University of London
Imperial College London
Department of Computing

Behaviour Model Extraction using Context Information

Lucio Mauro Duarte

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, November 2007

Abstract

This work describes an approach for behaviour model extraction merging static information, based on the control flow graph of the system, and dynamic information, obtained from traces of execution and a set of monitored attributes. The combination of control flow information and values of attributes forms what is called context information. More specifically, a context is defined as an abstract representation of a state of a system, composed of the block of code being executed, the evaluation of its associated control predicate and the current values of a set of attributes. This information, combined with a set of collected traces, provides the sequences of contexts reached during the execution and the actions performed in between them. It is demonstrated how context information can be used to guide the process of constructing Labelled Transition Systems (LTS) which are good approximations of the actual behaviour of the systems they describe. These models can be applied for automated behaviour analysis in a well-known model-checking tool, as well as for checking LTL properties. Augmentation of the set of values of attributes recorded in contexts produces further refined models and leads towards correct models, ruling out some false negatives. Completeness of the extracted models depends on the coverage achieved by the collected samples of execution, and may be slightly extended through the automatic inference of additional valid behaviours. The approach is partially automated by a tool called LTS Extractor (LTSE), which internally creates an implicit Labelled Kripke Structure (LKS) based on the gathered context information. The LKS is then mapped into a Finite State Processes (FSP) description, which is in turn used to obtain a graphical representation of the system behaviour as an LTS model. Results of two case studies are presented and discussed.

Acknowledgements

I would like to express my gratitude to:

- My supervisor, Jeff Kramer, for all his support and guidance during the development of my research. I learnt a lot from his experience and knowledge and I am also thankful for his friendship and understanding;
- My second supervisor, Sebastian Uchitel, for his friendship and his enthusiasm and dedication during our discussions. His invaluable insights and opinions helped me improve my understanding of my research area;
- Freeman Huang, for letting me use his implementation of the Bully Algorithm as one of my case studies;
- My colleagues Carlos Eduardo Thomaz, Robert Chatley, Howard Foster, Alberto Schaeffer, Eskindir Asmare, Daniel Sykes, Markus Huebscher, Paulo Henrique Maia, Leonardo Mostarda and Anandha Gopalan for their friendship and support. My special thanks to Daniel and Anandha for proof-reading chapters of this thesis;
- My sponsor, CAPES, for funding my research and giving me the opportunity to develop this work in an internationally renowned institution;
- All Nutford House residents who I came across during these years in London. My most special thanks to Paul Phibbs, Christina Malathouni, Marlène Monteiro, Lauren Trisk, Moreno Fasolo, Nicola Bianco, Neil Mangrolia and Jazmin Aguado, my family in the UK;
- All my friends, for their support during this entire journey. Their continuous friendship and care helped me through the times of loneliness;
- My family, for having taught me to remain positive and believe in myself even when problems come along the way. I am also thankful for their constant care and for understanding my reasons for being away for so long while trying to achieve my professional goals;
- God, for giving me such a wonderful family and for the gift of making another dream come true.

Dedication

Aos meus pais, Dinarte e Amelia, e a minha irmã, Maria Lúcia, com todo o meu esforço e dedicação.

A Cris, com todo meu amor.

‘People might not get all they work for,
but they must certainly work for all they get.’

Frederick Douglass

‘Things turn out best for the people who
make the best out of the way things turn out.’

Art Linkletter

Contents

List of Tables

List of Figures

List of Algorithms and Procedures

Chapter 1

Introduction

1.1 The Need for Behaviour Models

Software systems have become an important part of our everyday life. They control cash machines, telephone systems, medical devices, and many other equipment. Because of their relevance and wide range of applications, any misbehaviour of such systems could mean loss of money, shutdown essential services and put lives at risk [Pel01]. For this reason, ensuring the reliability of software systems is paramount.

Efforts to improve the confidence on software systems include the creation of *formal methods*, which have been successfully used for the development and verification of commercial and safety-critical software systems [CW96]. Formal methods include languages, techniques and tools based on mathematical concepts that can be applied to the specification and verification of programs[1] in order to check their correctness with respect to what they are expected to do. Though the use of formal methods cannot guarantee that a system is correct, it can improve confidence that certain misbehaviours will not occur [CW96].

Amongst the existing formal methods, one of the most used techniques is *model checking* [CGP99]. This technique employs the exhaustive checking of a finite-state model of a sys-

[1]We use the terms 'system' and 'program' interchangeably to refer to a software system.

tem against properties specified in some temporal logic, such as Linear Temporal Logic (LTL) [MP92], looking for possible violations. A violation consists of a behaviour proscribed by the property being checked.

Model checking has some advantages over other known validation techniques, such as simulation and testing [Pat06]. One of such advantages is that the model checking process can be fully automatic. This advantage is, to some extent, a consequence of the use of abstract models rather than concrete programs to analyse the behaviours of systems.

A *behaviour model* is an abstraction of the system that provides a restricted view of the system behaviour. This restrictive aspect may seem a disadvantage, since there is loss of information. Nonetheless, because they are abstract - and, therefore, reduced - representations of systems, models can normally be handled in situations where the real system could not [Lud03].

One of such situations involves the use of tool support: behaviour models can serve as inputs to one of the numerous available tools (e.g., Spin [Hol97] and LTSA [MK06]), where the system behaviour represented in these models can be analysed. These models have been successfully used in such tools to uncover errors that would go undetected otherwise, such as the identification of potential deadlock situations and violations of program properties [CW96].

In order to make use of such tools, it is first necessary to build behaviour models that comply with their input formal language. The problem is that constructing behaviour models normally requires some effort and expertise in the modelling language [UKM03]. This task is, therefore, much too often not trivial, even for experienced designers [Hol01].

1.2 The Model Construction Problem

It is common practice to build a model before implementing the system or in parallel with it, thus permitting an early analysis of the system behaviour. Properties can then be checked and the correctness of the model can be verified. Nevertheless, usually after producing a corresponding implementation, one cannot guarantee its correctness, as properties may not have been carried over to the program [GSVV04].

Even when it is possible to achieve a program that preserves the properties verified for a model, in many situations code and model tend to diverge as the system evolves [SC96]. The inclusion, removal or modification of parts of the system, in order to accommodate new features or versions, might happen without revisiting the model. It may also be the case that the model was not created taking into consideration some necessary features of the system, which were identified later on. The effort required to go back to the model to apply the changes may prevent developers from doing so. The mentioned situations may occur as a result of limited project time and/or budget. In all these cases, the model is rendered useless [Hol01].

In this work, we focus on how to build models of existing systems. The process of obtaining a model from an actual implementation is called *model extraction*. It involves using information from the implementation to construct a model of the system behaviour. Because the model is built based on the implementation, if any modification is applied in the code, then a new execution of the extraction process could generate a new model which includes the changes. Therefore, assuming that the model extraction process generates a correct translation from code to model, conformance could be easily maintained irrespective of alterations in the code.

There are, however, two essential requirements to be attended to when constructing behaviour models. Firstly, *the construction of the model must be much simpler and less time-consuming than building the system itself* [Hol01]. Otherwise, developers may opt for other types of analysis just because they require less time, effort or expertise, even though they cannot provide all the benefits of model checking. As a consequence, it is desirable that this process be executed (semi-)automatically, since the construction of models by hand is usually expensive and likely to introduce errors [CDH$^+$00]. Secondly, and more importantly, *the model should be a faithful representation of the system behaviour*, because any analysis based on an incorrect model may mislead the developer into an erroneous understanding of how the system behaves [JD00].

Research has been carried out on techniques for model extraction in recent years (e.g., [CW98], [HS99], [CDH$^+$00], [BR02] and [CCO$^+$04]) and the results have been encouraging. Nevertheless, the extensive use of model extraction, and, therefore, of model checking for existing systems, has been slowed down by what Corbett et al. call the *model construction problem* [CDH$^+$00]. This problem corresponds to finding a way of bridging the gap between the semantics of current programming languages and that of the less expressive languages used as inputs in model-checking tools.

1.3 Proposed Approach

The general objective of this work is *to present a possible solution to the model construction problem, which attends to the necessary requirements of simplicity, low time-consumption and faithfulness, and also complies with the modelling language of a model-checking tool.* We do so through the use of the concept of context. A *context* represents a specific situation in the execution of a system, which is the combination of the execution point in its control flow graph and its current state, represented by a set of values of selected program variables (attributes). Therefore, a context combines static and dynamic information to identify sequences of actions executed by the system and, this way, put these sequences together to build a behaviour model.

To obtain the necessary information, the source code of the program is statically instrumented with annotations that aim to collect control flow information and the values of a selected set of attributes. The instrumented version of the code is then executed, generating traces, where each trace is a sample of execution of the system. A set of test cases is usually employed to guide the trace generation phase, so that behaviours of interest are observed (e.g., behaviours that are relevant for the checking of a certain property).

The annotations in the traces are subsequently converted into *context information*, i.e., the block of code that created the context, the evaluation of its associated control predicate and the values of the set of attributes at that particular instant. The sequences of contexts found in the traces and the actions that appear between each two consecutive contexts are then translated into a Finite State Processes (FSP) specification [MK06], which describes a Labelled Transition System (LTS) [Kel76].

The behaviour models we build can be used to check temporal properties of single- and multi-threaded systems. Though different types of systems have been successfully used to test our approach, it has proved to be particularly suitable for reactive systems [MP92]. The reason is that in such systems it is possible to control the sequences of inputs. This way, we can choose the behaviours we would like to observe and record, thus generating a model that is very much focused on representing a set of behaviours that can affect the checking of a certain property.

We extract one model for each component of the system, so that they can be separately checked against local properties (properties pertaining to a single component). If the system includes more than one instance of a component, the observed behaviours of each instance, recorded in the traces, are combined to form the general behaviour of the component.

Multiple models can then be used to create a model parallel composition using tool support [MK06], making it possible to analyse how the components of the system work together and check program properties. Synchronisation on action names permits the representation of interactions and dependencies between components of a concurrent system.

The use of contexts not only allows us to combine multiple traces but also may result in the inclusion of additional behaviours to the model. These behaviours, though not observed in the traces, may be inferred based on the identification of similar contexts reached during the generation of the traces. For instance, this means that alternative paths may be included in the model even if each path appears in a different trace. In this case, these additional behaviours give us relevant information about the system behaviour, which might not be inferred directly from the traces. This feature is important if one considers that these additional traces may reveal violations that may not be detected by just looking at the traces.

The analyses using our models have demonstrated, through a number of case studies, that they are good approximations of the behaviours of the systems they describe with respect to properties to be checked. They can, therefore, be used to detect LTL property violations, thus increasing confidence on these systems regarding the absence of undesired behaviours, i.e., behaviours that violate the specification of the system.

Initial models can be augmented by the addition of more information, thus reducing its level of abstraction. This process generates a model which is a refinement [Mil71] of the initial model. The refinement process, by decreasing the level of abstraction, improves correctness with respect to a property being checked and prevents some false alarms (i.e., property violations in the model that are not actual violations in the code) from happening during property checking.

A generated model can also be improved through the inclusion of more observed behaviours. This allows an existing model to be updated using new observed behaviours. The addition of more behaviours not only improves the view the model provides of the program behaviour, but also reduces the possibility of missing out behaviours that might violate a property being checked. Focusing on behaviours that are relevant to check a property (by selecting appropriate test cases, for instance), it is possible to achieve a model that is complete enough to show whether the program satisfies or violates the property.

The effort devoted to the model extraction process is reasonably little and requires only basic knowledge of verification (mostly involving the interpretation of the outcomes of checking a property). In some situations, the user does not need to know either the programming language or the modelling language, as the necessary information can be collected and processed automatically. A tool called LTS Extractor (LTSE) has been developed to automate most of the process we describe here. The resulting model can be visualised and analysed using the LTSA tool [MK06], where linear temporal properties can be checked against it.

The main contribution of this work is, therefore, *the development of an approach for behaviour model extraction that uses the concept of context to combine static and dynamic information and whose resulting behaviour models can be used for property checking in an existing tool.* Secondary contributions include the development of a tool that implements this model extraction approach, demonstrating how the approach can be used for behaviour analysis and property checking and the description of a refinement technique that improves the model correctness with respect to a given property.

1.4 Thesis Outline

The next chapter presents the background theory of this work, discussing some issues related to behaviour models, including completeness and correctness of the models and model refinement, and some techniques for model extraction. Chapter 3 describes the proposed approach in more detail, presenting the types of information involved, introducing the concept of contexts and using an example to show the model extraction process. It also presents our approach for dealing with concurrency. In Chapter 4 there is a discussion about the formal foundations of the approach, mainly focusing on the abstractions used and on the formal definitions of the mappings applied during the process of constructing the final model.

Chapter 5 presents the tool support provided for the model extraction process and how it can be used to generate, visualise and analyse behaviour models. To demonstrate the proposed approach applied in a more complex scenario, Chapter 6 shows the results of two case studies, one based on the Single-Lane Bridge problem discussed in [MK06] and another involving an implementation of the Bully Algorithm [GM82] for leader election in a distributed system.

And, finally, Chapter 7 discusses the proposed approach in comparison to existing related work on model extraction, pointing out its main applications and known limitations. It also presents an evaluation of the tool support provided according to a set of criteria. The chapter ends with a summary of the thesis achievements and possible future work.

Chapter 2

Background

This chapter presents the background theory of this work. It includes a discussion on some types of behaviour models found in the literature and how they are employed to check properties. A discussion on the faithfulness of behaviour models and how they can be refined to improve correctness is also presented.

The second part of the chapter contains background information on model extraction techniques, commenting on their advantages and limitations, and a categorisation of our approach. We discuss our choice and the formalisms we have adopted.

2.1 Behaviour Models

Behaviour models are abstract descriptions of the intended behaviour of a system [UKM03]. They represent, therefore, a restricted view of all possible executions of the system. This view can be presented in many different ways and levels of abstraction, depending on the purpose of the model.

A widely used way of modelling behaviours for analysis and property checking is through finite-state machines. A *finite-state machine* (FSM) is composed of a finite set of *abstract states* $Q = \{q_0, q_1, ..., q_n\}$, representing (sets of) possible concrete states of a system, and a set of *transitions* connecting these states. A transition of the form $q_0 \rightarrow q_1$ defines that the system can evolve from a state q_0 to state a q_1. Thus, the behaviour of the system is described in terms of sequences of states that can be obtained by traversing the model using the transitions connecting these states, starting in some initial state.

Behaviour models described as FSMs have the required formal foundations for being used to support a rigorous analysis of systems and property checking [CCG+04]. However, some issues must be taken into account when working with these models:

- *Type of model*: It is important to choose the appropriate type of behaviour model. This choice depends on the type of application (e.g., model checking) it will have. In general, it is also related to the tool that shall be used, since the chosen type must be compatible with that accepted by the particular tool;
- *Expressiveness*: The model should provide ways of appropriately expressing relevant features of the types of systems to be modelled, such as concurrency, real-time conditions, probabilities, locations, among others;
- *Representation of States*: States may represent values of program variables, control locations (points where the system waits for the next noteworthy event), the value of the program counter, the contents of registers or any other abstraction used to describe the current situation of the system. States can also represent *local states*, describing only variables related to a specific process, or *global states*, which define a combination of the states of all processes composing the system;
- *Representation of Transitions*: Transitions may represent a variable assignment, the call or termination of a method, the beginning or end of a task or any other abstraction used to describe events that cause a change of state;
- *Types of Properties*: A *property* is the definition of an attribute that should hold for every possible execution of a system [MK06]. If the model is to be used for property checking, it is important to know what types of properties can be checked and what is the meaning of a violation according to the representation of states and transitions.

These and other issues are discussed next, when we present some types of behaviour models and their application for property checking and look into the question of model faithfulness. It is also discussed how behaviour models can be refined and the effects of this refinement.

2.1.1 Types of Behaviour Models

As suggested by Hansen et al. [HVV03], there are two basic approaches to represent behaviours using FSMs: state-based and action-based. These two approaches are presented next.

State-Based Approach

In the *state-based approach*, the behaviour of a system is represented as the sequence of states that can be reached according to the transitions available in each state. In this case, a *state* represents an instantaneous description of the system in terms of values of variables at a certain point in time [CGP99]. These variables may include, among others, the program counter, the execution stack state, the evaluation of program predicates or values of attributes.

Transitions define which states are reachable from the current state, showing how the values of the variables can change during the execution. Hence, the behaviour of a program is determined by the possible values these variables can be assigned and which combinations of variable values are allowed to occur.

One example of a stated-based behaviour model is a Kripke structure (KS) [CGP99]. In a KS, a state is described by a set of atomic propositions. Each atomic proposition represents a boolean expression over values of variables. Each state is labelled with the propositions that are true in that state. Therefore, a state in a KS defines a combination of true propositions at a certain instant of the execution of the represented system.

Definition 2.1. ***Kripke Structure***. A *Kripke Structure* $M = (Q, Q_0, T, AP, L)$ is a model where:

- Q is a finite set of states;

- $Q_0 \subseteq S$ is the set of initial states;
- $T \subseteq S \times S$ is a transition relation;
- AP is a set of atomic propositions; and
- $L : Q \rightarrow 2^{AP}$ is a function that labels each state with the set of true atomic propositions in that state.

An example of KS is shown in Figure 2.1. It describes a simple microwave oven system, based on a similar example presented in [CGP99]. Each state is labelled with the values of two atomic propositions: *on*, representing whether the microwave has been switched on, and *heat*, which indicates whether the microwave is cooking some food. As mentioned before, only true propositions in a state are used as labels in a KS. However, we also include false propositions in the labels - marked with the symbol $\sim$ - just to make it clearer the propositions that are false in that state.

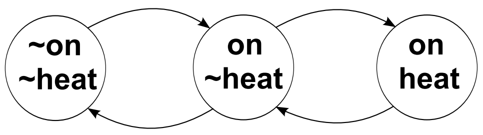


Figure 2.1: Kripke structure for a simple microwave oven.

The state where both propositions are false represents the initial state of the system. A KS can have several initial states, describing different possible initialisations of the system.

An execution of the system corresponds to a sequence of assignments to the set of propositions, causing transitions between states as the values of these propositions change. Note that the state where *on* is false and *heat* is true is not included in the structure presented in Figure 2.1. Therefore, this state is not part of a feasible execution of the system, i.e., it is unreachable.

Properties to be checked using state-based models describe how the values of the variables can vary during an execution. A given property is true in a model if the property is true in every state of the model. Hence, a violation of a property (the property is found to be false in the model) means that there is at least one state in the model which does not preserve the property being checked. This state constitutes an undesired combination of values of propositions and should, therefore, be disallowed by the system.

Action-Based Approach

Unlike the state-based approach, in the *action-based approach*, the behaviour of a system is described by the sequences of actions that it can execute. An *action* is an atomic event of the system that causes an indivisible change on the program state [MK06]. Actions are defined according to the level of abstraction involved [vG01], representing method calls, variable assignments, task completion or any other meaningful event. A sequence of actions describes a sequence of these events that the system can generate.

States represent points where the system waits for the next noteworthy action to happen. Thus, a change of state describes the occurrence of an action in the current state that causes the system to move to a next state.

Labelled transition systems (LTS) [Kel76] are frequently used to model behaviours according to this approach. As the name indicates, in an LTS, transitions are labelled rather than the states, as was the case in Kripke structures.

Definition 2.2. ***Labelled Transition System.*** A *labelled transition system (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 the actions that cause the system to progress from the current state to a new one. 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 an action with name a. Thus, a transition can only take place if the associated action occurs.

Figure 2.2 shows the LTS model of the same microwave oven system used to illustrate a KS. Note that states are labelled with a number just to identify them. Actions `swicthedOn` and `switchedOff` represent the events of turning the microwave on and off, respectively. Action `cook` signals the beginning of the cooking process, whereas action `done` indicates that the food being cooked is ready.

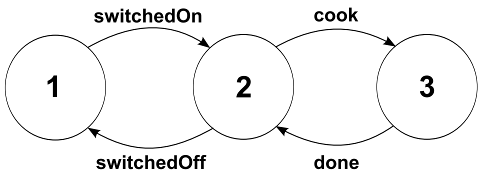


Figure 2.2: LTS of the microwave oven.

A property to be verified using this approach describes sequences of actions allowed in the system. If a property is true in a model, it means that the model does not allow any sequence not allowed by the property. Thus, a violation of a property represents that the model permits some sequence of actions that should not occur.

Our choice

We build models that follow the action-based approach. More specifically, we construct LTS models. Our choice is based on the fact that analysing a program in terms of the sequences of actions it can execute is a more intuitive way of reasoning about its behaviour [CCG$^+$04]. Furthermore, LTS models guarantee an easy way of describing system behaviours and, because they have well-defined mathematical properties [MK06], they can be used to model check sequential, concurrent and distributed systems.

The existence of tool support (e.g., the LTSA tool [MK06]) simplifies the process of checking properties. It also provides the possibility of visualising the generated models and obtaining the necessary feedback regarding property violations.

Following this choice, henceforth, when we mention a model we mean an LTS model. Similarly, when we talk about behaviours we refer to sequences of actions executed by the system.

2.1.2 Software Model Checking

Model checking refers to an automatic technique used to verify whether a model of a system satisfies a certain property. According to Clarke et al. [CGP99], the model checking process consists of the following three steps.

Modelling. The creation of a model that is accepted by a model-checking tool. In practice, as previously commented, the model is generally built using an FSM-based formalism, as this type of modelling language is the one used in most of the available tools (e.g. Spin [Hol97] and LTSA [MK06]). Because a model can be too large and/or too complex (leading to the *state-space explosion problem* [CGP99]), some technique may have to be applied to reduce the complexity and make the model tractable by a tool. Many of the available techniques apply abstract interpretation [CC77], predicate abstraction [GS97], symmetry reduction [ES96], partial order reduction [KP89] or slicing [Tip95]. Generally, the techniques are adapted to the properties to be verified and to the program to be abstracted. Therefore, the main idea is to keep as little information as possible, so long as this information suffices to verify the necessary properties about the program.

Specification. The definition of a set of properties the system is expected to satisfy, using the logic supported by the model-checking tool. The properties to be checked against the model usually are specified using some *temporal logic* [MP92]. These properties are formulas that describe particular behaviours a system is expected to exhibit.

Verification. The process of checking whether the specification is satisfied by the created model. When a property is checked against a model, there are two possible outcomes: either the property is *satisfied* (or preserved) by the model or it is *violated* by the model. The satisfaction of a property means that the model does not include any behaviour not allowed by the property. The violation, on the other hand, indicates that, at least, one behaviour in the model is not permitted by the specification. In this case, tools normally generate an *error trace* as output, which shows the behaviour that violated the property.

Issues related to the model checking process include, among others, how to support the specification step, in order to make it easier to specify properties and make sure these properties are accepted by model checking tools, such as through the use of specification patterns [DAC98]. How to provide better error trace interpretation techniques, helping the analysis of the outcomes of the verification process, is another focus of research [CDH$^+$00].

Although these are relevant issues, we concentrate on the modelling step. More specifically, we look into the issue of how to create models that are accepted by a model-checking tool (the LTSA tool [MK06]) and can be used to check properties specified using Linear Temporal Logic (LTL) [MP92].

2.1.3 Faithfulness of a Behaviour Model

When modelling a system, an important aspect is guaranteeing by construction that the model is a good approximation of the system behaviour. This requires choosing the appropriate abstractions to build the model, i.e., how to approximate the behaviour described in the model to that of the actual program.

If the model is an erroneous abstraction of the system, the fact that it satisfies some properties does not mean that the system also satisfies the same properties. Thus, if one cannot trust the model, then one cannot trust any result from any analysis based on it. Though it is not possible to completely ensure that the mapping from code to model is correct [Pel01], it should be guaranteed that at least some properties of the concrete system are present in the abstract model [Lud03].

Faithfulness

A behaviour model M is a *faithful* representation of the behaviour of a program $Prog$ if the satisfaction/violation of a certain property by M implies that $Prog$ also satisfies/violates the property. Hence, the objective when building a behaviour model is to achieve a faithful abstraction of the system it represents. This way, an analysis on the model would correspond, at a certain level of abstraction, to an analysis on the actual program.

Faithfulness is, nevertheless, not any easy requirement to fulfil. It essentially depends on the quantity and quality of the information used to build the model. Because the model is a restricted view of the complete behaviour of the program, some possible behaviours of the latter will probably not be represented in the former. As a result, if any significant behaviour is left out, the model can satisfy a property that is violated by the program.

Another important aspect that affects the faithfulness of a model is the choice of the level of abstraction, i.e., the level of details used to attribute meanings to states, transitions and actions. Too detailed information about the system behaviour can lead the model to a level of abstraction where its complexity and size prevent the use of a model-checking tool (for instance, a state-space explosion situation [CGP99]). On the other hand, too little information means that the model may allow behaviours that cannot be executed by the system, simply because it does not have enough details to identify those behaviours as infeasible.

Completeness and Correctness of a Model

The case where a property is satisfied by the model but violated by the code results in what is called a *false positive*. False positives are caused by the model not containing the behaviour that originates the violation. Hence, we say that the model is *incomplete*.

Definition 2.3. ***Completeness***. Let M be a behaviour model representing an implementation of a program $Prog$. If we represent the set of all behaviours of $Prog$ as $L(Prog)$ and the set of all behaviours of M as $L(M)$, then M is *complete* with respect to $Prog$ iff $L(Prog) \subseteq L(M)$.

Completeness, therefore, corresponds to the characteristic of a behaviour model that defines how representative the set of behaviours in the model is in relation to the set of all behaviours that can be observed in the real system it describes. The more representative this set of behaviours, the "more complete" is the model.

The situation where the model violates a property not violated by the code is called a *false negative*. False negatives are the result of the selection of an inappropriate level of abstraction. They indicate that the level of details of the model is too low and, for this reason, the model allows behaviours not allowed by the code. Thus, the model is said to be *incorrect*.

Definition 2.4. ***Correctness***. Let M be a behaviour model of an implementation of a program $Prog$. If we represent the set of all behaviours of $Prog$ as $L(Prog)$ and the set of all behaviours of M as $L(M)$, then M is *correct* with respect to $Prog$ iff $L(M) \subseteq L(Prog)$.

The correctness of a behaviour model is, therefore, related to the existence of invalid behaviours, i.e., behaviours not allowed by the system it describes. The fewer invalid behaviours it has, the "more correct" is the model.

Completeness can be improved by the addition of more behaviours of the program to the model, whereas correctness can be enhanced by adding more details to the model (e.g., adding more values of variables). Ideally, the set of behaviours should include only the necessary behaviours to check a property, so as not to significantly affect scalability and tractability by a model-checking tool. Following the same idea, the improvement of correctness should not increase complexity and size to levels not acceptable by a tool to be used. We discuss the completeness and correctness of the models we generate in Chapter 4.

2.1.4 Behaviour Model Refinement

In situations where a model generates false negatives, a change in the level of abstraction is necessary. Having an abstract model (hereafter called 'original model'), an *abstraction refinement* [Dam03] process aims at generating a new, more concrete model (hereafter called 'refined model'), where by "concrete" it is meant that the new model is a closer representation of the system [MG96]. In other words, the refinement process is used to improve the precision of a model that has too coarse a level of abstraction to check a certain property.

If the level of abstraction is too coarse, then the model is incorrect. Therefore, refining a model is a way of improving its correctness. This improvement involves the addition of more information about the program during the construction of the refined model, and should result in the refined model satisfying properties not satisfied by the original model.

The satisfaction of more properties is a consequence of the additional information used to build the refined model, which rules out some false negatives - i.e., eliminates some behaviours not allowed by the system. This elimination occurs, for instance, when states previously seen as equivalent are found to be distinguishable based on the additional information.

In practice, a refinement process is executed after the verification step. At this point, there are two possibilities: either the property is checked to hold or a violation is found. In the first case, the model checking process finishes, as the property has been checked to be satisfied by the model at the current level of abstraction.

If an error trace is obtained as a result of model checking a property, this counter-example is used to verify whether this is a feasible behaviour of the program. If the error trace is a feasible behaviour of the program, than a true violation has been found and the implementation should be modified as necessary. Nevertheless, if the program cannot reproduce the behaviour contained in the error trace, then one can conclude that it is a false negative and the refinement process starts. The refinement process may be repeated as many times as necessary until a suitable abstraction is found.

Refinement Relation

An important requirement of a refinement process, besides generating an abstraction that eliminates some infeasible behaviours, is the preservation of properties. The refined model should preserve all properties that held in the original model. This property-preservation relation can be guaranteed through the establishment of a formal relation between the behaviours described in the two models. The semantics of some existing formal relations are discussed in [vG01].

One well-known formal relation used to compare behaviour models is *simulation* [Mil71]. Simulation is a relation that considers the structures of two models, providing a way of relating their states according to the actions they enable and the states that are reachable from them. This means that, rather than comparing two models based only on the traces they can produce (as occurs, for example, in trace equivalence [Hoa85]), a simulation relation compares the systems according to the set of states and transitions that can produce these traces.

Definition 2.5. ***Simulation Relation*** (or Simulation Preorder). Given two LTS models $M = (S, s_i, \Sigma, T)$ and $M' = (S', s'_i, \Sigma', T')$, a relation $Sim \subseteq S \times S'$ is a *simulation relation* if:

1. $(s_i, s'_i) \in Sim$, and
2. for every pair of states (s_1, s'_1), with $s_1 \in S$ and $s'_1 \in S'$, if $s_1 \xrightarrow{l} s_2$ then $s'_1 \xrightarrow{l} s'_2$, for some $s_2 \in S$ and $s'_2 \in S'$, such that $(s_2, s'_2) \in Sim$.

M' is said to *simulate* M, denoted by $M \preceq M'$, if there is a simulation relation Sim over $S \times S'$.

A simulation relation over $S \times S'$ defines that any sequence of actions executed by M can be matched by M'. Hence, there is a relation between states of the two models in a way such that, whatever action M takes from its current state, M' can take the same action from its current state. Moreover, the new state of M' retains at least all the options of actions also available in the new state of M.

As will be discussed in Chapter 4, there is a simulation relation between the original model and the refined model that our refinement process generates. Though simulation is not as strong a behaviour relation as bisimulation [Par81], it suffices to guarantee property preservation from an abstraction of a system behaviour (initial model) to a more concrete representation of the same behaviour (refined model).

2.2 Model Extraction

Model extraction is the process of automatically generating a model from an existing implementation [HS99]. Because it is automatic, it reduces the occurrence of errors and the tedious work involved in building models by hand. However, the model extraction process should guarantee a certain level of faithfulness of the model so as to allow the checking of properties. It should also provide the possibility of refining an initial abstraction, so that some infeasible behaviours could be eliminated.

Several model extraction techniques have been proposed in the literature. They can be divided into three main categories, according to the type of information they collect to build a behaviour model[1]. We present each category and then insert our approach in one of them. The original characteristics of the proposed approach are pointed out and briefly compared to existing work in the same category.

2.2.1 Based on Static Information

In model extraction using *static information*, information is collected directly from the source code or compiled code of a program, without executing it. The information can be statically gathered through a complete control flow or data flow analysis [CDH$^+$00, HJMS02, BR02, CCG$^+$04] or through the insertion of annotations in the code to mark relevant points of interest and guide this analysis [HS99].

Information that can be obtained statically is usually related to the control flow of the program. A *control flow graph* (CFG) [ASU86] is a directed graph that represents statements of a program according to possible execution orderings based on the flow of control inside the program source code. Statements label the nodes and directed edges connect these nodes, showing the allowed sequences of statements. This graph shows control flow statements (selection and repetition statements) as points of decision where, depending on the evaluation of the *control*

[1]In this classification, we focus on the type of the information collected, regardless of how this information is obtained and of what sort of analysis will be carried out based on it.

predicate [RS02] associated with the statement, some paths can be taken and some others may be made infeasible.

A CFG has an initial node and a final node, representing, respectively, the beginning and the end of the computation inside the program. Therefore, the reasoning about the flow of control of a program described in a CFG is based on the paths allowed to be taken across the edges connecting the nodes in the graph, starting at the initial node and leading to the final node. Edges that can only be taken according to a control flow statement are labelled with values of the control predicate that enables that execution path. Edges coming out of nodes labelled with method names are not labelled, as they do not depend on any condition to be taken.

Having a CFG of a program code, it is possible to obtain a complete view of what the program can do. Therefore, the use of a CFG supplies valuable theoretical information on all possible behaviours of the system. If a certain sequence of nodes is not connected by edges, then we know that it is not a possible execution of the system. This information can then be used to model, analyse and verify the system.

A model built on static information needs to represent the system states in a way such that the real values are abstracted. This way, the model is tractable by a tool and the results of an analysis are true regardless of what these real values are [Ern03]. Though this *over-approximation* of the system behaviour guarantees no false positives, it can lead to the construction of a model that may yield several false negatives during verification [BPSH05] as a result of the decrease of precision.

This means that some of the paths, though possible according to the CFG, may not be feasible in the code when it is executed. This is mainly due to changes in the state of the system that cannot be easily tracked or predicted statically.

The use of *symbolic execution* [Kin76] can provide the necessary information to rule out some infeasible paths. However, if the control flow is dependent on inputs, an analysis of each case might be necessary. Therefore, the set of input classes may make it impossible to test all possible cases. Furthermore, symbolic execution usually also requires the help of a theorem prover, demanding some expertise from the user on how to guide this tool when dealing with a formula that cannot be solved automatically.

2.2.2 Based on Dynamic Information

Models constructed using *dynamic information* are the result of an inference process. Information on real executions is used to identify patterns of behaviour, which are then included in the model. These patterns can be obtained through techniques such as grammar inference [CW98] or machine learning [ABL02].

Information that can be gathered dynamically includes trace information. *Trace information* usually represents the behaviour of a system in terms of traces of execution. A *trace* is an output from the system generated during a real execution, normally in response to a set of inputs. Traces show values of program variables, the state of the execution stack, occurrence of method calls, the state of threads or any other run-time information.

Trace collection is done through the monitoring of the required information while the system is executed. To permit this monitoring, it is necessary to instrument the source code, the bytecodes, the virtual machine or the operating system [BPSH05]. During the instrumentation, annotations are inserted at certain parts of the code to record the necessary data when the program is running.

The gathering of information regarding actions that occurred during an execution of a program results in the generation of traces. Since these were observed behaviours of the system, one can be sure they are feasible behaviours. This is the most important advantage of working with dynamic information: this information includes only feasible behaviours. If some behaviour cannot happen, it will not appear in the traces.

Though the information is precise as to what the system can do, the knowledge is restricted to the particular observed behaviours, making it too dependent on the samples [Ern03]. Moreover, it is far from being a source of accurate knowledge about how some actions depend on others. Therefore, models created using this approach are *under-approximations* of the system behaviour and may either include behaviours not allowed by the system, generating false negatives, or leave out some important aspects of its behaviour, generating false positives.

As commented before, some inference techniques have been proposed to identify patterns of behaviours and, therefore, allow the generalisation of the behaviour of a program based on the patterns inferred from a set of traces. Nevertheless, neither the use of statistical information [CW98] nor of a machine learning technique [ABL02] provided satisfactory results to help understand the detected patterns and the identification of dependencies between actions. The main problem with these approaches is that their inference process is based only on sequences of actions where the reason for the ordering of actions and the meaning of these actions in the code are not known.

