$. Traversing this path backwards, from Q18, we discover that Q16 represents the point of the code where it is checked whether the document has been modified, thus enabling or not the execution of `save`. Its control predicate is $!(isSaved)$, indicating that the value of attribute `isSaved` determines the feasibility of this path. Consequently, we add attribute `isSaved` to the system state, refining the model and eliminating the violation.

As we commented before, this is just a heuristic for identifying attributes for refinement. It might happen that we backtrack all the way to the initial state without finding a suitable attribute. This could occur because context information might not be enough to represent the code behaviour when it depends on the value of local variables, as one of the case studies in Sect. 6 shows. Hence, even though this heuristic has been quite effective, it is restricted by the limitations of our abstraction, and we need to further investigate better ways of finding appropriate refinements. Nevertheless, we aim to improve it and eventually define an automatic process.

4.5 Contexts in concurrent systems

Thus far, we have discussed the extraction of single models, describing the behaviour of an isolated process (component). However, systems are normally formed by several processes that execute concurrently and may interact. Hence, we provide, as part of our approach, a strategy to create models of concurrent systems. This strategy involves three steps:

1. We use the process previously described to create a model for each single component;
2. When all the models have been created (ideally, they are correct and complete abstractions of each component's behaviour), we define how the components will synchronise to represent their interactions;
3. We apply a parallel composition of the models of the components to create a composite model.

As our model is based on actions, we use these actions to represent local behaviour as well as interactions between models. A *local action* of a process is an action that is visible only to the process itself and, thus, does not affect the execution of other processes (e.g. an internal method call). A *shared action*, on the other hand, represents an interaction between processes, defining when two or more processes synchronise. We use the *parallel composition* operation of the LTSA tool to generate a composite model. This operation is based on the CSP definition [31], considering local and shared actions, characterised by a mechanism of *synchronisation* on shared actions. Therefore, given two LTS models $M1$, with alphabet $\Sigma(M1)$, and $M2$, with alphabet $\Sigma(M2)$, if there exists an action with name a that is in $\Sigma(M1) \cap \Sigma(M2)$, then $M1$ and $M2$ will synchronise on a. In practice, this means that the two processes need to execute the action simultaneously, corresponding to a method invocation. Whereas shared actions cause synchronisation between models, the execution of local actions occurs independently. For this reason, their execution in composed models is described following the *interleaving semantics* [31]. Therefore, they are executed one at a time, in any order.

This strategy of creating one model from each component and then using parallel composition to create a global model provides the possibility of running analyses on isolated models, on partial combinations of models (integrating just a subset of processes), or on the entire system, thus exploring different scenarios. Moreover, it supports the possibility of reuse, as a model of a certain component can be used in several different composed models, just as, for instance, the corresponding component could be used in different projects. Evolution is also supported, because if a component of the system is modified, there is no need to recreate the models of all the other components; only the model of the modified component has to be extracted again, unless some shared action has been affected, in which case the synchronisation between processes would have to be checked.

Modelling synchronisation Process synchronisation in programming language implies some type of blocking mechanism. The most simple type of blocking is method invocation, where the caller has to wait for the callee to finish executing the method and return the control to it. Hence, the caller process blocks to wait for a response. Because we mark the beginning and the end of method calls and method bodies, we have the possibility of using this information to model blocking situations. As a default, the actions in our models represent the beginning of the execution of the method (*call mode*). However, when processes synchronise, the called method might not execute right away (e.g. if the method is guarded by a wait-notify mechanism) or it might take some extra actions from the callee process before it responds to the caller. To represent these situations, we can associate an action to the beginning of the method and another to the end, thus describing the point where the processes synchronise with the caller invoking a method from the callee, blocking the caller, and the point where the callee responds, releasing the caller. We call it an *enter-exit mode*, where, for a method m, its beginning is represented by an action $m.enter$ and its end, by an action $m.exit$. Although this makes it explicit in the model the blocking and releasing points, it significantly increases the number of states and transitions, since there are two actions for each method executed. Another option to represent a blocking situation, but in an implicit way, is to use a *termination mode*, where we include in the model only the actions marking their end. This way, the actions labelling transitions show the effective execution of the method. These three types of models allow the user to define different visions of the program execution, and the representation of synchronisation points, and choose the most appropriate for their needs.

Example As an example, we present a program based on the bounded buffer system described in [29]. The system is composed of three processes: a producer, a consumer, and a buffer. The producer and the consumer run as *active processes*, i.e. they have their own thread, whereas the buffer is a *passive process*, which receives calls from the other processes. These models were extracted executing the system considering different scenarios involving the number of items produced/consumed and the order in which the processes start. The (minimised and deterministic) LTS models of the producer, the consumer, and the buffer are shown, respectively, in Figs. 12, 13, and 14. The producer and the buffer synchronise on shared actions `put` and `halt`, whereas the consumer and the buffer synchronise through the execution of actions `get` and `halt_exception`, where this last action represents the generation of a `HaltException`, which happens when the consumer is still trying to read from the buffer, but the producer is no longer active and the buffer is empty. In this example, we used the *termination mode*, which means, for instance, that the action `get` in the model means that the corresponding method has been executed. Hence, the model of the buffer does not allow method `get` in the initial state because it cannot be thoroughly executed. Note, however, that it does not mean that method `get` cannot be invoked by the consumer, but rather that, if invoked when

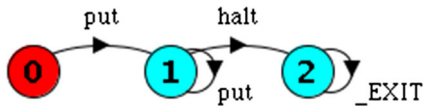

Fig. 12 LTS model of the producer

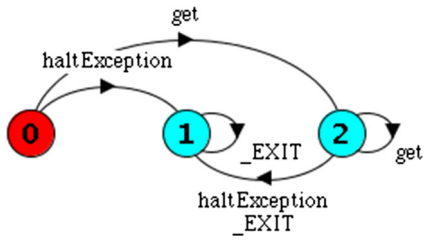

Fig. 13 LTS model of the consumer

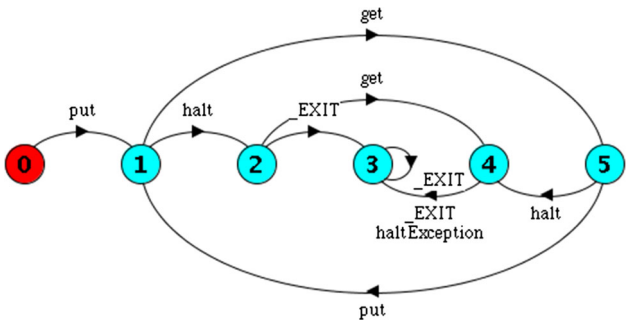

Fig. 14 LTS model of the buffer

there is nothing to consume from the buffer, it will start and block until an item is produced and put in the buffer (transition from state 0 to state 1). Based on these models, we could build the composite model `||BoundedBuffer = (Producer || Consumer || Buffer)` and use it to analyse the global behaviour of the system considering the local and shared actions.

Comments on modelling concurrency Using the call, termination, and enter-exit modes to represent synchronisations allows us to deal with different types of mechanisms used in programming languages. Our experiments have shown that we can correctly describe blocking situations using the different representations, specially in combination with context information, which includes the stack of blocked methods. This way, we can detect when a method is executed in a normal context (i.e. the method is called and executed without any blocking) and when the same method blocks during its execution or executes while another method is blocked. This type of information is not considered in other approaches that extract models from code. Another situation we found during our experiments was the need to mix the types of modes. In some situations, for example, we used the *termination mode* to represent passive processes, as they normally require synchronisation and include some blocking mechanism, and the *call mode* for active processes. The reason for that is that, from the active processes' perspective, when they call a method, they remain blocked until the method has been completely executed. Hence, focusing on the termination does not improve the model, but rather might make it confusing due to the order of the actions. Note, however, that composing models using different meanings for their actions does not prevent them from correctly representing the behaviour of the system. Since, when calling a method, a process does not need to know whether it will have a quick return or remain blocked for a while, it makes sense that they see it as just any other method call. In this situation, it is the process receiving the call who has to know how to handle it. This control has to appear in the model, thus the different model representations.

4.6 Tool support

The LTSE tool[6] partially automates the model extraction process. It implements the part of the process related to the processing of traces to collect context information, the storage of this information in a context table, the creation of an implicit LKS model, and the subsequent generation of an FSP description. It accepts inputs from a command line and generates results to the standard output. These results include messages of successful completion of tasks, error messages, and the contents of the context table created during the execution. The inputs to the tool are a list of action names (alphabet), a list of attribute names (system state), and a list of file names (files containing the execution traces). As it is implemented in Java, the only real requirement to execute the LTSE tool is the presence of a Java Virtual Machine. An additional requirement would be the installation of the TXL engine[7] to automatically instrument the source code. However, as commented before, it is not essential. Any form of instrumentation—even manual—may be used, provided that the appropriate annotations are introduced in the code following the predefined patterns.

If annotations from different instances of the same class are found in file, they are split so as to separate the particular behaviour recorded for each individual instance. However, the LTSE will eventually merge these behaviours into a single process guided by context information so as to produce a general model of the class, showing all the alternative observed behaviours. Annotations of different classes are also split, which means that the tool is able to work with files containing annotations from various classes and produce separate models for each different class identified during the parsing of the files. The representation of actions regarding the execution of method bodies and method calls can be selected

[6] Available at http://www.inf.ufrgs.br/~lmduarte/doku.php?id=ltse.

[7] Available from http://www.txl.ca.

according to what the user would like to observe, using a parameter of the LTSE tool. As we commented in Sect. 4.5, there can be three ways of representing the behaviour of a component. The *call mode* (option `-c`) associates one action to every enter-context annotation of the types `CALL_ENTER` and `MET_ENTER`, thus representing the call of a method and beginning of a method body execution, respectively; the *termination mode* (option `-t`), on the other hand, creates actions for exit-context annotations of the types `CALL_END` and `MET_END`, representing the return of a method call and end of execution of a method body, respectively; the *enter-exit mode* (option `-e`) combines the other two, representing both the beginning and the end of method calls and method executions, where the actions representing the beginning are marked with the suffix `.enter` and the ones representing the end with the suffix `.exit`. The use of these options has already been discussed in Sect. 4.5 and depends on the type of the process being modelled and on the purpose of the model.

5 Formal foundations

When we identify contexts, we are representing concrete states of the system using *abstract states*. Let Imp be a program with $CFG(Imp) = (Q, q_i, Act, \Delta)$ and a set of possible valuations of the system state $V(Imp)$. A *concrete state* $\theta = (q, v)$ of Imp comprises a control component $q = (bc_q, cp_q) \in Q$, where bc_q is a block of code and cp_q is its corresponding control predicate, and a data component $v \in V(Imp)$. We use $\Theta(Imp) = \{\theta_1, \theta_2, \ldots\}$ to denote the set of all possible concrete states of Imp and $\Omega(Imp) \subseteq \Theta(Imp) \times Act^* \times \Theta(Imp)$ to represent the transition relation between concrete states.