Combining multiple traces, as attempted in [Mar05], is also difficult, since one does not know when two occurrences of an action in the traces correspond to the same specific action in the code. The same action may be executed in different parts of the code, under different situations. For instance, a method might have several call sites in a program, each one of them reached through different paths in the code. For this reason, relevant relations between actions, such as an action that can only happen after another or an action that can be repeated because it is part of a loop, cannot be correctly inferred.

2.2.3 Hybrid

Because using just static or dynamic information has the limitations pointed out before, Ernst [Ern03] suggested that static and dynamic information could be used in combination, in a *hybrid approach*. In this approach, the knowledge gained from one type of information could be used to complement the knowledge obtained from the other.

Whereas static information can provide a general knowledge about the system behaviour, dynamic information could improve precision by supplying knowledge on particular, feasible behaviours. At the same time, whilst dynamic information can provide the necessary precision through feasible behaviours, the static information could be used to permit an accurate generalisation of the system behaviour based on the collected samples.

An attempt to use such an approach is presented in [NE02], where static information is analysed to confirm the existence of invariants inferred based on dynamic information. The authors successfully demonstrate the usefulness of complementing dynamic with static information. However, their approach applies two types of analyses and, consequently, requires the use of two different tools: one to infer the invariants from the dynamic information and another to check these invariants against the static information.

2.2.4 Categorisation of the Proposed Approach

We use a combination of static and dynamic information for the extraction of LTS models from Java source code, thus following the hybrid approach. The static information we collect is related to the system *control flow*, i.e., how the control of the system execution evolves while it is running. As for the dynamic part, we use *trace information*, which includes samples of real executions of the system (*traces*) and monitored values of program variables (*attributes*).

Trace information allows us to be precise about the actual behaviours of the system, whereas control flow information permits us to be accurate as to the relations governing the sequences of actions the system can execute. With this combination of information, we can identify feasible behaviours and, based on them, derive other behaviours not included in - or that could not be easily inferred from - the traces. Such behaviours represent, for instance, alternative paths, which, though not observed in any particular trace, can be detected by combining traces and identifying, according to the control flow structure, common subsequences of actions.

2.3 Summary and Discussion

In this chapter, we discussed the underlying theory of the work herein described. We presented concepts related to behaviours models, model checking, faithfulness of a model, model refinement and model extraction. These concepts will be used as basis for the discussions to be presented in the next chapters, when we describe our approach in more detail.

In the next chapter, we will show how we apply the combination of static and dynamic information to build action-based models, namely LTS models, and Chapter 4 will discuss how their correctness and completeness can be improved.

Chapter 3

Model Extraction Using Contexts

The proposed approach is now presented in more detail. In particular, we discuss the concept of context and the information used to identify contexts. We use a simple program as an example to present the basic ideas and the general process of building models using contexts. We discuss each part of the process and use the example to show the inputs and outputs in each step.

3.1 Contexts

Our model extraction process is based on the combination of control flow information and trace information. To support this combination, we also use state information to create our concept of context. This notion is central to this work, as it allows the combination of static and dynamic information in a way such that we can identify ordering relations between actions, connect multiple traces to form a single model and infer additional feasible behaviours.

In order to explain what we can obtain from each type of information and how we combine them to create our notion of contexts, we will use the piece of code shown in Figure 3.1 as an example. This is part of the code of a simple text editor, which allows the user to open a text file and execute operations on it (edit, print, save and close).

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

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

Figure 3.1: Running example code.

This system has two attributes: `isOpen` indicates whether a file is open and `isSaved` indicates whether the file contents have been saved. Method `open` sets `isOpen` to true, while `close` sets it to false. Method `save` defines `isSaved` as true, whereas method `edit` sets `isSaved` to false. If the file has been modified but not saved, the user has the option of choosing whether to save the document or not before terminating the program (see method `exit`).

3.1.1 Control Flow Information

Figure 3.2 shows the control flow graph (CFG) of the editor code, including the CFG of method `exit`[1]. The ellipses represent methods, whereas the diamonds define control flow statements. The arrows indicate the direction of the flow of control, from an entry point (`Enter`) to an exit point (`Return`).

When the system reaches a control flow statement, a control predicate is evaluated. Depending on its value, the flow of control can follow a different path inside the CFG. These alternative paths can be seen in the CFG in Figure 3.2 as multiple arrows leaving from the same diamond. The different values of the predicates label the arrows. We use dashed arrows to show the flow of control not affected by control flow statements (i.e., not dependent on the evaluation of any control predicate).

[1]We do not show the CFGs of the other methods because their implementations are irrelevant to our discussion.

The CFG shows us all possible paths of execution of the editor based on the control flow information collected from the code. It also reveals the control predicates that define the system behaviour and which statements and other predicates are dependent on each predicate. However, as mentioned earlier, this static information can include paths that cannot be taken during a real execution. Therefore, some of the paths presented in Figure 3.2, though possible according to the CFG, may not be feasible in the code when it is executed.

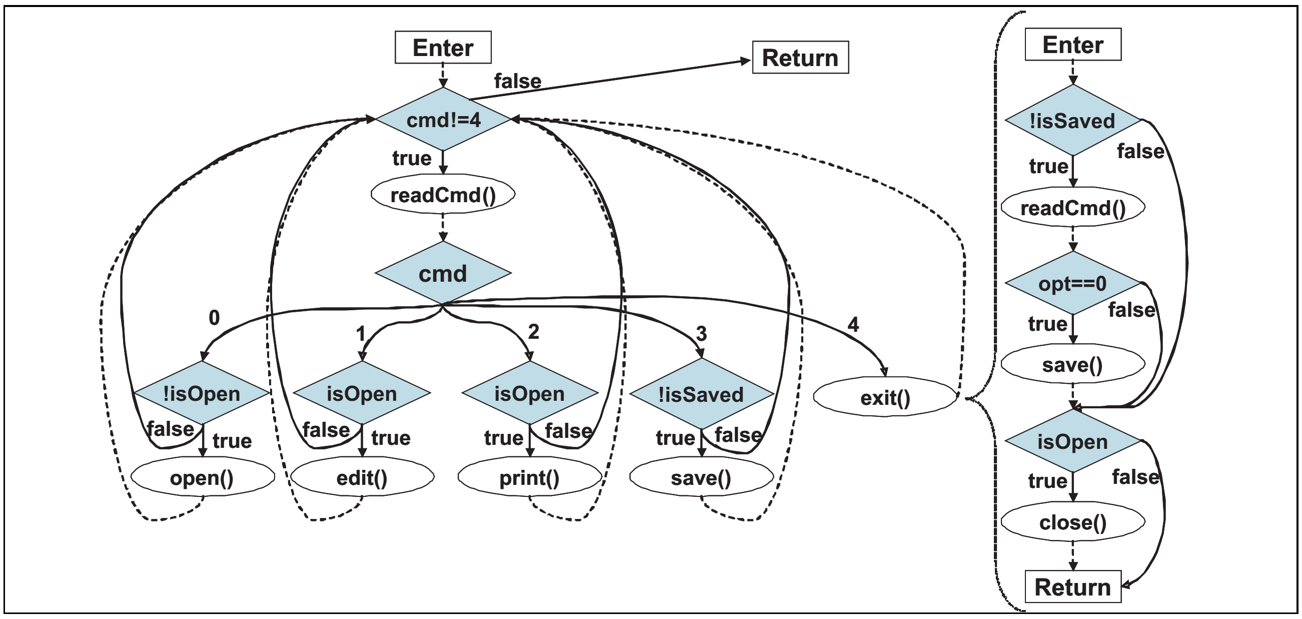


Figure 3.2: Control flow graph of the editor code.

To fully understand the feasibility of a path of execution, we need to know the values of predicates evaluated along that path, which are normally dependent on values of program variables. As these values can change during the execution (due to inputs or changes in the system state), some paths that were feasible at a certain point may become infeasible later on. If these dynamic changes are not considered, path feasibility can be hard to analyse.

3.1.2 Trace Information

Trace information provides sequences of actions executed by the system in the form of traces of execution. Thus, a trace represents a real execution of the system and can be used to identify feasible behaviours. However, traces are intrinsically connected to the inputs used to produce them and to the particular internal state of the system at the moment they were generated.

For example, let us consider the following sequence of inputs to the editor in Figure 3.1: ⟨0 1 3 2 1 1 2 3 2 1 4 0⟩. This generates the following trace (ignoring calls to method `readCmd`, which does not affect the results), where each name represents the occurrence of an action corresponding to the method with the same name:

⟨`open edit save print edit edit print save print edit exit save close`⟩

Looking at this trace, one can try to infer some relations between the actions. For example, one can see that `open` seems to be the initial action and `close`, the last. Actions `edit`, `save` and `print` appear more than once in the trace, indicating that they may be executed repeatedly. It also appears that, after saving a file, the next operation may be either `print` or `close`.

All these possible relations, however, are based only on the information from this specific trace, which can be misleading (and, in this particular case, it is). We do not have any concrete knowledge whatsoever on how the actions are actually related. Because of this, we could obtain a model like the one in Figure 3.3.

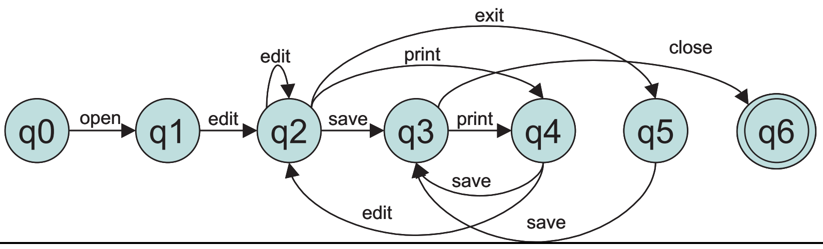


Figure 3.3: Inferred model based on trace information.

This model was built just by looking at the sequence of actions in the trace. We created one initial state from which only operation `open` is possible. Then, we followed to the next state with the next action (`edit`), creating a new state for each transition labelled with a new action. Whenever an action appeared again, the transition would lead back to the state where a previous transition labelled with the same action pointed to.

This model creates some possible sequences of actions that are not allowed by the system, such as repetitions of the sequence of actions $\langle$`save print`$\rangle$. The model also introduces a restriction that is not part of the system behaviour: contrary to what the model shows, the system allows that a document be closed at any moment once it has been opened.

The model was built using the assumption that all executions of an action occur under the same circumstances, thus always leading the system to the same point in its execution (same state in the model). However, the occurrence of an action may be influenced by the current state of the system and by previous actions. This information is, therefore, relevant to comprehend when and why an action can happen and when and why it cannot.

It may be argued that the model was created based only on a single trace, restricting the view of possible behaviours of the system. Nevertheless, even if we had more traces from which we could gather different behaviours, it would be hard to find similar subsequences of actions in order to join them. Because we do not completely understand why and when a certain action can happen, we cannot, for instance, be sure that the occurrence of an action depends on the occurrence of another.

The procedure we used to construct this model was an extremely simplified version of more elaborate approaches to model inference such as [CW98] and [Mar05]. Even these approaches, nonetheless, also suffer from the same problems previously mentioned. For this reason, their models can include invalid additional behaviours and impose unreal restrictions. Furthermore, the combination of multiple traces, supported by Mariani's work [Mar05], results in the inclusion of more invalid behaviours and restrictions, exactly because the inference process tries to merge traces without additional information on how they were produced.

To fully comprehend the relations between actions and how a trace was produced, we need information on how the code generated that trace for that particular set of inputs and under which circumstances. Based on this, we could also identify where one trace connects to another and, therefore, put them together in one single model.

3.1.3 Merging Control Flow and Trace Information

We merge the structural and general knowledge gained from control flow information with the dynamic and specific knowledge obtained from traces. The basic idea is to use the traces to identify, among all possible paths in the code, some feasible paths. Once we know that a path is feasible (i.e., there is a set of inputs and values of predicates that causes the system to exercise it), we can look at the control flow to understand how the trace was generated in the code and possibly infer alternative paths based on the control predicates.

Besides sequences of actions, we enrich trace information by including state information. *State information*, in this work, comprises the values of a subset of the set of program attributes. Attributes are normally used in control predicates, thus affecting the control flow and, consequently, the traces the system can generate. We call this set of attributes, used as state information, the *system state*.

In order to show the importance of using state information, Figure 3.4 shows an LTS that can be built combining only the CFG in Figure 3.2 and the trace used in the previous section[2]. We built the model following the paths in the CFG that needed to be exercised to output the sequence of actions included in the trace, in that same order. For instance, to generate an action `open`, it is necessary to get to the test of control predicate `cmd`, follow the arrow labelled with `0` and then take the arrow labelled with `true` after the test involving control predicate `!isOpen`. After that, we follow the dashed line back to the beginning and take the necessary path to produce action `edit`. We follow the same idea for all the other actions in the trace.

Note that the loop involving actions `open`, `edit`, `print` and `save` was correctly identified as a result of checking the paths in the CFG that originated the sequence of actions. Moreover, it is also correct that action `exit` ends the loop.

There is, nevertheless, a problem: the model does not show that some actions are not allowed to happen under certain circumstances. For instance, action `save` cannot occur if a document has not been edited (since control predicate `!isSaved` is false), but it is shown to be enabled

[2]State E represents the end of the execution.

every time the loop is executed. Therefore, this model only shows that certain actions *may* be enabled in a state, not the actions that *are* actually enabled in that state.

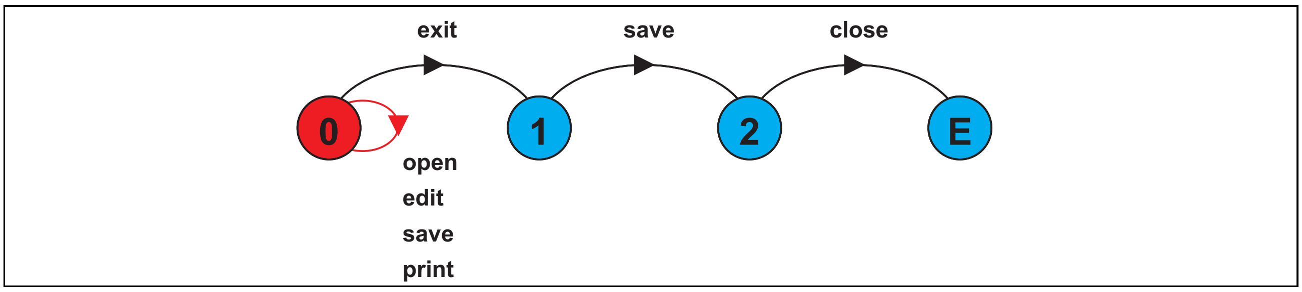


Figure 3.4: LTS of editor without state information.

As a very abstract representation of the editor, the model in Figure 3.4 would be fine. However, if we were to check a property stating that `save` cannot happen before an occurrence of `edit`, this level of abstraction would not suffice.

We could only check this property if we knew the values that affect the results of this analysis. In this case, this would correspond to knowing the value of attribute `isSaved`, as it controls the relation between `save` and `edit`. The use of values of attributes, in combination with control flow and trace information, to understand the feasibility of paths and the relations between actions is the idea upon which we build our concept of contexts.

3.1.4 Context Information

We deal with a control component, based on the implementation control flow, and a data component, composed of attributes of the system, to build our models. The control flow shows us the sequences of commands executed, whereas the attributes permit us to identify how the state of the program affects these sequences.

The control component is, as commented, obtained from a CFG of the implementation of the system. We now formally define the CFG of a program code:

Definition 3.1. ***Control Flow Graph.*** Let $Prog$ be a program. Then $CFG(Prog) = (Q, q_i, Act, \Delta)$ is its *control flow graph*, where:

- Q is a finite set of control components of $Prog$, where each control component $q \in Q$ is a pair (bc, cp), with bc representing a block of code and cp describing the logic test associated with bc (i.e., its control predicate),
- $q_i \in Q$ is a control component $(bc_i, true)$, where bc_i is the initial block of code,
- Act is the set of actions of $Prog$, and
- $\Delta \subseteq Q \times Act^* \times Q$ is a transition relation.

Because a method call represents the entrance to a new block of code (the method body), a method m of $Prog$ is also part of the set Q and its control predicate is always *true* (a method has no control predicate to prevent it from being executed once called). Moreover, the execution of a method also represents an action of $Prog$, such that, for every method $m = (bc_m, \texttt{true}) \in Q$, if we denote the name of m as $n(m)$, then $n(m) \in Act$. Hence, a method call in $CFG(Prog)$ describes that the control flow has reached a new block of code and that the execution of that block of code generates an action with the same name as that of the referred method.

Though the default interpretation of an action in this work is the execution of a method, the set of actions of the system can be expanded to represent other meaningful events, as will be discussed in 3.2.1. These other events are treated exactly as method executions, except that they do not represent contexts.

As for the data component, we adopt the values of attributes. Let $P(Prog)$ be the finite set of data components of $Prog$ and $val(p)$ be the value of an attribute $p \in P(Prog)$. A finite tuple $v = \{val(p_1), ..., val(p_n)\}$, where $n \geq 1$, represents one possible combination of values of attributes $p_1, ..., p_n \in P(Prog)$. The set $V(Prog) = \{v_1, ..., v_n\}$ is composed of all possible combinations of values of attributes of $Prog$, such that $v_1 = \emptyset$, representing the beginning of the execution, when values of attributes are yet unknown. The finite set $V(P) \subseteq V(Prog)$ represents all possible combinations of values of attributes $p_k, ..., p_m \in P$, where $1 \leq k \leq n$ and $m \geq k$, such that $P \subseteq P(Prog)$.

Definition of Contexts

We define a *context* as the combination of the current block of code, which is determined by the evaluated control predicates, and the current values of the attributes that define the system state. Therefore, a context puts together control and data components and forms our definition of abstract state. More formally:

Definition 3.2. *Context.* Given a program $Prog$, a context $C = (bc, val(cp), v)$ is the combination, at a certain point of the execution of $Prog$, of the current block of code bc being executed, the value $val(cp)$ of its control predicate cp and the current set of values $v \in V(P)$ of attributes in $P \subseteq P(Prog)$.

We assume that control predicates do not contain method calls. Hence, a control predicate is always a boolean expression mentioning only the values of local variables and attributes.

As an example, consider a singleton set of attributes $\{p\}$, where p is of boolean type, and the following control flow statement, which we will identify as the block of code if:

```
if (b == 1) {
   ...
}
```

A possible context $C_1 = (if, true, \{true\})$ represents a situation where this control flow statement is being executed, the control predicate $(b == 1)$ has been evaluated to be true and attribute p is also true. The same applies to all other control flow statements of the program.

According to our definition of contexts, the execution of a system can be seen as a sequence of contexts, with sequences of actions happening between two consecutive contexts. Note that, even though blocks of code can appear in a nested structure in the implementation, our definition of contexts only considers the sequence in which these blocks of code are reached during the execution. Therefore, if two blocks of code are executed sequentially or one inside the other, it still represents the occurrence of two consecutive contexts.

An execution starts in an *initial context*, where no control predicates have been evaluated yet and the initial values of attributes are still unknown. As the execution continues, and control predicates are tested and attributes are assigned new values, the context changes. A transition in our models, therefore, represents a change of context. In practice, it indicates that at least one of the components of the current context has been modified. This modification can be triggered by a new block of code being reached or by a new value being assigned to at least one of the attributes comprising the system state.

For instance, if we consider the code in Figure 3.1, we say that it was in a certain context C_1 before reaching the while-statement in line 10. At that point, a new control predicate is evaluated, thus causing a transition where a new context C_2 is created for which the predicate is, for example, true. Once the next input is read, the system reaches line 12, where a new predicate is evaluated. According to the command entered, the associated actions will be executed, possibly modifying the system state. Therefore, the next block of code reached will take into account this new state and, this way, create a different context.

This means that the same block of code may create multiple contexts, depending on the combinations of the evaluation of its associated predicate and the possible values of attributes. Therefore, contexts not only determine whether an action will be executed or not, but also show when this action happens under different circumstances, which may influence its result and the next possible actions.

The conjunction of a control component (control flow information) with a data component (state information) to identify a context is denominated *context information*. With context information, we can determine which behaviours are feasible based on the collected traces. We can also know how these behaviours are generated in the code, according to the control flow information gathered from the program. Moreover, we can identify the conditions (the evaluated predicates and the required value of the system state) under which these behaviours can occur.

3.2 Extracting Behaviour Models Using Contexts

Our model extraction approach builds behaviour models, in the form of LTS models, from Java source code. We begin the process by instrumenting the code using a source code transformation language to generate traces, usually, based on a test suite. Using the collected information, we identify the necessary context information. This information, combined with the sequences of actions in each context, obtained from the recorded traces, is used to create an FSP specification. This textual description of the system can then be used in the LTSA tool to obtain a graphical LTS model. A general view of the process is presented in Figure 3.5.

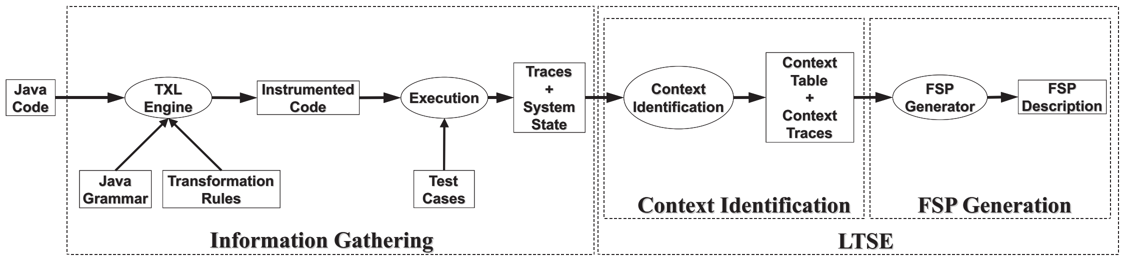


Figure 3.5: General view of the model extraction process.

Ellipses represent processing phases and boxes represent inputs/outputs of these processes. Horizontal arrows show the sequence of information processing, whereas vertical arrows describe extra inputs needed during the given process execution. The large block on the right-hand side represents the part of the process automated by our tool (presented in Chapter 5).

This process is described in more detail next. We refer to the code in Figure 3.1 to exemplify results from each phase.

3.2.1 Information Gathering

To collect context information, we first annotate the Java source code of the necessary classes of the system and then execute them to generate traces, which are recorded in log files. The methods used for instrumentation and trace generation are presented next.

Instrumentation

We obtain context information from the system through the instrumentation of the source code of Java classes that implement components of the system under analysis. All classes that produce actions mentioned in the properties should be instrumented. Each block of code of the classes is appropriately annotated during this process.

To carry out the instrumentation, we use the TXL engine [CDMS02] and a Java grammar specified using the TXL language[3]. Instrumentation is executed through transformations in the original code applied by the TXL engine, based on predefined rules. Each rule describes a pattern to be matched. Whenever one of these patterns is matched, the associated rule is applied, modifying that piece of code so that it includes the necessary annotations.

Annotation rules. We apply domain-independent rules to annotate control flow statements, call sites and method bodies. Annotations print out predefined labels identifying the type of statement to the standard error output, along with the values of attributes and other necessary information, such as an object ID and a block ID. The former is used to identify traces of different instances of a class, whereas the latter is a number automatically assigned to each block of code by the TXL engine and could be interpreted as an abstraction of the program counter. The TXL code for the rules is presented in Appendix A.

The rules of annotation presented in Figure 3.6 apply to *repetition statements*, where `cp` is the control predicate, `listCmds` is the list of commands executed in case `cp` is true, `attribs` is the list of values of attributes and `bid` is the block ID. We use `Print` to represent the output command `System.err.println` to make it shorter.

The annotations are included inside the statements to capture each iteration. Note that, in the case of a while- or for-statement, if the control predicate is evaluated to be false, then no output will be generated. This occurs because a false control predicate means that the statement is ignored, thus not affecting the future behaviour of the system.

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

```
while (<cp>) {
  Print("REP_ENTER:(<cp>)#"+this+"#{"+<attribs>+"}#<bid>");
  <listCmds>
  Print("REP_END:(<cp>)#"+this+"#<bid>");
}
```

```
do {
  Print("REP_ENTER:(<cp>)#"+this+"#{"+<attribs>+"}#<bid>");
  <listCmds>
  Print("REP_END:(<cp>)#"+this+"#<bid>");
} while (<cp>);
```

```
for (<init>;<cp>;<inc>) {
  Print("REP_ENTER:(<cp>)#"+this+"#{"+<attribs>+"}#<bid>");
  <listCmds>
  Print("REP_END:(<cp>)#"+this+"#<bid>");
}
```

Figure 3.6: Repetition statements annotation rules.

The rules presented in Figure 3.7 apply to *selection statements*. In the rules, `val(cp)` is the evaluation of `cp`. In switch-statements, we annotate each case-clause, since their execution depends on the control predicate and each one of them contains a different list of commands.

```
Print("SEL_ENTER:(<cp>)#"+(<val(cp)>)+"#"+this+"#{"+<attribs>+"}#<bid>");
if (<cp>)
  <listCmds>
Print("SEL_END:(<cp>)#"+this+"#<bid>");
```

```
Print("SEL_ENTER:(<cp>)#"+(<val(cp)>)+"#"+this+"#{"+<attribs>+"}#<bid>");
if (<cp>)
  <listCmds1>
else
  <listCmds2>
Print("SEL_END:(<cp>)#"+this+"#<bid>");
```

```
switch (<cp>) {
  case <v1> :
    Print("SEL_ENTER:(<cp>)#"+(<val(cp)>)+"#"+this+"#{"+<attribs>+"}#<bid>");
    <listCmds1>
    Print("SEL_END:(<cp>)#"+this+"#<bid>");
    break;
  ...
  default :
    Print("SEL_ENTER:(<cp>)#"+(<val(cp)>)+"#"+this+"#{"+<attribs>+"}#<bid>");
    <listCmds>
    Print("SEL_END:(<cp>)#"+this+"#<bid>");
    break;
}
```

Figure 3.7: Selection statements annotation rules.

In if-else-statements, the annotations are included in the same positions as in an if-statement. We can identify which list of commands (the one associated with the `if` part or the one associated with the `else` part) was executed according to the value of the control predicate.

Figure 3.8 shows the rule used to annotate method calls, where `met` is the method name. As commented before, a method call represents a block of code and the execution of an action. For this reason, it is doubly annotated: annotations for the beginning and end of the method execution and for the actions related to the call of the method and its termination. We will discuss the use of the termination annotation in Section 3.3.

```
 Print("CALL_ENTER:<met>#"+this+"#"+<obj>+"#{"+<attribs>+"}#<bid>");
 Print("ACTION:<met>#"+this);
<obj>.<met> (<params>);
 Print("CALL_END:<met>#"+this+"#"+<obj>+"#<bid>");
 Print("ACTION:<met>#"+this);
```

Figure 3.8: Method call annotation rule.

We also annotate the bodies of methods. This annotation occurs following the rule shown in Figure 3.9. For this annotation rule, we assume that the method has only one return statement, located at the end of the method body. According to the same idea used to annotate method calls, we include one pair of annotations to mark the block of code and another pair to signal the occurrence of the associated actions.

```
<modifiers> <ret_type> <met> (<params>) {
   Print("MET_ENTER:<met>#"+this+"#{"+<attribs>+"}#<bid>");
   Print("ACTION:<met>#"+this);
  <listCmds>
   Print("MET_END:("+"<cp>"+"#"+this+"#<bid>");
   Print("ACTION:<met>#"+this);
  return(<ret_value>)
}
```

Figure 3.9: Method body annotation rule.

Annotating the method body we can capture external calls to that method and represent this in the model. Because an external method call is annotated in the caller, we can have synchronisation on action names between the caller and the callee. This is used for modelling concurrent components, as will be discussed in Section 3.3.

When analysing the annotations generated in the log files (see next sections), we ignore actions generated by internal method calls (i.e., calls where the caller and the callee are the same). The reason for this is that we already capture calls to internal methods when the method body is accessed. Therefore, we can discard the actions created by internal method calls and keep the ones created by method bodies. However, the annotation of internal method calls is still important to identify calls happening in different contexts and at different call points.

Because every method execution generates two annotations in the log file, one to indicate the context generated by the method and another to signal the occurrence of the action it represents, we will refer to the latter as an *action annotation* to distinguish it from the former. Annotations generated by other instrumented blocks of code will be called *context annotations*.

User-defined actions. Besides the automatically identified actions, we allow the user to define their own actions. *User-defined actions* represent actions other than the execution of a method. They are important in situations where, for example, reaching a given point in the code 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). The format of a user-defined annotation is

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

where `name` refers to the name of the action being defined. User-defined actions can be manually inserted in any part of the code, using this predefined format. They are automatically converted into the appropriate annotation (`ACTION` annotation) when the code is instrumented. They are also considered action annotations.

User-defined attributes. In the same way we allow a user to define actions other than the execution of methods, we provide support for the definition of additional attributes. 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. For instance, the real temperature of a boiler is not the main information needed by the system controlling it; the system needs to know that the temperature is not above/below some safety threshold. Therefore, being able to code a set of values of an attribute into a simple expression can reduce the model, as one will not have one separate state (context) for each possible different value of this attribute. The format of a user-defined attribute is

```
#attribute:"<name>"=<expression>;
```

where `name` is the name of the defined attribute and `expression` is the expression associated with this attribute, which should mention only predefined attributes. User-defined attributes should be manually inserted using this format, in the declaration area of the code, where the other attributes are defined. They are automatically added to the attribute list when the code is instrumented. After being added to the attribute list, user-defined attributes can be used to refine the model just like any other attribute.

Instrumented file. The inclusion of the annotations generates an instrumented file. The instrumented file can then be used to execute the system and obtain the necessary information to construct the model. This information is produced in the form of execution traces.

Part of the instrumented version of the code of the editor is shown in Figure 3.10. Action annotations are clearly identified by the label `ACTION`. All other annotations, therefore, are context annotations.

Known limitations. Due to some difficulties using the TXL language, the automatic instrumentation process still suffers from some limitations regarding statements that can be annotated. Among these limitations are method calls that include the returned value of another method call as a parameter (e.g., $m_1(m_2()))$ and the use of commands that alter the normal execution flow, such as return-statements, continue-statements and break-statements. The only situation where a break-statement is currently supported is in case-clauses of switch-statements. A single return-statement is supported at the end of a method body. Note, however, that these limitations are not limitations of the annotation formats. Therefore, these problems can be overcome by manually editing the code to include the necessary annotations where needed. Moreover, the use of TXL to annotate the code is not mandatory. Any other tool could be used to execute this task so long as the predefined annotation patterns are followed.

```
public class Editor {
  ...
  public Editor () {
    while (cmd != 4) {
      System.err.println("REP_ENTER:(cmd!=4)#"+this+"#{isOpen="+isOpen+" isSaved="
        +isSaved+"}#14");
        ...
        switch (cmd) {
          case 0 :
            System.err.println("SEL_ENTER:(cmd)#"+cmd+"#"+this+"#{isOpen="+isOpen
              +" isSaved="+isSaved+"}#7");
              System.err.println("SEL_ENTER:(!isOpen)#"+(!isOpen)+"#"+this
                +"#{isOpen="+isOpen+" isSaved="+isSaved+"}#0");
              if (! isOpen)
                System.err.println("CALL_ENTER:open#"+this+"#"+this
                +"#{isOpen="+isOpen+" isSaved="+isSaved+"}#13");
                System.err.println("ACTION:open#"+this);
                open ();
                System.err.println("CALL_END:open#"+this+"#"+this+"#13");
                System.err.println("ACTION:open#"+this);
              System.err.println("SEL_END:(!isOpen)#"+this+"#0");
            System.err.println("SEL_END:(cmd)#"+this+"#7");
            break;
          ...
        }
      System.err.println("REP_END:(cmd!=4)#"+this+"#14");
    }
  }

  void exit () {
    System.err.println("MET_ENTER:exit#"+this+"#{isOpen="+isOpen+" isSaved="
      +isSaved+"}#8");
    System.err.println("ACTION:exit#"+this);
    {
      ...
    }
    System.err.println("MET_END:exit#"+this+"#8");
    System.err.println("ACTION:exit#"+this);
  }
  ...
}
```

Figure 3.10: Example of instrumented code.

Trace Generation

The trace generation is done by logging the output produced by executing the instrumented code. In order to be able to select the behaviours we want to observe, we usually use a test suite. A *test suite* is a set of *test cases*, where each test case contains a sequence of inputs used to probe the system.

We do not currently use any particular technique for selecting test cases. Our test cases are chosen based on the knowledge we have of the system and on the behaviours we would like to observe. These behaviours are usually related to properties we intend to verify, so that we can observe behaviours that influence the preservation/violation of these properties.

Each class selected for instrumentation must be instrumented and executed in turn to generate traces. This means that the instrumentation and the trace generation process must be repeated for each one of them. This is necessary to guarantee that each log file contains only information about a single class.

The result of executing the instrumented code is the creation of a set of logged traces. One log file is created for each trace. Part of the log for an execution of the code in Figure 3.1 is shown in Figure 3.11. It includes the beginning of the log, where the first input was the command to open a file and the next command was to edit the file. We used the same sequence of inputs used to obtain the model in Figure 3.3 to generate this log file. Note that, for simplification, annotations produced by internal calls are not included.

```
REP_ENTER:(cmd!=4)#Editor@12b6651#{isOpen=false isSaved=true}#14
SEL_ENTER:(cmd)#0#Editor@12b6651#{isOpen=false isSaved=true}#7
SEL_ENTER:(!isOpen)#true#Editor@12b6651#{isOpen=false isSaved=true}#0
MET_ENTER:open#Editor@12b6651#{isOpen=false isSaved=true}#8
ACTION:open#Editor@12b6651
MET_END:open#Editor@12b6651#8
ACTION:open#Editor@12b6651
SEL_END:(!isOpen)#Editor@12b6651#0
SEL_END:(cmd)#Editor@12b6651#7
REP_END:(cmd!=4)#Editor@12b6651#14
REP_ENTER:(cmd!=4)#Editor@12b6651#{isOpen=true isSaved=true}#14
SEL_ENTER:(cmd)#1#Editor@12b6651#{isOpen=true isSaved=true}#7
SEL_ENTER:(isOpen)#true#Editor@12b6651#{isOpen=true isSaved=true}#1
MET_ENTER:edit#Editor@12b6651#{isOpen=true isSaved=true}#9
ACTION:edit#Editor@12b6651
MET_END:edit#Editor@12b6651#9
ACTION:edit#Editor@12b6651
SEL_END:(isOpen)#Editor@12b6651#1
SEL_END:(cmd)#Editor@12b6651#7
REP_END:(cmd!=4)#Editor@12b6651#14
...
```

Figure 3.11: Example of log file.

3.2.2 Context Identification

The output from the information gathering phase consists of log files containing the blocks of code executed, their respective values of control predicates and the values of the attributes of the system at that point of the execution. In the context identification stage, this information is used to discover contexts of the system recorded in the log files.

Each log file records a sequence of contexts reached during the execution and the actions that occurred in between them. This sequence of alternating contexts and sequences of actions forms our traces.

Definition 3.3. ***Trace.*** Given a program $Prog$ with set of actions Act, a trace t of $Prog$ is a finite sequence $\langle C_1\alpha_1C_2\alpha_2...\alpha_nC_n\rangle$, where $C_1, C_2, ..., C_n$ are contexts of $Prog$ and $\alpha_1, ..., \alpha_n$ are sequences of actions of $Prog$, such that, for $1 \leq j \leq n$, $\alpha_j = \{a_1...a_m\}$, where $m \geq 1$ and $a_1, ..., a_m \in Act$.

We use $Tr(Prog)$ to denote the set of all traces that program $Prog$ can produce when instrumented and executed. F_{Prog} is the set of log files containing traces of $Prog$. Therefore, $Tr(F_{Prog})$ represents the set of all traces of $Prog$ recorded in F_{Prog}, such that $Tr(F_{Prog}) \subseteq Tr(Prog)$.

The Context Table

We now discuss the structure used to record contexts identified during the analysis of the log files. To simplify the discussion, we will treat the control flow information as two additional attributes. This way, a context can be seen simply as a combination of values of a set of attributes, where two of these attributes necessarily encode the control flow information, composed of an identification of the executed block of code (block ID), and the evaluation of the control predicate associated with this block.

The block ID (BID) is obtained statically during the instrumentation of the code. A unique sequential number is assigned to each new instrumented block of code. The value of the control predicate and those of the other attributes are gathered dynamically, i.e., through the execution of the instrumented code and the monitoring of their values.

The context information collected from context annotations is recorded in a *context table* $CT = \{c_1, ..., c_n\}$, where $c_1, ..., c_n$ are entries of the table. Each entry is assigned a *context ID* (CID), which is a unique sequential numeric identifier created whenever a new entry is inserted in the CT. An entry of the CT contains a set of values of attributes defining a context.

Before executing the procedure to create the context table, each log file is split into as many files as the number of different instances (object IDs) found in the logs. This way, we can isolate the particular behaviour exhibited by each instance and, eventually, merge them to create a single model. Note that this is the same as having multiple traces from a single class, with the only difference that they were all collected during the same execution.

The CT is initialised with the initial context, which receives CID 0. This is the initial context for every model we generate. The BID for the initial context is always -1.

The basic procedure to construct the CT is to read each annotation from the log files and collect the context information from the context annotations. After the context information has been gathered, the identified context is compared with every other context already recorded. A context C is identified as in the table if, when compared with a context C' (stored as an entry c' of the CT), C and C' have the same context information, i.e., the same set of values for the attributes, including the BID and the value of the control predicate.

If no context in the CT is found to have the same context information, the context is new (i.e., not in the table yet). A new entry is then created to store its context information, which is assigned a newly created CID.

Table 3.1 shows part of the CT generated based on the log presented in Figure 3.11. The first column contains the CIDs. The second column describes the predicates evaluated in the contexts. No predicate is associated with the initial context and the name of the method is used for contexts representing a method execution. The last column contains the system state, where the first value describes the BID, the second value presents the evaluation of the associated control predicate and the last two values represent the values of attributes `isOpen` and `isSaved`, respectively.

Context Traces

The result of the context identification phase is the creation of a CT and the generation of a set of *context traces* for each class of the system. These context traces are sequences of CIDs

Context ID	**Predicate**	**State**
0	-	{-1,`true`,-,-}
1	`(cmd!=4)`	{14,`true`,`false`,`true`}
2	`(cmd)`	{7,0,`false`,`true`}
3	`(!isOpen)`	{0,`true`,`false`,`true`}
4	`Editor.open`	{8,`true`,`false`,`true`}
5	`(cmd!=4)`	{14,`true`,`true`,`true`}
6	`(cmd)`	{7,1,`true`,`true`}
7	`(isOpen)`	{1,`true`,`true`,`true`}
8	`Editor.edit`	{9,`true`,`true`,`true`}
9	`(cmd!=4)`	{14,`true`,`true`,`false`}
...	....	{...}

Table 3.1: Example of context table.

and actions, representing the contexts the system went through during the execution and the actions that happened in between them.

Each context trace is a result of the analysis of a log file during the CT construction. As annotations are read from a log file, a *context file* is used to record the context trace. Reading a context annotation causes the corresponding CID to be written on the context file, whereas reading an action annotation results in the action name being inserted in the file.

Context traces are, therefore, processed traces, where each context annotation is replaced by a CID and each action annotation is replaced by an action name. The construction of the CT and the generation of the context traces occur at the same time. Algorithm 3.1 shows this procedure in more detail.

Note that the attribute comparison is restricted to a set of attributes $P \subseteq P(Prog)$ received as input (line 11). This is necessary because the value of every single attribute is always recorded in the annotations, even though just a subset of them might be used to build the model. The reason for that is that, as we will see in Chapter 4, our refinement process depends on changes in the set P to modify the level of abstraction of the model. Therefore, having the values of all attributes makes it easier to build models with different sets of attributes by just providing a new set P and filtering out those attributes not in it. Otherwise, it would be necessary to collect the information on the additional attributes by again running the instrumented code.

Algorithm 3.1 $BuildCT(F_{Prog}, P)$

Inputs: F_{Prog}: finite non-empty set of log files, P: finite set of attributes, CT: empty context table

Outputs: CF: a finite non-empty set of context files

```
1  $nextID = 0$
2  $CF = \emptyset$
3  $c_{initial} = (nextID, \{-1, \texttt{true}, -\})$ // Creates initial context
4  $nextID = nextID + 1$
5  $CT = \{c_{initial}\}$ // Initializes CT with initial context
6  for all log files $f \in F_{Prog}$ do
7     Create new context file $cf$
8     for all annotations $an \in f$ do
9        Read $an = (a, v)$
10       if $an$ is context annotation then
11          if $\exists c = (s_c, v_c) \in CT$ s.t. $v_c == v \cap P$ then
12             Write $s_c$ in $cf$ // Context found and written in context file
13          else
14             New CID $s = nextID$
15             $nextID = nextID + 1$
16             New CT entry $c' = (s, v \cap P)$
17             $CT = CT \cup \{c'\}$ // Adds context to the table
18          end if
19       else
20          Write $a$ in $cf$ // Writes action name in context file
21       end if
22    end for
23    $CF = CF \cup \{cf\}$
24 end for
25 return $CF$
```

An example of context trace can be seen in Figure 3.12. It represents the context trace generated using the log file shown in Figure 3.11. CIDs are preceded by the symbol # (e.g, line 1) to differentiate them from action names (e.g., line 6).

Line 1 of the context trace corresponds to the initial context, which is always automatically included. The CID in line 2 of the context trace (`#1`) was created based on line 1 of the log file (see Figure 3.11). Similarly, the second and the third lines of the log file were translated, respectively, into the CIDs in lines 3 and 4 of the context trace. Lines 4 and 5 of the log file were generated when the system entered method `open`. The context annotation generates the context identified as `#4` in the context trace and the action annotation originates the action `open` (lines 5 and 6 of Figure 3.12).

```
#0
#1
#2
#3
#4
open
#5
#6
#7
#8
edit
#9
#10
#11
#12
save
#5
#13
#14
#15
print
#5
#6
#7
#8
edit
#9
#16
#17
#18
edit
#9
#19
#20
#21
print
#9
#10
#11
#12
save
#5
#13
#14
#15
print
#5
#6
#7
#8
edit
#9
#22
#23
exit
#24
#25
#12
save
#26
#27
close
#END
```

Figure 3.12: Example of context trace.

Note that annotations in lines 6 to 10 in Figure 3.11 do not create any corresponding element in the context trace. However, the action annotation in line 7 could have caused the inclusion of an action name in the context trace. In Section 3.3 we will discuss when and how this is possible. As for the annotations marking the end of control flow statements, they are merely used to understand the nested structure of these statements.

The context trace puts together the context information, summarised in the CIDs, and the trace information related to the executed sequences of actions. This means that, analysing its contents, we can identify the contexts the system went through to be able to execute a certain action or sequence of actions. Moreover, we can know in which context each execution of an action happened and, therefore, detect executions of the same action in different contexts.

3.2.3 Model Generation

As previously stated, we generate LTS models using context information. However, in order to use the values of attributes during the construction of the models, we need an intermediate structure, which can deal with both actions and states (i.e., contexts). The formalism we have adopted is described as follows, based on the definition presented in [CCO+04]:

Definition 3.4. ***Labelled Kripke Structure.*** A *Labelled Kripke Structure* (LKS) $K = (S, s_i, P, \Gamma, \Sigma, T)$ is an abstract 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 the one presented in [CCO+04] in that we use attributes instead of propositions. However, the only difference is that attributes are not always boolean variables and, consequently, may have a wider range of values than that of propositions, which can only be either true or false.

Because of that, in our case, the state-labelling function Γ always labels every state with the values of all attributes in P. Therefore, the number of possible labels will be a result of all possible combinations of values for each attribute in P. Note, however, that even though the number of combinations is potentially infinite, generally not all of them can actually occur when the system is executed.

We always use a singleton set of initial states. The reason is that, ultimately, we will generate an LTS model, which has only one initial state (see Definition 2.2). Therefore, having just a single initial state in the construction of the LKS makes it easier to convert it into an LTS. Moreover, the initial state represents the initial context, which is also unique.

Another small difference from our definition to that of Chaki et al. [CCO+04] is that we allow an atomic sequence of actions to label the transitions. This represents the possibility of more than one action occurring in between two consecutive contexts. They are atomic because no other states can happen between actions in a sequence.

It is important to notice that the use of sequences of actions does not affect the LKS definition, because a transition is still a connection between two states that is labelled with actions from the alphabet. In our case, this label is a compound label, comprising all actions that can happen in between two states. Hence, the underlying semantics of a transition in an LKS does not change.

Basic Definitions

The following definitions apply to an LKS according to Definition 3.4. A *behaviour* is a finite sequence of actions $\pi = \langle a_1 ... a_n \rangle$ such that $a_1, ..., a_n \in \Sigma$. The set $L(K) = \{\pi_1, \pi_2, ...\}$ of all behaviours of K is called its *language*.

For a state $s \in S$, $E(s) = \{\alpha \in \Sigma^+ | \exists s' \in S \cdot (s, \alpha, s') \in T\}$ represents the non-empty finite set of sequences of actions *enabled* in s. A *path* $\lambda = \langle s_1 \alpha_1 s_2 \alpha_2 s_3, ... \rangle$ is a sequence of alternating states and sequences of actions labelling transitions connecting these states, such that, for $i \geq 1$, for every transition (s_i, α_i, s_{i+1}) composing the path, $\alpha_i \in E(s_i)$. A path should always start and end with a state rather than with a sequence of actions. We use $\Lambda(K)$ to denote the set of all paths of K.

Parameters

The construction of an LKS model from context traces is executed according to three parameters: an alphabet, a set of attributes and a set of traces. As we generate models to check properties, we attempt to specify these parameters according to a property to be checked.

Alphabet. The actions that are part of the alphabet label the transitions of the model. The user has total freedom to choose these actions, i.e., the user can select which subset of the alphabet of the system will compose the alphabet of the model. However, only actions mentioned in the property are required. Hence, some of the actions in the alphabet of the system may be irrelevant and can be discarded.

Let us suppose we would like to check a property ϕ, with alphabet $\Sigma(\phi)$, using a model K to be constructed using our approach. If $\Sigma(K)$ is the alphabet of K, then it should include the alphabet $\Sigma(\phi)$, such that $\Sigma(\phi) \subseteq \Sigma(K)$.

Moreover, the greater the number of actions chosen, the bigger and more complex the model will be. Focusing only on the actions required to check a property ϕ results in a more compact model, tailored for checking that particular property. This allows a better visualisation of the model and, therefore, facilitates all types of analysis. Therefore, ideally, $\Sigma(K) = \Sigma(\phi)$.

System State. The selection of attributes to compose the system state is a simple way of defining the level of abstraction of the model. If no attributes are selected, the model is at its most abstract version, since states are defined only by control flow statements (i.e., essentially, the resulting model represents a CFG). On the other hand, if all available attributes are chosen, the model is at its most concrete version.

To allow visualisation, we must select a subset of attributes rather than using all the ones available. A reasonable approach for selecting attributes is to identify which of them affect the property being checked. Usually, attributes used in control predicates are relevant, because they influence the decision-making during program execution.

It is also important to consider the range of possible values of an attribute. Attributes of the boolean type are normally good candidates to be used in the system state because their range of possible values is very short (just two, in fact) and they are generally used in control predicates. Attributes that store references, however, should not be included. They tend to change values quite often during the execution and their contents are usually irrelevant for property checking. In cases where one just wants to know whether a reference is null or not, user-defined attributes should be used to model this abstraction.

Set of Traces. The selection of the set of traces has great influence on the results of the model extraction process. Therefore, they must be chosen in a way such that they offer a complete coverage of the relevant behaviours of the system. The best criterion is to drive the trace generation according to the property to be checked. This means that, if we use test cases to produce traces, the tests should force the observation of those behaviours that could cause a property violation and of those that show the correct behaviour of the system in relation to the property. The test suite should, therefore, prevent, or at least minimise, the occurrence of false positives.

LKS Construction

Once the context traces have been produced, we proceed to the construction of the LKS model, which is our intermediate representation of the system behaviour and will serve as basis for the generation of the final model (LTS model). Algorithm 3.2 presents the abstract algorithm used to generate an LKS model.

The algorithm receives the parameters previously discussed. In our example of the editor, we used alphabet $\Sigma = \{\texttt{open},\texttt{edit},\texttt{print},\texttt{save},\texttt{close}\}$, set of attributes $P = \{\texttt{isOpen},\texttt{isSaved}\}$ and the context file presented in Figure 3.12. The context file was created using the log file in Figure 3.11 and the same set P of attributes.

Note that the algorithm does not explicitly build an LKS model. The construction of the implicit LKS model actually occurs in two steps:

1. In the first step, states (contexts) are identified and "labelled". The labelling corresponds to executing Algorithm 3.1 to identify contexts and include them in the CT according to the value of the system state. Therefore, the implicit state label is the combination of values of the attributes. These implicitly labelled states are then used to generate the context traces;

2. In the second step (implemented by Algorithm 3.2), context traces are analysed to identify states and transitions. An abstract state is associated with each CID so that each state of the LKS model corresponds to a context recorded in the CT. If two states s_1 and s_2 appear consecutively in a context trace and a finite sequence of actions α occurs in between them, then a transition (s_1, α, s_2) is added to the set T.

Algorithm 3.2 $CreateLKS(CF, P, \Sigma)$

```
Inputs: $CF$: finite non-empty set of context files, $P$: finite set of attributes, $\Sigma$: finite non-empty alphabet
Outputs: $K$: an LKS model
State $initialState, previousState, currentState$
Set of states $S = \emptyset$
Set of transitions $T = \emptyset$
Sequence of actions $\alpha = \langle\rangle$
Boolean $ended$ =false
for all context files $cf \in CF$ do
  $initialState = previousState = currentState = -1$
  for all entries $e \in cf$ do
    Read $e$
    if $e$ is context ID then
      $currentState = e$ // Context set as current state
      $S = S \cup \{e\}$ // Adds state to the set
      if $e == END$ then
        $ended$ =true
      end if
      if $initialState == -1$ then
        $initialState = previousState = e$
      else
        if $\alpha = \langle\rangle$ then
          $append(\alpha, \epsilon)$
        end if
        New transition $t = (previousState, \alpha, currentState)$
        $\alpha = \langle\rangle$ // Resets the sequence of actions
        $T = T \cup \{t\}$ // Adds transition to the set
        $previousState = currentState$ // Updates previous state info
      end if
    else
      if $a \in \Sigma$ then
        $append(\alpha, a)$ // If part of alphabet, adds to sequence
      end if
    end if
  end for
  if (not $ended$) OR ($\alpha \neq \langle\rangle$) then
    New state $FINAL$ // Create final state
    $S = S \cup \{FINAL\}$
    New transition $t_f = (currentState, \alpha, FINAL)$
    $T = T \cup \{t_f\}$
    New transition $t_e = (FINAL, \langle\_EXIT\rangle, FINAL)$
    $T = T \cup \{t_e\}$
  end if
end for
Create model $K = (S, initialState, P, \Gamma, \Sigma', T)$, where $\Gamma : S \rightarrow N^P$, with $N$ representing the sum of the ranges of all attributes in $P$, and $\Sigma' = \Sigma \cup \{\epsilon\}$.
return $K$
```

Therefore, Algorithm 3.2 builds an LKS model based on a previously executed implicit state-labelling procedure. For this reason, it does not mention state labels, using the predefined CIDs to refer to states when adding states and transitions to the model.

Note, however, that there may not always be actions occurring between two consecutive contexts in a trace (e.g., lines 1 and 2 of the context trace in Figure 3.12). This is a result of either the selection of the model alphabet Σ or the existence of nested blocks of code. In this last case, the control of the execution passes from the outer block to the inner block without methods being executed (or user-defined actions being reached). For example, the selection statement in line 2 of the log file in Figure 3.11 is executed inside the repetition statement in line 1, with no action in between. Therefore, they originate two consecutive CIDs in the context trace that are not connected by actions (see lines 2 and 3 of Figure 3.12).

To represent that, we use an action ϵ (line 20) to mean an *empty action sequence*, i.e., a transition from one context to another with no actions happening in between. Thus, irrespective of the actions chosen to compose the alphabet Σ of the model, ϵ is always added to this alphabet (line 42). This is necessary in order to be consistent with Definition 3.4, which states that a transition is always labelled with some action from the alphabet of the model.

The execution that generated the logs may terminate normally or abruptly. A program execution may be precociously ended because of user intervention, an exception or any other situation where the program is aborted. Knowing whether the execution finished successfully is important during the analysis.

Because it is difficult to know what caused the termination of the program just by analysing the context traces, we use a special annotation to guide us. This annotation is introduced at the end of the body of the main method and includes a checking of the termination of the program[4]. It produces an `END` output in the trace, which is interpreted as a context annotation when creating the context traces. However, it is not added to the context table, but just written on the context trace.

[4]In the case of Java, we check that all threads created by the program have terminated.

When the end of a context trace is reached, if the last element found is `END`, then the execution ran until the end and, therefore, terminated normally. Note, however, that this means that the class containing the main method must always be annotated along with the relevant classes. We use a flag to signal that this annotation was found (line 14 in Algorithm 3.2). In this case, the produced FSP will contain a transition from the last context to the predefined state `END`, which is used to represent a normal termination.

In a situation where `END` is not found, a *final state* is created to represent a sink state, indicating an abnormal termination (line 34). The terminal state $FINAL$ has always only one transition looping back to it, which is labelled with an *exit action* (line 38). This guarantees that no false deadlock alarms are generated when checking the model.

It is possible that, when combining context traces from different files, we find the `END` annotation in some of them but not in others. For this reason, we also create a final state if we reach the end of the context trace and the last element is an action (i.e., the last sequence of actions is not empty), even if an `END` had been found before in another context trace. This ensures that no actions are lost during the process and also that the final model shows the traces leading to a normal termination as well as the ones that ended abnormally.

Final Model Generation

At this stage, the LKS model is translated into an FSP description. *Finite State Process* (FSP) [MK06] is a process algebra for describing LTS models. 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. A simple process `Proc` can be defined in FSP as follows

```
Proc = (a -> b -> Proc).
```

where `a` and `b` are actions. The *action prefix* `->` is used to define that the process executes a sequence of actions and then behaves like the process at the end of the sequence. In this case, the process executes action `a` and then action `b` and behaves again in the same way, in a recursive manner.

FSP also allows for local definitions, which are described as subprocesses. A *subprocess* is used to describe a specific behaviour of a process. For example, the following process definition

```
Proc = (a -> b -> SubProc),
SubProc = (c -> Proc).
```

describes a subprocess `SubProc`. This defines that process `Proc` executes actions `a` and `b` and then behaves as described in `SubProc`, i.e., it executes action `c` and then behaves as process `Proc` again.

The process algebra also provides a *choice operator* `|`, which, as the name indicates, allows the representation of alternative behaviours, such as in the process definition below.

```
Proc = (a -> b -> SubProc
       |d -> Subproc2),
SubProc = (c -> Proc),
SubProc2 = (e -> f -> SubProc
           |g -> Proc).
```

Note that the execution of the behaviour defined in a subprocess can also lead to another subprocess, and that that subprocesses can have alternative behaviours as well.

We translate the LKS model $K = (S, s_i, P, \Gamma, \Sigma, T)$, which was built by Algorithms 3.1 and 3.2, 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 $\alpha = \langle a_1 ... a_n \rangle$ and $a_1, ..., a_n \in \Sigma$, 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 the representation of the empty sequence ϵ in the FSP description;

- Alternative behaviours of a subprocess are defined using the choice operator.

Figure 3.13 presents part of the FSP description generated for our example of the editor. Subprocess `Q0` represents the initial state.

```
Editor = Q0,
Q0 = (null -> Q1),
Q1 = (null -> Q2),
Q2 = (null -> Q3),
Q3 = (null -> Q4),
Q4 = (open -> Q5),
Q5 = (null -> Q6
     |null -> Q13),
Q6 = (null -> Q7),
Q7 = (null -> Q8),
Q8 = (edit -> Q9),
Q9 = (null -> Q10
     |null -> Q16
     |null -> Q19
     |null -> Q22),
...
```

Figure 3.13: Example of generated FSP description.

It can be seen that our models may have non-deterministic choices, in particular involving action `null`. This is usually a consequence of the chosen alphabet and 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, as it 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`.

In other situations, the lack of the necessary information to distinguish states results in two or more similar states being merged, which also originates points of non-determinism. This means that the level of abstraction does not provide sufficient detail about the contexts to resolve the non-determinism. Note that the necessary additional information may involve the value of local variables used in the control predicates, which are not considered in our models.

Having the produced FSP description, we generate a graphical representation of the LTS described in FSP using the LTSA tool[5] [MK06]. With the FSP description of the editor, we produced the LTS shown in Figure 3.14.

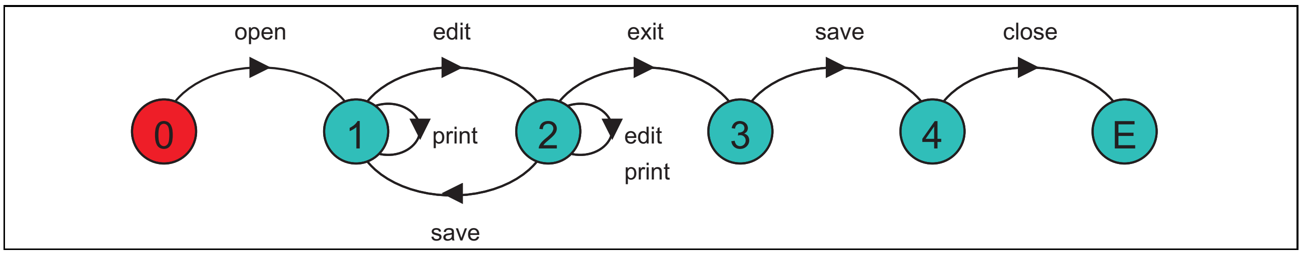


Figure 3.14: LTS model of the editor.

Note that, to produce this model, we applied a hiding operation [MK06] to action `null`. The hiding operator for this action is introduced automatically on the creation of the FSP description. We also minimised the model and made it deterministic using the algorithms provided by the LTSA tool[6]. Although these operations are not required when checking a property, they allow a better visualisation of the model. Moreover, minimising and making the model deterministic does not change the analysis results, since these operations produce an equivalent model that has no empty transitions and no non-deterministic choices.

Even though it is based on a single trace, the model shows some correct relations between actions of the editor, according to its implementation (Figure 3.1). For example, action `save` cannot happen before an occurrence of action `edit`. Moreover, the model also shows that `open` must happen before any occurrence of `edit`, `print`, `save` or `close`. However, the lack of more traces prevented the model from including the possibility of exiting the program at any time.

It is also important to note that this model includes behaviours that were not described in the trace, such as the possibility of repeating the command `print` infinitely. Actually, the trace did not even include a sequence of two consecutive actions `print`. This additional behaviour

[5]Available at http://www.doc.ic.ac.uk/ltsa.
[6]See Chapter 5 for more information on these operations.

could be inferred because the context trace shows that this action happens inside a loop (the execution goes back to the same context) and that it is always an enabled action after a file has been opened (when attribute `isOpen` is true). For the same reason, `print` can also be repeated infinitely on state 2, which represents the context where the file has been edited.

3.3 Dealing with Concurrency

The approach herein presented can also be applied to build models of concurrent systems. Concurrent systems presuppose the parallel execution of multiple processes that can interact. This interaction can be direct, through the exchange of messages between them, or indirect, involving the use of shared resources.

In this work, a *process* is an executing instance of a component. In Java, a component usually corresponds to a class and a process is, therefore, an instance of this class running as a thread. Thus, we model concurrent systems through the construction of multiple models - where each model represents the behaviour of a component - and their possible interactions through actions. Actions are, therefore, used to represent dependencies between processes.

3.3.1 Model Composition

The model of a concurrent system is created by composing the models of each component of the system, generated as described in the previous section. To do so, we apply a parallel composition operation based on the one defined in CSP [Hoa85], where there is a distinction between local and shared actions. A *local action* of a process is an action that is visible only to the process and, thus, does not affect the execution of other processes. A *shared action*, on the other hand, represents an interaction between processes that depend on each other to execute.

This dependence is characterised by a mechanism of *synchronisation* on shared actions. Therefore, given two LTS models $M1$, with alphabet $\Sigma(M1)$, and $M2$, with alphabet $\Sigma(M2)$, they will synchronise when composed if $\Sigma(M1) \cap \Sigma(M2) \neq \emptyset$. In practice, this means that one of the processes described by one of the two models can invoke a method of the other. When the invocation occurs, the caller process stops, waiting for the callee to respond, after which it resumes its execution.

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* [Hoa85]. Therefore, they are executed one at a time, in any order.

This idea of synchronisation on shared actions and interleaving of local actions is represented in FSP using the parallel composition operator `||`. Hence, writing `P || Q` means that a process `P` runs in parallel with a process `Q`. In this composition, therefore, shared actions cause synchronisation and local actions are interleaved. A process definition of this form `||R = (P || Q)` describes a process `R` that is a parallel composition of processes `P` and `Q`.

Note, however, that the model composition is not fully automatic. We build one separate model for each class of the system, but do not generate the process definition of the composition. Doing it this way, we allow the user to experiment with different scenarios involving the generated models.

3.3.2 Active and Passive Processes

As said before, each model represents a component of the system that can originate processes. According to [MK06], these processes can be of two types:

- *Active*: An active process is one that runs with its own thread and may interact with other processes during its life cycle;
- *Passive*: A passive process does not have an internal thread. Its execution depends on other processes invoking its methods. A passive process usually represents a monitor that controls access to a shared resource (e.g., a buffer or a shared printer).

Active processes can interact with either other active processes or passive processes. Passive processes can only interact with active processes. In this case, the interactions are always initiated by the active process.

As commented before, the interaction between processes occurs through shared actions. It involves the blocking of its initiator until a response is obtained. This blocking represents a method call where the caller waits for the callee to complete the execution of the method and return the control to it. When relating only active processes, a shared action can represent that a call to a method occurred and the callee immediately executed the method called, returning the control to the caller. In this case, except for abnormal situations (e.g., abortion of execution), the action in the model describes a method that was called and executed through.

A different scenario occurs when the interaction involves a passive process. Because they normally describe shared resources controlled by a monitor, they include an extra blocking mechanism. This mechanism is implemented within the method body using the wait-notify scheme, which restricts the access to a method according to a certain condition (guard). In practice, this means that, though the method is called, its complete execution is not necessarily immediate, as it depends on the evaluation of the condition guarding the method.

In the above mentioned situation, the active process may call the method but become blocked inside of it. In this case, we need to use a different approach to be consistent with our representation of an action, where the execution of a shared action means the releasing of the caller process to resume. That is, we cannot abstract actions of a passive process in the same way as we do for an active process.

For this reason, we provide the possibility of building different models for active and passive processes. This distinction is determined by the way we use method annotations. Remember that we insert an action annotation at the beginning and another at the end of a method body or method call (see Figures 3.8 and 3.9). For active processes, we build the models using the annotation indicating the beginning of a method execution, whereas we use the annotation representing the end of the method execution when creating the model of a passive process. This guarantees that, in both cases, an action means that the corresponding method was executed.

Note, however, that it is not easy to automatically identify active and passive processes. In fact, even deciding which processes will be active and which will be passive during the development of a system is already a hard task [MK06]. Therefore, we choose to leave the option to the user. The LTSE tool, presented in Chapter 5, allows the choice of which representation (active or passive) should be used to build the model. Thus, it is possible to define how to appropriately interpret the occurrence of an action in the model of a process and its participation in the interactions in a composed model.

3.3.3 Example of Concurrent System

An example of model extraction applied to a concurrent system is now presented. It is based on the bounded buffer system described in [HP00]. The system is composed of three processes: a producer, a consumer and a buffer. Their source codes are shown in Figure 3.15, Figure 3.16 and Figure 3.17, respectively.

```
 1  class Producer extends Thread {
 2    static final int COUNT = 6;
 3    private Buffer buffer;
 4
 5    public Producer (Buffer b) {
 6      buffer = b;
 7      System.out.println (”Producer started”);
 8      this.start ();
 9      #action:”p_starts”;
10    }
11
12    public void run () {
13      AttrData ad = null;
14      for (int i = 0; i < COUNT; i ++) {
15        ad = new AttrData (i, i * i);
16        buffer.put (ad);
17        yield ();
18      }
19      buffer.halt ();
20      System.out.println (”Producer ends”);
21      #action:”p_stops”;
22    }
23  }
```

Figure 3.15: Source code of producer.

The main method, implemented in a separate class, starts one producer and one consumer. Note that we have introduced some user-defined actions (e.g., lines 9 and 21 of Figure 3.15), which are used to help understand the behaviour of the processes.

```
class Consumer extends Thread {
  static final int COUNT = 6;
  private Buffer buffer;

  public Consumer (Buffer b) {
    buffer = b;
    System.out.println ("Consumer started");
    this.start ();
    #action:"c_starts";
  }

  public void run () {
    AttrData[] received = new AttrData[COUNT];
    int count = 0;
    try {
      while (count < COUNT) {
        received[count] = (AttrData) buffer.get ();
        count++;
      }
    }
    catch (HaltException e) {
      #action:"halt_exception";
    }
    System.out.println ("Consumer ends");
    #action:"c_stops";
  }
}
```

Figure 3.16: Source code of consumer.

The producer and the consumer were identified as active processes, whereas the buffer was considered a passive process. Following our approach, one model is built for each component, using alphabet $\Sigma = \{$`p_starts`, `c_starts`, `p_stops`, `c_stops`, `put`, `get`, `halt`, `p_waits`, `c_waits`, `halt_exception`$\}$ and a set of attributes $P = \{$`usedSlots`, `halted`$\}$.

Three situations were considered when generating the traces:

1. The consumer waits (no alteration of the code);
2. The consumer starts with a delay and makes the producer wait (delay introduced in the initialisation of the consumer);
3. There is a delay between each attempt of the consumer to access the buffer, so that the producer terminates early enough to cause the halt exception (delay introduced before each call to method `get`).

The (minimised and deterministic) LTS models of producer, consumer and buffer are shown, respectively, in Figures 3.18, 3.19 and 3.20. Producer and buffer synchronise on shared actions

```
 1  class Buffer implements BufferInterface {
 2    static final int SIZE = 3;
 3    protected Object[] array = new Object[SIZE];
 4    protected int putPtr = 0;
 5    protected int getPtr = 0;
 6    protected int usedSlots = 0;
 7    protected boolean halted;
 8
 9    public synchronized void put (Object x) {
10      while (usedSlots == SIZE)
11        try {
12          System.out.println (”producer wait”);
13          #action:”p_waits”;
14          wait ();
15        }
16        catch (InterruptedException ex) { }
17      System.out.println (”put:” + putPtr);
18      array[putPtr] = x;
19      putPtr = (putPtr + 1) % SIZE;
20      if (usedSlots == 0)
21        notifyAll ();
22      usedSlots++;
23    }
24
25    public synchronized Object get () throws HaltException {
26      while (usedSlots == 0 && !halted)
27        try {
28          System.out.println (”consumer wait”);
29          #action:”c_waits”;
30          wait ();
31        }
32        catch (InterruptedException ex) { }
33      if (halted) {
34        System.out.println (”consumer gets halt exception”);
35        #action:”halt_exception”;
36        HaltException he = new HaltException ();
37        throw (he);
38      }
39      System.out.println (”get:” + getPtr);
40      Object x = array[getPtr];
41      getPtr = (getPtr + 1) % SIZE;
42      if (usedSlots == SIZE)
43        notifyAll ();
44      usedSlots--;
45      return x;
46    }
47
48    public synchronized void halt () {
49      System.out.println (”producer sets halt flag”);
50      halted = true;
51      notifyAll ();
52    }
53  }
```

Figure 3.17: Source code of buffer.

`put` and `halt`, whereas consumer and buffer synchronise through the execution of actions `get` and `halt_exception`.

It is important to notice that the model of the buffer (Figure 3.20) includes alternative behaviours in state 2, where actions `put`, `c_waits` or `halt` may be executed. This represents the blocking point in the buffer code (lines 26-32 in Figure 3.17), where the consumer may block,

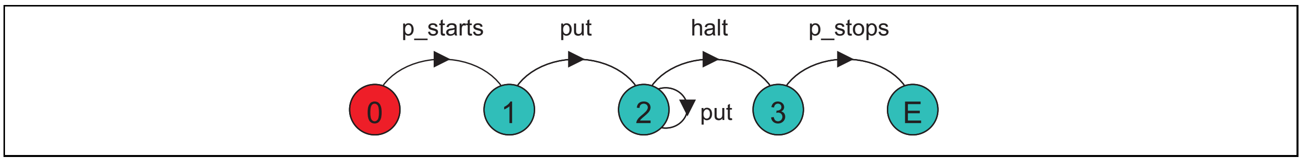


Figure 3.18: LTS model of the producer.

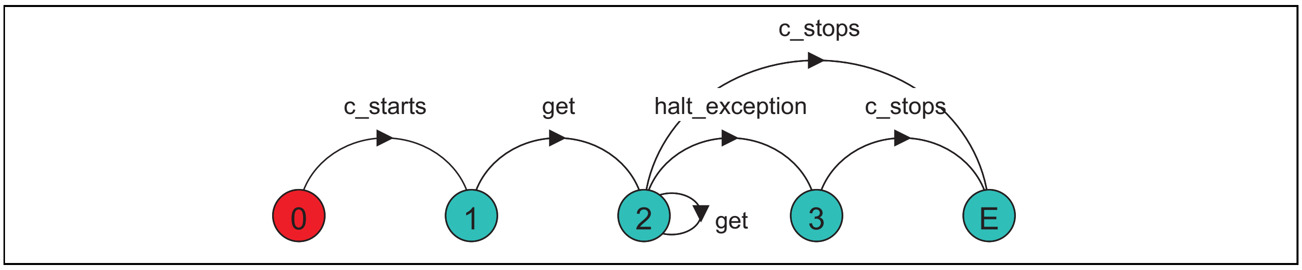


Figure 3.19: LTS model of the consumer.

waiting for the producer to insert the next item in the buffer (if all items have been collected, then action `halt` is executed). The situation where the producer can also become blocked (lines 10-16 in Figure 3.17) appears in state 5.