5.1 Mapping from code to an LKS model

Our mapping from the context information collected from Imp to an LKS $K = (S, s_i, P, \Gamma, \Sigma, T)$ involves translating (a set of) concrete states of Imp to abstract states of K and modelling the change between concrete states as abstract transitions in K. This occurs as described bellow:

- Every concrete state $\theta = (q, v) \in \Theta(Imp)$, where $v = \{val(p_1), \ldots, val(p_n)\} \in V(Imp)$, is modelled by an abstract state $s \in S$. This abstract state s is derived from a context ID from the context traces generated by Imp and includes only the values of attributes in a selected set $P \subseteq P(Imp)$, such that $\Gamma(s) = v'$, where $v' \subseteq v$. For this reason, each abstract state s may represent a set of concrete states $\Theta(Imp)' = \{\theta_1, \ldots \theta_x\}$, where $\Theta(Imp)' \subseteq \Theta(Imp)$. These concrete states are indistinguishable when the information to be used for comparison is restricted to system states considering only the attributes in P;
- The initial state $s_i \in S$ models a concrete state $\theta_i = (q_i, v_i) \in \Theta(Imp)$, where $v_i = \emptyset$ and, thus, $\Gamma(s_i) = \emptyset$;
- $\Sigma \subseteq Act$ and, therefore, the alphabet of the model can also be restricted to a subset of that program;
- The transition relation T is defined in the following way. Given a set of attributes $P \subseteq P(Imp)$, let s and s' be two abstract states of K. Abstract state s models a set of concrete states $\Theta(Imp)_s = \{\theta_1, \ldots, \theta_n\}$, such that $\Theta(Imp)_s \subseteq \Theta(Imp)$, where, for $1 \geq l \geq n$, $\theta_l = (q_l, \{v_l\} \cap V(P))$. Abstract state s' models a set of concrete states $\Theta(Imp)_{s'} = \{\theta'_1, \ldots, \theta'_m\}$, such that $\Theta(Imp)_{s'} \subseteq \Theta(Imp)$, where, for $1 \geq j \geq m$, $\theta'_j = (q'_j, \{v'_j\} \cap V(P))$. Let $\alpha = \langle a_1 \ldots a_t \rangle$ be a sequence of actions such that $a_1, \ldots, a_t \in \Sigma \cup \{\varepsilon\}$. A transition $(s, \alpha, s') \in T$ exists in K iff there exists a concrete transition $(\theta, \alpha, \theta') \in \Omega(Imp)$ such that $\theta \in \Theta(Imp)_s$ and $\theta' \in \Theta(Imp)_{s'}$.

This mapping guarantees that no invalid paths of Imp will be included in K, according to the level of abstraction provided by the set of attributes P. Hence, at the selected level of abstraction, there will be no transitions connecting two abstract states if the system does not allow a transition between their corresponding sets of concrete states. Note, however, that this does not mean that infeasible paths will not be part of the model, as the model describes the behaviour of the system at an abstract level. As we discussed in Sect. 4.4, it is possible to decrease the level of abstraction to eliminate some of these invalid behaviours, if necessary. In Sect. 5.3, we formally show that our refinement process is property-preserving.

5.2 Mapping from an LKS to an LTS

As we do not explicitly build an LKS model, we apply a transformation from this intermediate structure, which implicitly includes state labels, to a simpler structure that does not. This process is necessary for the creation of the FSP description. For this reason and for simplicity, we will call this mapping a *state-label elimination* (SLE) process.

Let $K = (S, s_i, P, \Gamma, \Sigma, T)$ be an LKS model of a program Imp (as presented in Definition 7) that was obtained through the previously described translation. Using K, we apply a new translation to generate an LTS model $M = (S', s'_i, \Sigma', T')$. Essentially, an LKS is an LTS where states are labelled with values of attributes using a state-labelling function Γ. Therefore, an LTS can be obtained from an LKS simply by ignoring state labels, i.e. the values of attributes in P labelling states of K are not taken into consideration. This can be done in the following manner:

– Every state $s' \in S'$ corresponds to a state $s \in S$, such that s' is the same as s but without its label, i.e. $\Gamma(s') = \Gamma(s) \backslash P$;
– $\Sigma' = \Sigma$; and
– $T' = T$.

As can be seen, the alphabet and the transition relation do not change when mapping an LKS into an LTS. Based on that, we claim that this mapping is property-preserving when we consider LTL properties that do not predicate over attributes of K, but only refer to actions in Σ. In this restriction of LTL formulas, we follow the ideas presented in [41], where LTL is applied to CSP according to an association of propositions with actions (called ALTL in [25]). Considering this association, the set of propositions of an LTL formula about a certain model corresponds to the set of actions in the model alphabet Σ. In this case, LTL formulas are defined on behaviours (traces) of a model such that a model K satisfies an LTL property ϕ over Σ iff, for all $\pi \in L(K)$, $\pi \models \phi$.

Theorem 1 *Let $K = (S, s_i, P, \Gamma, \Sigma, T)$ be an LKS. Applying the SLE process to K results in an LTS $M = (S', s'_i, \Sigma', T')$ such that, given an LTL property ϕ over Σ, if $K \models \phi$, then $M \models \phi$.*

Proof Let us assume that $K \models \phi$. If K satisfies ϕ, then, for all $\pi \in L(K)$, $\pi \models \phi$. This means that all behaviours in $L(K)$ preserve property ϕ. Remember that a behaviour is a possible sequence of actions, determined by a sequence of transitions labelled with these actions. Hence, the set of behaviours is directly dependent on the alphabet (which defines the actions used to label transitions) and on the transition relation. The transition relation results from the actions enabled in each state, which are associated with outgoing transitions and the destinations of these transitions. Because the alphabet and the transition relation do not change when mapping an LKS into an LTS using the SLE process ($\Sigma' = \Sigma$ and $T' = T$), they share the same set of behaviours, i.e. $L(M) = L(K)$. Consequently, for all $\pi' \in L(M)$, $\pi' \models \phi$, and thus, $M \models \phi$. □

From this, we can conclude that this mapping preserves LTL properties over Σ. Note that we could use either the LKS or the LTS model to check properties. We map from an LKS to an LTS model only because of the formalism used in the tool we have adopted. Also remember that we build an implicit LKS, and therefore, the elimination of state labels in practice only means that we no longer use the CT, but analyse directly the context traces. This means that it does not matter any more how contexts were distinguished (i.e. the context information used to identify them) but only the sequences of context IDs and the actions happening in between these contexts, which will be used to create the FSP description as described in Sect. 4.3.

5.3 Refinement relation

In [10], the following definition is presented for an abstraction relation considering LKS models:

Definition 8 *Abstraction.* Let $K = (S, s_i, P, \Gamma, \Sigma, T)$ and $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ be two LKS models. K_A is an *abstraction* of K, denoted by $K \sqsubseteq K_A$, iff:

1. $P_A \subseteq P$,
2. $\Sigma_A = \Sigma$, and
3. For every path $\lambda = \langle s_1 a_1 \ldots \rangle \in \Lambda(K)$, there exists a path $\lambda' = \langle s'_1 a'_1 \ldots \rangle \in \Lambda(K_A)$ such that, for each $n \geq 1$, $a'_n = a_n$ and $\Gamma_A(s'_n) = \Gamma(s_n) \cap P_A$.

Hence, K_A is an abstraction of K if the propositional language accepted by K_A contains the propositional language accepted by K when the language is restricted to the set of propositions of K_A. Ultimately, this means that K_A is an over-approximation of K, such that $L(K) \subseteq L(K_A)$. Remember that we consider this relation in terms of attributes, which just means that the set of values for each element of state labels may be different from $\{true, false\}$.

Our goal was to demonstrate that our refinement process creates this relation of abstraction between an initial model and a more refined one using attributes, rather than propositions. We now show that our refinement process produces a model K that is a refinement of an initial model K_A, given that K_A has a smaller set of attributes than K. If K is a refinement of K_A, then all properties valid for K_A are also valid for K, but K does not contain some behaviours allowed in K_A, since there is more information to distinguish states that were considered the same in K_A.

Theorem 2 *Let $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ be an LKS model obtained from Imp following our mapping, using the set of context traces $CTraces(Imp)$ generated during the context table construction, and a set of attributes $P_A \subseteq P(Imp)$. If $CTraces(Imp)$ is used with a set of attributes $P \subseteq P(Imp)$, such that $P_A \subseteq P$, then we obtain an LKS $K = (S, s_i, P, \Gamma, \Sigma, T)$ such that $K \sqsubseteq K_A$.*

To prove the theorem, we have to show that all items of our definition of abstraction (Definition 8) are satisfied by our refinement process. Item 1 of the definition is trivially satisfied by our definition of refinement: since we add more attributes to the initial set, it is always the case that $P_A \subseteq P$. Item 2 is also readily satisfied, because we do not alter the alphabet[8] and, thus, $\Sigma_A = \Sigma$.

Proof of item 3 is broken into three separate partial proofs, presented next, considering the focus on showing state abstraction, proving that the set of actions enabled in

[8] Remember that both alphabets also include action ε.

a refined state is a superset of the set of actions enabled in the more abstract state, and demonstrating that the refinement process is path-preserving. These three parts together determine behaviour preservation from the abstract to the refined model with respect to common attributes. After this, we discuss our proof of refinement based on them. In all proofs, we use $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ and $K = (S, s_i, P, \Gamma, \Sigma, T)$ to represent the initial and the refined model, respectively. Note that, to simplify the discussion, we will refer to states of the models rather than to the contexts originating these states. Since there is a one-to-one relation between contexts and abstract states, it does not change the results of the proofs. Therefore, when we talk about a trace, we will treat it as a sequence of states with actions in between, instead of a sequence of contexts. It is also important to mention that, in the proofs, we will use only transitions labelled with single actions, rather than with sequences of actions. This makes the proofs simpler and yet does not affect the results, since we are just using sequences of actions that contain only one action.

Partial Proof 1: state abstraction The first step is to show that every state of K is related to a state of K_A. As states are created based on the labels they receive, we will use the following relation:

Definition 9 *State-Labelling Relation (SL).* Given two LKS models, defined as $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ and $K = (S, s_i, P, \Gamma, \Sigma, T)$, such that $\Sigma_A = \Sigma$ and $P_A \subseteq P$, $SL \subseteq S_A \times S$ is a state-labelling relation such that, given a state $s_A \in S_A$ and a state $s \in S$, $(s_A, s) \in SL$ iff $\Gamma_A(s_A) = \Gamma(s) \cap P_A$.

This relation determines that, for every state in the more abstract LKS, there is a corresponding state in the refined LKS. This relation is determined by the set of attributes labelling these states and their respective values. Considering this definition, we have the following lemma:

Lemma 1 *For every state $s \in S$, there is a state $s_A \in S_A$ such that $(s_A, s) \in SL$.*

Proof Let us suppose a state $s_A \in S_A$ labelled with a set v_A of values of attributes in a set P_A. Let us also define $An = \{an_1, \ldots, an_n\}$ as the set of context annotations in the set of traces $Tr(Imp)$ that refer to the context represented by state s_A. Since the inclusion of new attributes expands the labels used to distinguish states, there can be two possible situations when analysing the context annotations in An if a new attribute p is added to the set P_A, creating a set P:

1. In every context annotation $an \in An$, p has the same value; or
2. p has more than one value registered in context annotations in An.

In situation 1, the addition of p to the set of attributes does not reveal any new state from s_A. Hence, all context annotations in An will result in the inclusion in the model of a single state s labelled with $v = v_A \cup \{val(p)\}$, that is, the addition of the value of p is ignored, s_A and s have the same label and, consequently, represent the same state. Therefore, $\Gamma_A(s_A) = \Gamma(s) \cap P_A$, which confirms that $(s_A, s) \in SL$.

As for situation 2, the inclusion of p does make a difference. Because p has more than one value when analysing annotations in An, given two states $s, s' \in S$, where $\Gamma(s) = v_A \cup \{val(p)\}$, $\Gamma(s') = v_A \cup \{val(p)'\}$, and $val(p) \neq val(p)'$, these states are distinguishable. Nevertheless, it is easy to see that s and s' are the same state when the value of p is abstracted away. Then, if the set of attributes was restricted to P_A, $\Gamma(s_A) = \Gamma(s) \cap P_A = \Gamma(s') \cap P_A$. Hence, $(s_A, s), (s_A, s') \in SL$.

Therefore, every state $s \in S$ is related to a state $s_A \in S_A$ in a way such that, if attributes labelling s and not labelling s_A are ignored, then they represent the same abstract state and, thus, $(s_A, s) \in SL$. □

Partial Proof 2: enabled actions preservation The next step is proving that every action enabled in a state s_A of K_A is also enabled in at least one of the refined states of K related to s_A through relation SL.

Lemma 2 *Given $s_1, \ldots, s_n \in S$ and $s_A \in S_A$ such that $(s_A, s_1), \ldots, (s_A, s_n) \in SL$, $E(s_A) = \bigcup_{j=1}^{n} E(s_j)$.*

Proof Lemma 1 showed that, if an attribute p is ignored, such that $p \in P$ and $p \notin P_A$, then a set of states $S' \subseteq S$ will have the same label as a more abstract state $s_A \in S_A$. When generating K, the only input to the algorithms that changes is the set of attributes. The alphabet remains the same and so does the set of traces $Tr(Imp)$ used to build K_A.

Let us suppose that a state $s_A \in S_A$ originates a set of states S' in K when an attribute p is added to the set of attributes P_A, originating a set P, such that $P_A \subset P$ (i.e. for every state $s \in S'$, $(s_A, s) \in SL$). Because the set of traces $Tr(Imp)$ will also be used to construct K, the effect of using P instead of P_A will be that, in every context trace derived from traces in $Tr(Imp)$, the context represented by s_A will now be identified as one of the contexts represented by states in S'.

Remember that the algorithm creates a transition between two consecutive contexts (states) in a context trace and labels it with the sequence of actions happening in between. Thus, given an action $a \in \Sigma_A$, if there is a transition $(s_A, a, s'_A) \in T_A$, it means that the context represented by s_A and the context represented by a state s'_A happen consecutively in a context trace ctr, created based on a trace in $Tr(Imp)$, and action a occurs in between them. If s_A is replaced in ctr by a state $s \in S'$, then a transition (s, a, s'_A) is obtained, since the sequence of contexts in ctr did not change, but just the states used to

represent these contexts (i.e. the CIDs created when building the CT). This means that if $a \in E(s_A)$ in the more abstract model, then now $a \in E(s)$ in the refined model. Because each refined state in S' will take a share of the transitions of s_A, the union of all actions enabled in states $s_1, \ldots s_n \in S'$ will result in the same set of actions enabled in the more abstract state s_A. Therefore, $E(s_1) \cup \ldots \cup E(s_n) = E(s_A)$. □

Partial Proof 3: abstract path preservation The last proof involves showing that every refined path in K can be mapped into an abstract path in K_A.

Lemma 3 *For every pair* $(s_A, s) \in SL$, *if* $(s, a, s') \in T$, *then there exists* $(s_A, a, s'_A) \in T_A$, *such that* $(s'_A, s') \in SL$.

Proof Given a state $s \in S$, Lemma 1 determines that there exists a state $s_A \in S_A$ such that $(s_A, s) \in SL$. Let us now suppose that there is a transition $t = (s, a, s') \in T$, where $a \in \Sigma_A$ and $s' \in S$. Based on Lemma 2, $E(s) \subseteq E(s_A)$. Hence, if $a \in E(s)$, then $a \in E(s_A)$, and therefore, there must be a transition $t_A = (s_A, a, s') \in T_A$, where the more concrete state s is replaced in t by the more abstract state s_A, which it is related to through relation SL.

According to Lemma 1, s' must be state-labelling related to a state $s'_A \in S_A$. Consequently, if $(s'_A, s') \in SL$, then s' can be replaced in t_A by s'_A just as s was replaced by s_A. This results in a transition $(s_A, a, s'_A) \in T_A$, which is the more abstract representation of transition t, such that $(s_A, s), (s'_A, s') \in SL$. □

Proof of property-preserving refinement

Proof Proving Theorem 2. As a result of Lemmas 1, 2, and 3, every state of the more refined model K is related to a state of the more abstract model K_A through the state-labelling relation, and all outgoing transitions of a state s_A of K_A are preserved in K as outgoing transitions of a set of states related to s_A. Furthermore, every transition of the refined model can be mapped back into an abstract transition. Hence, every path $\lambda = \langle s_1 a_1 s_2 a_2 s_3 \ldots \rangle \in \Lambda(K)$ can be mapped into a path $\lambda_A = \langle s'_1 a'_1 s'_2 a'_2 s'_3 \ldots \rangle \in \Lambda(K_A)$ such that, for $n \geq 1$, $a_n = a'_n$ and $(s'_n, s_n) \in SL$. Therefore, Theorem 2 holds. □

In [10], the authors present a logic that is a superset of LTL, called SE-LTL. They show that, if a property ϕ is expressed in this logic and mentions only actions in the alphabet Σ_A, then if ϕ holds for K_A, then it also holds for K. Based on this and on Theorem 2, we can conclude that:

Corollary 1 *For every LTL property* ϕ *over* Σ_A, *if* $K_A \models \phi$, *then* $K \models \phi$.

Therefore, our refinement process between LKS models preserves LTL properties that consider only actions of the alphabet of the more abstract model. Hence, given that there is a property-preserving refinement relation between two LKS models built with different sets of attributes and that the mapping from an LKS to an LTS model is property-preserving (see Theorem 1), the generated LTS models also have a property-preserving relation between them. This relation is also a refinement:

Theorem 3 *Let* $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ *and* $K = (S, s_i, P, \Gamma, \Sigma, T)$ *be two LKS models such that* $K \sqsubseteq K_A$. *If* K_A *is mapped into an LTS* $M_A = (S'_A, s'_{i_A}, \Sigma'_A, T'_A)$ *and* K *is mapped into an LTS* $M = (S', s'_i, \Sigma', T')$, *then, given an LTL property* ϕ *over* Σ_A, *if* $M_A \models \phi$, *then* $M \models \phi$.

Proof The proof of Theorem 1 demonstrated that eliminating the state labels from an LKS, we obtain an LTS that preserves the same properties. Hence, given an LTL property ϕ over Σ_A, if $K_A \models \phi$, then $M_A \models \phi$, and if $K \models \phi$, then $M \models \phi$. Since Corollary 1 holds, guaranteeing that a refined LKS preserves the same LTL properties of its abstraction when these properties are restricted to actions in the alphabet of the more abstract model, if $K_A \models \phi$, then $K \models \phi$.

Therefore, if K_A preserves ϕ, then so does the LTS M_A derived from it, and so will its refinement K. Because M is an LTS obtained from K through the same property-preserving process that generated M_A from K_A and $K \models \phi$, then M also preserves this property. As a result, if $M_A \models \phi$, then $M \models \phi$. □

Note that, ignoring the state labels, the relation described in Definition 8 is a *simulation relation* [50], where the more abstract model simulates the more refined one. Therefore, it is possible to say that M_A simulates M. This relation between the models guarantees inherent properties of a simulation relation.

6 Case studies

This section presents four case studies developed using our model extraction approach based on contexts. They involve part of the code of a remote agent described in [59], the PipedOutputStream class from the JDK 1.7, the SMTPProtocol class from the ristretto library, and the ThreadedPipeline application presented in [15]. We discuss the results of these case studies[9] considering the use of the approach to produce models of sequential and concurrent systems. Our main goal was to evaluate the applicability and scalability of our

[9] The complete data and results are available at http://www.inf.ufrgs.br/~lmduarte/doku.php?id=ltse.

approach, including our data abstraction and model refinement techniques, as well as its effectiveness in different scenarios. To do so, we focused our choice of case studies on programs whose source code is publicly available and for which there is some type of specification, so that we had a description of the intended behaviour of the program, which could be used as an oracle to evaluate our models.

We initiated all experiments with an empty set of attributes, which means that the initial models considered only the control component of contexts. In some cases, we had to modify parts of the original code to enable the correct automatic instrumentation, due to difficulties we encountered using the TXL language. For instance, we had to split some compound commands into multiple simpler ones (e.g. changing `int i = 1 + m();`, where `m` returns an integer, into the following sequence of commands: `int r; r = m(); int i; i = 1 + r;`). However, these modifications did not, in any way, alter the program behaviour.

RAX This case study involved part of the code of the Remote Agent, which is an AI-based spacecraft controller [29,59] used in a NASA project. The agent executes processes in the form of tasks, which exchange events. There are three classes involved in this part of the code: the `Event` class, which implements the events, and classes `FirstTask` and `SecondTask` that implement two different tasks that exchange signals using events. In [59], a deadlock situation in this system is reported, which was uncovered using the Java-PathFinder (JPF) tool. Hence, we selected this case study to analyse whether our strategy for concurrent systems could be used to detect this error even though we generate a separate model for each element of the system. The models were created using only two traces, considering the two possible orders of initialisation of the tasks, obtained by manually modifying the main class code to force the necessary situations. The produced models were composed in the LTSA tool, where we manually created two instances of `Event`, one associated with (synchronised with) each one of the tasks. When checking the composed model for deadlock-freeness, we obtained a counterexample showing the situation where the `SecondTask` signals an event, notifying all the waiting threads, and, right after that, the `FirstTask` starts to wait for the signal. Because the `FirstTask` missed the signal, it blocks (calls method `wait`). After sending the signal, the `SecondTask` waits for the return signal and blocks as well. Since both threads are suspended, the system reaches a deadlock situation. Note that neither of the traces used to create the models presented a deadlock situation, but the behaviour captured for each component and their composition allowed us to identify the problem.