Based on this, the composite model `||BoundedBuffer = (Producer || Consumer || Buffer)` (not shown here) includes behaviours where the producer starts before the consumer (i.e., `p_starts` happens before `c_starts`) and others where it is the consumer that starts first. Similarly, `p_stops` and `c_stops` can be also interleaved.

3.4 Summary and Discussion

This chapter presented the main ideas of the approach for model extraction using contexts. Contexts identify particular situations during the execution of a system, considering the current block of code and the system state, which is composed of values of a selected set of attributes.

We showed how the source code is instrumented using predefined annotations to capture context information, involving the blocks of code executed and the values of monitored attributes. The instrumented code is then executed to produce traces, which are recorded in log files.

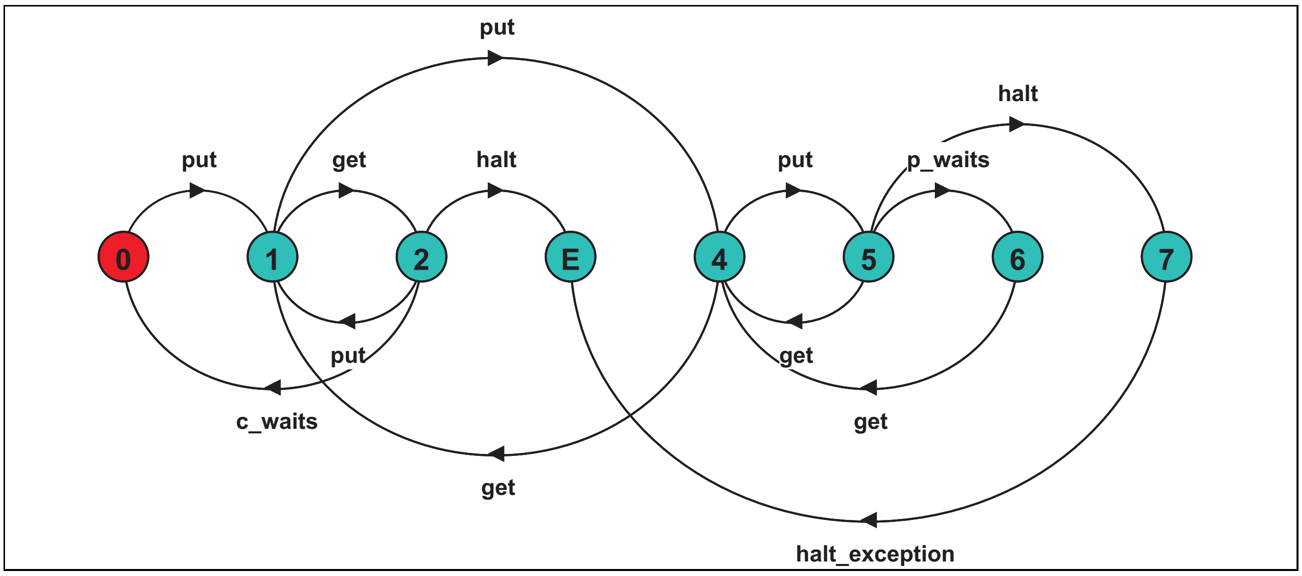


Figure 3.20: LTS model of the buffer.

Using this context information, we produce an implicit LKS model whose states are labelled with the values of the system state and whose transitions have the executed actions as labels. Based on this LKS model, we then create an FSP description where a process definition is generated for each class of the system for which traces were collected.

These models can be combined in a composed model, allowing the application of our approach to concurrent systems. We use the parallel composition ideas from Hoare's CSP [Hoa85] determining that models synchronise on shared actions and local actions are interleaved.

In the next chapter it will be discussed the formal basis of the described approach. The formal description of the mappings applied during the model extraction process and the relations between the models we generate are presented. Among these relations, the focus is on the refinement relation, which allows the augmentation of an existing model with the purpose of modifying its level of abstraction and yet preserve previously checked properties.

Chapter 4

Formal Foundations of the Approach

This chapter contains a discussion on the formal foundations of our approach, presenting our general mapping from a concrete system to an LKS model based on context information. We then show how we map this LKS model into an LTS model by eliminating the state labels.

An analysis on the abstractions our approach produces is also presented, discussing the completeness and correctness of the models we generate with respect to the concrete systems they represent. We comment on how completeness can be improved by the addition of new traces to an existing model and correctness could be achieved by refining the model. We present our refinement technique and formal proofs that this is a property-preserving process.

To give a general view of the mappings we apply and the relations between the source and the target of each of these mappings, we present the diagram in Figure 4.1.

$$\begin{tikzcd}
Prog_P \arrow[r, "m_1"] & K_P \arrow[r, "m_2"] \arrow[d, "r_1"] & M_P \arrow[d, "r_2"] \\
Prog_{P'} \arrow[r, "m_1"] & K_{P'} \arrow[r, "m_2"] & M_{P'}
\end{tikzcd}$$

Figure 4.1: Diagram of mappings and relations.

In the diagram, m_1 is a mapping that builds an LKS K_P based on the context information extracted from a program $Prog_P$ of a system, using a set of attributes $P \subseteq P(Prog)$. M_P is an LTS model obtained from K_P through a mapping m_2.

$Prog_{P'}$ represents the same program code as $Prog_P$, but using a set of attributes P' instead of P, such that $P \subseteq P'$. We build an LKS $K_{P'}$ from the context information provided by $Prog_{P'}$ through the same mapping m_1 applied before. Using mapping m_2 again, $M_{P'}$ is the LTS model generated based on $K_{P'}$.

Relation r_1 corresponds to a property-preserving relation between the LKS models K_P and $K_{P'}$, such that the latter preserves LTL properties [MP92] that hold in the former. The same applies to relation r_2, which defines $M_{P'}$ as preserving LTL properties of M_P. We will show that both relations are in fact refinements. We assume that the exact same test cases are used to produce traces in both programs and that the system is deterministic.

4.1 Formal Mappings

When creating a model of a system, we are producing an *abstraction*. This abstraction is a representation of something *concrete* where some details are left out. In our case, a model M is an abstraction of a concrete system $Prog$, where M does not include implementation details such as data structures used in $Prog$. Therefore, M *abstracts* $Prog$.

As previously discussed (see Chapter 2), the use of abstractions helps model checking tools cope with the complexity and size of systems by allowing the generation of models that are tractable by these tools. However, the model construction should guarantee enough precision to allow the checking of the necessary properties.

This section describes the two mappings from the diagram in Figure 4.1. The first mapping (m_1) corresponds to building an LKS model from context information from an existing implementation. As for mapping m_2, it describes how we create an LTS model using the LKS model generated using mapping m_1.

4.1.1 Mapping from Implementation to Abstract Model

We use a formalism based on finite-state machines to build models. Thus, we interpret the behaviour of a system as a set of reachable states and events (actions) that trigger a change of state. Recalling Section 3.1, we consider states (i.e., contexts) comprising a control component, which is the combination of a block of code and its associated control predicate, and a data component, representing the valuation of a set of attributes (system state).

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

Our mapping from the context information collected from $Prog$ to an LKS $K = (S, s_i, P, \Gamma, \Sigma, T)$ (m_1 in Figure 4.1) involves translating concrete states of $Prog$ into abstract states of K and modelling the change from one concrete state to another as transitions in K. This occurs as described bellow:

- Every concrete state $\theta = (q, v) \in \Theta(Prog)$, where $v = \{val(p_1), ..., val(p_n)\} \in V(Prog)$, is modelled by an abstract state $s \in S$. This abstract state s is derived from a context ID appearing in the context traces generated by $Prog$ and includes only the values of attributes in a selected set $P \subseteq P(Prog)$, such that $\Gamma(s) = v'$, where $v' = \{val(p_k), ..., val(p_m)\}$, for $1 \leq k \leq n$ and $m \geq k$, and $val(p_k), ..., val(p_m) \in P$. For this reason, each abstract state s may represent a set of concrete states $\Theta(Prog)_s = \{\theta_1, ...\theta_x\}$, where $\Theta(Prog)_s \subseteq \Theta(Prog)$. These concrete states are indistinguishable when the information to be used for comparison is restricted to system states considering only attributes in P;

- The initial state $s_i \in S$ models a concrete state $\theta_i = (q_i, v_i) \in \Theta(Prog)$, where $v_i = \emptyset$ and, thus, $\Gamma(s_i) = \emptyset$;

- $\Sigma \subseteq Act$ and, therefore, the alphabet of the model is also restricted to a subset of that of the program;

- The transition relation T is defined in the following way. Given a set of attributes $P \subseteq P(Prog)$, let s and s' be two abstract states of K. Abstract state s models a set of concrete states $\Theta(Prog)_s = \{\theta_1, ..., \theta_n\}$, such that $\Theta(Prog)_s \subseteq \Theta(Prog)$, 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 $\Theta(Prog)_{s'} = \{\theta'_1, ..., \theta'_m\}$, such that $\Theta(Prog)_{s'} \subseteq \Theta(Prog)$, where, for $1 \geq j \geq m$, $\theta'_j = (q'_j, \{v'_j\} \cap V(P))$. Let $\alpha = \langle a_1 ... a_t \rangle$ be a sequence of actions such that $a_1, ..., a_t \in \Sigma \cup \{\epsilon\}$. A transition $(s, \alpha, s') \in T$ exists iff there exists a concrete transition $(\theta, \alpha, \theta') \in \Omega(Prog)$ such that $\theta \in \Theta(Prog)_s$ and $\theta' \in \Theta(Prog)_{s'}$.

This mapping guarantees that no invalid paths of *Prog* will be included in K, considering 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 two concrete states modelled by these abstract states.

Note, however, that it does not mean that infeasible paths will not be part of the model. Infeasible paths may be in the model as a result of the selection of the level of abstraction, which determines the set of concrete states represented by each abstract state. As we will discuss in Section 4.2, it is possible to decrease the level of abstraction to eliminate some of these invalid behaviours.

It is also important to emphasise the comment made in the previous chapter that Algorithm 3.2 does not explicitly build an LKS model. Though it applies the mapping described above to obtain an abstract representation of a concrete system, the LKS model is only used as an intermediate structure that allows us to store the information contained in context files and, subsequently, produce an LTS model from it. Transition labels in the LKS model that the algorithm builds are explicit and correspond to the names of actions happening between contexts in a context trace. State labels, on the other hand, are implicit and used to uniquely identify different contexts when converting traces into context traces (Algorithm 3.1).

4.1.2 Mapping from an LKS to an LTS Model

We now present the mapping m_2 shown in the initial diagram (Figure 4.1), which corresponds to converting an LKS model, created using mapping m_1, into an LTS model. As we do not explicitly build an LKS model, the mapping described here is a transformation from this intermediate structure, which implicitly includes state labels, to a simpler structure - an LTS model - that does not contain state labels. This process is necessary for the creation of an FSP description, since FSP is the process algebra we use to describe LTS models. 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 $Prog$, as presented in Definition 3.4, which was obtained through the model construction process discussed before. Using K, we apply a mapping 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, and thus the definitions previously presented for an LKS (enabled actions, behaviour, language and path) also apply for an LTS. Therefore, an LTS can be obtained from an LKS simply by ignoring state labels - i.e., if 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 without its label, i.e., $\Gamma(s') = \Gamma(s) \setminus P$;
- $\Sigma' = \Sigma$; 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 [LMC01], where LTL is applied to CSP according to an association of propositions with actions (called ALTL in [GM03]). 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 4.1. *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 $K \models \phi$, where ϕ is an LTL property over the alphabet Σ. If K satisfies ϕ then, for all $\pi \in L(K)$, $\pi \models \phi$. This means that all behaviours in $L(K)$ are behaviours defined according to 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, resulting 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 mapping m_2 in Figure 4.1 does preserve 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 model-checking 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.

4.2 Evaluation of Behaviour Models

In Section 2.1 we discussed the definitions of completeness and correctness of an abstract model with respect to a given system. We presented completeness as the characteristic that describes how much of the complete behaviour of the system a model includes. Correctness was defined as describing how much of the behaviour in a model is not valid (i.e., cannot be executed by the real system).

Our objective is that the models we build allow us to check the necessary properties and have the appropriate level of abstraction to guarantee confidence on the results. For this reason, completeness and correctness (and, therefore, faithfulness) are here considered with respect to the checking of a property of interest, rather than with respect to all behaviours of the system.

It is now discussed the completeness and correctness of the generated models and possible ways of improving them. Moreover, situations considering different possible models according to their completeness and correctness are presented and discussed.

4.2.1 Completeness

The completeness of the generated models depends on the coverage provided by the set of traces used to build them. If the set of traces provides full coverage of the system behaviour, then it is possible to identify all reachable concrete states of the system and all valid transitions. This would allow the construction of a complete model, such that $L(Prog) \subseteq L(M)$. However, this is normally not the case and, therefore, the model is generally an *under-approximation* of the behaviour of the system (i.e., $L(M) \subset L(Prog)$). Thus, it represents only the part of the behaviour observed during the trace generation phase.

Despite being an under-approximation of the behaviour of the whole system, the produced model may be an *over-approximation* of the set of observed behaviours. As commented in Section 3.2, some additional behaviours may be inferred using the context information, which allows the merging of multiple traces and the identification of alternative and recurrent behaviours (loops). This way, the model might include more behaviours than those exhibited by the system and recorded in the collected traces.

When checking a property, it might not be necessary to achieve a complete model. As Jackson [JD96] pointed out, many errors can be identified in a small finite portion of a possibly infinite state space of a program. However, ideally, the model should include all the relevant behaviours to allow the checking of the property. Therefore, one should aim at observing the behaviours that might influence the analysis of the system regarding a specific property and include them in the model.

One possible way of selecting the relevant behaviours is to use a test suite. By choosing test cases, it is possible to control the inputs to the system and, this way, force it to exhibit some particular behaviours. Though testing is not directly connected with this work, the use of test cases to observe specific behaviours can help the construction of models tailored for the checking of properties of interest. For instance, one of the various existing testing coverage criteria [Pat06] could be used to provide this focused generation of traces.

Regardless of the technique used to generate the traces (testing, profiling or monitoring), our approach allows new traces to be incrementally incorporated to the model. Therefore, missing traces can be added to the model to provide information on executions not considered before. This way, it is possible to gradually improve completeness even if an initial model fails to include all the necessary behaviours.

4.2.2 Correctness

A model built using just control flow information is the most abstract model that can be obtained using our approach. This is a model where the set of attributes is empty, and can be used as a starting point. Regrettably, such a model tends to generate violations that are not real; they just occur because the abstraction is too coarse and permits behaviours that are not actual behaviours of the system. The existence of invalid behaviours can affect the checking of a property because one of these invalid behaviours may violate the property during the verification, generating a misleading result.

In this work, correctness can be enhanced through the inclusion of more attributes into the set used to define the system state. Therefore, the correctness of the models depends essentially on the selection of the attributes to form the system state, used to define contexts.

The addition of new attributes to the state information results in a model that is less abstract than the original. Hence, it precludes some behaviours that were allowed in the original model. These precluded behaviours are invalid behaviours that were removed from the model because of the additional behavioural information.

The augmentation of an initial model permits, for example, to tune the model according to a property to be checked. This approach can eventually lead to a model that is correct with respect to a given property (model does not violate the property if it is not violated in the real system), even though it may still contain behaviours not allowed by the implementation of the system. These additional behaviours do not interfere in the checking of the property and, therefore, do not need to be removed.

4.2.3 Interpretation of Property Checking Results

Ideally, if a property ϕ holds in a behaviour model M, then it should also hold in the program $Prog$ represented by M. However, this cannot always be guaranteed unless M is complete and correct with respect to ϕ. In this case, the set of behaviours of the system that affect ϕ and the set of behaviours of the model are the same. Therefore, any LTL property holding in M would also hold in $Prog$, and vice-versa.

As commented before, the generated models cannot always be complete and correct. The quantity and quality of the set of traces used to build them can make them more or less complete, depending on how much coverage is provided. In the same way, the modification of the set of attributes forming the system state alters the correctness of the model. This correctness depends on the relevance of the selected attributes with respect to the system behaviour and the property to be checked.

We now comment on situations where a model is incomplete or incorrect. We discuss the interpretation of property checking results and how to improve completeness and/or correctness in each case.

Complete and Incorrect. By definition, a complete model ($L(Prog) \subseteq L(M)$) does not necessarily imply a correct model ($L(M) \subseteq L(Prog)$). The model may not only be able to completely reproduce all behaviours of the system that are relevant to check a property ϕ, but also describe behaviours not belonging to $L(Prog)$. In this case, though the model is complete, it may also also be incorrect ($L(Prog) \subset L(M)$).

Despite being incorrect, a complete model can be used for checking properties. Because it describes the complete behaviour of the system (and perhaps some other additional inferred behaviours), if a property ϕ holds in the model, it is guaranteed to hold in the system. On the other hand, property violations detected in the model can be actual violations or just false negatives, i.e., violations caused by behaviours not in $L(Prog)$ but present in its superset $L(M)$.

False negatives can usually be eliminated from the model using an abstraction refinement technique. In our approach, as mentioned before, the abstraction refinement process corresponds to the addition of more attributes to the system state (decreasing the level of abstraction). The inclusion of additional attributes should rule out some false negatives.

Note that, in spite of the fact that adding all available attributes to the system state would make the model as closest as possible to the real behaviour of the system, this may be not desirable. Some attributes may have a wide range of values, causing the model to become too large. The number of attributes also contributes to making the model more complex to analyse.

Because of the mentioned reason, a subset of the set of the available attributes should be selected to comprise the system state, instead of just using the whole set. Although this approach does not guarantee a correct model, it helps maintain the model in a size amenable for verification and visualisation. Furthermore, as said before, if the model is complete, the lack of correctness does not completely hamper the checking of properties. It just may make the process longer if false negatives are generated and need to be eliminated.

Incomplete and Correct. A correct model may not be complete. Though it describes only actual behaviours of the system, it may lack some of the behaviours found in $L(Prog)$ (i.e., $L(M) \subset L(Prog)$). In this case, the absence of violations in the model does not necessarily mean that the property holds in the system. This situation may originate a false positive, i.e., the behaviour that violates the property belongs to $L(Prog)$ but not to its subset $L(M)$.

False positives are more difficult to detect than false negatives since there is no indication of a possible problem with the model (no counter-example). Therefore, the only way of preventing false positives is to assure completeness. Some techniques for model extraction guarantee completeness by obtaining the complete CFG of the system [BR02, HJMS02, CDH$^+$00]. As expected, this results in an over-approximated abstraction of the system, which can yield a number of false negatives but guarantees the absence of false positives.

In our approach, assuring completeness corresponds to providing complete behaviour coverage with respect to a property to be checked. If this coverage is not achieved, the absence of violations during the model checking process does not imply that no violation exists. It only means that it is guaranteed that no violation is caused by behaviours in $L(M) \cap L(Prog)$. Any behaviour in $L(Prog) \setminus L(M)$ cannot be guaranteed to not violate the property.

As commented before, it is possible to improve completeness by means of adding new traces to the model. This corresponds, for example, to adding new test cases to a test suite used to generate the traces. The addition of test cases increases the tested situations and may reveal an unknown behaviour, which may violate the property being checked.

Incomplete and Incorrect. In the worst case, a model is incomplete and incorrect. This situation occurs if the set of traces does not include some relevant behaviours with respect to a property to be checked and the set of attributes does not provide the right level of abstraction to avoid false negatives when checking this property.

Since an incorrect model is likely to generate a false negative, correctness is improved first. Once a suitable level of abstraction has been found (i.e., one that does not generate false negatives during the checking of a property), then it is possible to improve completeness with the inclusion of new traces.

In Section 4.4, it will be discussed the procedure applied to checking properties and dealing with false negatives and false positives. This will describe the steps taken during the checking of properties and how to deal with its possible outcomes.

4.3 Abstraction Refinement

This section presents the formal description of the relation between two LKS models generated using different sets of attributes, where one set is a subset of the other. This relation corresponds to relation r_1 in Figure 4.1.

A generated LKS model has the states identified during the construction of the CT and the transitions collected from the logs. It is used to generate an LTS model, as described in Section 4.1, which will be checked against a property. After the model checking process, the model may prove to be too abstract (i.e., incorrect), yielding false negatives. In this case, we apply a refinement process to generate a less abstract model that preserves properties of the original model.

The refinement process is carried out by enlarging the set of attributes used to generate a model, i.e., we add more attributes to an initial set, while using the same traces as before. This enlargement causes a change in the labelling of the states, since the labels have to be modified in order to include the values of the newly added attributes.

Because the state labels change to consider more values of attributes, this can also potentially lead to an increase in the size of the state space. The reason is that the new attributes can reveal different states that were not distinguishable before. For instance, let us suppose a state s labelled with $\langle \texttt{true} \rangle$, representing the value of a boolean attribute p_1. Now, let us say that a new boolean attribute p_2 is added. In this case, s could then be seen as two distinct states: s_1, labelled with $\langle \texttt{true}, \texttt{true} \rangle$, and s_2, labelled with $\langle \texttt{true}, \texttt{false} \rangle$, where the second values in the tuples describe values of attribute p_2.

For a more concrete example, consider the model in Figure 3.4. That was the model of the editor code presented in Figure 3.1, built with an empty set of attributes. In contrast, the model shown in Figure 3.14 was generated using attributes `isOpen` and `isSaved`. Therefore, the former model is more abstract than the latter, which is its refinement. Note how, for example, state 0 of the more abstract model was split into states 0, 1 and 2 in the model considering attribute values.

4.3.1 Refinement Relation

We now present a formal description of the relation between a model and its refinement. This is used to support our claim that augmenting context information results in a more refined model which preserves properties that were satisfied in the original model.

In a refinement process, an original model is said to be an *abstraction* of a refined model, as it does not include some information included in its refined version. In [CCO⁺04], the following definition is presented for an abstraction relation considering LKS models:

Definition 4.1. ***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 $\lambda = \langle s_1 a_1 ... \rangle \in \Lambda(K)$ there exists a path $\lambda' = \langle s'_1 a'_1 ... \rangle \in \Lambda(K_A)$ 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 is 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. Hence, we aim to prove the following theorem:

Theorem 4.2. *Let F_{Prog} be a set of log files recording traces of a program Prog. $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ is an LKS model obtained from Prog following our mapping, using the set of traces $Tr(F_{Prog})$, collected from F_{Prog} during the context table construction (see Algorithm 3.1), and a set of attributes $P_A \subseteq P(Prog)$. If the same set of traces $Tr(F_{Prog})$ is used with a set of attributes $P \subseteq P(Prog)$, such that $P_A \subseteq P$, then we obtain an LKS $K = (S, s_i, P, \Gamma, \Sigma, T)$ such that $K \sqsubseteq K_A$.*

Item 1 of the definition of abstraction (Definition 4.1) is 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 satisfied, because we do not alter the alphabet[1] and, thus, $\Sigma_A = \Sigma$.

Proof of item 3 is broken into three separate proofs, presented next, after which we discuss our proof of refinement based on them. In all proofs, we will 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 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.

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

It is also important to mention that, in the proofs, we will use only transitions labelled with single actions, rather than 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.

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 4.2. ***State-Labelling Relation.*** Given two LKS models $K_A = (S_A, s_{i_A}, P_A, \Gamma_A, \Sigma_A, T_A)$ 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 $\Gamma_A(s_A) = \Gamma(s) \cap P_A$.

Lemma 4.1. *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 $An = \{an_1, ..., an_n\}$ as the set of context annotations in the set of traces $Tr(F_{Prog})$ 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, if the value of p is ignored, s_A and s have the same label and, consequently, are 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 identified as different. Nevertheless, it is easy to see that s and s' are the same state when the value of p is abstracted. Then, if the set of attributes is 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, $(s_A, s) \in SL$. □

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

Proof. Lemma 4.1 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 $Tr(F_{Prog})$ 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 $Tr(F_{Prog})$ 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 $Tr(F_{Prog})$, 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 $Tr(F_{Prog})$, 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 $s_1, ...s_n \in S'$ will result in the same set of actions enabled in the more abstract state s_A. Therefore, $E(s_1) \cup ... \cup E(s_n) = E(s_A)$. □

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 4.3. *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 4.1 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 4.2, $E(s) \subseteq E(s_A)$. 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 4.1, 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 $(s_A, s), (s'_A, s') \in SL$. Therefore, Lemma 4.3 holds. □

Proof of Property-Preserving Refinement

Proof. **Proving Theorem 4.2.** As a result of Lemmas 4.1, 4.2 and 4.3, 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 $\lambda = \langle s_1 a_1 s_2 a_2 s_3 ... \rangle \in \Lambda(K)$ can be mapped into a path $\lambda_A = \langle s'_1 a'_1 s'_2 a'_2 s'_3 ... \rangle \in \Lambda(K_A)$ such that, for $n \geq 1$, $a_n = a'_n$ and $(s'_n, s_n) \in SL$. Therefore, Theorem 4.2 holds. □

In [CCO+04], the authors present a logic that is a superset of LTL, called SE-LTL. They show that, if a property ϕ is expressed in the authors' 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 4.2, we can conclude that:

Corollary 4.1. *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.

4.3.2 Property-Preserving Mapping

The last relation is the one represented by r_2 in the diagram at the beginning of this chapter. It defines that, given two LKS models K_A and K, such that $K \sqsubseteq K_A$, if we generate two LTS models M_A and M, respectively from K_A and K, then there should be a property-preserving relation between them. This is shown in the diagram in Figure 4.2, where `SLE` is the state-label elimination mapping described in Section 4.1 and `Ref` is the refinement relation.

We claim that, 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, the generated LTS models also have a property-preserving relation between them. This relation between the LTS models is also a refinement.

$$\begin{tikzcd}
K_A \arrow[r, "\texttt{Ref}"] \arrow[d, "\texttt{SLE}"] & K \arrow[d, "\texttt{SLE}"] \\
M_A \arrow[r, "\texttt{Ref}"] & M
\end{tikzcd}$$

Figure 4.2: Property-preserving relations diagram.

Theorem 4.3. *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 4.1 (see Section 4.1) demonstrated that eliminating the state labels from an LKS, we obtain and 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 4.1 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 the LTS M_A it generates will also preserve the property, 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 4.1 is a simulation relation [Mil71], 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.

4.4 Property Checking

Once a model has been extracted as described in Chapter 3 and formally presented in Section 4.1, it can be used for model checking. Checking a property against a model means comparing the behaviour specified in the property with that permitted by the model.

It is now discussed the types of properties that can be checked against our models. For each type, it is presented how a property of that type can be specified and how results of the model checking process are interpreted.

After this, it is presented the procedure applied in this work to check properties and adjust a model to allow this checking. An example of property checking is discussed to demonstrate this procedure in practice.

4.4.1 Specification of Properties

Our models can be used to check that safety properties are satisfied. Safety-property checking is executed using the support provided by the LTSA tool.

A *safety property* describes a set of behaviours that a system should include. It is used to check that the system does not engage into any undesired behaviour during its execution [MK06]. Safety properties to be checked against the models we produce can be specified in three different ways, presented next.

Property Automata

The simplest way of specifying a property is using the FSP process algebra. Properties specified in FSP correspond to a deterministic process definition, which is marked with the keyword `property`. It determines sequences of visible actions allowed to happen (trace semantics). These sequences may either correspond to the actions in the alphabet of the model or just be a subset of it. Figure 4.3 shows an example of a property specified in FSP. It describes a simple property determining that a system should execute an action `in` and then an action `out`, in this order.

```
    property IN_OUT = IN,
    IN = (in -> OUT),
    OUT = (out -> IN).
```

Figure 4.3: Example of property specified in FSP.

An FSP property automaton is composed with the model of a system to ensure that the model does not include behaviours not allowed by the property. In the case of our example, invalid behaviours would be ⟨`in in`⟩, ⟨`in out out`⟩ and any other behaviour that does not follow the pattern of alternating between `in` and `out`, where the former is always the first action to happen.

The execution of an invalid behaviour, leads the property model to an *ERROR* state. An *ERROR* state is represented in the LTS model of the property by number -1. All transitions leading to this state correspond to disallowed transitions, i.e., transitions that cause the execution of an invalid behaviour. Figure 4.4 presents the LTS model that describes the property specified in Figure 4.3.

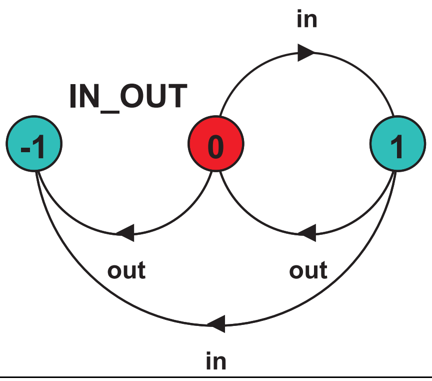


Figure 4.4: Example of LTS model of an FSP property.

When the property automaton is composed with the system model, if the *ERROR* state is unreachable, then the property holds. Otherwise, it is violated and the behaviour that causes the violation is a sequence of actions that leads from the initial state to the *ERROR* state.

Consider the process definition presented in Figure 4.5 of a process `P1`. As it is possible to see, process `P1` does not violate the property. Therefore, the composition of `P1` with property `IN_OUT` does not include the *ERROR* state, as shown in Figure 4.6.

```
    P1 = (in -> process -> out -> P1).
```

Figure 4.5: FSP of process `P1`.

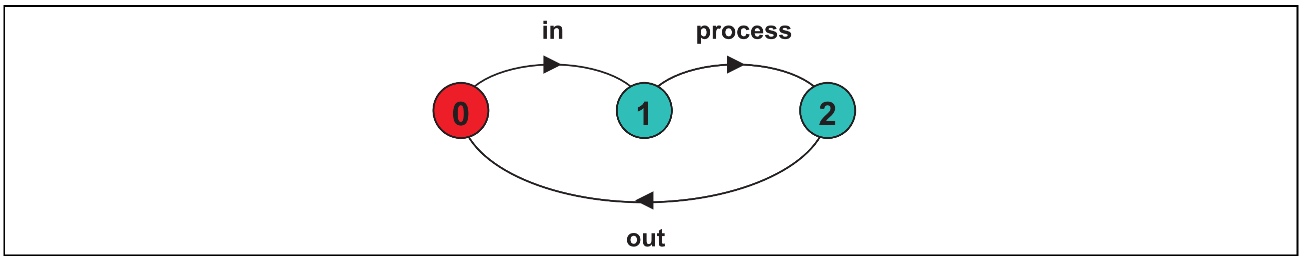


Figure 4.6: LTS model of process `P1` composed with property `IN_OUT`.

While `P1` does not violate the property, the same does not happen with process `P2`, whose definition is presented in Figure 4.7. It is easy to see that the process executes the action `in` twice before executing `out`. This is made clear in the composition with property `IN_OUT`, shown in Figure 4.8. Note that the *ERROR* state is reachable and, therefore, the model allows an invalid behaviour, which violates the property.

```
    P2 = (in -> in -> process -> out -> P2).
```

Figure 4.7: FSP of process `P2`.

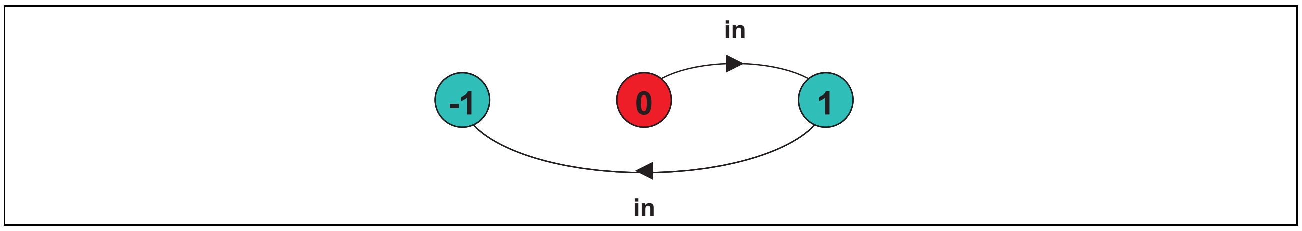


Figure 4.8: LTS model of process `P2` composed with property `IN_OUT`.

LTL Properties

As discussed before, only LTL properties over actions in the system alphabet are considered. Therefore, LTL formulas are specified using action names in combination with the Boolean and LTL temporal operators[2] shown in Table 4.1.

Logical Operators	**Temporal Operators**
$\neg$ (logical negation)	$\Box$ (always)
$\wedge$ (logical AND)	$\Diamond$ (eventually)
$\vee$ (logical OR)	$\bigcirc$ (next)
$\Rightarrow$ (implication)	$\mathcal{U}$ (until)
$\Leftrightarrow$ (equivalence)	$\mathcal{W}$ (weak until)

Table 4.1: LTL logical and temporal operators.

Using these operators, one can create LTL formulas (properties) to be checked against LTS models. The model satisfies the property if every behaviour in its language satisfies the property.

As an example of an LTL property, let us consider the editor presented in Chapter 3. One property that the system should satisfy is that a document, after having been opened, cannot be saved if it has not been modified. In other words, one can only save a document that is currently open and has been edited. This can be expressed in LTL using the property $\Box(open \Rightarrow (\neg save\ \mathcal{W}\ edit))$.

This property defines that, after the execution of action `open`, action `save` cannot happen before an execution of action `edit`. The property automaton describing this property is shown in Figure 4.9.

When composing this property automaton with the LTS model in Figure 3.14, the *ERROR* state is not reachable. This means that no behaviour accepted by the language of the model permits a behaviour that does not comply with the restriction imposed by the checked property.

[2] Refer to [MP92] for an explanation on the semantics of these operators and to [LMC01] for their meaning in formulas over actions.

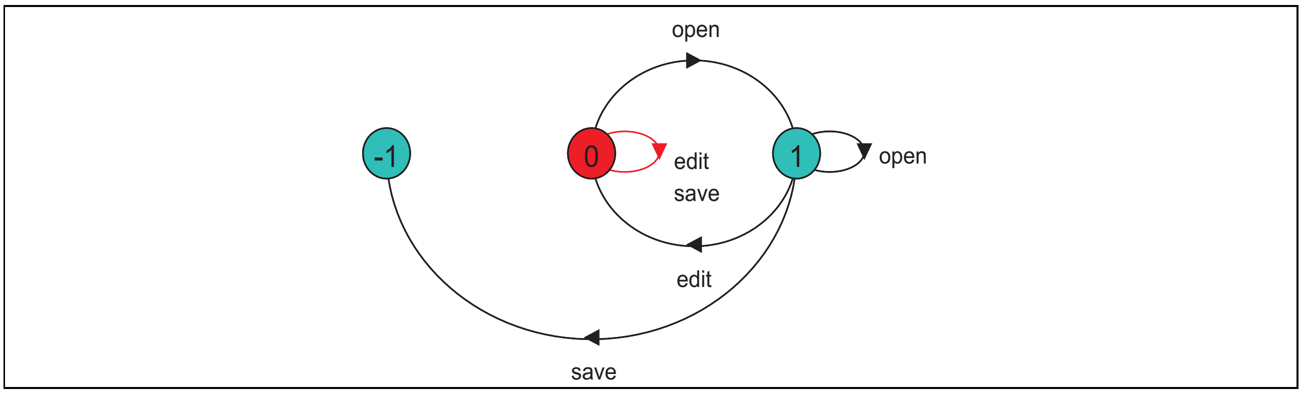


Figure 4.9: Property automata of LTL property.