PipedOutputStream The Java class `PipedOutputStream` implements an output stream that connects to a corresponding input stream (`PipedInputStream`) to exchange data. This code is part of the open source libraries of the Java language, included in the `java.io` package of the JDK 1.7. The work described in [26] presents an iLTS (interface LTS) model as a specification for the `PipedOutputStream`, which was obtained using a combination of an inference algorithm and symbolic execution. The original iLTS model has three states and nine guarded transitions, containing an error state to represent that an exception has been thrown. The guards represent conditions for the transitions, where the most important guard is that of the transition representing the execution of method `connect`, which is successful if the `PipedInputStream` received as a parameter is not null and is not currently connected. We used this iLTS as specification to evaluate the correctness and completeness of our model. To work with this iLTS as specification, we converted it to an LTS, representing the guards and the error states using actions: action `connected` occurs after a `connect` in the initial state if the connection is successful; otherwise, the next action is `exception`, representing the transition to the error state of the iLTS; the same action `exception` was used to represent other situations that lead to errors. This procedure produced the LTS described in Fig. 15, which, when converted to a property, determines that any behaviour not described in the LTS is considered a violation (i.e. it leads to an error state). We created two user-defined actions to represent actions `connected` and `exception`. The first one was placed at the end of method `connect`, representing a successful connection, whereas action `exception` was introduced in all points of the code where an exception could be thrown.

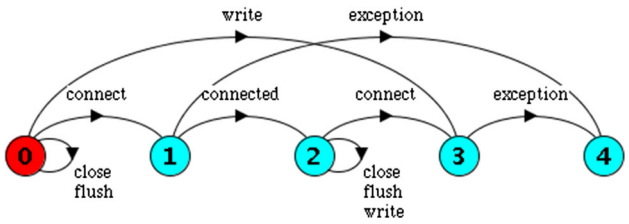

Fig. 15 Property automaton for the `PipedOutputStream` created based on the iLTS shown in [26]

A test suite with a total of 12 JUnit[10] test cases is distributed by Oracle (available with the openJDK source[11]) and the Apache Harmony project.[12] They provide 85.7 % of statement coverage and 72.2 % of branch coverage.[13] We applied the test suite to generate the traces to build an initial model

[10] http://junit.org.

[11] http://openjdk.java.net.

[12] http://opensourcejavaphp.net/java/harmony/org/apache/harmony/luni/tests/java/io/PipedOutputStreamTest.java.html.

[13] Measured with CodeCover: http://codecover.org.

and evaluated the correctness and completeness of this model with respect to the specification. We used the model-checking capability of the LTSA to compare the specification model with the model we produced, resulting in the detection of a violation that allowed the stream to successfully connect even if already connected. We detected that it was a spurious counterexample, since method `connect` has a guard that only allows a successful connection if no previous connection exists. Therefore, we detected a correctness problem due to the level of abstraction of our model. We applied the ideas presented in Sect. 4.4 to identify that adding attribute `sink` to the system state should eliminate the spurious behaviour. The value of this attribute determines the current state of a connection, where a *null* value indicates that the stream is not connected. Because any value other than *null* signals that a connection exists and the set of possible values is, therefore, potentially infinite, we created a user-defined attribute called `isSinkNull` and associated it with the expression $sink == null$. Hence, since what matters is whether `sink` is *null* or not, we created a predicate that checks specifically this. Checking the refined model, a new violation was found: ⟨`connect connected write exception`⟩. The counterexample indicates that, after a successful connection, the execution of method `write` could produce an exception, which is not allowed by the specification model. This behaviour was identified as real, because there are two implementations of method `write` in the class: one that takes only an integer as input—and that indeed never generates exceptions if there is a connection—and another that takes an array of bytes and two integers as parameters, which generates an exception if the array is *null* or the integers provided are outside a certain range. Introducing one user-defined action for each method to differentiate them, we observed that the violation occurred because the specification model was created based on a version with a single method `write`, which only fails if invoked when there is no valid connection. However, the counterexample contained the added action that shows that the violation was related to the other implementation of method `write`. Hence, in this case, not only our model correctly described the behaviour of the code at the defined level of abstraction, but also we could detect a problem in the specification regarding a simplification due to the overloading of a method. In a real scenario, our model would help improve the specification or, at least, the situation could be taken to stakeholders for a decision. To proceed with our experiment, we removed the transitions related to the execution of the version of method `write` that violated the specification, and no more counterexamples were found. Thus, we concluded that we found a correct representation of the system behaviour with respect to the specification.

To analyse the completeness of our model, we did a reverse checking, using the specification as model and our model as a property automaton. This way, we could detect which behaviours allowed by the specification are not present in our model. To avoid differences in alphabet, we hid all actions not mentioned in the specification. We also added action `_EXIT` as an outgoing transition from state 4 of Fig. 15 to a created `FINAL` state, where we introduced a loop of action `_EXIT` so as to avoid a possible alarm of a deadlock situation during verification (this procedure follows the idea presented in Algorithm 2). The first missing behaviour detected was the possibility of executing `close` in the initial state. With this information, we added a new JUnit test case to force the execution of `close` as the first action and produced a new trace. The new trace was added to the existing model, creating a transition from the initial state to the `FINAL` state. We repeated this procedure until no more violations were found, concluding that our model included all behaviours present in the specification. This cycle involving model checking and test cases was executed 22 times, adding 22 new test cases to the test suite, reaching 88.1 % of statement coverage and 83.3 % of branch coverage. As the introduction of new behaviours could have included some behaviour that could violate the specification, we executed a new checking of our model against the specification, where we detected violations related to action `_EXIT`. As commented before, we use this action because we do not know whether, in order to produce finite traces, the program successfully terminated or its execution was interrupted at some point. In this case, since class `PipedOutputStream` describes a passive process, it is expected that the execution could end at any moment (i.e. there should be a transition labelled with action `_EXIT` from every state to the `FINAL` state). Nevertheless, the specification assumed that the execution would only terminate when an exception occurred. Just to guarantee that everything else was correct, we removed all occurrences of action `_EXIT` not allowed by the specification, checked again, and no more violations were found, which means that our model was correct and complete with respect to the specification.

SMTPProtocol The source code for this experiment is an implementation of the client side of the SMTP protocol[14] obtained from the ristretto library.[15] The traces for this case study were generated using the 5 JUnit test cases distributed along with the source code. In [17], the authors present typestate [56] models representing this class. For this experiment, we considered the three models available from the authors' online repository:[16] one *initial* model, built by their tool using the test suite from the ristretto library, an *enriched* model also built by their tool, containing the results of mutant test cases, and a *complete* model manually created by the authors. We

[14] http://tools.ietf.org/html/rfc821.

[15] http://ostatic.com/ristretto.

[16] http://www.st.cs.uni-saarland.de/models/tautoko/materials.html.

applied the same idea used in the previous case study, focusing on determining the correctness and completeness of the extracted model with respect to the specifications. We produced a model based on the traces generated by the set of five test cases. Visually comparing this model with the *initial* model, which contained three states and 10 transitions, we noticed that the specification did not include some actions present in our model (which had 10 states and 13 transitions), such as action `ehlo`, and it did not present the description of exceptional states, as discussed in [17]. We checked our model against this model, resulting in a counterexample that described the sequence of actions ⟨`openPort auth authReceive`⟩. However, one of the test cases provided with the program and used to build the models showed that the mentioned sequence is a feasible behaviour. In this case, in a real scenario, there would have to be a discussion whether the specification was incomplete or the program was incorrect. In any case, our model correctly included this behaviour that was observed during the trace generation, based on the same test suite used by the authors of the specification. We then proceeded to use the *enriched* model as specification. This model had five states and 33 transitions, including behaviours describing situations where an exception could be generated. We modelled a transition labelled with a given action *a* from a state *S* to the exception state (`Ex`) as a transition labelled with *a* from *S* to an intermediate state and a transition from this state, labelled with an action `exception`, to a `STOP` state. Doing the check, we found the counterexample: ⟨`openPort noop`⟩, which was produced because action `noop` did not appear in our model (i.e. it was not on the observed traces). We then created a new JUnit test case to force this sequence and included the produced trace in our model. However, when checked again, we obtained the same counterexample, because the *enriched* model did not include this behaviour as valid. Once again, we found a feasible behaviour of the code that is prohibited by the specification and should, therefore, be analysed. Considering that the authors produced it based on test case generation and specification mining, this is probably because of the specification incompleteness. For this reason, we decided to compare our model with the *complete* model, manually created by the authors, which contained seven states and 89 transitions. Our first verification presented the absence of action `verify` in our model (indicated by the counterexample ⟨`openPort verify`⟩), which led to the creation of a new test case to observe this specific behaviour, which was a feasible behaviour. Checking the model with this additional behaviour against the *complete* model, the same violation was found, presenting this as an invalid behaviour. Nevertheless, the execution of the test case showed it to be feasible. Hence, in comparison with all three models, our model built with a test suite with an equal or smaller number of test cases (i.e. observed traces) could detect discrepancies between what the models (used as specifications) described and the real behaviour of the program. More importantly, even with a small set of traces, our model correctly described only feasible behaviours of the program and provided more coverage than their initial model.

ThreadedPipeline This program is described in [15] and implements the transportation of integer values through stages of a pipeline. Each one of the three stages runs on its own thread, and they are connected to each other by a connector, which is responsible for receiving the values from the previous stage and passing them on to the next . The connector implements a monitor, which means that it includes a *wait-notify* mechanism used in Java for synchronisation. Besides stages and connectors, the program also includes a `Main` class (used to create all elements of the program, start the threads, and provide values to the pipeline) and a `Listener` class, which is connected to the last stage of the pipeline, consuming the value produced by the stages and printing it out. The following two properties proposed and codified in LTL by the authors in [15] were used as specification, since the authors demonstrated them to be valid for this system: (i) *calling method `stop` of a pipeline stage eventually leads it to shut down*, and (ii) *no pipeline stage stops prematurely* (i.e. they do not stop sending new values without receiving the message to stop, after the first stage has stopped). We created one model for each element of the scenario and then manually composed them in the LTSA tool, creating the necessary instances of connectors and establishing the appropriate synchronisations between actions from each element. We did not create a model for class `Main`, as it just starts all other elements. This way, we can analyse the general behaviour of the application and not only the particular scenario defined in the `Main` class. The traces were generated by randomly executing the application 10 times. We again used the model-checking feature of LTSA to check our composed model against the specification. The result of the verification was a violation of the property that ensures that no stage can shut down before the first connector of the pipeline stops. This was not a real behaviour because a `Connector` instance does not allow the next stage to read a value (execute method `take`) prior to having received a value from the previous stage (executing method `add`) or stopping (method `stop`). Therefore, it was identified that it was necessary to consider whether the value of attribute `queue` is less than 0, which is the condition used to block threads (meaning that neither a value has been received or the pipeline has stopped). This attribute was used to refine the model. Verifying the properties against the refined model produced the counterexample ⟨`stage1.start stage2.start stage3.start listener.start c1.add c1.take c2.stop stage1.stop c1.stop`⟩ to the same property, where `stage1`, `stage2`, and `stage3` represent the

pipeline stages, `listener` is the component that reads the value from the last stage, and `c1` and `c2` are the connectors between `stage1` and `stage2` and between `stage2` and `stage3`, respectively. The counterexample shows that `stage2` could be told to stop before `stage1` stops. Analysing the code, we discovered that it could be a feasible behaviour. This would happen in the following situation:

1. `Main` generates a single value, which is 0; hence, it adds this value to Connector `c1` and then stops `c1`;
2. `stage1` reads the value from `c1` and tests it. Because the value is 0, it identifies that the pipeline has shutdown, tells `c2` to stop, and then stops;
3. Depending on the scheduling of the machine, the order in which `Main` tells `c1` to stop and `stage1` stops may cause the sequence presented in the counterexample.

However, considering the original scenario (based on the code of class `Main`), where the values to be transmitted range between 1 and 9 (therefore, no possibility of being 0), this would be an infeasible behaviour. This means that, in this particular scenario, the violation is spurious, whereas in a generic scenario, where one may use different versions of the `Main` class to instantiate and start the pipeline components, this would be a possible behaviour that would violate the property. If we consider the particular scenario proposed by the authors, we would need to control the number of values sent to the pipeline, so that we could guarantee that it would only stop after sending all the values. Hence, we would have to create a model for the `Main` class to count the number of calls to method `add` before sending a stop signal. This model cannot be produced automatically using the extraction approach because the value that controls the repetition comes from a local variable and, therefore, cannot be used as part of the context information. Hence, we would have to build it manually. In order to differentiate the case when a connector has stopped and when it has some value to pass on, we would have to introduce a user-defined attribute that would tell whether the value of attribute `queue` is equal to 0. This way, we would create two equivalence classes of behaviour: one where the connector stops (value of `queue` is 0) and another where the connector is still active (any value different from 0). Carrying out these modifications, another violation was detected during the verification of the new model: ⟨`stage1.start stage2.start stage3.start listener.start c1.add c1.add c1.add c1.add c1.add c1.add c1.add c1.add c1.take c2.stop stage1.stop c1.stop`⟩. This is due to the fact that, even though the model describes that there is more than one value to be transmitted, it does not contain the information about the specific value that is being transmitted; thus, it still includes the possibility that a value 0 could be added to the pipeline, causing `stage1` to terminate right after the first value is received from connector `c1`. Given the information used to identify contexts, this sort of knowledge about the program behaviour is not accessible. Even if each method call is labelled with the concrete value added/read, the limitations of our approach make it not possible to automatically identify that, for instance, after adding 1 to a connector, the `take` action from the next connector would work on the same value 1. Moreover, it is not possible to know that, after a stop, the value returned by calling method `take` must be 0.

6.1 Evaluation of results

Table 2 summarises the results of the case studies in terms of the number of classes involved, the total number of lines of code, the total number of states and transitions in the extracted models, and the physical space used and the total time consumed to build all the models. The space is the sum of the size of the context table and the size of the temporary structures used to store the model before generating the FSP description. The total time considers the time elapsed from the moment that the first log file (the file containing a trace) starts being parsed until the point where the complete FSP description was generated. The number of states and number of transitions for the case studies with a single class (PipedOutputStream and SMTPProtocol) correspond to the model described in the FSP file created by our tool, i.e. it contains the `null` transitions, which can then be hidden to reduce the number of transitions and, perhaps, cause the merging of states. In the case of applications with multiple elements (RAX and ThreadedPipeline), the number of states and number of transitions refer to the composed model, and the total space and time are the sum of the space and time used to build the model of each element.

The case studies demonstrated the applicability and usefulness of our approach for extracting models that can be used to describe the behaviour of a program. Even if not complete, the use of contexts guarantees that they include only valid behaviours considering a certain level of abstraction. For this reason, any violation of a specification is either a real problem or a false alarm caused by an incorrect level of abstraction. Hence, they can serve for property checking to uncover existing errors, such as the one detected in the RAX case study, which could not have been identified by testing alone, and the problems with the specifications in the `PipedOutputStream` and `SMTPProtocol`, created based on inference methods. However, it might be required that they be, at least, complete with respect to the specification in order to check more complex safety properties. Liveness properties might call for a higher level of completeness, nevertheless. The case studies also showed that completeness can be achieved through an incremental

Table 2 Results from the case studies

Program	Classes	LoC	States	Transitions	Space (b)	Time (ms)
RAX	3	60	141	365	3294	941
PipedOutputStream	1	165	34	70	8462	881
SMTPProtocol	1	982	255	269	253,050	1132
ThreadedPipeline	7	96	28,056	146,133	13,992	3315

process, even though it might require multiple steps to reach the desired model. Our strategy for building models of concurrent systems has produced good results, and its greatest advantage is the fact that the user can concentrate on the specific behaviour of each element rather than worrying about the integrated behaviour. This way, as achieving complete coverage of a single component is easier than covering all the combined behaviours, it is possible to generate traces to produce a complete model of each component, which can be checked against local properties, and then combine these models in the LTSA to check program properties. Moreover, because each model is independent of the particular system being analysed, they represent the generic behaviour of the components, allowing the possibility of reuse and exploring different scenarios, as we discussed in the `ThreadedPipeline` case study.

Next, we discuss some aspects of our approach based on the results of the case studies and on our experience with other experiments.

Applicability of the approach Even though our approach currently extracts LTS models from Java source code, it can be used to generate LTS models from programs in other programming languages, as long as the source code is available for instrumentation. The real effort would be to adapt the annotation rules. Our extracted models have been used for program comprehension, validation, model checking, and model-based testing. Due to the possibility of incrementally enhancing the model, this approach could be used as part of an agile development process, gradually constructing the model as the system grows, providing an artefact to be presented to stakeholders. It also can be part of a CEGAR process [12], as we presented in the case studies, although the refinement is still not automatic. The output of our approach could also be adapted to serve as input to tools other than the LTSA, such as Spin [32] and PRISM [40], thus allowing the models to be used for other types of analysis.

Customisation of models Our approach provides a high level of flexibility, allowing the user to customise the model and the process. This way, it is possible to define actions other than methods, select the program variables for the system state, create predicates over program variables to refine the model, define the model alphabet, and select the interpretation of method actions. However, this requires some effort and, in some cases, some expertise. In the case of concurrent systems, the possibility of creating individual models for the components of the system and then composing them might reduce the effort to generate traces, as each component can be instrumented and, for example, tested independently. Because creating test cases for individual classes is relatively simple (it took us just a few seconds to build the test cases for our case studies), this is an advantage. Nonetheless, to compose these individual models, the user needs to know how to correctly synchronise the necessary actions and instantiate each element.

Effort to run the experiments The `PipedOutputStream` was the one that took longer (about 30 min) and was harder to execute. Much of the time and effort was due to the cycles to complete the model based on the specification. Each cycle would take two minutes on average, considering a refinement or a new trace generation and the production of the new model. In general, the instrumentation process, even if manual corrections are required, is simple and takes less than a minute. Trace generation can be done with random executions or using a test suite. Random executions reduce the time to start producing traces, but we have no control over the traces to be observed. Test suites, on the other hand, allow to determine the traces to be generated based on a given coverage criterion (e.g. a testing coverage or a specification coverage). Because we have been working on Java, we can create JUnit test cases, which is a simple task, in particular because these test cases usually only involve instantiating the necessary objects and applying the desired sequence of method calls. For the same reason, creating new test cases based on counterexamples is straightforward. In the experiments with concurrent systems, an additional effort was dedicated to composing the models, which required basic knowledge of the FSP language and semantics.

Complexity to extract models The extraction process is affected by the length, the quantity, and the quality of the traces used to build the model. The length of the trace refers to the number of annotations collected during the execution: the larger this number, the longer it takes to process the trace. As a consequence, the number of traces to be analysed also increases the time to complete the parsing process. However, the quality of the traces—i.e. the coverage of all the possible different contexts that the program can produce—does

not have much effect on the time to extract the model, as all the annotations are processed, one by one, anyway. Nevertheless, it influences the space required to complete the extraction. Remember that each different context means a new entry in the context table, which, in turn, means more space used to store the context information required to construct the model. Hence, there is no direct relation between time and space when building models using our approach. For instance, the `SMTPProtocol` was the case study that consumed more space because of its variety of contexts, but it was not the one that took longer to finalise.

Scalability The two elements that affect the most the space and time required to complete the extraction process are the length of the traces and the size of the CT. The length of the traces determines the number of annotations that have to be processed, but also the type of the annotation defines its effect on the size of the CT and the model. Each new enter-context annotation found produces a new entry in the CT and the inclusion of a new state in the model. Hence, the number of states in the model is exactly the number of different contexts identified in the traces (which is also the number of entries in the CT). Exit-context annotations do not produce any modifications in the context table nor in the model, as they are only used to control the end of contexts. Thus, the maximum number of contexts that can be identified from a trace is equal to the total number of enter-context annotations. Because there is a corresponding exit-context annotation for each enter-context annotation, given a trace with n annotations, there can be identified $n/2-a$ different contexts, where a is the number of action annotations. The number of transitions in the model corresponds to the number of changes in context (each pair of consecutive enter-action annotations, ignoring exit-context and action annotations). Each change in context causes the inclusion of a new transition in the model (labelled with the `null` action if no action happened between the two consecutive contexts). Hence, the maximum number of transitions created based on a trace containing n annotations is $n - 1$. Regarding the time necessary to process the traces, it takes $O(n * m)$ to process n annotations and a context table with m entries. The elements that affect the size of a CT are those structures that originate contexts (i.e. method bodies, method call sites, and selection and repetition statements) and the set of attributes composing the system state. Method bodies and call sites create each at least one entry in the context table. Depending on the observed traces, each guarded command will also create at least one entry in the CT (if only one possible evaluation of the control predicate has been observed, and the set of attributes is empty). Hence, let m be the number of method bodies of a class, c be the number of call sites, and g be the number of guarded commands (selection and repetition statements), the minimum size of the corresponding CT is $m + c + g$. However, in general, the control predicates of guarded commands are either Boolean expressions (structures of the types `if-then-else`, `while-do`, `do-while`, or `for-do`) or expressions over types that can be converted into integers or enumerations (structures of the type `case` or `switch`). Thus, the size of the CT is affected by the cardinality of the set of possible values resulting from the evaluation of expressions used as control predicates, since there can be a different context for each possible different value. Therefore, the maximum number of contexts that a given program can produce with an empty set of attributes is given by $m + c + \sum_{i=1}^{g} rangeOf(g_i)$, where $rangeOf(g_i)$ gives the number of distinct values that the control predicate of the guarded command g_i can assume. If the set of attributes (system state) is not empty, then the maximum number of contexts that a program can produce is $(m + c + \sum_{i=1}^{g} rangeOf(g_i)) * v$, where v is the number of different valuations of the system state. Therefore, the size of the CT grows exponentially according to the ranges of values of the attributes that are added to the system state. For this reason, it is important that only attributes that are really necessary be added to the system state. A sensible approach is to start with an empty set of program variables and add attributes as they are needed to achieve correctness. User-defined attributes also help reduce the possible number of valuations when it is only necessary to evaluate a certain predicate over the value of an attribute.