FLTL Properties

The *Fluent LTL* (FLTL), introduced in [GM03], is a variation of LTL tailored for checking event-based systems. It expands our previous definition of LTL properties over actions with the introduction of the concept of fluent.

A *fluent* is a proposition whose value varies over time according to the execution of actions. A fluent Fl is defined by a set I_{Fl} of initiating actions and a set T_{Fl} of terminating actions. The following formal definition is presented in [GM03], considering a system with alphabet Act:

$$Fl \equiv \langle I_{Fl}, T_{Fl} \rangle, \text{ where } I_{Fl}, T_{Fl} \subset Act \text{ and } I_{Fl} \cap T_{Fl} = \emptyset$$

Fluents can be initialised as true or false. At some point in time, a fluent is true if it has been initialised as true or after the execution of an action $a \in I_{Fl}$. It becomes false if it has been initialised as false or when an action $a' \in T_{Fl}$ is executed. Therefore, generally speaking, a fluent is true from the point in time when an initiating action is executed until a terminating action occurs and it remains false from the moment of the execution of a terminating action until another initiating action occurs.

An action $a \in Act$ naturally defines a fluent that becomes true when a is executed and false when any other action $a' \in Act \backslash \{a\}$ occurs. Therefore, properties specified using LTL over Act can be expressed in FLTL. This means, for example, that the LTL property $\Box(open \Rightarrow (\neg save\ \mathcal{W}\ edit))$ is also an FLTL formula, where, assuming an alphabet $\Sigma \subseteq Act$, each action a in the formula defines a fluent of the form

$Fluent(a) = \langle \{a\}, \Sigma \setminus \{a\} \rangle \ Initially_a$ =false

where $Initially_a$ defines the initial value (true or false) of fluent a.

For the editor system, for example, it would be possible to create the following fluents considering actions of the system alphabet:

$Fluent(Closed) = \langle \texttt{close}, \texttt{open} \rangle \ Initially_{Closed}$ =true

$Fluent(Edited) = \langle \texttt{edit}, \texttt{save} \rangle \ Initially_{Edited}$ =false

$Fluent(Saved) = \langle \texttt{save}, \texttt{edit} \rangle \ Initially_{Saved}$ =true

Using these fluents, we can define an FLTL formula that states that an edited document is eventually saved before being closed. This formula can be specified in FLTL as follows:

$\Box(Edited \Rightarrow (\neg Closed \ \mathcal{W} \ Saved))$

The property automaton for this property is presented in Figure 4.10. It is easy to see that the model in Figure 3.14 does not violate the property, as it is not possible to reach the *END* state without executing `save`.

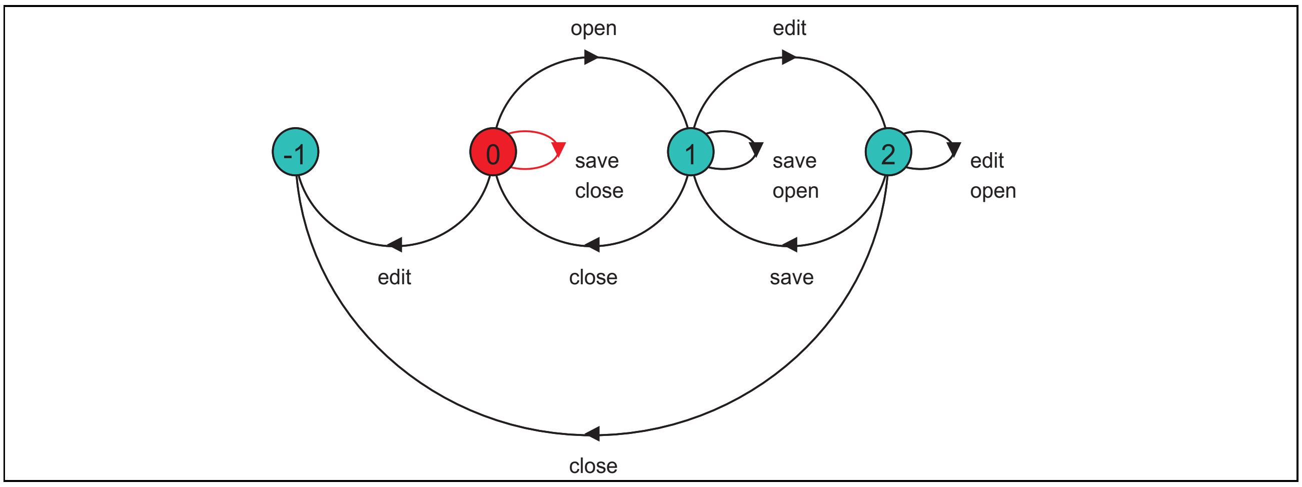


Figure 4.10: LTS model of FLTL property.

4.4.2 Adopted Procedure

Following the model checking process described in Chapter 2, we use the algorithms presented in Chapter 3 to extract a model M from a program $Prog$ (modelling step) and specify a property ϕ as described before (specification step). For the verification step, we use a model-checking tool (LTSA) to check whether $M \models \phi$.

The procedure for property checking applied in this work is described in Procedure 4.1. The steps are presented in the form of subprocedures.

Note that these steps may be taken for each individual model or directly for the composed model. Preferably, the model of each component should be checked against local properties - if they exist - and, after that, then the composed model should be produced and checked against global properties. This allows the application of techniques of compositional reasoning, such as the assume-guarantee paradigm [GL94]. Such techniques can be used to alleviate the state explosion problem [CGP99].

Lines 1 to 7 of the procedure correspond to the steps described in Chapter 3. Firstly, the code is instrumented. Subprocedure $InstrumentCode$ represents the application of the annotation rules using the TXL language, where $Prog$ is the original program and $Prog'$ is its instrumented version. After the instrumentation, traces are generated. We use TC to represent a set of test cases used to generate traces from the instrumented version of the code (subprocedure $GenerateTraces$). This creates the set of log files F_{Prog}.

Next, the alphabet of the model (Σ) is selected as a subset of the alphabet of the system (Act), such that it matches the alphabet $\Sigma(\phi)$ of a property ϕ to be checked, which is supplied by the user according to one of the formats previously presented. The attributes to compose the system state are selected from the set of attributes of the system (line 4).

Subprocedures $BuildCT$ and $CreateLKS$ correspond to the executions of Algorithm 3.1 and Algorithm 3.2, respectively. As a result, an LKS model K is generated, which is then mapped into an LTS model M using the mapping described in Section 4.1.

Procedure 4.1 *PropertyChecking*

```
 1 $Prog' = InstrumentCode(Prog)$
 2 $F_{Prog} = GenerateTraces(TC, Prog')$
 3 Select $\Sigma \subseteq Act$ such that $\Sigma == \Sigma(\phi)$
 4 Select $P \subseteq P(Prog)$
 5 $CF = BuildCT(F_{Prog}, P)$
 6 $K = CreateLKS(CF, P, \Sigma)$
 7 $M = MapToLTS(K)$
 8 $e = CheckProperty(\phi, M)$
 9 if $e == \langle a_1 ... a_n \rangle$ (i.e., $\exists e \in L(M)$ s.t. $e \notin L(\phi)$) then
10   Violation found:$M \not\models \phi$
11   Create test case $tc$ to try to reproduce $e$ in $Prog$
12   $TC = TC \cup \{tc\}$
13   Execute $tc$ using $Prog$
14   if $e \in L(Prog)$ then
15     Real violation found: $Prog \not\models \phi$
16     Fix error: modify $Prog$ so that $L(Prog) = L(Prog) \setminus e$
17     Go back to 1
18   else
19     False negative occurred: $e \notin L(Prog)$
20     Select $P' \subseteq P(Prog)$ such that $P \cap P' == \emptyset$
21     $P = P \cup P'$
22     Go back to 5
23   end if
24 else
25   No violation found: $L(\phi) \subseteq L(M)$, therefore, $M \models \phi$
26 end if
```

Subprocedure *CheckProperty* represents the verification step of the model checking process, when property ϕ is checked against model M using the model checker. As in any other property checking process, there are two possible outcomes: either a violation is found (i.e., $M \not\models \phi$) or no violation is detected. As discussed earlier in this chapter, the correctness and completeness of the model decisively influence the results of the verification step.

If a violation is found (lines 9-10, where e is an error trace), it may be real or a false negative. Using the error trace e generated by the model-checking tool, one can try to replay this behaviour using the code, for example, by applying a new test case (line 11). This test case can then be added to the existent test suite (line 12).

If the real system can behave as described in the error trace (line 14), then we have found an actual violation and the code needs to be fixed[3], after which, we go back to the beginning to generate a new model.

If the violation is not real (line 19), we enter the refinement process. The model is refined adding more attributes to the initial set (lines 20-21). This process is manual and requires knowledge about the implementation of the target system. There are no predefined rules as to which attributes should be chosen and it may turn out to be a trial-and-error situation.

From the experience gained from the development of several examples during this work, it seems that attributes involved in control predicates are the choice most likely to provide the necessary increase in correctness. If the infeasible behaviour caused a point of choice that is not clear at the current level of abstraction, then adding the attributes that affect this choice might introduce a split of paths, thus possibly eliminating the invalid behaviour. The information on the error trace can - and should - also guide this choice.

After the refinement process, which may be repeated as many times as necessary, we go back to the construction of the LKS model, but now using the new set of attributes. If the resulting model now preserves the checked property (line 25), then the property checking process terminates and the property holds.

Once again, note that reaching the end of the property checking process only means that an abstraction suitable for the desired specification was found after possibly successive refinements. If the model is incomplete with respect to the system behaviour, $M \models \phi$ does not necessarily imply $Prog \models \phi$.

In our procedure, we employ the use of test cases, although this is not required to apply the proposed approach for model extraction. This is due to two main reasons: firstly, using test cases, it is possible to control the situations for which traces are generate and, this way, we can observe behaviours of interest (e.g., behaviours that affect the property being checked); and, secondly, error traces can serve as a basis for the creation of new test cases, which helps improve the completeness of the model.

[3]We assume that the property is always correct and, therefore, a violation is a result of a problem in the code.

By following this procedure, we believe that it is possible to improve confidence on the correct behaviour of the systems being analysed. The use of testing to generate the traces allows an early detection of some errors and their correction even before the model extraction. The posterior application of model checking can possibly uncover additional errors missed during the testing phase, if any exists, thus increasing the set of behaviours analysed and, consequently, reducing the number of non-detected existing errors.

4.4.3 Property Checking Example

We now apply our property checking procedure to check properties of the editor system presented at the beginning of Chapter 3 (see Figure 3.1). For this experiment, we used the following initial parameters:

- Set of test cases TC
  - $T1 = \langle$ `0 1 3 2 4` $\rangle$
  - $T2 = \langle$ `0 1 2 4 0` $\rangle$
  - $T3 = \langle$ `0 1 2 4 1` $\rangle$
  - $T4 = \langle$ `0 2 1 1 3 4` $\rangle$
  - $T5 = \langle$ `0 1 1 3 4` $\rangle$
- Alphabet $\Sigma = \{\texttt{open}, \texttt{edit}, \texttt{print}, \texttt{save}, \texttt{exit}, \texttt{close}\}$
- Set of attributes $P = \emptyset$

The properties we would like to check are defined as follows, where we refer to the fluents *Closed*, *Edited* and *Saved* previously defined:

$\phi_1 : \ \Box(\mathit{Closed} \Rightarrow \neg(\texttt{edit} \lor \texttt{print} \lor \texttt{save} \lor \texttt{exit}))$

$\phi_2 : \ \Box(\mathit{Saved} \Rightarrow \Box(\neg \bigcirc \texttt{save} \ \mathcal{W} \ \mathit{Edited}))$

Property ϕ_1 determines that actions `edit`, `print`, `save` and `exit` are not allowed to be executed before a document has been opened. Property ϕ_2 specifies that, if a document has not been modified, then action `save` cannot occur until the document is edited.

Applying the model extraction process, the model presented in Figure 4.11 was produced. Checking these properties against the model in Figure 4.11, we verify that the model violates both of them. We first analyse property ϕ_1.

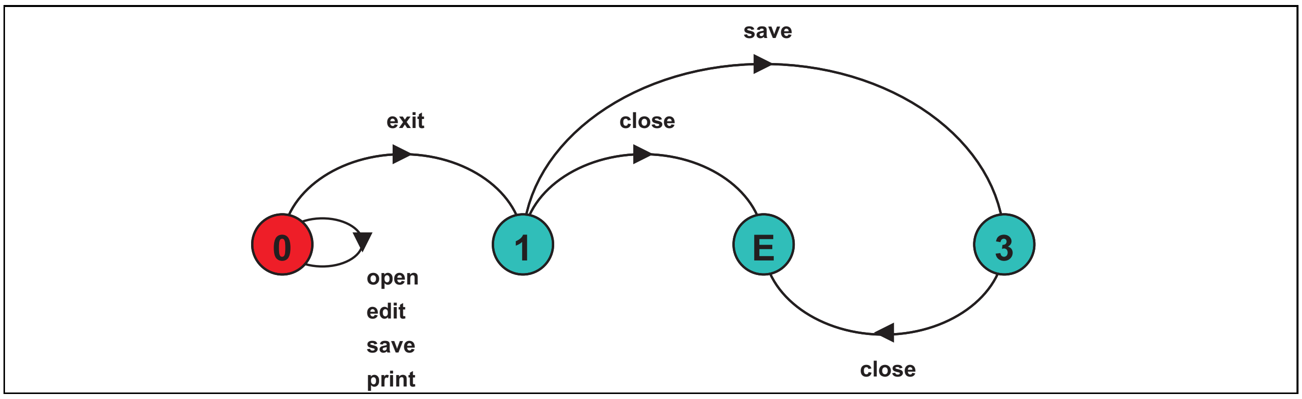


Figure 4.11: Initial model of the editor system.

For ϕ_1, the error trace is $\langle$ `edit` $\rangle$, which indicates that action `edit` can happen before a document has been opened. However, it is not possible to replay this trace in the code in Figure 3.1. Hence, we have found a false negative.

In this example, there are only two attributes we can use to refine the model: `isOpen` and `isSaved`. Intuitively, attribute `isOpen` is a good option, since it is used in the control predicate of the block of code including the call to method `edit` (see lines 16-17 in Figure 3.1). Also, it is this attribute that indicates whether a document is open, which affects the value of fluent *Closed*. We then update the set P, so that now $P = \{$`isOpen`$\}$. The alphabet and test cases remain the same.

After the inclusion of the new attribute in the system state, we generate a new model, shown in Figure 4.12, which is a refinement of the model in Figure 4.11. Note that the refined model successfully shows action `open` as the only enabled action in the initial state. As expected, this model satisfies property ϕ_1. Unfortunately, it still violates property ϕ_2, producing the error trace $\langle$ `open save` $\rangle$.

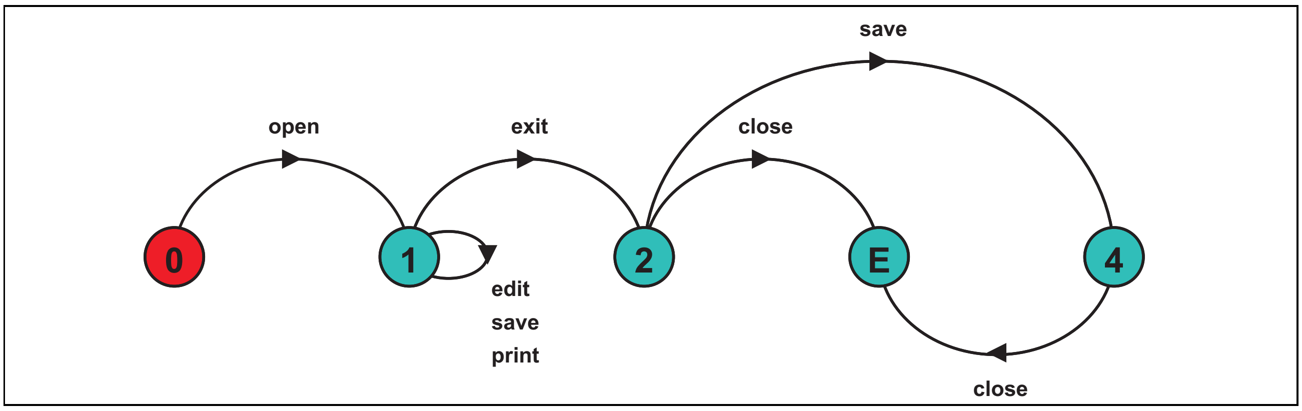


Figure 4.12: Refined model of the editor system.

The error trace shows that it is possible to save a document that has not been edited, contradicting what is stated in property ϕ_2. Once again, we verify that this trace is not feasible using the implementation in Figure 3.1 and, therefore, we face another false negative.

At this point, we select attribute `isSaved` to refine the model. Even though it is the only available attribute to be added to the system state, it would surely be our first choice if the same heuristic used before were applied again, since it controls whether a document has been saved or not. Using $P = \{\texttt{isOpen}, \texttt{isSaved}\}$, we produce a more refined model, which is presented in Figure 4.13.

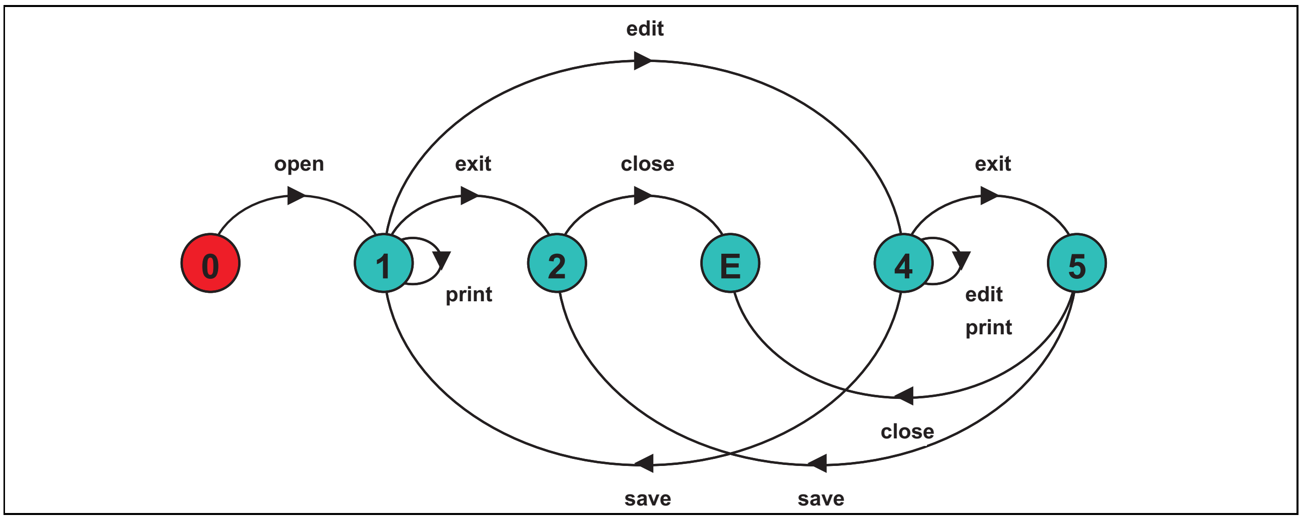


Figure 4.13: Final model of the editor system.

This model satisfies both properties. Therefore, we have achieved an abstraction that matches the specification and the property checking process is concluded.

4.5 Summary and Discussion

In this chapter, the formal basis of the proposed approach was described. The mappings from an implementation to an LKS model and from this model to an LTS model were discussed. We showed that, when mapping an LKS to an LTS model, all LTL properties over actions in the alphabet of the LKS model are preserved in the resulting LTS model.

The completeness and correctness of the models and their influence in the results of property checking were explained. We discussed in more detail the refinement process, formally demonstrating that it is property-preserving from the more abstract to the more refined model. This allows us to decrease the level of abstraction of a model to improve correctness and eliminate false negatives, and yet guarantee that previously proved properties still hold.

Our procedure for property checking was presented. Properties are specified as FSP property automata, LTL formulas or FLTL formulas. An explanation about the procedure was provided, discussing issues such as the selection of attributes to refine a model and the use of test cases to generate traces. We also discussed the use in combination of testing and model checking using our models as a possibility of increasing confidence on the correct behaviour of a system.

Finally, a practical example of property checking showed the use of our refinement process to eliminate false negatives. It demonstrated that there is indeed a refinement relation between a more abstract and a more refined model, where the latter preserves the properties checked to hold in the former.

Next chapter presents the tool support for our approach. It describes the LTS Extractor (LTSE), which implements the model extraction processes based on contexts and the LTSA tool, used to convert the FSP description generated by the LTSE tool into an LTS model, which can be analysed.

Chapter 5

Tool Support

This chapter presents the tool support provided to our approach by the LTS Extractor (LTSE). It automates most of the model extraction process, generating a Finite State Process (FSP) description from a set of inputs, including context information.

We show how the tool can be used and how the FSP description it generates can serve as an input to the LTS Analyser (LTSA), where its graphical representation can be generated and analysed. Moreover, we discuss how to use the generated model to check properties using the features of the LTSA tool.

5.1 The LTS Extractor

The LTSE tool has been developed to give support to the ideas presented here for model extraction. Tool support is always important as it makes the process easier and faster. Moreover, the user does not need to know the internal computations, but just benefit from the results.

This tool partially automates the model extraction process. As described in the diagram in Figure 3.5, in Chapter 3, the information gathering phase is not automatic. The tool implements the part of the process related to the processing of logs to collect context information, the storage of this information as a context table, the creation of an implicit LKS model and the subsequent generation of an FSP description.

The tool does not directly support the instrumentation of the code. However, as commented before, we used the TXL engine [CDMS02] to support this task in the development of the examples presented in this work, including the case studies discussed in Chapter 6.

5.1.1 Implementation

The LTSE tool is implemented entirely in Java. 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, a list of attribute names and a list of log file names. When building the model, the action names are used as the model alphabet and the attribute names define the set of attributes forming the system state. The log files should contain traces of the system, from which context information is extracted. At the end of its execution, the tool outputs an FSP description based on the inputs provided.

The class diagram in Figure 5.1 shows the main classes of the tool implementation and their relations. The *Interest Filter* component implements the creation of a filter to select the model alphabet, based on the list of action names given as input, whereas the *State Refiner* provides the creation of the system state according to the input list of attributes.

The *Log Splitter* is the component responsible for analysing the log files and identifying annotations related to different instances of a component of the system. This identification is based on the object ID assigned to the instance during the execution. One new log file is created for each instance identified and each annotation regarding that instance is copied from the original log file to the instance log file.

When each log file contains only events of a single instance, the *Context Annotator* executes the gathering of context information. This process is the implementation of Algorithm 3.1,

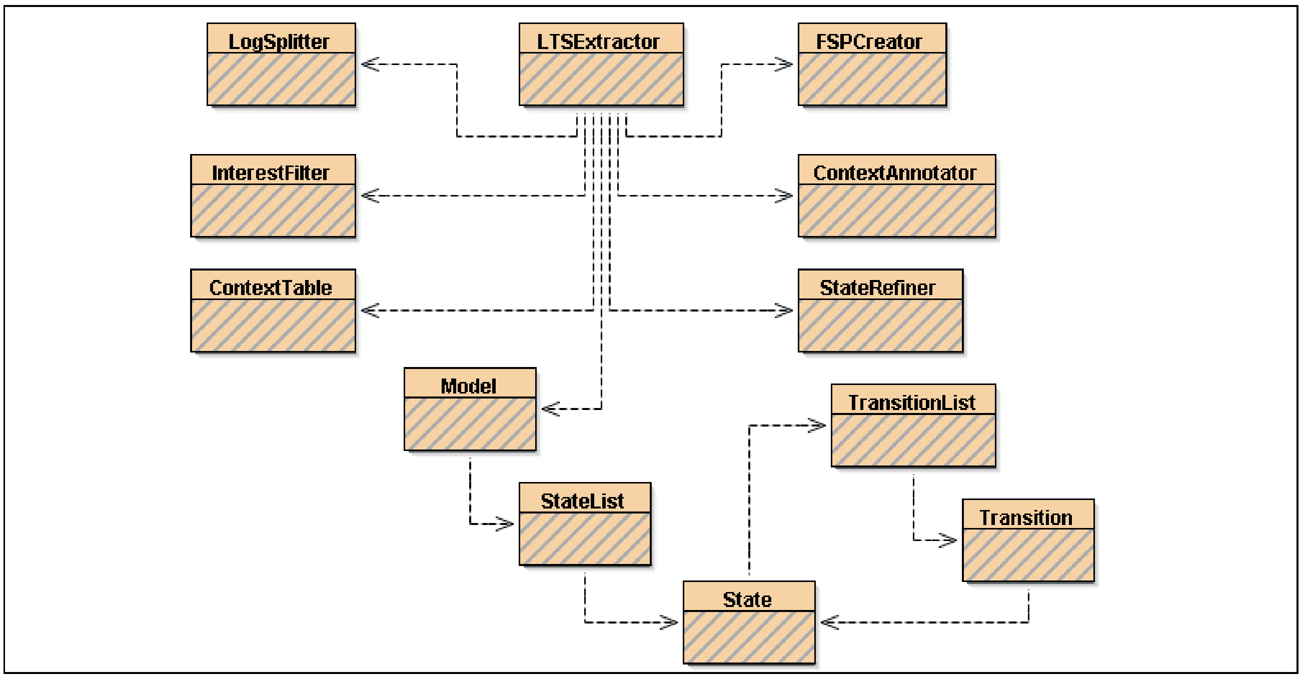


Figure 5.1: Class diagram of the LTSE tool main classes.

presented in Chapter 3. Each annotation of each log file is processed using the filter created by the *Interest Filter* component and the system state produced by the *State Refiner*. This processing produces the context table and a set of context files.

The *FSP Creator* uses these context files to build a structure to store an LKS model according to Algorithm 3.2, discussed in Chapter 3. The resulting LKS is mapped into an LTS model through an implementation of the formal mapping discussed in Section 4.1, in Chapter 4. This LTS model is output as an FSP description, so that it can be subsequently used in the LTSA tool for visualisation and property checking, as will be discussed in Section 5.3.

5.1.2 Requirements

The only real requirement to execute the LTSE tool is the presence of a Java Virtual Machine. An additional requirement could be the installation of the TXL engine[1] to automatically instrument the source code. However, as commented before, it is not essential. Any form of

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

instrumentation - even manual - may be used, provided that the appropriate annotations are introduced in the code following the patterns presented in Chapter 3.

Though those patterns apply to Java only, they could be easily redefined to annotate source codes written in other imperative languages, such as C. Because the annotations mark control flow statements, which are common to all imperative languages, an automatic instrumentation of other languages would just require the adaptation of such annotations to the grammar of the target language.

The trace generation could also be supported by a test case selection tool, which would help create the appropriate test cases according to a property of interest. As discussed in Chapter 4, the correct selection of test cases can originate a set of log files containing relevant traces with respect to the property one would like to check. Though it is also not a requirement, the use of an automatic test selector, especially one that could choose test cases based on a given property, could produce better results.

5.2 Extracting Models with the LTSE Tool

Generating models using the LTSE requires three basic steps. The first one is to provide the necessary parameters, which will guide the construction of the final model and influence its subsequent analysis. The second step is to determine the meaning of actions based on the types of processes involved. The last step consists of creating the FSP descriptions for each process using the defined parameters and types of processes.

We now present these steps and discuss how parameters are provided and in which format. The commands and results of generating models using the tool are also part of the discussion.

5.2.1 Providing Parameters

As discussed in Chapter 3, the model extraction process takes three parameters: an alphabet, a set of attributes (system state) and a set of traces. These parameters are defined according to the components of the system selected to compose the model[2].

During the processing carried out by the LTSE tool, these parameters are treated as local. This means that each parameter selection should consider only the component for which a model will be created. It is important to follow this approach to avoid mistakes such as instrumenting and collecting traces from more than one component at a time, resulting in mixed log files (i.e., logs containing information about two or more components).

Note that mixing annotations from two different components in the same log does not generate a model that is the composition of the behaviours of these two components. Remember that annotations regarding different instances are split into new log files and treated as different behaviours of a single component. Therefore, if annotations produced by different components are found in the same set of log files, then they will be considered two instances of the same component, which does not lead to the expected outcome.

Alphabet. The first parameter to be provided is the model alphabet. The LTSE tool receives the alphabet input in the form of a *filter file*, which has the extension *'.flt'*. This file contains the list of actions to compose the model alphabet. Its format is simply a list of action names, one per line, like this:

```
<actionName1>
<actionName2>
...
<actionNameN>
```

The LTSE tool reads the information from the file and creates a filter. Using this filter, action annotations regarding actions not on the list are ignored. Thus, if the file is empty, then all actions are ignored. If no such file is provided, the tool includes in the model alphabet all actions found in action annotations in the log files.

System State. The set of attributes is provided to the LTSE in a *refinement file*, which has the extension *'.ref'*. This file stores a list of attribute names, one per line, like this:

[2]Refer to Chapter 3 for a discussion on how to select parameters.

```
<attributeName1>

<attributeName2>

...

<attributeNameN>
```

The information contained in this file is used to create a state refiner, which reads the state information from each context annotation and obtains the values only of those attributes on the list. If the refinement file is empty, no attributes will be selected. Providing no refinement file produces the same result.

Set of Traces. As commented before, the set of traces produced during the execution of the system is stored in log files, whose format was presented in Chapter 3. The LTSE tool receives the log files directly, using the defined format, and processes each annotation, applying the filter and the state refiner, if provided.

Unlike the filter file and the refinement file, the tool can receive a list of more than one log file name as a parameter. Each log file on the list is processed as described in Algorithm 3.1, in Chapter 3. Therefore, the resulting model will be a combination of the behaviours recorded in each file.

5.2.2 Choosing the Interpretation of Actions

The LTSE tool allows the representation of actions in three modes. Each mode assigns a different interpretation to actions, causing the tool to use some action annotations related to methods and ignore others. This can be applied to a single model only and the meaning of actions in the model depends on the chosen mode.

Call Mode. The default mode is the *call mode* (c-mode), which corresponds to interpreting an action as a *method call*. In this case, when processing the traces in the log files, the tool uses the action annotations that refer to the beginning of method bodies for internal methods and to action annotations of the beginning of a method call for external methods. The actions corresponding to the end of a method body and the return of an external method call are, therefore, ignored in this mode.

Termination Mode. The *termination mode* (t-mode) corresponds to the opposite of the c-mode. When processing the logs, the tool uses the action annotations referring to the end of a method body and the return of an external method call. Thus, an action in the model represents the *termination of a method execution*, be it internal or external. The other action annotations related to methods are ignored.

Enter and Exit Mode. The *enter and exit mode* (e-mode) combines the c-mode and the t-mode. All action annotations related to methods are considered. Actions used in c-mode receive the suffix '*.enter*', whereas actions used in t-mode receive the suffix '*.exit*'. Therefore, in this mode, for each action m, where m represents a method body or a method call, there is an action *m.enter* to describe its beginning and an action *m.exit* to represent its end.

The use of these modes provides a different view of the behaviours described in the model. In general, the c-mode should be used to create models of active processes, whereas the t-mode should be used with passive processes. As commented in Chapter 3, this allows the synchronisation between actions of active and passive processes and provides an abstract way of representing the blocking mechanism usually implemented in passive processes (monitors).

Active processes may be represented using t-mode. Nevertheless, it is necessary to be aware that, because an action represents the completion of a method, nested method calls will cause the actions to appear in the reverse order of their actual occurrence (i.e., the call to the inner method would appear in the model before the call to the outer method). This may cause confusion during the analysis.

Similarly, the use of c-mode for a passive process model would mean that an action would appear in the model even though the corresponding method was called but not necessarily completely executed (e.g., if a method is called but an internal choice prohibits its execution). Therefore, when using the c-mode, if the system allows a method to be called then there will always be an action in the model to represent it.

The e-mode is an alternative to clarify the relation between inner and outer methods in nested method calls and blocking mechanisms. For example, let us suppose an internal method m_2 is called inside an internal method m_1. In c-mode, the behaviour to be introduced in the model would be $\langle m_1\ m_2 \rangle$, showing the order of calls. In t-mode, the behaviour would be $\langle m_2\ m_1 \rangle$, thus presenting the order of termination of methods. In e-mode, we can distinguish a method call from a method termination. For this example, the trace produced would be $\langle m_1.\mathit{enter}\ m_2.\mathit{enter}\ m_2.\mathit{exit}\ m_1.\mathit{exit} \rangle$. However, this option obviously enlarges the model, as there are two actions (enter and exit) for each method called.

It should also be noted that, if e-mode is chosen, the model of all processes involved in the system must adhere to this mode as well so that synchronisation is possible when they are composed. The same does not apply to the other modes, where models created using c-mode can be composed with models created using t-mode.

There are no restrictions as to which mode should be used. The option of creating different combinations of representations and types of processes gives the possibility of trying them out to identify which combination suits best the problem being analysed. This choice can be made through a parameter of the LTSE tool. Because all annotations are stored in the log files used to build a model, the change of views can be carried out without having to re-generate the traces, just changing the value of the parameter.