Termination Even if some variable has an infinite range of possible values, it does not prevent the model extraction process from terminating. All traces given as input to the approach are finite. Hence, we always have, for each trace, a finite number of annotations to analyse to build a model, which means that every context trace created based on these traces will also be finite. Likewise, as the set of traces is finite, the set of context traces is finite as well. Therefore, Algorithm 1 always terminates. Because Algorithm 1 always terminates, producing a finite CT and a finite set of finite context traces, Algorithm 2 also always terminates.

6.2 Threats to validity

The results presented in this section are subject to some threats to validity. We describe these threats below, distinguishing between threats to internal, external, and construct validity.

Threats to internal validity The flexibility of our approach is strongly related to the possibility of the user customising the model. However, too many configurations, most of them manual, can also increase the chance of making some mistake that could affect the final result. Hence, the user has to, at least, have a good knowledge of the modelling formalism, so that they can appropriately adjust these parameters and check

the resulting model to see whether they are correct for the specific purpose. When dealing with concurrent systems, the global model is a composition of the models of the elements of the system. As the composition is also created manually, the choice of which actions to hide/relabel/synchronise is left to the user. Once again, there is a trade-off between customisation and required knowledge to obtain the desired results.

An important issue is the fact that our refinement process is still manual, which means that the user has to define how to select the attributes to be added to the system state. Since the values of program variables may have an infinite range, data abstraction is required to avoid state-space explosion. So far, we provide user-defined attributes as a means of creating predicates over the values of attributes when the actual values are not necessary at the required level of abstraction. However, this data abstraction demands manual effort and knowledge, and has a limited effect if the actual value of an attribute is necessary to achieve correctness. Moreover, as the refinement is manual, introducing new values to the system state may also lead to a state-space problem if the additional attributes are not correctly chosen and data abstraction is not appropriately implemented. Although we provide some insight on how to select these attributes, these are still preliminary ideas and might not be applicable to all situations.

As mentioned at the beginning of this section, in some of the experiments, we had to modify the original source code to break complex commands into multiple simpler ones, due to some limitations in using the annotation language. Although the process itself should not change the general behaviour of the applications, it was conducted manually, and thus, we could have made some mistake.

Threats to external validity In this paper, we applied our approach to a set of four small applications. We have not yet applied our approach to extract models of large-scale systems, which means that we have not yet dealt with a huge number of elements and extremely long traces. However, the number of elements should not be a problem, as each element can be treated separately, extracting each model based on the instrumentation of only the specific target element, generating traces that concern only information about this element. The problem could arise when composing multiple models, as the composed model might get too large. In this case, we offer the possibility of user-defined attributes, which can reduce the space state. Another possibility is to hide away all unnecessary actions, such as local actions that do not interfere on the type of analysis being conducted. In general, issues related to model composition might involve more aspects inherent to the formalism we adopt than have to do specifically with our approach, since we just produce the models that could have been created manually. Our goal was to ease the process of generating the model. As for the length of the traces, it only means that it might take longer to parse them and, perhaps, require more space to store the necessary context information. Nevertheless, depending on the behaviour of the component, a few, short traces might be enough to capture its complete description. The length and the quantity of the traces can be adjusted by focusing on specific behaviours or particular coverage criteria.

Our approach is not well-suited for programs where most of the control flow is based on local variables and/or most of the code behaviour does not involve method calls. If the control flow is based on local variables, it might not be possible to find an appropriate refinement, as we refine models based on (expressions over) the values of program variables. This situation appeared in the `ThreadedPipeline` case study, where a local variable value was an important information regarding the program behaviour. If the program makes little use of method calls, a large number of user-defined actions might have to be introduced to describe relevant events during the execution, such as assignments and results of mathematical expressions, increasing the manual effort.

The requirement of access to the source code limits the applicability of our approach, and the need to introduce annotations in the code causes an effect on the program execution time, which might be relevant in real-time systems. In concurrent systems, the fact that the annotations include the evaluation of expressions and values of attributes may result in a anomalous behaviour being introduced in the model. For instance, if shared variables are used and the access to these variables is not controlled, the value of a variable when the annotation is recorded might be different from the value actually evaluated by the program. This situation did not occur in our experiments, but it is a possibility to be considered. We do not handle local synchronisations, which also makes our approach not suitable for programs that use such mechanism.

Threats to construct validity The effort to produce a complete model with respect to a certain specification/coverage depends on the initial set of traces and the required coverage. Although we could incrementally create a model based on an initial small set of traces (perhaps, just a single trace), the coverage provided by these traces and the desired coverage will determine how many steps will be necessary to complete the task. As this process is not automatic, it significantly affects the effort of producing the models.

In the case studies, we applied different comparisons to evaluate the extracted models. When evaluating completeness, we compared our models to existing specifications. In some cases, these specifications were not presented as an LTS, which required a conversion. This means that our comparisons did not consider the original models, but LTS versions of them. Even though the underlying formalisms are similar, we may have introduced some error in this conversion. Nevertheless, the fact that all the case studies are based on previous results of related work allows us to believe that

such errors, if they existed, would have been spotted. Our choice of type of comparison for each case study may also have influenced our results. However, they seemed appropriate for each case and demonstrated the effectiveness of our approach in different scenarios and the different uses of our extracted models.

7 Related work

Our work was originally inspired on ModEx [34], which proposed to manually insert annotations in C code. The user had to provide a mapping between the C commands to commands in PROMELA, which is the input language of the Spin model checker [32], to allow the extraction of models that could be used for model checking. Unlike the ModEx approach, our mapping from the programming language to the modelling language is predefined and automatic. In [37], the authors propose an approach where the user provides rules to translate an intermediate model, generated from the source code, to the final model. This rules are basically syntactical but incorporate the semantics of a specific domain or application to apply the appropriate abstractions. Hence, whereas ModEx uses purely syntactical rules, they provide a way of associating semantics to these rules. Our rules are related to the structure of programs and can be adapted to different programming languages, just as their configuration rules can be adapted for each application and domain. Although our rules can be seen as basically syntactical, they contain the semantics required to correctly describe the control flow of the program and combine it with dynamic information, which is not considered by their work, as they extract the model based only on static information.

Some techniques guarantee completeness by obtaining the complete CFG of the system [4,11,30]. As expected, this results in an over-approximated abstraction of the system, which can yield a number of false alarms. They rule out these false alarms by applying an automatic refinement process based on *predicate abstraction* [28]. Though we still do not provide automatic abstraction refinement, our refinement process has proved to successfully eliminate some false alarms. This process is simple and, unlike the aforementioned related work, does not require the support of a theorem prover. However, our model extraction process usually leads to the creation of an incomplete model, as we depend on the set of observed traces.

Static checkers, such as ESC/Java [23], also work on the source or compiled code to detect possible run-time errors (e.g. null references, wrong type assignments). As static checkers, they do not require the execution of the system, but have to rely on some sophisticated analysis technique to help the process. In the specific case of ESC/Java, the programmer has to provide annotations using a predefined language in order to mark certain points of interest, regarding invariants and preconditions, and enable a theorem prover analysis. We also use annotations in the code, but they can be automatically inserted and are generic, rather than containing elements that are specific for each program. Moreover, these static checkers only produce reports on detected errors and warnings, whereas we generate a model of the behaviour of the system. Hence, our result is an artefact that not only can be used to document the software in a higher level than their annotations, but also serves for other purposes, such as the analysis of different types of properties.

The Bandera toolset [15] extracts models from Java source code based on provided temporal properties, using a slicing technique to reduce the size of the code, eliminating parts that do not affect the validity of a given property. In common with this approach, we have the possibility of creating user-defined data abstractions to reduce model complexity. Moreover, we also allow the model construction to focus on a property of interest, so that completeness is considered with respect to the property and not to the entire behaviour of the system. Nevertheless, we do not use a reduced version of the code to generate models. This way, we do not need to conduct control and data flow analysis to guarantee the correctness of our models. On the other hand, Bandera is capable of dealing with local information, which can be useful when local variables are relevant for the program behaviour.

We apply the analysis of execution traces, as described in [14], where collected traces were used to infer a model based on techniques such as neural networks and statistical analysis. Generally, similar techniques, based on *automata learning*, such as [1,2,55], share with our approach the dependence on the samples of execution to achieve completeness. However, they usually do not provide means of refining models to improve correctness. Moreover, though the work presented in [47] describes an incremental approach, the increase in completeness of an existing model usually causes the decrease in correctness. This is a consequence of the lack of information about how to combine different traces without creating infeasible behaviours. Such information is provided by our context abstraction; thus, the inclusion of new traces is sound, considering the current level of abstraction.

Aiming to allow a sound state-merging, the work described in [43] combines the samples of sequences of method invocations with values of their parameters to generate *extended finite state machines* (EFSM). They apply a merging technique based on similar possible futures of states, where these possible futures are considered up to a specific, predefined length. Instead of gradually adding information, context information already provides us with more reliable information regarding the possibility of merging different traces correctly, so that improving completeness does not affect the correctness of the models. Moreover, we do not need to limit the length of the paths analysed, since we only need to com-

pare states based on the contexts they represent according to the level of abstraction. This analysis guarantees that, when we merge two states, they represent the same context and, therefore, any paths from these states, however long they might be, are always valid at the defined level of abstraction. The work presented in [61] describes an iterative approach to build EFSMs from a combination of names of events (function calls or I/O events) and values of some variables. They use a data classifier technique to classify each event according to a class of values. The idea is to improve the inference process by adding data information to the events, which means that they enrich the traces. An initial model is built based on the classified traces and a state-merging algorithm, based on scoring pairs of states to find out which ones are more likely to be equivalent, is iteratively applied to obtain more compact and general models. Though the combination of names of events and values of variables resembles our context information, they do not include any control flow information. This means that they can improve the model inference process based on the collected data, but they still can infer incorrect sequences due to the absence of any information about which were the control predicates evaluated to enable each event and at which point of the code the event was generated. For instance, consider a certain state `s` that is reached through the occurrence of an event `e` and with the set of variable values `v`. If there is another state `s'`, which is reached with the same event `e` and the same set of variables values `v`, then the inference process will deem these states equivalent and merge them. However, event `e` might occur in more than one location along the code and even have different effects depending on its specific location. In our approach, on the other hand, because of the control component of contexts, we can safely identify events that are indeed the same (i.e. generated under the same conditions, considering location and variable values). Moreover, also because of contexts, we do not need a state-merging algorithm. Contexts provide the necessary information to identify equivalent states. Hence, we always produce the more general and compact model of the system behaviour that can be obtained from the collected traces (with respect to the defined level of abstraction).