5.2.3 Generating an FSP Description

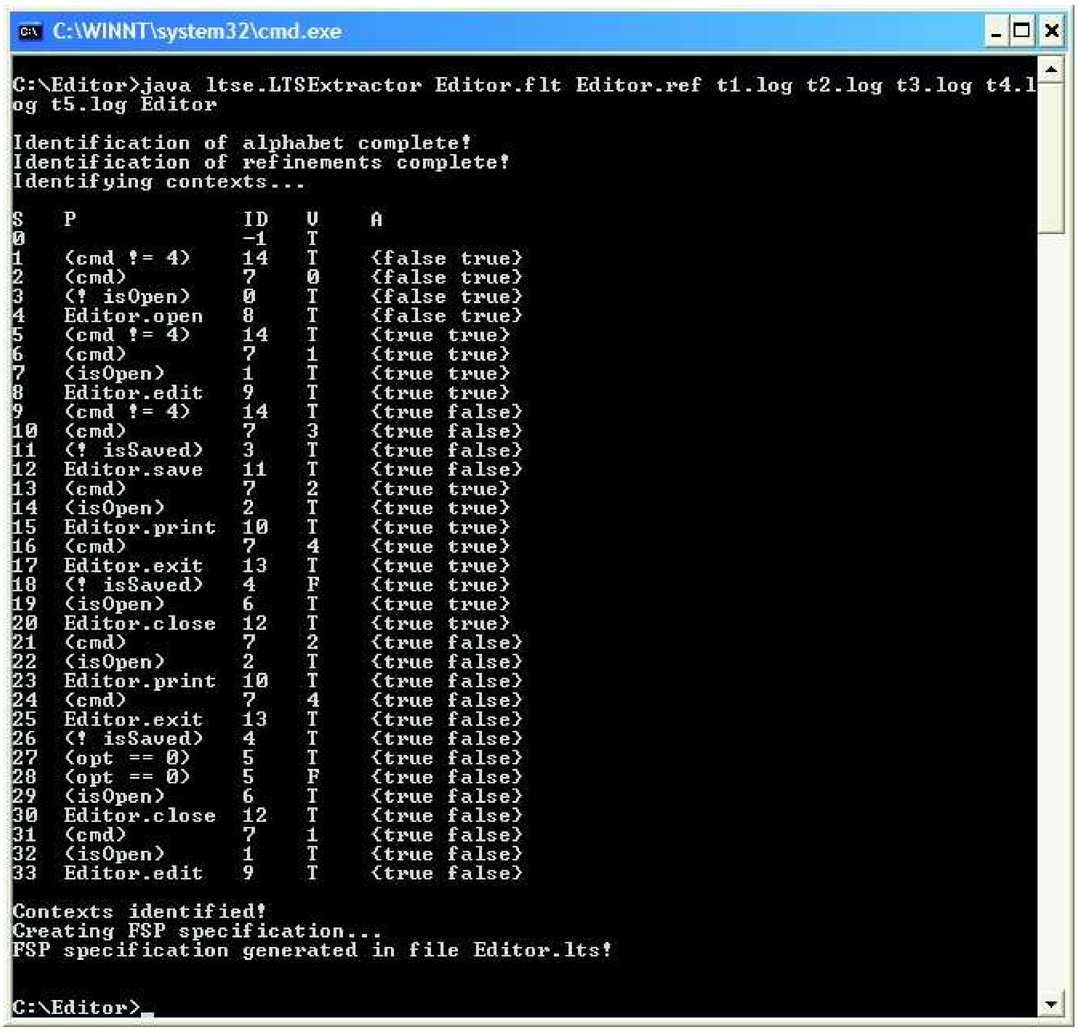
Using the parameters and one of the available modes, the basic command to generate a model would be:

```
java ltse.LTSExtractor [filter] [refiner] [-c|-t|-e] <logs> <name>
```

where `filter` is a filter file, `refiner` is a refinement file, `logs` is a list of log file names of the form `log1.log log2.log ... logN.log` and `name` is the name of the resulting model.

The interpretation of the meaning of actions is selected using the options `-c` for c-mode, `-t` for t-mode and `-e` for e-mode.

As an example, let us consider the creation of the model shown in Figure 4.13, in Chapter 4. For that model, the filter file `Editor.flt` contained the action names `open`, `edit`, `print`, `save`, `exit` and `close` and the refinement file `Editor.ref` contained the attribute names `isOpen` and `isSaved`. These files were used as parameters along with a list of five log files, one for each trace presented in Section 4.4. The chosen mode was the default (c-mode). The command line used to generate the model and the outputs for this example are shown in Figure 5.2. This created a file `Editor.lts`, which contained the FSP description obtained using the parameters provided and adapted to the mode selected.



```
C:\Editor>java ltse.LTSExtractor Editor.flt Editor.ref t1.log t2.log t3.log t4.l
og t5.log Editor

Identification of alphabet complete!
Identification of refinements complete!
Identifying contexts...

S   P              ID   V    A
0                  -1   T
1   (cmd != 4)     14   T    {false true}
2   (cmd)          7    0    {false true}
3   (! isOpen)     0    T    {false true}
4   Editor.open    8    T    {false true}
5   (cmd != 4)     14   T    {true true}
6   (cmd)          7    1    {true true}
7   (isOpen)       1    T    {true true}
8   Editor.edit    9    T    {true true}
9   (cmd != 4)     14   T    {true false}
10  (cmd)          7    3    {true false}
11  (! isSaved)    3    T    {true false}
12  Editor.save    11   T    {true false}
13  (cmd)          7    2    {true true}
14  (isOpen)       2    T    {true true}
15  Editor.print   10   T    {true true}
16  (cmd)          7    4    {true true}
17  Editor.exit    13   T    {true true}
18  (! isSaved)    4    F    {true true}
19  (isOpen)       6    T    {true true}
20  Editor.close   12   T    {true true}
21  (cmd)          7    2    {true false}
22  (isOpen)       2    T    {true false}
23  Editor.print   10   T    {true false}
24  (cmd)          7    4    {true false}
25  Editor.exit    13   T    {true false}
26  (! isSaved)    4    T    {true false}
27  (opt == 0)     5    T    {true false}
28  (opt == 0)     5    F    {true false}
29  (isOpen)       6    T    {true false}
30  Editor.close   12   T    {true false}
31  (cmd)          7    1    {true false}
32  (isOpen)       1    T    {true false}
33  Editor.edit    9    T    {true false}

Contexts identified!
Creating FSP specification...
FSP specification generated in file Editor.lts!

C:\Editor>_
```

Figure 5.2: Screenshot of an execution of the LTSE tool.

The outputs essentially present messages regarding the model extraction process and the context table created during the identification of contexts. Column A contains the values of the attributes `isOpen` and `isSaved`, respectively. Note that the control predicates associated with the contexts are presented as well (column P of the table).

In some situations, looking at the context table and at the resulting FSP description provides a good understanding of why a certain behaviour was included in the model or even how an invalid behaviour was generated. For example, it is possible to see that method `edit` can be executed in two situations: when a document is open and has not yet been modified (column S of the table, state 8) or when a previous call to the same method caused the file to be modified (state 33). The same occurs with method `print` (states 15 and 23). This shows that they are allowed to execute irrespective of the value of attribute `isSaved`. However, this is not true for method `save`, which can occur only if `isSaved` is false (state 12).

5.3 Model Visualisation and Analysis

The FSP description produced by the LTSE tool can be imported into the LTSA tool. There, it is possible to visualise a graphical representation of the LTS model described using the process algebra. It also allows the checking of properties against the models. Error traces can be replayed in the models so that it is possible to identify how violations were produced.

This resulting model can also be used for purposes other than property checking. One of such purposes is simulation of behaviours. This is done by presenting the user with the alphabet of the model and allowing them to choose, in each state, one action to be taken among the actions enabled in that state. Therefore, the user can 'execute' behaviours in the model and analyse whether they are valid or not. It helps understand the system behaviour, especially in the presence of concurrency.

5.3.1 Visualising the LTS Model

The model described in FSP can be modified using some operations implemented in the LTSA tool. These operations are applied before the graphical representation is generated.

Hiding Operator. The user can define a set of actions $\{a_1, ..., a_n\}$ of the model that are not relevant - or not visible to other processes - to be hidden using the *hiding operator*. Applying the hiding operation $/\{a_1, ..., a_n\}$ to a process $Proc$, results in every action a_i in the set, for $1 \geq i \geq n$, being removed from the alphabet of $Proc$.

When generating the visual model, each occurrence of one of these actions is replaced by a `tau` action, which represents a silent action. We use this operation to remove the `null` actions used in our models and make them silent. If an action is silent, it means that it is a local action and does not interfere in the behaviour analysis. During simulation, these actions represent transitions that are always enabled (i.e., no actions required).

As an example of the use of the hiding operator, consider the FSP description in Figure 5.3 of a simple traffic lights system, which alternates between green lights and red lights, using yellow lights as a transition colour. Action `change` represents the event that triggers the change of colour, which we choose to hide. This FSP generates the graphical LTS model shown in Figure 5.4. Note that occurrences of `change` have been replaced by `tau` actions.

```
Lights = (change->yellow->green->change->yellow->red->Lights)
         \{change}.
```

Figure 5.3: FSP of the lights example.

Minimisation. It is also possible to minimise a model so that a more compact version is obtained. This more compact version corresponds to a model produced applying the observational equivalence defined by Milner [Mil99]. Essentially, this operation reduces the model by eliminating `tau` actions whenever possible, such that observational behavioural equivalence between the reduced model and the original model is preserved.

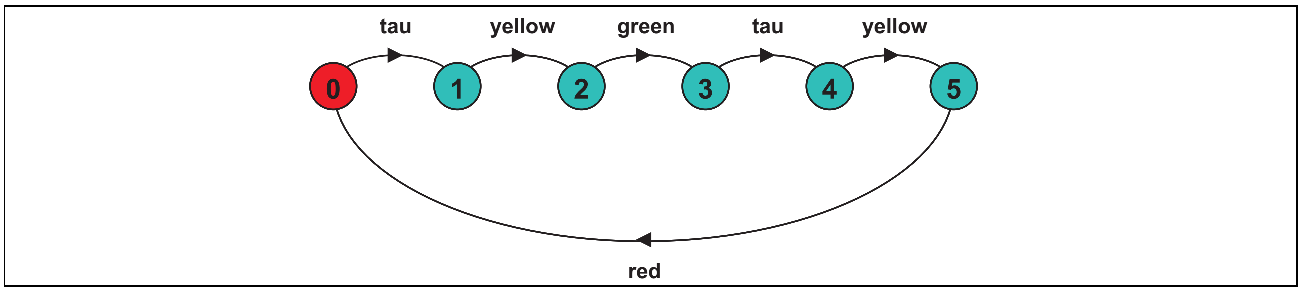


Figure 5.4: LTS model of the lights example.

If we apply the minimisation operation to the model shown in Figure 5.4, we obtain the model presented in Figure 5.5. It is observationally equivalent to the other model but all transitions labelled with `tau` have been removed.

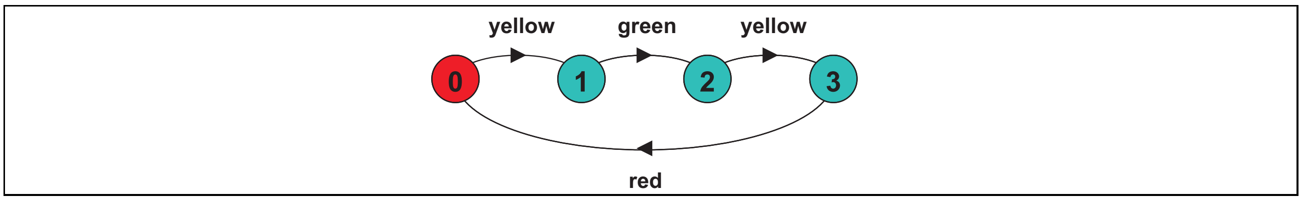


Figure 5.5: Minimised LTS model of the lights example.

Deterministic. Another possible operation is the elimination of non-determinism. Consider now a version of the traffic lights system where a random choice is used to decide which colour will be selected next. The FSP of this example is shown in Figure 5.6. Note that there is a non-deterministic choice involving actions `change` and `yellow`. We decide not to hide any action this time.

```
RandomLights = (change->yellow->green->RandomLights
               |change->yellow->red->RandomLights)
               \{change,yellow}.
```

Figure 5.6: FSP of the random choice traffic lights system.

The model produced using this FSP is shown in Figure 5.7. If we try to minimise this model, the result is the exact same model, as the non-deterministic choice cannot be removed (not even if actions `change` and `yellow` are made silent).

The LTSA tool provides an operation to convert non-deterministic models into deterministic ones during the generation of their graphical representation. This operation is applied when the keyword `deterministic` is inserted in front of the FSP process definition. The deterministic model is a result of a Non-Deterministic Finite Automaton to Deterministic Finite Automaton (NFA-DFA) transformation [HU79]. The deterministic version of the previous non-deterministic model of the traffic lights system is presented in Figure 5.8.

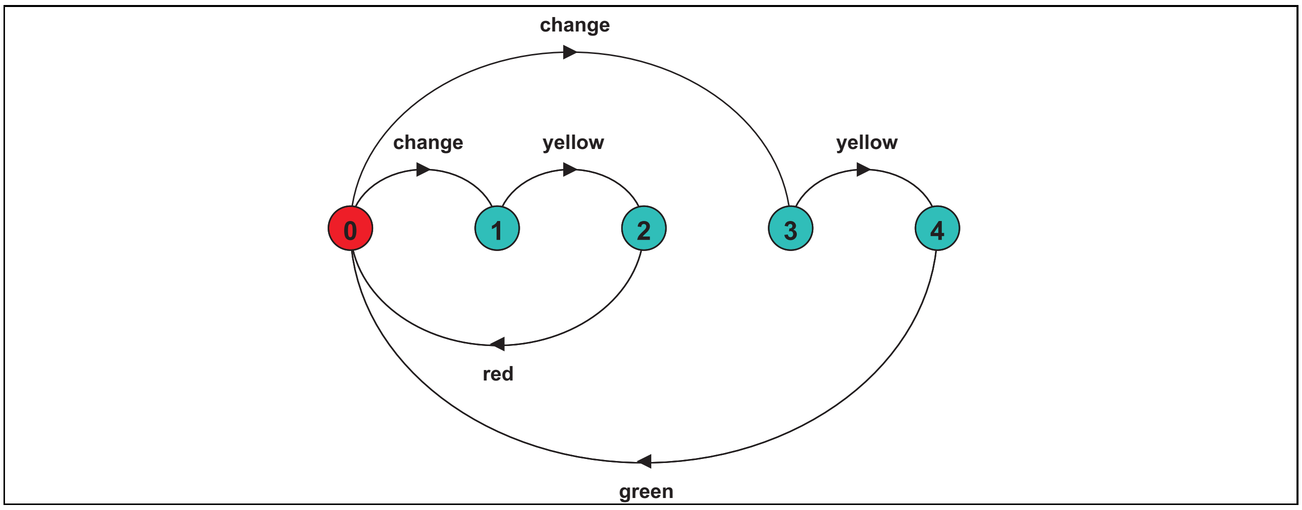


Figure 5.7: LTS model of the random choice traffic lights system.

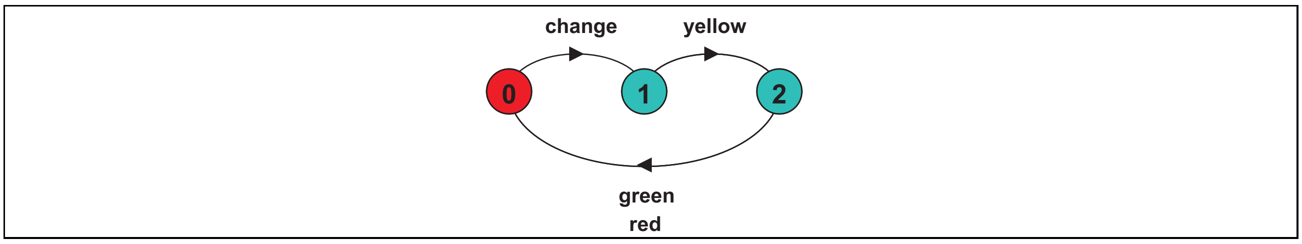


Figure 5.8: Deterministic LTS model of the random choice traffic lights system.

The deterministic model shows the same possibility of turning the lights to either green or red without the non-deterministic choice. However, the random characteristic of the model still exists, since after executing the sequence ⟨`change yellow`⟩ it randomly chooses between `green` and `red` to execute next.

5.3.2 Property Specification in the LTSA tool

The types of properties that can be checked using models generated using our approach were discussed in Chapter 4. These properties can be specified using an FSP property automaton, an LTL formula or an FLTL formula.

In the LTSA tool, an FSP property automaton is defined as discussed in Chapter 4, prefixing a process definition with the keyword `property`. When generating a model this process definition is converted into a deterministic finite automaton containing an $ERROR$ state, denoted by state -1 in its graphical representation.

LTL properties are specified using names of actions as propositions and the logical and temporal operators shown in Table 5.1. These operators correspond to their equivalent symbols presented in Table 4.1.

Logical Operators	**Temporal Operators**
`!` (logical negation)	`[]` (always)
`&&` (logical AND)	`<>` (eventually)
`||` (logical OR)	`X` (next)
`->` (implication)	`U` (until)
`<->` (equivalence)	`W` (weak until)

Table 5.1: LTL logical and temporal operators in the LTSA tool.

Using this operators, an LTL formula is defined as an assertion, as in the example below

```
assert SaveIfEdited = [](open->(!save W edit))
```

where `SaveIfEdited` is the name associated with the property. Multiple properties may be defined to be checked against a model.

A fluent is defined in the LTSA tool as in the example below

```
fluent Edited = <edit,save> initially 0
```

where `Edited` is the name of the fluent, `edit` is the initiating action, `save` is the terminating action and `0` corresponds to the value false (`1` for true).

Once defined, a fluent is used in a property defined as described for LTL formulas. The fluent name is used in the very same way as action names:

```
  assert SaveIfEdited = [](open->(!save W Edited))
```

Any of the discussed formats can be used to specify properties and check them against a generated model, according to the preference of the user. It is just important to bear in mind the semantics of each formalism so that results can be correctly interpreted. Next, it will be discussed how to analyse the results of a property checking process.

5.3.3 Checking Properties

When checking properties against a model, the result can be that either no violation is found or an error trace is generated. In the first case, the LTSA tool produces an output such as the one presented in Figure 5.9 to inform the property satisfaction by the model. This example of report was generated using the formula `Allowed = [](!Open -> !(edit || print || save || exit))`. Part of the report is omitted because it is not important for this discussion.

```
...
Composition: DEFAULT = Editor || Allowed
State Space:
 6 * 2 = 2 ** 4
Analysing...
Depth 4 -- States: 6 Transitions: 15 Memory used: 1740K
No deadlocks/errors
Analysed in: 0ms
```

Figure 5.9: Example of report of no violation found.

If a violation is found, the report includes the error trace, as shown in Figure 5.10, where formula `SaveEdited = [](Saved -> [](!X save W Edited))` is checked (fluents `Saved` and `Edited` are the ones defined in Chapter 4). If fluents are used in the formula, the error trace indicates the sequence of actions and also the fluents that were true at each point of the error trace. Otherwise, it only presents the sequence of actions.

```
...
Composition: DEFAULT = Editor || SaveEdited
State Space:
 5 * 3 = 2 ** 5
Analysing...
Depth 2 -- States: 2 Transitions: 3 Memory used: 3468K
Trace to property violation in SaveEdited:
    open    Saved
    save    Saved
Analysed in: 0ms
```

Figure 5.10: Example of report of violation.

In this example, the error trace violates the property because it demonstrates that `save` can happen when `Saved` is true, contrary to what is specified in the property. The error traces produced by the tool always show the shortest sequence of actions leading to the error state.

5.4 Summary and Discussion

This chapter showed that the combination of the LTSE tool and the LTSA tool supports a complete model checking process. The LTSE tool supports the modelling step, whereas the LTSA provides an environment for the specification and verification steps.

The LTSE creates a model that can be adapted to specific needs. This adaptation is achieved through the choice of the parameters provided to the tool. Selecting the alphabet and system state and producing the appropriate set of traces allows the extraction of a model tailored for a particular system and level of abstraction.

The model constructed by the LTSE tool serves as input to the LTSA tool, where its graphical representation can be generated and visualised. Properties can be specified using FSP property automata, LTL formulas or FLTL formulas. This way, the property can be described in the more convenient formalism according to the user's expertise and needs.

Finally, properties can be checked against the extracted model. The LTSA tool can identify property violations and produce error traces to describe the behaviour that violates the property. Error traces can guide the identification of invalid behaviours and, subsequently, lead to modifications in the program code or the refinement of models.

In the next chapter, we combine these tools to develop two case studies. We show how these tools can help the understanding and analysis of systems which involve the concurrent execution of multiple threads and their interactions.

Chapter 6

Case Studies

During the development of this work, the approach herein presented was evaluated using several case studies, some of which are discussed in Appendix B. This chapter presents the results of two selected case studies with the purpose of demonstrating how the approach can be applied and its strengths and limitations.

The first case study was based on the Single-Lane Bridge problem described in [MK06]. Though this system was quite simple, it helped us apply and evaluate our representation of active and passive processes. Moreover, manually created models are presented in [MK06], allowing us to compare them with our automatically generated models. The case study also provided a complete example of our procedure for model extraction and property checking.

The second case study shows the results of our approach applied to a more complex system. The system was an implementation of Garcia-Molina's Bully Algorithm [GM82]. The complexity of the system and some details of the particular implementation used in this case study provided an interesting challenge for our model extraction process. This case study revealed some unknown limitations of our approach and yet proved its uselfulness.

6.1 Single-Lane Bridge Problem

The Single-Lane Bridge problem involves a bridge over a river which has only one lane. Hence, only cars moving in the same direction are allowed to cross the bridge at the same time. This scenario can be seen in Figure 6.1, which shows a screenshot of an applet implementing the problem and discussed in [MK06].

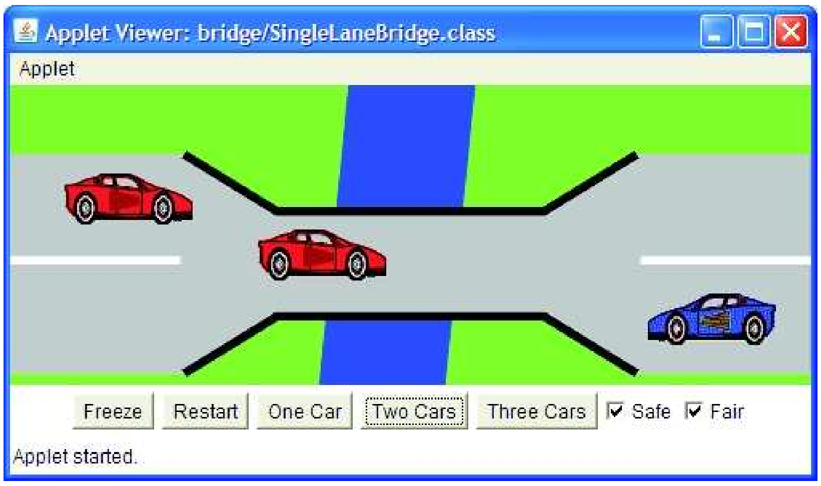


Figure 6.1: Visual representation of the Single-Lane Bridge problem.

In the implementation found in [MK06], cars crossing the bridge from left to right are identified as red cars and cars moving in the opposite direction are identified as blue cars. Each car is implemented as a thread that tries to obtain access to the bridge. The bridge is a passive entity that provides methods for allowing cars to enter and exit it, as shown in Figure 6.2. As one can see, in this implementation the method bodies are empty.

6.1.1 Model Generation

We defined cars as active processes and the bridge as a passive process. Models were created for each class involved in the system (`RedCar`, `BlueCar` and `Bridge`). The alphabet of the models included the names of the methods provided by the bridge to allow cars to enter and exit, namely `redEnter` and `redExit` for red cars and `blueEnter` and `blueExit` for blue cars. The initial set of attributes was empty.

```
    class Bridge {
        synchronized void redEnter () throws InterruptedException {}
        synchronized void redExit () {}
        synchronized void blueEnter () throws InterruptedException {}
        synchronized void blueExit () {}
    }
```

Figure 6.2: Source code of the bridge.

The execution of the applet provided the necessary traces to generate the models. The interface allowed the execution with one, two or three cars in either direction. We executed the system once using each one of these settings.

The behaviour of the cars does not depend on the numbers of cars crossing the bridge and, therefore, a single model was built for either type of car. The deterministic models of a red car and a blue car are shown, respectively, in Figure 6.3(a) and Figure 6.3(b).

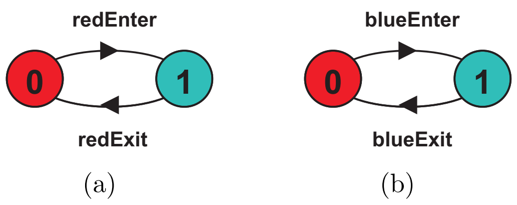


Figure 6.3: LTS models of (a) the red car and (b) the blue car.

As for the bridge model, we used the following definitions, adapted from the ones in [MK06]:

```
const N = 3
range T = 0..N
range ID = 1..N

NOPASSRED1   = C[1],
C[i:ID]   = ([i].redEnter -> C[i%N+1]).
NOPASSRED2   = C[1],
C[i:ID]   = ([i].redExit -> C[i%N+1]).
```

```
||RED_CONVOY = ([ID]:RedCar || NOPASSRED1 || NOPASSRED2).


NOPASSBLUE1    = C[1],
C[i:ID]    = ([i].blueEnter -> C[i%N+1]).
NOPASSBLUE2    = C[1],
C[i:ID]    = ([i].blueExit -> C[i%N+1]).
||BLUE_CONVOY = ([ID]:BlueCar || NOPASSBLUE1 || NOPASSBLUE2).


||CARS = (RED_CONVOY || BLUE_CONVOY).
```

Processes `NOPASSRED1` and `NOPASSRED2` define that one red car should not pass another while on the bridge. Processes `NOPASSBLUE1` and `NOPASSBLUE2` do the same for blue cars. Note that an identification was used as a prefix for the names of actions to determine which car executed which action. Composite processes `RED_CONVOY` and `BLUE_CONVOY` composed the model of the red cars (`RedCar`) and the model of blue cars (`BlueCar`), respectively, with the processes prohibiting cars to overtake each other. Composite process `CARS` put both red cars and blue cars together.

Initially, we generated a model of the bridge for the situation where only one car of each type was trying to cross it (therefore, `N=1` in our definitions). Figure 6.4 shows this model.

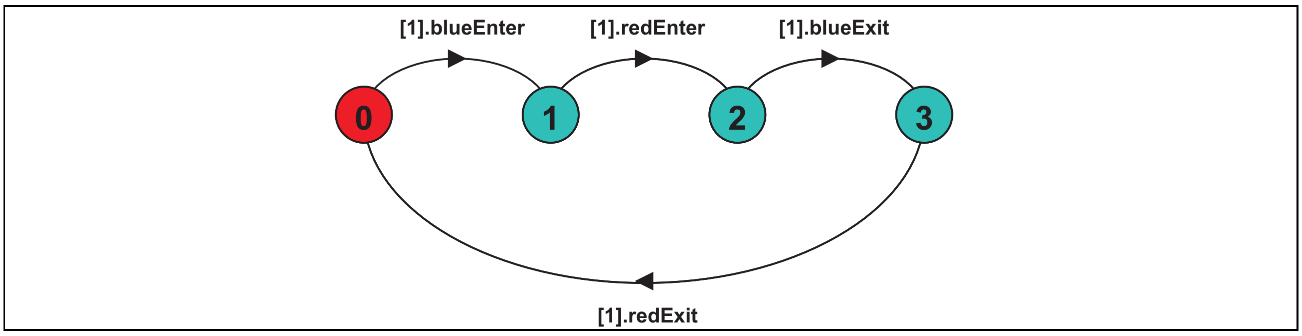


Figure 6.4: LTS model of the bridge for one car of each type.

Note that the original names of the methods have been already relabelled so that they synchronise with the action names used in the composite model CARS. This way, the model of the bridge also contains the identification of the car originating the action.

6.1.2 Property Checking

The important property that the single-lane bridge should preserve is that cars of different types should not be on the bridge at the same time. This property specification as an FSP automaton is presented in [MK06]. We used the same property, just changing the names of the actions so that they matched those of our previous models (e.g., [ID].redEnter instead of red[ID].enter). The adapted property specification is shown in Figure 6.5.

```
property ONEWAY = ([ID].redEnter -> RED[1]
                  |[ID].blueEnter -> BLUE[1]),
RED[i:ID] = ([ID].redEnter -> RED[i+1]
            |when (i==1) [ID].redExit -> ONEWAY
            |when (i>1) [ID].redExit -> RED[i-1]),
BLUE[i:ID] = ([ID].blueEnter -> BLUE[i+1]
             |when (i==1) [ID].blueExit -> ONEWAY
             |when (i>1) [ID].blueExit -> BLUE[i-1]).
```

Figure 6.5: FSP specification of property OneWay.

This property defines that either a red or a blue car is allowed to access the bridge and, once one of them has done it, only cars of the same type can enter the bridge until all cars of that type have left it. Hence, if cars of different types are simultaneously on the bridge at any time, the property is violated.

Composing this property with processes CARS and Bridge - i.e., the process corresponding to the model shown in Figure 6.4 -, it is possible to check that the property is not preserved. This can be seen by using the LTSA tool to create the composition and executing a safety check. The error trace produced is presented in Figure 6.6.

```
Trace to property violation in ONEWAY:
    1.blueEnter
    1.redEnter
```

Figure 6.6: Error trace for property `OneWay`.

This error trace describes a situation where a blue car enters the bridge and then a red car can do the same, thus violating the property. This result confirms the violation described in [MK06] for the same composition but by using a manually created model.

As commented in [MK06], this violation occurs because the methods of the bridge implementation do not include any control over the cars entering and exiting the bridge. As a solution, the authors present an implementation of a safe bridge (Figure 6.7), which uses the wait-notify mechanism and counters to control access to the bridge.

```
class SafeBridge extends Bridge {

    private int nred  = 0;
    private int nblue = 0;

    synchronized void redEnter() throws InterruptedException {
        while (nblue>0) wait();
        ++nred;
    }

    synchronized void redExit(){
        --nred;
        if (nred==0)
            notifyAll();
    }

    synchronized void blueEnter() throws InterruptedException {
        while (nred>0) wait();
        ++nblue;
    }

    synchronized void blueExit() {
        --nblue;
        if (nblue==0)
            notifyAll();
    }
}
```

Figure 6.7: Source code of the safe bridge.

Generating a new model and executing the safety check again results in no violation found. The model of the safe bridge is presented in Figure 6.8. For this model, the system state was composed of the attributes `nred` and `nblue`, which count the number of red cars on the bridge and the number of blue cars on the bridge, respectively. These attributes are necessary to distinguish the context where a car can enter the bridge because no car of the other type is on it at that moment from the context where the car has to wait for cars of the other type to leave the bridge (process is blocked). Note that once again the attributes used to refine the model are part of control predicates in the code.

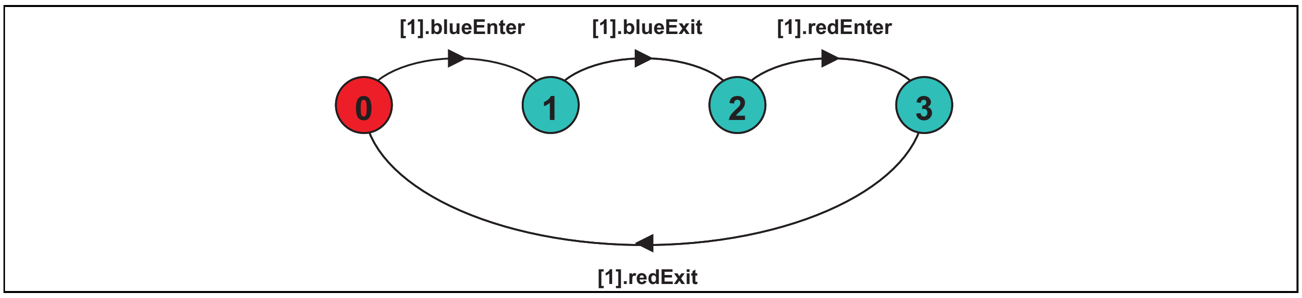


Figure 6.8: LTS model of the safe bridge for one car of each type.

The model shows that the blue car enters first and the red car is only allowed to enter after the blue car has exited the bridge. The blue car only returns to the bridge when the red one has left it. The order of the cars was determined by the collected traces and is irrelevant to the checking of the property, which, in this case, is not violated.

To make sure the system does preserve the property irrespective of the number of cars, we generated the models for two and three cars of either type. In the case of two cars, no violation was found. However, the model generated for three cars of either type, shown in Figure 6.9, described a problem: red cars could never enter the bridge. The property is violated not because cars of different types are allowed on the bridge at the same time, but because the actions involving red cars are not part of the alphabet of the model.

This problem is solved in [MK06] by introducing turns for cars of either type entering the bridge. The code of this fair bridge - obtained from [MK06] - is presented in Figure 6.10 and the model extracted for this bridge is shown in Figure 6.11.

The system state used for this model was the same used when creating the safe bridge model plus the new attribute `blueturn`. This additional attribute allows the identification of the contexts that described the alternated access to the bridge by either type of car.

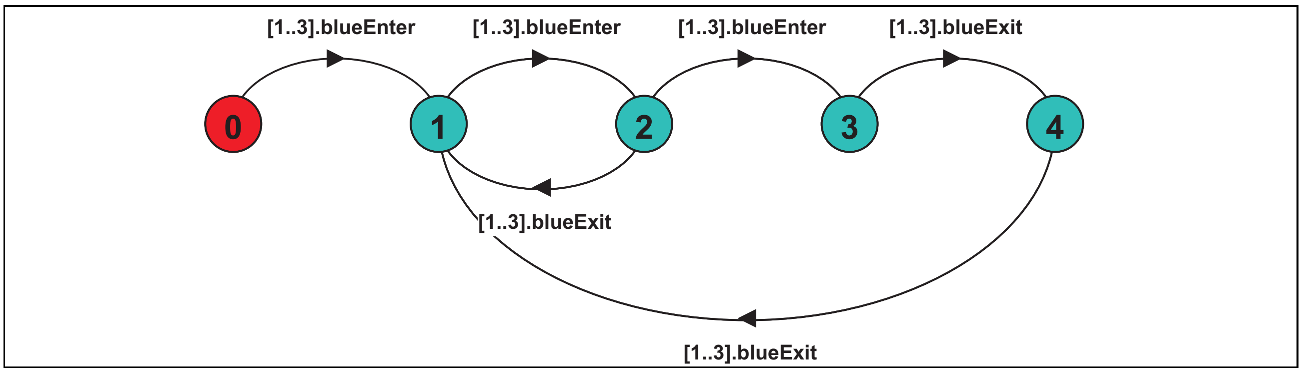


Figure 6.9: LTS model of the safe bridge for three cars of each type.

```
class FairBridge extends Bridge {

    private int nred  = 0;
    private int nblue = 0;
    private int waitblue = 0;
    private int waitred = 0;
    private boolean blueturn = true;

    synchronized void redEnter() throws InterruptedException {
        ++waitred;
        while (nblue>0 || (waitblue>0 && blueturn)) wait();
        --waitred;
        ++nred;
    }

    synchronized void redExit(){
        --nred;
        blueturn = true;
        if (nred==0)
            notifyAll();
    }

    synchronized void blueEnter()  throws InterruptedException {
        ++waitblue;
        while (nred>0 || (waitred>0 && !blueturn)) wait();
        --waitblue;
        ++nblue;
    }

    synchronized void blueExit() {
        --nblue;
        blueturn = false;
        if (nblue==0)
            notifyAll();
    }
}
```

Figure 6.10: Source code of the fair bridge.

The checking of this model did not reveal any violation of the property. This confirms the results of the authors in [MK06].