The work of Bodhuin et al. [8], like ours, attempts to extract behaviour models based on traces collected from instrumented Java programs. However, they obtain the models from the Java bytecodes. If, on one hand, it allows their approach to be used even when no source code is available, on the other handm it makes their approach applicable only to Java programs. As mentioned before, the annotation rules we have defined are based on common programming structures and, therefore, can be adapted to any programming language that contains these structures. Hence, if the annotation of source code imposes a restriction, it also provides flexibility and a language-independent approach. Moreover, as they only present examples of their approach applied to single traces, it is not clear whether their approach also considers the combination of multiple traces to create a global model.

The work described in [17] presents an approach that combines test case generation (TAUTOKO tool) with typestate mining (ABADU tool). They use test case generation to systematically enrich dynamically mined specifications and enable the detection of illegal interactions. Our case study about the `SMTPProtocol` class showed that, compared with their work, we obtained a model that contained more behaviours than their initial typestate model, using the same set of test cases. Moreover, with a smaller set of traces, we built models that, when checked against their enriched and complete models, were able to detect feasible behaviours of the code that were not allowed by their models. This is a result of the use of contexts, which allow us not only to generalise over the observed traces, but also guarantee the correct merging of all traces, preserving each original trace in the global model. We used the model of `PipedOutputStream`, presented in [26], as a specification in one of our case studies. Their work describes a combination of the *L* automata learning algorithm* [3] with *symbolic execution* [39] to learn component interfaces. Their models describe the behaviour of a class in terms of the effect of executing methods of the class interface depending on the evaluation of control predicates involving local variables. Hence, they aim to produce specifications on the usage of interfaces of components. Our models show a more complete description of the component internal behaviour, including internal calls (methods not available in the component interface), the control flow, and the component state. As we demonstrated in the case study, this approach can be complimentary to ours, providing a specification that can be directly used to evaluate the correctness and completeness of our models. The work presented in [36] extends the approach described in [26] by improving performance through the more extensive use of concrete executions rather than symbolic analysis, but it does not include any additional improvement related to the model. Approaches for creating object usage models [54,62] aim to create models that describe the correct usage of objects/classes. They resort to inference methods to create their models, either based on source code, examples of code, or observed executions. Our models describe the actual behaviour of the implementation and do not require any inference process to be built. The work described in [63] also derives models from code, but their objective was to create *goal models*, which are more related to design strategies towards defining requirements of the system than involving its actual behaviour.

The approach presented in [6] (which extends the work described in [7]) uses the CSight tool to produce *Communicating Finite State Machines* (CFSM) from logs of execution based on regular expressions provided by users. They deal with logs of concurrent systems and require that vector

timestamps be inserted in the traces to determine the partial order of the concurrent events. Based on that, they create one model for each component, just as we do. Also similar to our approach, the composed model requires that the user provides how the components interact through reliable FIFO channels. Hence, they model interaction in a way similar to CCS [51]. We do not need the user to determine how the components interact (unless some different configuration is necessary), since we use the simple synchronisation on shared actions to describe interactions. As these interactions are basically related to components calling methods of other components, they come naturally in the models of the involved components. To extract our models, we do not require any regular expressions or timestamps to determine the identification of components or the order of events. Because we model each component separately, considering their general execution rather than their behaviours in a particular program, we only need local ordering of events, which is guaranteed by the order in which the annotations are inserted in the traces. Considering that every annotation contains the object ID, we can distinguish the events pertaining to each different component and even separate the behaviour of each instance of a component. Our tool can then merge the behaviours of these instances into a generic description of the behaviours of all instances of the specific type of component.

Our combination of static and dynamic information was inspired on the work of Nimmer and Ernst [52], where the authors combined a tool that dynamically infers invariants from an existing implementation with a static checker. They used the static checker to confirm the results obtained with the dynamic analysis. In our case, the context analysis merges static and dynamic information into a single abstraction, so that we can simultaneously work with these two types of information. For this reason, we can use a single tool to run the analysis and produce the models and better explore the hybrid information.

Work on *Learning-Based Testing* (LBT) [21,48,49] combines model checking with inference algorithms, using a learned model to automatically derive test cases. Hence, the focus is on automating test case generation following an approach similar to CEGAR [12], but directed to testing, where counterexamples represent new test cases and queries to the inference algorithm. Our main goal was not to create models for a specific purpose, such as deriving test cases or model checking, but rather providing an approach for extracting customisable, multi-purpose models. Moreover, unlike LBT, our model extraction process is not based on an inference/learning algorithm; we actually merge observed samples of execution, guided by the context information, to generalise the behaviour of a system over a set of traces, without the risk of introducing infeasible behaviours. Thus, the results of our approach could be used in combination with LBT, providing a model that correctly describes the system behaviour at a certain level of abstraction and that could be generated based on random traces. This means that the initial model would not be a hypothesis model, but a partial representation of the actual system behaviour. The use of contexts would also guarantee that the model would fit well in the incremental characteristic of LBT.

Although we obtained the same result as the analysis using JPF [59] for the RAX case study [29], there are differences in the two approaches. JPF acts directly in the Java bytecodes, executing a customised Java Virtual Machine to obtain run-time data from the code. It gradually explores possible paths by commanding the execution and conducting backtracks when some path has been completed. Because it only stores some run-time information, JPF does not generate a proper model, as it builds only an internal representation that allows it to control the state space and memory usage. We believe that having a model is important not only for verification, but also for many other purposes. Hence, even though JPF is applicable in situations where our approach cannot be used, because it works on bytecodes, we produce models that can be visualised, manipulated, and customised for different types of analysis.

Considering all the discussed related work, we believe that our approach advances the current state of the art by combining the following characteristics:

- Defining an abstraction (contexts) to represent states of a system that, unlike most of the work with traces, provides confidence on the correctness of the model generalised from the set of observed behaviours, without resorting to any type of inference algorithm to determine state equivalence;
- Enabling an incremental approach that can iteratively improve a model, both by refining it (many approaches do not provide support for refinement) in a property-preserving way and by allowing the later addition of new observed traces, which can eventually lead to a complete model, without introducing infeasible behaviours (which is a common problem of dynamic approaches);
- Providing support for the customisation of the model for different purposes and representations of actions. This gives the flexibility that is not present in other approaches, specifically those where the model is tailored for a particular use;
- Allowing the modular extraction of models for concurrent systems, which supports reuse, modular analysis, and the exploration of different scenarios of instances and compositions. These models are composed based on a simple principle, provided by the adopted formalism, which can be directly mapped from programming languages. The use of a call stack as part of the context information also supports the identification of situations that cannot be detected with any other approach, involv-

ing the analysis of the state of a component when methods execute isolated or with pending calls.

8 Conclusions and future work

Our model extraction process generates models based on traces containing context information. They can be used for multiple purposes and customised according to the user's needs. The completeness of these models depends on the quantity and the quality of the observed behaviours and can be gradually improved with the inclusion of more behaviours. The correctness of the models is affected by the set of attributes used as the system state. Improvement of correctness can be achieved by the addition of more attributes to contexts, thus ruling out false alarms. This refinement process is property-preserving provided that the properties only predicate over actions of the more abstract model. The LTSE tool automates the process of creating a representation of an LTS model from a set of collected traces.

Results of case studies have demonstrated that our approach can generate models that are useful for a number of applications, including validation and verification of specifications. The approach is also applicable to concurrent systems in a modular manner, allowing the user to build individual models for elements of the system and then apply parallel composition. The refinement process has proved to effectively eliminate false alarms when the counterexample is produced by the absence of information about some global variable of the system. Although completeness is not always possible to obtain, including only relevant behaviours (e.g. for checking a certain property) improves coverage and tailors the model to the specific purpose.

The manual refinement of models is a limitation of our approach. We intend to implement an automatic refinement mechanism based on the ideas discussed in Sect. 4.4 and the CEGAR process [12]. Moreover, the `ThreadedPipeline` case study demonstrated that, on some occasions, the refinement of contexts does not rule out infeasible behaviours. Hence, we plan to investigate how we could handle these situations.

As future work, we also intend to apply techniques for the automatic selection of test cases based on a specification, such as [22,44], to produce our traces. This would facilitate the identification of which behaviours could affect the specification in order to choose an appropriate test suite. This investigation will also enhance our knowledge on how much results of an analysis using our models can improve and/or complement previous analyses based on testing outcomes. Another possible path to be followed could be the application of *program slicing* [15] or *predicate slicing* [42] to eliminate unnecessary parts of the code and allow the instrumentation and execution of a reduced version of the implementation. Using a specification as the criterion to create the slice, we might be able to more quickly achieve completeness with respect to this specification.

We will analyse whether work on *trace sampling* [53] and *trace compression* [9,24] could be used to reduce the length of traces we have to analyse. As this is an important issue when it comes to the performance and, therefore, scalability of our approach, decreasing the number of lines to be parsed would enhance our approach.

The improvement of tool support is considered as well, in particular concerning the development of a graphical interface for the LTSE tool. A next step would then be the integration of the tool with the LTSA, resulting in an environment for a complete model analysis process.