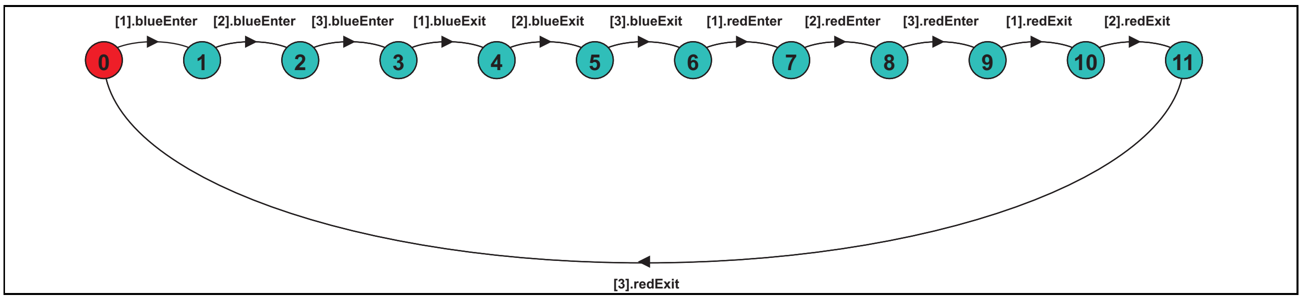


Figure 6.11: LTS model of the fair bridge for three cars of each type.

6.1.3 Evaluation

The Single-Lane Bridge case study provided us with an example where models had been manually created and the results of checking those models were available. This allowed us to compare the models we generated with those models. Except for the model of the fair bridge with three cars, all the models reproduced the behaviour modelled by the authors in [MK06], thus generating the same results during the analysis.

The difference in the model of the fair bridge for three cars was due to the fact that the manually created model defined the expected behaviour of the code, whereas the extracted models showed its observed behaviour. Therefore, the results were different because the expected behaviour included the possibility of red cars accessing the bridge at some point, even though a progress check could reveal that this actually might never occur. The observed behaviour, on the other hand, showed this situation directly in the model.

Though the complexity of this application was low, the concurrency aspect was useful to test our approach when dealing with multiple threads accessing a shared resource. It proved the applicability of our approach for extracting models of concurrent systems and its usefulness for detecting invalid behaviours.

6.2 Leader Election Algorithm

A *leader election algorithm* [CDK05] is a distributed algorithm for choosing one of the processes composing a distributed system to be the leader (or coordinator). The leader works as a server to all processes - including itself - that share a particular resource. It receives requests from a process, accesses the resource and possibly returns a confirmation to the process that made the request. However, unlike the client-server architecture, the leader may change over time if a current leader fails. In this case, an election occurs to find a new leader.

The election procedure involves the exchange of messages between processes to decide which one of them will be the leader. The choice is normally based on an identifier assigned to each process. Identifiers may be values of any type but must be unique and totally ordered [CDK05]. The leader is usually the process with the highest identifier. Each member has a variable `elected` that contains the identifier of the current leader. All processes must agree on the elected process [CDK05].

The Bully Algorithm [GM82] is a leader election algorithm where a new election starts whenever a process is detected to have failed or recovered. If a process that had failed recovers and its identifier is higher than any of those of the processes still alive, then it becomes the leader. This is the reason for the name of the algorithm: the process with the highest identifier will always take over as leader, even if a current leader exists and has not failed.

The algorithm makes three main assumptions: 1) message delivery is reliable, so that no message is lost; 2) the system is synchronous and, therefore, timeouts are used to detect that a process has crashed; and 3) each process knows all other processes and their respective identifiers, which allows all processes to take part in the election procedure and rejoin the system after recovering from a failure.

6.2.1 Implementation

For this case study, we used an implementation of the Bully Algorithm available on the Internet[1] and presented in Appendix C. This code was developed as part of the Distributed Systems course at Queen's University, Canada, to teach the algorithm to students.

Because it was aimed to teach students, the implementation includes not only the components of the algorithm but also components to support the simulation of failures and recoveries. These components allow the user to send messages to a process telling it to simulate a failure, which causes the process to shutdown by stopping all its running threads. The user can also decide to simulate a recovery of a failed process, thus sending it a message to restart all its threads. Therefore, even though the threads execute and communicate in an actual distributed system, the simulation components help analyse the effects of failures and recoveries on the behaviour of the processes of the system.

The implementation includes two types of components: an election console and election members. The *election console* controls the start and ending of the execution and provides a textual interface where the user can choose processes to simulate a failure or recovery. An *election member* is a process involved in the election, including threads to check whether other members are alive, to receive messages from other members and to participate in the election process. There are also threads that simulate an application where the leader has access to a printer. The leader receives printout requests from other members and sends back confirmations.

Subsystem Analysed

In order to reduce the complexity of the model to be generated and concentrate on the election procedure, we chose to analyse only the threads involved in the process of choosing a leader. Therefore, we focused on a subsystem where each election member runs only the subcomponents necessary to execute the election process.

[1] http://www.cs.queensu.ca/~huang/cisc833/BullyElection.pdf.

This subsystem is initialised by starting the console and then starting each member, one by one. Each member registers with the console, so that, at the end of the registration phase, the console has the list of all participants and can send it to every member.

Every time a new member registers, it receives an identifier that corresponds to the current value of a counter controlled by the console. Hence, the first member to register is identified as 1, the second as 2, and so on. In this implementation, the value of the identifier inversely corresponds to the priority of a member to become leader during an election process.

We worked with a system consisting of three members. In this case, the third member to register with the console - thus, being assigned priority 3 - can only be the leader if the other two members fail. If all members are alive, the leader should always be the one with priority 1. The member with priority 2 can be the leader as long as the member with priority 1 is down.

The implementation defines two means of communication between threads. Local communication (i.e., communication between threads of the same member) is via method invocation, whereas remote communication is via UDP [Pos80] sockets. The component `MessageManager` (`MM`) is responsible for internal communication of a member and communication between it and the other members. The `MM` receives messages sent by other members via UDP sockets, identifies the component the message is addressed to and forwards it to its destination using method invocation.

When communication is local, the `MM` invokes a method `setMsg` of the receiver. This method causes the message to be stored in a local variable, which works as a single-slot buffer. The component execution blocks in a loop (where it regularly checks its buffer) until a message has been received. In remote communication, the sender immediately resumes its execution after sending a message, while the receiver blocks until a message arrives.

Protocols

In the chosen subsystem, there are three protocols. The first protocol involves the communication between the console and the election members. The execution starts with the console sending a *start* command to every member. Each member, upon reception of this command, initialises all its internal components. If the console then sends a message to simulate a *failure*, the member executes the reversed sequence of actions, sending commands to every component to stop. At this point, a *recovery* message causes the member to restart its components, whereas a command to *close* terminates the execution.

The second protocol corresponds to the election process. This protocol follows the original definition of the Bully Algorithm and is executed in four phases:

1. Each member sends a check to every member that has higher priority than it. If it receives at least one response, it aborts the election process and waits to be notified of the new leader. On the other hand, if no response is received within a certain period of time (timeout), it assumes that all other candidates have failed and, therefore, it is the *leader candidate*, i.e., the member with the highest priority still alive;

2. The leader candidate then sends an `EnterElection` message to all other running members telling them that a new leader election is in progress. All other members stop their execution and wait for the identification of the new leader to be received. They respond to confirm they are alive and aware of the new election process;

3. After that, the leader candidate sends a `SetCoord` message to all other members. This message contains the candidate's identifier. The other members receive this message, set their `elected` variable (`coordinator` in this particular implementation) to the identifier received and send a confirmation;

4. Finally, the leader candidate sends the new state of the system to every member. The state contains the status of each member, which can be “coord”, if the member is the leader, “normal”, if the member is not the leader but is alive, or “down”, if the member has failed. Members update their internal view of the state of the system using the state received and send a confirmation. From this point on, the leader candidate is officially the new leader.

This protocol can be seen in Figure 6.12. At each point of the protocol, if the leader candidate does not receive a response from at least one of the members supposedly still alive, a new election begins. A new election also occurs at any time a failed member recovers. The new election process is initiated by the recovered member.

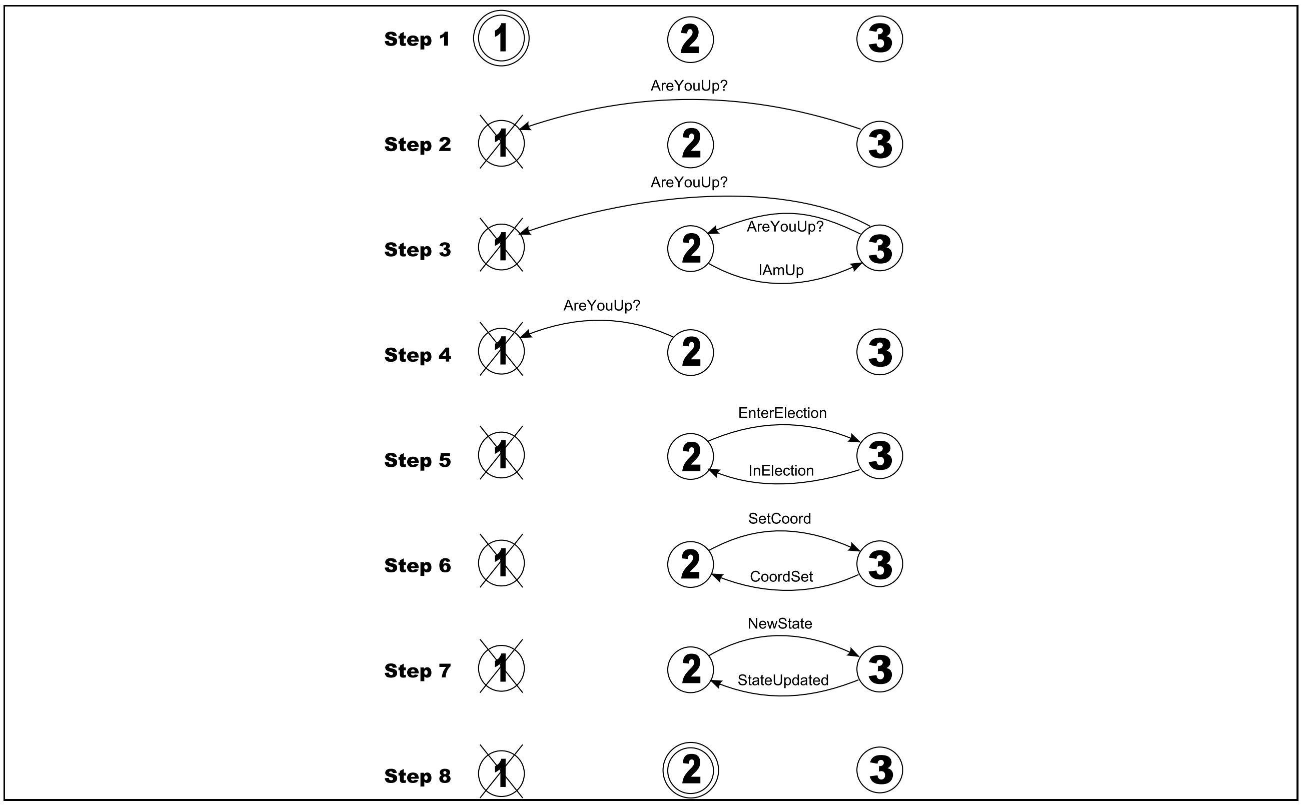


Figure 6.12: Election protocol of the Bully Algorithm implementation.

The third protocol is executed after an election. Once the election process is finished, the new leader starts the coordinator component, which is used to regularly check that all members are still running normally. If a member does not respond to the check in time, a new election starts. Each member also has a component that keeps checking on the leader. If the leader does not respond in time, the member that detected the failure initiates a new election.

6.2.2 Model Generation

The model generation involved three tasks: 1) selecting the test cases; 2) defining how to model the communication between components of a member and between members; and 3) the model creation. These tasks are further described below.

Test Suite Selection

The selection of the test cases was based on the operations allowed by the interface of the election console. These operations were *start*, *fail*, *recover* and *close*. They tell a member to start executing, simulate failure, simulate recovery and shut down completely, respectively.

Using these operations, a set of test cases was created to collect traces from executions involving one, two and three members. The selected test cases were the following, where S represents the command to start, F represents the command to fail, R represents the command to recover, C represents the command to close and the numbers between brackets define the priorities of the members executing the operations:

1. S(1), F(1), R(1), C(1)
2. S(1), F(1), R(1), F(1), C(1)
3. S(1,2), F(2), F(1), R(1), R(2), F(1), F(2), R(2), R(1), C(1,2)
4. S(1,2), F(1), F(2), C(1,2)
5. S(1,2,3), F(3), F(2), F(1), R(1), R(2), R(3), F(1), F(2), F(3), R(3), R(2), R(1), C(1,2,3)
6. S(1,2,3), F(2,3), R(2,3), F(1,3), R(1,3), F(1,2), R(1,2), F(1,2,3), C(1,2,3)

Each test case involved the abstract states of each executing member, which comprised its functional status (alive or down) and its membership status (normal or leader). These test cases were chosen with the purpose of producing traces where each member appears with different combinations of values of these two types of status.

Note that when a member was down, it did not matter which was its member status. Therefore, tests with only one member involved the abstract states {alive,leader} and {down}. Tests with two members included the same abstract states for the member with priority 1 and the abstract states {alive,normal}, {alive, leader} and {down} for the member with priority 2. As for the tests with three members, we had the same abstract states mentioned before for the members with priorities 1 and 2. The member with priority 3 had the same abstract states as those of the member with priority 2.

Modelling Communication

We modelled communications between threads as shared actions, regardless of their type (local or remote). Because it is important to specify the type, origin and destination of a message, we could not just synchronise on method names. The method name would tell us only the operation executed. The rest of the information is sent as parameters and our automatic instrumentation process does not collect information from parameters.

For this reason, we created user-defined actions to represent communication operations and included the necessary parameters in the names of the actions. The format of an action representing a communication was defined as

$\langle Op\rangle\langle Msg\rangle[\langle orig\rangle][\langle dest\rangle]$

where *Op* is the operation executed (send or receive), *Msg* is the type of the message, *orig* is the priority of the sender and *dest* is the priority of the receiver. Therefore, an action `sendAreYouUp[2][1]` corresponded to a message `AreYouUp` being sent from the member with priority 2 to the one with priority 1. It matched action `receiveAreYouUp[2][1]` from the receiver[2]. Relabelling operations [MK06] were used to synchronise "send" actions with their corresponding "receive" actions.

Following our definitions of active and passive processes (Chapter 3), we identified that this implementation contains elements which are both active and passive (see source code in Appendix C). Each component of the election member runs an internal thread that checks received messages and executes the necessary operations. In addition, each one of them has methods

[2]Note that, when the graphical representation is generated in the LTSA tool, the square brackets are replaced by dots, so that `sendAreYouUp[2][1]` becomes `sendAreYouUp.2.1`.

that can be called by other components, characterising a passive entity. We, therefore, modelled each component as the composition of two processes: one that models the internal thread (active behaviour) of the component and another that represents the access to its methods (passive behaviour).

We generated models of the active behaviour automatically and manually created the models of the passive behaviour. We decided to concentrate on the active behaviour because it represents the more interesting part of the behaviour of the components. Moreover, the passive behaviour is quite simple, being restricted to receiving external method calls and changing the value of internal attributes.

The model of the passive behaviour defined actions which synchronised "send" action of external components with "receive" actions of the local `MM`, representing that the local component received a message and stored it in the local variable (buffer). The collection of a message from the buffer was modelled as a synchronisation between the component and its local variable.

Some components included methods used by other components to tell them to stop (method `close)` or check whether there was any message in the single-slot buffer waiting to be collected (method `msgConsumed`). In this case, the synchronisation occurred directly between the component calling the method and the component whose method was being called.

Model Creation and Modification

Although the implementation defined all components as inner classes of a main class (see Appendix C), we created separate models for each one of them. This allowed us to look at each individual model and reason about the behaviours it described. Furthermore, with separate models it was possible to apply different refinements to each model, achieving models with different levels of abstraction when necessary.

Two values were important when creating the models: the priority of the member being modelled and the number of members involved in the execution. The priority of the member defined its specific behaviour and the messages it could send and receive. The number of members was also important to restrict which members other members could send messages to and receive messages from. The priorities of members were obtained from the value of the attribute `priority`, whereas a user-defined attribute `members` was created to provide the information about the number of members. These two values were used in the system state of all models.

The models created with these two values as system state showed that, from the initial state, multiple paths were possible to be taken. Each path represented a combination of the priority of the member and the number of members, both ranging from 1 to 3. This characterised the same problem found in the Dining Philosophers example, in Appendix B, where the model shows that there are different behaviours for philosophers with odd identifiers and those with even identifiers but either of them can be taken at the beginning.

To make it clear which behaviour a certain path describes, we manually modified the models by creating parameters for their FSP process definitions. Each process definition received the priority of the member and the number of members as parameters. Guards where used to define path restrictions according to the values received. The part of the FSP description of the Election Thread containing the guards can be seen in Figure 6.13.

```
deterministic ElectThread (P=1,N=1) = Q0,
Q0 = (when (P==1 && N==1) null -> Q1
     |when (P==1 && N==2) null -> Q13
     |when (P==2 && N==2) null -> Q48
     |when (P==1 && N==3) null -> Q77
     |when (P==2 && N==3) null -> Q112
     |when (P==3 && N==3) null -> Q155),
...
```

Figure 6.13: Example of use of guards in the FSP description.

Parameter `P` is the priority of the member and parameter `N` is the number of members. Therefore, we could create different instances of a process using different combinations of values for these parameters. Figure 6.14 shows the model of the Election Thread for a member with priority 2 and number of members equals to 2.

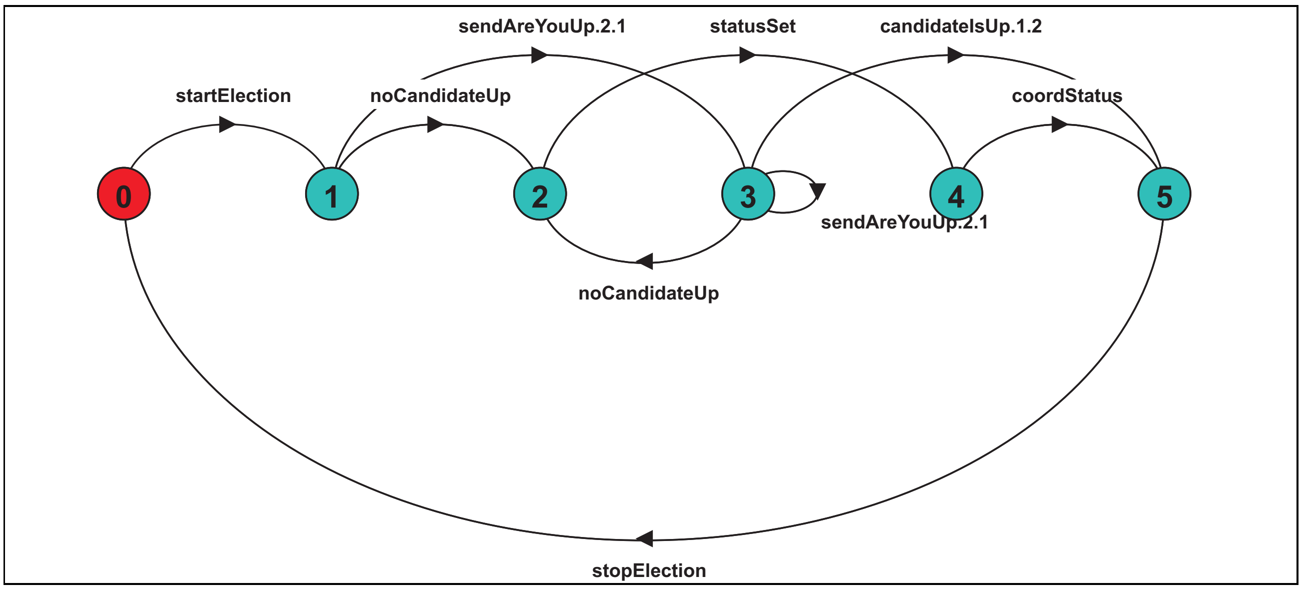


Figure 6.14: LTS model of the Election Thread for priority 2 and two members.

Another modification that proved to be necessary was replacing transitions leading to the `END` state with transitions leading back to the initial state. This change was due to the dynamic creation of threads involved in the system. Connecting the final state to the initial state represented that a thread could finish and then start again, simulating that a new thread had been created.

The timeouts of the system were not modelled. Rather, user-defined actions were created to represent the situation where a timeout occurred. The possibility of a normal execution or a timeout was represented by alternative paths from the same state (e.g., state 3 in the model in Figure 6.14, where `candidateIsUp.1.2` represents the reception of a response and `noCandidateUp` represents the timeout).

6.2.3 Property Checking

The essential property of the algorithm is a safety property that should guarantee that there can only be one leader at all times. Therefore, the property is violated if two members claim to be the leader at the same time. The FSP specification of this property is presented in Figure 6.15.

```
property OneLeader (N=3) = NO_LEADER,
NO_LEADER = ({m[i:1..N].monitor.normalStatus,
              m[i:1..N].memberFails} -> NO_LEADER
            |m[i:1..N].election.coordStatus -> LEADER[i]),
LEADER[i:1..N] = ({m[i].monitor.normalStatus,
                   m[i].memberFails} -> NO_LEADER
                 |when (i>1 && i==N)
                   {m[j:1..N-1].monitor.normalStatus,
                    m[j:1..N-1].memberFails} -> LEADER[i]
                 |when (i>1 && i<N)
                   {m[j:1..i-1].monitor.normalStatus,
                    m[j:1..i-1].memberFails} -> LEADER[i]
                 |when (i>=1 && i<N)
                   {m[j:i+1..N].monitor.normalStatus,
                    m[j:i+1..N].memberFails} -> LEADER[i]).
```

Figure 6.15: FSP specification of property `OneLeader`.

The action named `memberFails` represents that a member received a command to fail. Action `normalStatus` represents that the member status has been set to "normal", whereas `coordStatus` indicates that the member has been elected leader (coordinator). The names prefixing the actions correspond to the threads that executed them.

To check the property, the models of all components of each member were composed. Therefore, there was a composite model representing the complete behaviour of each member. However, when trying to compose two or more members, the resulting model would become too large and the LTSA tool would run out of memory. The main factor for that seemed to be the great number of alternative paths in each model, in particular the model of the `MM`.

Even though the composite model could not be generated, it was still possible to check the property, as the safety check of the LTSA tool may not need to generate the whole composition to detect violations. Applying the safety check to the composition of the composite models of two members and the FSP automaton of the property, the tool detected a violation. The error trace reported was the one presented in Figure 6.16. Actions prefixed with `m.1` belong to the member with priority 1 and actions prefixed with `m.2` belong to the member with priority 2. The line numbers have been added to help during the discussion.

```
Trace to property violation in OneLeader(2):
    1  m.1.memberStarts
    2  m.1.startMonitor
    3  m.1.startMM
    4  m.1.exp.startElection
    5  m.1.election.noCandidateUp
    6  m.1.enterElection.1.2
    7  m.1.election.timeout.2
    8  m.1.election.statusSet
    9  m.1.election.coordStatus
    10 m.2.memberStarts
    11 m.2.startMonitor
    12 m.2.startMM
    13 m.2.exp.startElection
    14 m.2.election.noCandidateUp
    15 m.2.election.statusSet
    16 m.2.election.coordStatus
```

Figure 6.16: Error trace for property `OneLeader`.

This error trace showed that the member with priority 1 (m1) might start earlier than the member with priority 2 (m2) and proceed to the election process (lines 1-4). Because it was the candidate with highest priority, m1 ignored any other candidate (line 5) and tried to send a message to m2 to let it know that an election was in progress (line 6). However, m2 was taking too long to start and a timeout occurred (line 7). At this point, m1 assumed it was the only one alive and, therefore, went on to change its status to “coord” (lines 8 and 9).

When m2 started (line 10), it also did all the necessary initialisations and began a new election process (line 13). Due to a possible delay in the communication, m2 assumed it was the only member alive (line 14) and claimed to be the current leader (lines 15 and 16). Therefore, there were two leaders at the same time, thus violating the property.

Two comments need to be made on the error trace found. The first comment regards the problem it revealed: if communication between members is too slow, the system might reach a state where there is more than one leader. To check that this is indeed true, we modified the code so that delays were included before each communication between members. It demonstrated that small delays in communication did not affect the correctness of the algorithm execution.

Nevertheless, from a certain value of delay, the system would fail to elect just one leader, having two members claiming to have been elected. Therefore, though we could not produce the entire composition of models and property, we could still find a real error.

The second comment on the error trace is that, even though it prompted us to check a problem that was proved to exist, the trace itself was infeasible. Looking at the code of the election thread (class `ElectThread`, in lines 505-685, in Appendix C), it is possible to identify that, when the thread starts, it will always check all members that have higher priority (i.e., members with priority values smaller than its own) by sending them an `AreYouUp` message (line 522). Therefore, there should be an action `m.2.sendAreYouUp.2.1` in the error trace between lines 13 and 14 to represent the sending of message `AreYouUp` from m2 to m1.

The model of the Election Thread, presented in Figure 6.14, shows that this error trace is possible from state 1. Note that, if the model takes the other path, it does include the check and then allows the trace where the candidate failed to respond in time or the trace where an answer was received (state 3), which is the expected behaviour of the component.

To understand the origin of the problem, one needs to remember how our approach works. When annotating a repetition statement, we put an annotation as the first statement of the loop and another one as the last. This way, every repetition of the loop leads us back to the same point: the beginning of the loop and our initial annotation marking a context. Because none of the available attributes (nor possible user-defined attributes) was modified at each new iteration of the loop, the LTSE tool interpreted all iterations as starting in the same context (the beginning of the loop), executing some actions and coming back to the initial point.

That is why there is a self-loop in state 3 in the election model (Figure 6.14). It is the situation where the check message was sent but no response was received. Because the message was sent to the first member on the list and there were more members to check, the LTSE created a transition from the previous context (the beginning of the loop) to the next context, which was the same.

The incorrect path was created in the very same way. The loop began, the member sent a message to a higher-priority member (priority 1) and did not receive a response. Then, it ignored itself and exited the loop to execute action `noCandidateUp`, since no candidate with higher priority replied to the check. Therefore, the last sequence the LTSE tool identified was that the context marking the beginning of the loop was found, then no action was taken inside the loop (member ignored itself) and then the execution went on to action `noCandidateUp`. Hence, the trace shown in the violation - in which the member seems not to send a check before assuming no candidate is up - was produced because the context originating the transition where the check happened and the one originating the one where the member ignored itself were understood to be the same.

As commented before, no refinement could be found to eliminate the invalid path. However, an experiment showed that the behaviour could be removed from the model if the value of the loop counter were used as an attribute. Note that we could not use local variables as attributes to solve the problem. Attributes are global variables and, thus, their values are accessible throughout the execution. Local variables, on the other hand, have valid values only in part of the execution and cannot be added to annotations, which are placed all over the program.

6.2.4 Evaluation

This case study was a great challenge to our approach. Characteristics such as distributed components and communication, dynamic creation of threads, great number of concurrent threads and components where two threads can execute at the same time were not involved in any of the previous systems we applied our approach to.

As expected, distributed components are supported by our technique. Components of the same type, executing at different locations, produce separate log files. These files can then be put together to represent different traces of the same type of component, thereby allowing us to generate a generic model based on the particular behaviour of each instance of the component.

Communication between remote components has to be abstracted using user-defined actions, so that the type of the message, the sender and the receiver can be identified. If such information is not relevant (for example, in a system with only two components that exchange just one type of message or where the essential information is that a message has been received), then method names could be used.

The dynamic creation of threads was an important aspect in the system. The Election Thread, for instance, could itself create a new election thread. During our tests, this situation did not happen, probably due to the incompleteness of our test suite. However, had it happened, the automatic generation of the model would not have inferred that there was a creation of a new thread. Because we merge the behaviours of all instances to create a model of the component type, we do not automatically identify distinct instances where one can create the other and both can execute simultaneously. In [MK06], the authors describe a way of modelling such behaviour and we intend to investigate the possibility of doing that automatically.

Even though our approach had proved to support the modelling of concurrent behaviour, the development of this case study showed some new important issues. The high level of concurrency, where usually six to seven threads would execute and interact in each member, demonstrated that our approach indeed produces models that represent abstractions of the behaviour of real processes. The fact that it was possible to find an existing property violation confirms that the models are useful for model-checking concurrent systems. Nevertheless, we also encountered some problems regarding concurrency.

One of such problems was that the value of some control predicates were incorrectly evaluated in the produced traces. That was not a problem related to the format of the annotations, but rather a problem related to the fact that their value was dependent on another thread. This situation appears in the timeout loops, where a thread is controlling the timer while waiting for a message to be received.

See, for example, lines 528-541 in Appendix C. The timeout loop has an internal test where the control predicate is `(inPacket!=null)`. The true value of the predicate indicates that a message has been stored in variable `inPacket`. Every time the loop executes, the current value of the predicate is evaluated in the annotation placed right before it in the code and, just then, evaluated in the actual statement. Since the change of the value of variable `inPacket` depends on another thread calling method `setMsg`, the call may happen just after the predicate evaluation in the annotation terminated. Therefore, the annotation would record the predicate as being false when it actually was true.

This shows the need for a better strategy of instrumentation to avoid this situation. Although this disagreement between annotation and actual value of predicate occurred during the development of the case study, its impact on the resulting models was not serious. In fact, the discrepancy could be easily spotted, as obvious invalid sequences appeared in the model, indicating that there was some problem in the traces. Nonetheless, in other situations, this might not be as easy to detect and have relevant consequences for the model generation and analysis.

Another problem found was related to the components combining active and passive characteristics. The possibility of having an internal thread and an external thread executing at the same time demonstrated that they should be treated separately. Mixing the behaviours of the two parts - active and passive - increased the complexity of the model and created some restrictions to the behaviour of the components which did not correspond to real restrictions. When modelling the two parts independently, we could look at each component as if they had two subcomponents: one that actively executed actions in sequence and another that passively awaited for methods to be called by other components. The impact of the execution of the methods on the active part was easily modelled as an interaction between the two subcomponents.

The issue of the invalid behaviour in the loop execution provided an insight that some refinements may need to be done only in parts of the code. As commented, the only way of refining the model to proscribe the invalid behaviour would be using a local variable as an attribute. However, this is not supported by our definition of system state, where attributes must be global, and so must user-defined attributes, which can only include expressions over attribute values. In this specific case, the solution would be a refinement executed only on that particular area of the code. This would be similar to the approach implemented by BLAST [HJMS03], in which different regions of the code may have different levels of abstraction.

Apart from the aforementioned problems and the fact that the composite model of two members could not be entirely generated in the LTSA tool, the approach proved to help the modelling and analysis of programs. The error trace, though not feasible, indicated a potential violation, which was confirmed by a practical experiment.

The error found was not actually a problem in the code, but a result of the influence of the environment on the execution of the system. Although the error may not be fixed by a simple modification in the code, the awareness of its existence allows users to be prepared for such a situation and strive to guarantee that the environment provides at least the minimum conditions to avoid the problem. Thus, the result of the analysis improved the knowledge about the system and correctly warned users about a possible violation of an essential property.

Note that the models contained some inferred additional behaviours, which were not observed during the trace generation phase. The addition of behaviours that were not detected using our test suite extended the coverage of this test suite, thus providing extra information about the program behaviour. This information is usually important when checking properties, since a non-observed behaviour might violate the property. If, however, an invalid behaviour is inferred, then the user may be prompted to further analyse the system and either identify real bugs, as was the case in this experiment, or detect false negatives. In this last case, a refinement should be applied to remove the infeasible behaviour.

6.3 Summary and Discussion

In this chapter, we presented the results of two case studies with different levels of complexity and obtained from two different sources. The major difficulty, however, was to find source codes to be used in the case studies. Even though many applications can be easily downloaded from the Internet, it is not usual to find their source code available. This demonstrates that our requirement of having access to the program source code limits the application of our approach to situations where either the source code is available, as in our first case study - which is not usually the case - or where the developer grants us access to it, as was the case in our second case study.

The case studies confirmed the application and importance of the idea of context when extracting models. Moreover, the possibility of creating user-defined actions permitted us a high degree of customisation of the models, so that relevant parts of the code without method calls could also be represented in the model. User-defined attributes extended this customisation to the possible expansion of the system state by defining expressions over existing attributes. This was particularly useful in the leader election application, where we would like to know the number of members involved in the execution but we only had the attribute which contained the data structure storing the list of members. An expression over the contents of this structure (method `size`) provided us with the necessary information.

Known limitations were also confirmed and will be further studied. Nevertheless, the case studies unveiled unknown problems as well. Anomalies in the models related to highly concurrent systems and to the necessity of local refinements revealed the need for further investigation and possible extension of the approach with mechanisms to minimise or eliminate these problems.

The development of these two case studies and several other examples gave us confidence on the usefulness of applying our approach for model extraction and subsequent model analysis. Though we encountered the mentioned limitations, the results of the analysis phase identified real property violations in both applications. Therefore, despite the existing problems, the approach proved to achieve its main objective, which is the extraction of models that can be used for checking properties and reveal potential violations.

Chapter 7

Evaluation and Conclusions

This chapter concludes the presentation of our approach for behaviour model extraction. It presents an evaluation of the approach according to the faithfulness of the generated models and their usefulness for model checking.

Our work is compared to other techniques found in the literature with which we share some ideas and characteristics. Each related work is analysed based on its advantages and disadvantages in relation to our model extraction process.

This chapter also includes an evaluation of the tool support according to some criteria and a summary of the contributions of this work. Possible future improvements are discussed.

7.1 Evaluation of Approach

This section presents an evaluation of the proposed approach for model extraction. It discusses the faithfulness of the generated models, their usefulness for model checking and how the approach presented compares to existing related work.

7.1.1 Faithfulness of Models

An important aspect of a model extraction process is to know when it can be considered successful. One way of measuring the success of such a process is to evaluate the faithfulness of the model it creates in relation to the program this model represents. Following this idea, the model extraction process is successful if the generated model is a faithful representation of the program behaviour.

Because completeness is not always possible, due to the size and complexity of current systems, we consider that our model extraction process is successful if it meets two requirements:

1. The created model includes all the necessary traces for checking a required property; and
2. Given a certain level of abstraction, the model contains only behaviours that are valid behaviours of the actual system.

Requirement 1 defines that the model does not need to be complete, but should contain the necessary behaviours to allow the checking of a property of interest. Requirement 2 defines that, using the information at disposal (i.e., the attributes of the system), it is possible to obtain a model that does not include any behaviour that the system cannot execute when considering the property being checked - i.e., the model is correct with respect to the property.

Based on these requirements, *the extraction process is considered to be successful if the model it generates is a correct, though partial, representation of the system behaviour, which can be used for model checking.* According to this, a model is a faithful representation of a program if it fulfils these requirements.