References

1. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: ACM POPL, pp. 98–109. ACM, New York, NY, USA. doi:10.1145/1040305.1040314
2. Ammons, G., Bodìk, R., Larus, J.R.: Mining specifications. In: ACM POPL, pp. 4–16. ACM, Portland, OR, USA (2002). doi:10.1145/503272.503275
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: ACM POPL, pp. 1–3. ACM, New York, NY, USA (2002). doi:10.1145/565816.503274
5. Belinfante, A.: Jtorx: a tool for on-line model-driven test derivation and execution. In: Esparza, J., Majumdar, R. (eds.) TACAS, LNCS, vol. 6015, pp. 266–270. Springer, Berlin, Germany (2010)
6. Beschastnikh, I., Brun, Y., Ernst, M.D., Krishnamurthy, A.: Inferring models of concurrent systems from logs of their behavior with CSight. In: Jalote, P., Briand, L., van der Hoek, A. (eds.) ICSE, pp. 468–479. ACM, New York, NY, USA (2014). doi:10.1145/2568225.2568246
7. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: ACM ESEC/FSE, pp. 267–277. ACM, Szeged, Hungary (2011). doi:10.1145/2025113.2025151
8. Bodhuin, T., Pagnozzi, F., Santone, A.: Abstracting models from execution traces for performing formal verification. In: Šęzak, D., Kim, T-H., Kiumi, A., Jiang, T., Verner, J.,Abrahã, S. (eds.) ASEA 2009, CCIS, vol. 59, pp. 143–150. Springer, Berlin, Heidelberg (2009). doi:10.1007/978-3-642-10619-4_18
9. Burtscher, M.: VPC3: a fast and effective trace-compression algorithm. ACM Perform. Eval. Rev. **32**(1), 167–176 (2004)
10. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., Sinha, N.: Concurrent software verification with states, events, and deadlocks. Form. Asp. Comput. **17**(4), 461–483 (2005)
11. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE TSE **30**(6), 388–402 (2004)
12. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. JACM **50**(5), 752–794 (2003)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts, USA (1999)
14. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM ToSEM **7**(3), 215–249 (1998)

15. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: ACM ICSE, pp. 439–448. IEEE, Limerick, Ireland (2000). doi:10.1145/337180.337234
16. Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A.: Source transformation in software engineering using the TXL transformation system. IST **44**(13), 827–837 (2002)
17. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ACM ISSTA, pp. 85–96. ACM, Trento, Italy (2010). doi:10.1145/1831708.1831719
18. Duarte, L., Kramer, J., Uchitel, S.: Towards faithful model extraction based on contexts. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE, LNCS, vol. 4961, pp. 101–115. Springer, Berlin, Germany (2008)
19. Duarte, L.M.: Behaviour model extraction using context information. Ph.D. thesis, Imperial College London, University of London (2007)
20. Duarte, L.M., Kramer, J., Uchitel, S.: Model extraction using context information. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS, LNCS, vol. 4199, pp. 380–394. Springer, Berlin, Germany (2006)
21. Feng, L., Lundmark, S., Meinke, K., Niu, F., Sindhu, M., Wong, P.: Case studies in learning-based testing. In: Yenigün, H., Yilmaz, C., Ulrich, A. (eds.) Testing Software and Systems, LNCS, vol. 8254, pp. 164–179. Springer, Berlin, Heidelberg (2013)
22. Fernandez, J., Mounier, L., Pachon, C.: Property oriented test case generation. In: Petrenko, A., Ulrich, A. (eds.) FATES, LNCS, vol. 2931, pp. 147–163. Springer, Berlin, Heidelberg (2003)
23. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: ACM PLDI, pp. 234–245. ACM, Berlin, Germany (2002). doi:10.1145/512529.512558
24. Gao, X., Snavely, A., Carter, L.: Path grammar guided trace compression and trace approximation. In: IEEE HPDC, pp. 57–68. IEEE, Paris, France (2006). doi:10.1109/HPDC.2006.1652136
25. Giannakopoulou, D., Magee, J.: Fluent model checking for event-based systems. In: ACM ESEC/FSE, pp. 257–266. ACM, Helsinki, Finland (2003). doi:10.1145/940071.940106
26. Giannakopoulou, D., Rakamarić, Z., Raman, V.: Symbolic learning of component interfaces. In: Proceedings of the 19th International Conference on Static Analysis, pp. 248–264 (2012)
27. Gradara, S., Santone, A., Villani, M.L., Vaglini, G.: Model checking multithreaded programs by means of reduced models. ENTCS **110**, 55–74 (2004)
28. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. LNCS **1254**, 72–83 (1997)
29. Havelund, K., Pressburger, T.: Model checking java programs using java pathFinder. STTT **2**(4), 366–381 (2000)
30. Henzinger, T., Jahla, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: ACM POPL, pp. 58–70. ACM, Portland, OR, USA (2002). doi:10.1145/503272.503279
31. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs, NJ, USA (1985)
32. Holzmann, G.: The model checker Spin. IEEE TSE **23**(5), 279–295 (1997)
33. Holzmann, G.: From code to models. In: ACSD, pp. 3–10. IEEE Computer Society, Washington, DC, USA (2001). http://doi.ieeecomputersociety.org/10.1109/CSD.2001.981759
34. Holzmann, G., Smith, M.: A practical method for verifying event-driven software. In: ACM ICSE, pp. 597–607. ACM, Los Angeles, USA (1999). doi:10.1145/302405.302710
35. Holzmann, G.J., Smith, M.: Software model checking: extracting verification models from source code. STVR **11**(2), 65–79 (2001)
36. Howar, F., Giannakopoulou, D., Rakamarić, Z.: Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In: ACM ISSTA, pp. 268–279. ACM, Lugano, Switzerland (2013). doi:10.1145/2483760.2483783
37. Ichii, M., Myojin, T., Nakagawa, Y., Chikahisa, M., Ogawa, H.: A rule-based automated approach for extracting models from source code. In: Oliveto, R., Poshyvanyk, D., Cordy, J., Dean, T. (eds.) WCRE, pp. 308–317. IEEE, Kingston, ON, Canada (2012). doi:10.1109/WCRE.2012.40
38. Keller, R.: Formal verification of parallel programs. CACM **19**(7), 371–384 (1976)
39. King, J.: Symbolic execution and program testing. CACM **19**(7), 385–394 (1976)
40. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic symbolic model checker. In: P. Kemper (ed.) Proceedings of the Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems, pp. 7–12 (2001)
41. Leuschel, M., Massart, T., Currie, A.: How to make FDR Spin: LTL model checking of CSP by refinement. LNCS **2021**, 99–118 (2001)
42. Li, H.F., Rilling, J., Goswami, D.: Granularity-driven dynamic predicate slicing algorithms for message passing systems. ASE **11**(1), 63–89 (2004)
43. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ACM ICSE, pp. 501–510. ACM, Leipzig, Germany (2008). doi:10.1145/1368088.1368157
44. Machado, P.D.L., Silva, D., Mota, A.C.: Towards property oriented testing. ENTCS **184**, 3–19 (2007)
45. Magee, J., Kramer, J.: Concurrency: State Models and Java Programming, 2nd edn. Wiley, Chichester, England (2006)
46. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York, USA (1992)
47. Mariani, L.: Behavior capture and test: dynamic analysis of component-based systems. Ph.d., Università degli Studi di Milano Bicocca (2005)
48. Meinke, K., Niu, F., Sindhu, M.: Learning-based software testing: a tutorial. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, Communications in Computer and Information Science, pp. 200–219. Springer, Berlin, Germany (2012)
49. Meinke, K., Sindhu, M.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) Tests and Proofs, LNCS, vol. 6706, pp. 134–151. Springer, Berlin, Germany (2011)
50. Milner, R.: An algebraic definition of simulation between programs. In: Society, B.C. (ed.) 2nd IJCAI, pp. 481–489. Morgan Kaufmann Publishers Inc., London, England (1971)
51. Milner, R.: Communication and Concurrency. Prentice-Hall Inc, Passau, Germany (1989)
52. Nimmer, J., Ernst, M.: Automatic generation of program specifications. In: ISSTA, pp. 232–242. ACM, Rome, Italy (2002). doi:10.1145/566172.566213
53. Odom, J., Hollingsworth, J., DeRose, L., Ekanadham, K., Sbaraglia, S.: Using dynamic tracing sampling to measure long running programs. In: ACM/IEEE SC, p. 59. IEEE, Seattle, WA, USA (2005). doi:10.1109/SC.2005.77
54. Pradel, M., Gross, T.R.: Automatic generation of object usage specifications from large method traces. In: Ceballos, S. (ed.) IEEE/ACM ASE, pp. 371–382. IEEE Computer Society, Washington, DC, USA (2009). doi:10.1109/ASE.2009.60
55. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. STTT **11**(5), 393–407 (2009)
56. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE TSE **12**(1), 157–171 (1986)
57. Utting, M.: How to Design Extended Finite State Machine Test Models in Java. CRC Press, Boca Raton, FL, USA (2011)

58. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA (2007)
59. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. ASE **10**(2), 203–232 (2003)
60. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse engineering state machines by interactive grammar inference. In: Penta, M.D., Maletic, J.I. (eds.) WCRE, pp. 209–218. IEEE Computer Society, Washington, DC, USA (2007). doi:10.1109/WCRE.2007.45
61. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. In: Lähmmel, R., Oliveto, R., Robbes, R. (eds.) WCRE, pp.301–310. IEEE, Koblenz, Germany (2013). doi:10.1109/WCRE.2013.6671305
62. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: ACM ESEC/FSE, pp. 35–44. ACM, Dubrovnik, Croatia (2007). doi:10.1145/1287624.1287632
63. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., Sampaio do Prado Leite, J.: Reverse engineering goal models from legacy code. In: IEEE RE, pp. 363–372. IEEE, Paris, France (2005). doi:10.1109/RE.2005.61

Lucio Mauro Duarte is a Senior Lecturer at the Department of Theoretical Computer Science of the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), where he teachs Algorithms and Software Verification. He holds a PhD degree in Computing (Imperial College London, University of London), and his main research areas are Validation and Verification of Systems, Software Testing, and Software Modelling. His Web page is www.inf.ufrgs.br/~lmduarte.

Jeff Kramer is a Professor at Imperial College London. He was Head of the Department of Computing from 1999 to 2004, Dean of the Faculty of Engineering from 2006 to 2009, and Senior Dean from 2009 to 2012. His research work is primarily concerned with software engineering, focusing on software architecture, behaviour analysis, the use of models in requirements elaboration and architectural approaches to adaptive software systems. He was Editor in Chief of IEEE TSE from 2006 to 2009 and was awarded the 2005 ACM SIGSOFT Outstanding Research Award and the 2011 ACM SIGSOFT Distinguished Service Award. He is co-author of books on Concurrency and on Distributed Systems and Computer Networks, and the author of over 200 journal and conference publications. He is a Fellow of the Royal Academy of Engineering, a Chartered Engineer, Fellow of the ACM, Fellow of the IET, Fellow of the BCS, and Fellow of the City and Guilds of London Institute.

Sebastian Uchitel holds a Professorship at the Department of Computing, FCEN, University of Buenos Aires, and a Readership at Imperial College London. He is also an Independent Researcher at CONICET. Dr. Uchitel was Associate Editor of IEEE Transactions on Software Engineering and is currently Associate Editor of the Requirements Engineering Journal and the Science of Computer Programming. He was program co-chair of the 21st IEEE/ACM International Conference on Automated Software Engineering and of the 32nd IEEE/ACM International Conference on Software Engineering. Dr. Uchitel has been distinguished with the Philip Leverhulme Prize, the ERC Starting Grant award, the KONEX prize. and by the Argentine National Academy of Exact Sciences.