The development of the case studies presented in the previous chapter and other examples (see Appendix B for results of some of them) have indicated that the models we generate do fulfil the requirements when the traces used to build the model provide the necessary coverage with respect to a property of interest and the appropriate system state is selected. However, there were cases where the model was not complete or correct enough at first.

As discussed in Section 4.2, in the cases where the models do not contain all the necessary traces, violations may not be identified. Though the resulting model might contain more traces than those observed during the execution, since we can infer some additional behaviours, some behaviours cannot be inferred if a certain execution path is not exercised. Focusing the creation of the set of traces on the properties to be verified - perhaps using a test suite - could provide the necessary coverage to generate the models, thus including the required behaviours.

If an inappropriate set of attributes is selected, the model may contain invalid behaviours, causing false negatives. Thus far, we do not provide any particular heuristics as to how to select the appropriate level of abstraction. However, the use of our refinement process has proven that it can eventually lead to the extraction of a correct model with respect to a given property if this property is not violated by the implementation.

The creation of user-defined attributes provides the possibility of adding information to the model that is not directly available from the existing attributes. Furthermore, it represents a simple means of achieving data abstraction and, thus, a way of reducing the size of the models whilst still including the necessary information.

The experience gained from developing this work indicates that the correct selection of parameters (alphabet, traces and attributes) allows the generation of a compact and correct representation of a system. This representation is an approximation of the program behaviour and can usually be considered a faithful abstraction according to the established requirements.

It is important to mention that we assume that the systems from which we extract models are deterministic. This guarantees that every execution using a certain sequence of inputs will result in the generation of the same trace. In non-deterministic systems, one can obtain different outcomes for the same sequence of inputs. This means that the same test case would have to be executed several times in order to produce traces for each possible output. Even so, guaranteeing coverage of all possibilities could make it impracticable. Moreover, checking whether an error trace is real or not would also require multiple executions. If none of the executions reproduce the error trace, it is still not possible to be completely sure that eventually that sequence of actions will occur. Therefore, assuming determinism simplifies the process of improving completeness and correctness.

7.1.2 Usefulness for Model Checking

Our models can serve as inputs to a model-checking tool (LTSA), where temporal properties can be checked. Besides model checking, the tool allows using the models for behaviour simulation and model parallel composition.

The characteristic of our approach of creating one separate model for each component (class) of the system permits a modular checking of properties. If the properties refer to a single component (local properties), it is possible to create only the model of this specific component. When capturing traces, we execute the entire system, even if we are only interested in one component. This way, all interactions of this component with other components are recorded and, therefore, the actions included in the model of the component are a result of these interactions.

If multiple instances of a component (i.e., multiple objects of the same class) exist in the system, our approach permits the gathering of traces from all of them. These traces are then sorted according to the instance that they refer to, so that each instance behaviour is isolated from the behaviour of the others. Traces of different instances are treated as different traces from the same component and, as such, are then combined to form the general observed behaviour of the component.

Models of single components can be composed to form the behaviour model of the whole system or part of it. This way, it is possible to focus on the components in isolation, check the composition of a subset of components or even analyse the complete system. Properties regarding each subsystem can be checked independently.

The meaning of the actions in the alphabet of the model can be adapted to the type of process being represented (passive or active). The selection of the alphabet, which can be augmented using user-defined actions, also permits one to focus on the actions that are meaningful when verifying a certain property. This gives our approach a great level of flexibility and customisation, since the model can be tuned according to the user's needs.

Note that we normally use test suites to generate traces because they provide control over the behaviours we would like to observe. Furthermore, new test cases can be created based on identified violations of properties, thus improving an existing test suite. Nevertheless, the approach we described here is not necessarily connected to testing. The only requirement is the generation of traces, which can be done with random inputs to the system or just through a passive profiling of system actions.

7.1.3 Comparison to Related Work

We now discuss how the approach presented herein compares to some other existing techniques for model extraction. We focus on techniques that are well-known and have somewhat influenced or inspired our own technique.

For this discussion, we divide the approaches into four categories, based on the type of information collected to build models. We use the categories presented in Chapter 2 plus an additional category concerning custom-made model checkers. At the end, we comment on a general comparison of these techniques to ours.

Based on Static Information

In this category, we consider only techniques using static information, such as control flow information, to build models. For this reason, all approaches cited in this category differ from ours in that they do not need to execute the system to collect information. Other differences and similarities are commented for each related work.

Modex. The *Modex* approach [HS99, Hol01] was based on the idea of manually annotating ANSI-C source code for the extraction of high-level verification models. The user annotated relevant statements of a procedure using a predefined format. A mapping from the ANSI-

C statements to Promela commands had to be provided by the user to guide the extraction process. The result of the process was the creation of a Promela description of the system that was used as input to the Spin model checker [Hol97].

We share the idea of code instrumentation and a mapping from the programming language to a formal language, which serves as input to a model-checking tool. Using the TXL engine, we have been able to automatically annotate the source code and still allow the user to include their own annotations, if necessary.

Unlike the Modex approach, our mapping from the programming language (Java) to the modelling language (FSP) is predefined and automatic, which avoids mistakes during the translation. Moreover, the user usually needs little or no knowledge on either of these languages for most of the model extraction process. Knowledge on the programming language is only required for the inclusion of user-defined actions, whereas knowledge on the modelling language is necessary for the creation of composite models, guards, and other features of FSP.

Bandera. The *Bandera* toolset [CDH$^+$00] supports a property-driven model construction from Java code. Properties are specified using predefined patterns [DAC98]. Slicing [DH99] is used to remove statements and variables that do not matter for checking a given property and produce a reduced version of the code [HD01]. This reduced version can then be used to create a Promela model to be checked in Spin, a SMV input model [McM93] or be directly checked using the Java PathFinder (see Custom-Made Model Checkers category). The domain of values of variables is restricted using data abstraction.

We also direct our model construction by a property to be verified, but the properties we verify do not follow any previously created pattern. Furthermore, we do not use a reduced version of the code to generate models. Rather, we use the complete program to generate the traces and then apply a selective analysis to them, according to the actions required to be in the model, and the level of abstraction defined by the set of attributes composing the system state.

The instrumentation of the code to generate traces, without modifying it, guarantees that the behaviour of the system is not affected. When slicing is applied, it is necessary to verify that the reduced version of the code contains all the statements to ensure that no modification of the control and data flow of the system has occurred.

We apply a simple, manual data abstraction technique. Our technique is essentially based on the user's knowledge about the system. In the Bandera tool, however, data abstraction is automatic and, therefore, does not require user intervention.

SLAM. The *SLAM* project [BR02] considers sequential C programs to generate abstractions called boolean programs. *Boolean programs* [BR00] are programs where the control flow of the C programs is preserved but all variables are of boolean type. A boolean program is automatically obtained from a C program using predicate abstraction [GS97] and can be refined by the addition of new boolean variables representing predicates [BCDR04]. This is done with the help of a theorem prover and the refinement process follows the Counter-example Guided Refinement (CEGAR) paradigm [CGJ+03].

Whereas modifying the level of abstraction in our work is simple and involves only the selection of additional attributes to be monitored, SLAM uses a more complex approach. The automatic refinement of abstractions permits SLAM to achieve an appropriate level of abstraction without user intervention.

Statically deciding the feasibility of a path is, however, not always possible, in which case the SLAM tool returns a 'don't know' result. Though our refinement is manual, feasibility can be tested by attempting to replay an error trace in the program to check whether the behaviour is spurious or not. This can also be used to improve an existing test suite.

SLAM supports only sequential programs, as it is directed to the checking of device drivers. Our approach handles sequential programs as well as concurrent programs, being applicable to a wider range of types of systems.

BLAST. The *Berkeley Lazy Abstraction Software verification Tool* (BLAST) [HJMS03] is a tool written in Objective Caml and aimed to the checking of safety properties of C programs. The programs are represented as *Control Flow Automata* (CFA). A CFA is a CFG in which edges are labelled with statements executed between two control locations, represented by the nodes.

The tool implements the *lazy abstraction algorithm* [HJMS02] in which a reachability tree is built, representing the reachable abstract states of the program. Each node of this tree corresponds to a vertex of the CFA and is labelled with a formula. This formula represents the state of the system at that point, considering a set of predicates. If an error is found, an analysis of the sequence of states from the root to the error node is carried out to check whether it is a real counter-example. If it is an actual error, new automatically-identified predicates are added to refine only the subtree where the spurious error occurred. Similarly to SLAM, they apply an on-demand automatic refinement.

Instead of predicates, we use attributes to refine a model, although user-defined attributes may be employed to define predicates over attribute values. Our refinement is applied to the entire model, as opposed to the BLAST approach, where different areas of the reachability tree may have different levels of abstraction. Therefore, they reduce the costs of refinement by applying it only in specific parts of the model.

We do not provide such a selective refinement of a single model, but different precisions can be achieved for each model to be composed, thus allowing different levels of abstraction (different set of attributes) for each component of the system. Furthermore, as does SLAM, BLAST relies on a theorem prover to execute a refinement and, because of that, may encounter problems scaling to large programs, in particular, if they involve complex data structures [KGC04].

MAGIC. The *Modular Analysis of proGrams In C* (MAGIC) [CCO$^+$04, CCO$^+$05] tool is a C model-checker used to check properties of state machines. These state machines are presented as LKSs, thus combining states and actions (events). The properties are described using a logic called SE-LTL, which is an extension of LTL to support the use of actions in LTL formulas. The refinement process follows the CEGAR paradigm [CGJ$^+$03].

As we do, they follow a compositional approach to build models, so that checking and refinement can be applied component-wise and the complete model can be achieved via parallel composition. Though they use an LKS as their final model, we use it as an intermediate representation before building a final LTS model.

Their refinement process, though applying the same abstract-verify-refine idea as ours, uses predicate abstraction, whereas we apply a refinement through the expansion of a set of attributes considered during the model construction. The use of FLTL also allows us to mention actions in the formulas we check as well as states in very much the same way as they do in SE-LTL.

Model Reduction. In [GSVV04], the authors present an approach for model checking multi-threaded Java programs. They propose a mapping from Java to CCS [Mil89]. The CCS specification is then used as input to the CWB-NC model-checking tool [CS96] to check properties described using selective mu-calculus [BDFSV99].

Rather than performing an abstraction mapping, as we do from the programming language (Java) to a modelling language (FSP), the authors apply a more syntactical mapping. They translate Java statements into CCS operations according to some transformation rules. Because of this translation, they need to use Bandera to obtain a formula-based abstraction of the code. Though a reduction of the code to obtain a property-based version in our approach could reduce the costs of the model extraction process, it is not a requirement.

Their translation also requires that they make two assumptions. Firstly, they assume that the number of objects in the system is fixed and statically defined. This means that attributes and methods of a class are replicated for each object of this class during the translation. We, however, do not assume a static number of objects. Since we annotate the class code, any object created or destroyed dynamically will produce traces. These traces are recorded in the log file and merged with other traces from other instances when generating the model. Nevertheless, our approach does not deal well with instances that can create other instances of the same class (see the Leader Election case study in Chapter 6).

The second assumption is that the domains of values of all variables are finite. They apply a data abstraction technique to guarantee this. Our approach does not include a data abstraction technique to control the different values that attributes can be assigned. As previously commented, we implement a simple technique of data abstraction, which relies on the user creating expressions over the values of attributes. Though it does not limit the number of different values an attribute can be assigned, it allows the user to define which information about the attribute is relevant. For example, it might be more relevant knowing whether a document is open or not in an editor than knowing the name of the document.

FLAVERS. The *FLow Analysis for VERification of Systems* (FLAVERS) [TAC04, CCO02] employs data flow analysis techniques to check behavioural properties of Ada programs. These properties must be specified in terms of sequences of events, so that it can be translated into a finite-state automaton. The tool automatically extracts a *Trace Flow Graph* (TFG) from the source code and applies an algorithm called *state propagation* to associate states of the property with states of the TFG. The checking of the property corresponds to identifying the set of states of the property associated with the final state of the TFG. If only accepting states are in this set, then the property is said to hold. Otherwise, a counter-example is provided.

An idea that our approach shares with FLAVERS is the incremental addition of information to eliminate false negatives. Whereas they add constraints to prevent some paths from being taken, we add attributes that split states into a set of distinguishable states. In both approaches, however, this increment of precision is not automatic and requires user intervention.

An important difference between our technique and FLAVERS is that they require the modelling of the constraints meant to eliminate spurious errors, in the form of finite-state automata. Hence, the user needs to know the modelling approach and how to create a model that will add the expected constraint to the program model. The addition of attributes is simpler and only requires the user to select one or more attributes from a finite set of attributes the system possesses or define expressions over them.

Based on Dynamic Information

In this category we consider techniques using dynamic information, such as samples of execution (traces), to build models. For this reason, all approaches cited in this category differ from ours in that they do not require access to the source code. Nevertheless, they do not take into account the influence of the program control flow to the generation of actions recorded in the traces. Other differences and similarities are commented for each related work.

Grammar Inference. Cook and Wolf's work [CW98] proposed the use of *grammar inference* for process discovery. They presented three techniques, namely RNet, KTail and Markov, that could be used to analyse a trace containing a sequence of events (method calls) produced by a system. Treating the trace as a string, they applied the techniques to infer an FSM from it.

In all techniques, the inference process was based only on the observed sample, making the resulting model totally dependent on the trace produced. For instance, the Markov technique used statistical information to identify common subsequences of actions. Subsequences of two or three events appearing in the trace with a frequency under a certain threshold were discarded and, thus, not included in the model. This work was then extended by Mariani's [Mar05] to allow the merge of multiple traces, also using a statistics-based technique. Common subsequences of events in two or more traces are assumed to represent the same sequence of actions and, therefore, are merged when the model is generated.

Although the models produced with our approach are also somewhat dependent on the input traces, the use of the concept of context allows us to infer additional behaviours based on information collected from the source code. Therefore, we do not need any statistical analysis to discover sequences that indicate an ordering relation between actions, since we can, accurately, obtain this information from the program control flow graph. Moreover, our merging procedure is also based on accurate information and does not introduce invalid behaviours at the selected level of abstraction, as it is the case with the techniques presented in [CW98].

The KTail algorithm has been augmented with information on values of parameters for each call of a method [LMP06]. The goal is to improve the accuracy of the merging of models inferred from different traces. Daikon [ECGN01] has been used to calculate invariants over the values of parameters and, therefore, allow similar traces to be put together. However, it is still completely dependent on the observed behaviours and does not offer the level of accuracy necessary to guarantee that no invalid behaviours are included in the model.

Specification Mining. *Specification mining* was the approach presented in [ABL02]. A machine learning technique was used to obtain a model of protocols followed by applications to interact with an application program interface (API) or an abstract data type (ADT) implemented in C. A tracer was used to instrument the code in order to record interactions of the program with the API or ADT. The tracing part involved the creation of wrappers to capture calls to methods of the API or ADT. The system was then executed to generate the traces.

The traces were refined using a flow dependence annotator. Flow dependencies represent the connection of attributes of interactions that change the state of an object to interactions that use the state of this object. It was necessary to have an expert who would determine which attributes of interactions defined objects and which used objects. Interaction scenarios were obtained from the annotated traces, identifying sets of interdependent interactions. An automaton learner generated a *Probabilistic Finite-State Automaton* (PFSA) that accepted a superset of all the strings in the training interaction scenarios, which was then converted to a non-deterministic finite automaton (NFA).

The results of this technique, which could be applied only for single-threaded systems, depended on the length of the scenarios to be inferred from execution traces, which was limited and predefined by the developer. Moreover, they assumed that frequent behaviours are usually correct behaviours, whereas rare behaviours indicate faulty executions. Based on this assumption, they would discard the “uninteresting” behaviours, thus losing information that could be relevant for finding errors.

Regular Extrapolation. Hagerer and et al. [HHNS02] propose a technique called *regular extrapolation* for automatically creating models focusing on certain aspects of a system using machine learning and finite automata theory. They create an automaton that extrapolates from observed finite executions to infinite behaviours, according to regular patterns. The patterns they discover depend on the test suite they use to learn frequent sequences of events and on experts’ knowledge to establish distinctions between states that could not be achieved using the learning algorithm. Experts also have to rule out some patterns, when necessary.

They also guide the creation of the model by the definition of properties to be checked, just as we do. However, as the authors themselves point out, their models are unsafe, since they do not build models using any safe information. We, on the other hand, take the control flow graph of the system as our guide to allow us to “extrapolate” from observed behaviours.

To some extent, their use of experts’ knowledge to provide information to distinguish states and eliminate patterns is similar to the input we use to provide the necessary refinements to rule out infeasible paths. In our approach, the user cannot influence the behaviours included in the model, in the sense that all observed sequences are feasible behaviours and, therefore, must be incorporated into the model. However, the user can define a set of attributes that is used to distinguish states and, this way, exclude invalid behaviours.

Based on Static & Dynamic Information

To our best knowledge, the only work to effectively attempt to put static and dynamic information together to generate an abstraction of the system is the one described in [NE02]. It presents a combination of static and dynamic information to recover program specifications in the form of a set of program invariants and verify the absence of runtime errors. Daikon [ECGN01] dynamically detects possible program invariants and annotates the detected invariants in the Java source code and then the static verifier ESC/Java [LNS00] analyses the annotated program to check which invariants can be statically verified.

Apart from the benefits of combining these two tools, the combination also put the deficiencies of both of them together. Daikon can detect only a limited number of likely invariants. Out of the types of invariants detected by Daikon, ESC/Java can verify just a subset of them. This restricts the use of such tools for testing some invariants that might be of interest.

We share the same underlying idea of [NE02] of putting static and dynamic information together. However, their focus is on state properties, such as invariants of attributes of a class, rather than the dynamic behaviour of a component in terms of its required and provided services. Furthermore, our combination actually combines static and dynamic information to build an abstraction, whereas they use results from an analysis on one type of information to confirm the results from an analysis on the other. The actual combination of static and dynamic information through context information also permits us to work with a single tool.

Custom-Made Model Checkers

Verisoft [God03] and *Java PathFinder* (JPF) [VHB$^+$03] present the possibility of controlling the execution through a custom-made environment to verify feasible behaviours. Therefore, they allow checking properties without having a proper model. For this purpose, they dynamically store state information about the system and explore possible paths, recording results.

Because these tools are integrated into the execution environment of the system, they are theoretically independent of programming language. Moreover, because no model is generated, they also do not require any specific modelling language. Nevertheless, we believe that having a model is useful for a range of purposes other than just verifying properties, such as simulations, animations, performance analysis and model parallel composition to be used, for example, for compositional reasoning and software evolution.

The scalability of these tools and their use for checking distributed systems is restricted by the amount of information they have to deal with to keep track of paths already taken and avoid redundant information. The use of our approach can easily scale to large systems, since all the storage space it requires corresponds only to the length of the log files of the local components of the system for which we record traces. The use of abstractions produces compact models, which take considerably little memory space.

General Comparison

We now present a summary of the comparison to the cited work. For this, six criteria will be used:

1. *Type of information* (Info): defines which type of information is used to build a model: static, dynamic or static and dynamic (SD);

Approach	**Info**	**PL**	**ML**	**SC**	**Conc.**	**Ref.**
Modex	Static	C	FSM	$\surd$	$\surd$	-
Bandera	Static	Java	Promela	$\surd$	$\surd$	-
SLAM	Static	C	Boolean program	$\surd$	X	Auto
BLAST	Static	C	CFA	$\surd$	X	Auto
MAGIC	Static	C	LKS	$\surd$	$\surd$	Auto
Model Reduction	Static	Java	CCS	$\surd$	$\surd$	-
FLAVERS	Static	Ada	TFG	$\surd$	$\surd$	Man.
Grammar Inference	Dynamic	-	FSM	X	X	-
Specification Mining	Dynamic	C	PFSA	X	X	-
Regular Extrapolation	Dynamic	-	FSM	X	X	Man.
Daikon & ESC/Java	SD	Java	Invariants	$\surd$	X	-
Verisoft/JPF	Dynamic	C/Java	-	$\surd$	X/$\surd$	-
LTSE	SD	Java	LTS	$\surd$	$\surd$	Man.

Table 7.1: Comparison to related work on model extraction.

2. *Programming language* (PL): the programming language from which a model is obtained;

3. *Modelling language* (ML): the language or formalism used to present the resulting model;

4. *Source code* (SC): defines whether the source code is required;

5. *Concurrency* (Conc): defines whether the approach provides support for modelling concurrency;

6. *Refinement* (Ref): defines the form used to refine initial models: manual or automatic.

Table 7.1 presents the values of these criteria for each work discussed before. The table highlights the general idea that approaches based on static information require access to the source code and are, therefore, dependent on the programming language. Approaches based on dynamic information do not require the source code and can usually be applied to any programming language. The table also shows that formalisms based on finite-state automata are the most used, thus giving support to our option for one of these formalisms.

Our approach provides a type of combination of static and dynamic information that is not allowed by any previous work. The use of contexts ensures that we can cope with multiple traces without introducing invalid behaviours, given a certain level of abstraction. Unlike most of the approaches cited here, we are able to deal with concurrency and allow the user to refine an initial model when false negatives are identified. Furthermore, our models can be used to check properties not supported by other techniques, such as FLTL properties.

Our discussion on related work demonstrated that the proposed approach presents some advantages over other techniques, hence, indicating its relevance and contribution to the area. The comparison to other techniques also showed that our approach can learn more from related work, in particular towards providing automatic refinement of initial abstractions.

7.1.4 Main Applications

The proposed approach has, as its main goal, the generation of a model to be used for checking temporal properties. Therefore, the resulting models are applicable for model checking in the LTSA tool - or any other tool that could accept an FSP specification as input.

Taking advantage of the features of the LTSA tool, models can also be used to analyse and understand the behaviour of a system. This applies to sequential as well as to concurrent systems. The application of our approach for the analysis and model checking of concurrent systems is important, since some errors related to concurrency are not easily detected and may generate significant consequences to the overall behaviour of the system. Thus, their identification and correction is of great relevance.

We have observed that our approach is particularly suitable for reactive systems [MP92]. The reason for that is that these systems provide a behaviour that corresponds to our way of generating traces, which is using test cases to send stimuli to the system and collecting its reactions. Therefore, having a system whose behaviour follows exactly this idea of receiving inputs and reacting to them in some manner, makes it easier to control the behaviours that will appear in the traces.

The application of the technique for distributed systems was tested during the development of the case studies discussed in Chapter 6. Because we can collect information locally and interactions between distributed components can be identified via action synchronisation, components, wherever they may be, can be annotated and executed locally. Once the models for each component have been generated, a global model can be obtained by composing these models. However, the need for a representation of location still has to be further investigated, as it may be relevant in the context of certain distributed systems, where components behave differently when interacting locally or remotely.

Representation of locations is also important to allow the checking of mobile applications [FPV98], where components can change their locations dynamically. Hence, keeping track of their moves is essential to understand their behaviour and identify problems.

7.1.5 Limitations

As any other technique, despite its advantages, our approach has some known limitations. Though they were mentioned in the previous chapters, we here summarise the most important limitations we are aware of.

It seems that the approach's main limitation is that it requires access to the source code of the components of the system - a limitation that it shares with some related work (e.g., SLAM and Bandera). This requirement restricts its application to systems where the implementation is available. Moreover, because it relies on the source code to collect the necessary information, the information gathering process needs to be adapted for each particular programming language.

The influence of the coverage provided by the set of traces on the resulting models has already been commented on and is another limitation of the approach. The part of the system behaviour described in the model is directly related to the collected traces. Hence, models generated using our approach will allow the checking of properties based on the observed behaviours and, perhaps, some inferred additional behaviours. Therefore, completeness cannot be usually guaranteed and one should focus on selecting a set a traces which shows the behaviours that are relevant for checking properties of interest.

Correctness is guaranteed up to the level of abstraction provided by the set of attributes. Usually, this level corresponds to that necessary to prove that a certain property holds. Hence, correctness is assured up to the property being checked. For this reason, the adequate selection of attributes can eventually produce a model that is correct with respect to the property. However, finding this adequate set of attributes may not be simple. Currently, our approach fails to provide appropriate guidelines on how to choose attributes. All we offer are some ideas on attributes that are likely to help refine the model and those who should be avoided (see Chapter 4).

As a consequence of not having a well-defined heuristic to select appropriate attributes to refine a model, our refinement process is still manual. In spite of the freedom it gives to the user to try different combinations of sets of attributes, it makes the process too dependent on the knowledge the user has about the system. As in many cases the person executing the verification is not the same who implemented the system, the essential knowledge to guide this process may not be at hand.

The leader election case study (Chapter 6) also demonstrated that, on some occasions, the refinement of contexts does not rule out infeasible behaviours. We limit the possible refinements to attributes and expressions over their values. However, certain situations may ask for a refinement which takes into account the value of local variables as well. This is a feature that our approach currently does not support.

Using our approach to model systems where it is important to distinguish the behaviours of different instances does not produce good results. The reason is that we assume that the composition of the behaviours of various instances makes the general behaviour of the component being instantiated. Hence, if there is something in the parameters of instantiation that define different behaviours for instances of the same component, then the model will not include this information. This can be overcome by using guards in the FSP generated by the model extraction process, as we did in the Leader Election case study (see Chapter 6).

Our approach does not support the automatic modelling of timed and dynamic systems. Timed systems require the modelling of timers, used to define time constraints on the occurrence of certain actions. Dynamic systems involve components that can be dynamically created by the system and may terminate before the system execution ends. Thus far, timers are represented by an action describing the occurrence of a timeout, indicating that a time constraint was not respected. As for dynamic components, the extracted models need to be manually modified to describe the dynamic instantiation of components.

7.2 Evaluation of Tool Support

This section presents our evaluation of the tool support provided for our approach. Essentially, we analyse how easy it is to use the LTSE tool and what can influence its performance and scalability. We also discuss some known limitations.

7.2.1 Usability

The principal aim of a tool is to make easier the execution of a task that would otherwise be complex, tedious or time-consuming. Hence, a tool should serve to reduce the complexity of the task and speed up its completion.

In model extraction, the use of a tool is even more important. Converting a program into a model that is tractable by a model-checking tool is generally complex enough to be time-consuming and error-prone. By automating this process, one can gain time and decrease the possibility of errors. Furthermore, it allows even those who are not experts to apply the approach to obtain models from their codes.

Based on this, we consider that the LTSE tool is adequate for its purpose of implementing our approach. Although it does not automate the information gathering and trace generation phases, it provides an automatic way of generating an FSP description based on a set of traces from the system, guided by parameters provided by the user (alphabet, system state and interpretation of actions). This is the most difficult step of the approach and likely to introduce errors. Thus, the process implemented by the LTSE tool is essential and the combination of this tool with the LTSA tool provides complete support for the model checking process.

An important characteristic of the LTSE tool is its portability. As it is implemented in Java, it can run in any machine where a Java Virtual Machine has been installed. This is especially important if one intends to analyse a distributed system running in a heterogeneous network.

7.2.2 Performance and Scalability

Besides being useful and easy to execute, a tool should ideally not require much processing power and produce results quickly. The process executed by the LTSE tool demands reasonably little processing effort, being most of its work related to operations on files and access to data structures (i.e., the context table and the model structure). At most, the tool operates on two files simultaneously - one to read from and another to write in - when applying one of the necessary mappings.

Knowing that most of the processing is connected to operations on files and data structures, it is clear that the general performance of the tool depends on the size of the files and data structures it has to handle. Long log files will generate long context files. However, long log files tend to include redundant sequences of actions. Thus, the resulting FSP description file is much smaller than it could be according to the size of the original logs.

The size of the data structures is, to a great extent, also dependent on the size of the log files. The more context the traces in the files include, the more entries the context table will have and the bigger the model structure will be. Redundant information in the logs can also mean that the structures will not be as large as they could, in particular the context table. Nevertheless, the size of the model structure is more easily affected by the length of the logs. Whereas the context table only grows with the discovery of new contexts, the model structure grows also when new transitions between contexts are detected. In general, this growth is not very significant and the size of the structures is perfectly manageable.

The types and ranges of the attributes of a system also influence the performance of the tool and its memory usage. In order to accelerate the refinement process, we collect the values of all attributes. This means that, irrespective of which attributes will be actually used, each entry in the log file registers the values of every attribute available. However, this information overhead is not carried over to the context files, since they only record context IDs and action names. The ranges of values of the attributes included in the CT affect the CT size, the size of the context files and that of the resulting model structure.

Table 7.2 shows some performance data collected from the programs used as examples in this work, including those presented in Appendix B and the case studies discussed in the previous chapter. These values were obtained executing the tool in a 2.4 GHz Pentium 4 machine with 512 MB of RAM running Windows XP. Rows containing the same log size represent results from the same program with and without attributes being considered, respectively (e.g., the first and the second rows).

The sizes of models and context tables and the processing time are approximate. The model and the context table sizes correspond to the quantity of memory occupied by these structures during the model creation. The log size is the sum of the size of all logs used in the model extraction process.

Log size (KB)	**CT size (KB)**	**Model size (KB)**	**Time (ms)**
8	0.8	0.4	31
8	1.5	0.5	31
60	7.5	1.2	63
60	8.9	1.3	62
257	3.7	0.4	63
1,913	21.2	15	312
96,390	18.9	3.1	14,235
462,445	7.7	1.5	58,593

Table 7.2: LTSE performance data.

The table shows how the size of the logs influences the sizes of the CT and that of the model. Note that, as commented before, the increase in the log size does not necessarily mean an increase in the size of the other structures.

Let us take as an example the fifth row of the table. It shows that, though the size of the log is large if compared to other log sizes in the table, the sizes of the CT and of the model happen to be smaller than, for example, those shown in the row immediately above, where the log size is about four times smaller. This indicates that the redundancy of information in the log described in the fifth row is much smaller than that in the log described in the fourth row. Logs that generate larger models than logs (e.g., the sixth row compared to the seventh) are a result of the quantity and type of attributes used to refine the model and the fact that these refinements can create more contexts and, consequently, more states.

7.2.3 Known Limitations

The LTSE tool has some known limitations. One of such limitations is the absence of a graphical interface. Although it means that the execution of the tool requires less processing power, since no graphics processing is required, it would be desirable to have a more friendly interface.

The representation of a method execution as an action whose name matches that of the method it describes seems a natural choice. However, it hampers the use of some features provided by programming languages, in particular related to object-oriented programming. The overload of methods cannot be represented in the model, since a call to any version of a method m will result in the introduction of an action m in the model, regardless of the parameters and return type. Hence, polymorphism is not represented either.

Support for inheritance is not provided, as it requires access to the code of the superclasses, which may not be available, especially if they belong to some third-party library. Moreover, as mentioned before, the tool does not distinguish between methods that have the same name, even if they are in different classes. Therefore, possible overridings would not be captured.

In order to handle the end of a log file and guarantee that no information is lost, the tool builds models under the assumption that the execution of a component always terminates (normally or abnormally). This means that it introduces either a reference to the predefined END state (if normal termination) or to a $FINAL$ state (for an abnormal termination). In cases where the execution does not finish (infinite loop) but is interrupted, the inclusion of a $FINAL$ state means that the tool could not find a next context to connect the last one to and, therefore, it has connected the last context to the $FINAL$ state. During the analysis, this transition may be misleading. Nonetheless, thus far, this seems the best solution for this problem and the users of the tool need to be aware of this assumption.

7.3 Summary of Contributions

The main contributions of this work are summarised below:

- *Definition of contexts*: Our approach presents the combination of static and dynamic information. Though this idea had been advocated before [Ern03] and put into practice [NE02], we propose the concept of contexts as a way of merging the two types of information as part of a model extraction process. The use of contexts has proven to allow the creation of models that accurately represent the behaviour (at a certain level of abstraction) of the systems they describe and has demonstrated to provide a solid ground for inferring additional valid behaviours. This approach helps bridge the gap between programming and modelling languages, which have slowed down a wider use of model-checking techniques and tools;
- *Generated models are useful for property checking*: The models created following our model extraction process can be used for checking temporal properties. The possibility of generating such models, which can serve as inputs to a model-checking tool, makes the checking process easier and accessible even to non-experts;
- *Definition of a refinement process*: Initial models can be further refined through a simple process of adding attributes to the set of attributes included in contexts. The refinement of models can eventually lead to models that are correct with respect to a given property;
- *Compositional modelling and checking*: Our approach allows each component of the system to be modelled independently, thereby enabling the individual analysis of components and the checking of local properties. Through parallel composition, which is supported by the formalism we have adopted, it is possible to combine the models of each compo-

nent into a single model. Global properties can then be checked on this composed model, which represents the behaviours allowed by the components when executing in parallel;

- *Inference of additional behaviours*: Models may include more traces than those observed during the trace generation. This means that the inference process we apply to create the models may infer additional behaviours from those found in the set of collected traces, thus improving the completeness of the model. If a test suite is used to generate the traces, then these inferred behaviours may increase the coverage provided by the test cases and reveal real errors which may not have been detected during the testing phase;

- *Support for concurrency*: It is possible to obtain models of components of concurrent systems. Interactions between components are represented as synchronisations on action names. The access to shared resources is modelled as active components that interact with passive components. The support for modelling concurrency is important to detect errors that could not be easily detected only based on testing;

- *Development of tool support*: The creation of the LTSE tool to implement the approach facilitates its application and reduces the complexity of the model extraction task. Moreover, the tool support allows the use of the technique even by users who have limited knowledge of either the programming or the modelling language, as the mapping from one to the other is mostly automatic;

- *Models can be adapted to user's needs*: The approach provides enough flexibility for the user to customise the resulting models according to their needs. The model alphabet can be extended using user-defined actions, which can mark relevant points in the code that do not correspond to method calls or method bodies. The meaning of actions in the model can be selected to represent a method call, method completion or to describe "method enter" and "method exit" events. The system state can also be expanded by the creation of user-defined attributes. They allow the user to define expressions, using the available attributes, in case these attributes are not enough to achieve the necessary level of refinement. Furthermore, the approach can be easily adapted to imperative programming languages other than Java, thus permitting the extraction of models for virtually any system written in a language falling into this category.

7.4 Future Work

As future work, we plan to improve the approach and the LTSE tool so as to eliminate current limitations. One of such limitations is the requirement of needing access to the source code. Particularly, when dealing with Java programs, we could create an instrumentation scheme to annotate the bytecode rather than the source code. The availability of the necessary context information in this language requires further investigation.

We also plan to extend the approach to other imperative languages, such as C and C++. This will require the creation of specific rules of annotation for these languages. If TXL is used, it will be necessary to construct a TXL grammar for each new language as well. In fact, we intend to analyse other alternatives to carry out the instrumentation process so that it could be more precise and overcome current restrictions and problems.

Another future work concerns the possibility of further automating the model extraction process. One step would be identifying and selecting the necessary parameters without user intervention. As we direct our model construction by the property we would like to check, the property specification could be a source of inference of the necessary information. For instance, we could identify the actions that should be part of the model alphabet according to the alphabet of the property.

The refinement process is another part of the approach that we envisage as a possible automatic procedure. Before that is possible, we have to create heuristics to guide the selection of candidate attributes. Though we have already identified that attributes used in control predicates are more likely to produce better results during the refinement, we still need to find a more formal definition of the influence of these attributes regarding the checking of properties.

Automatic data abstraction techniques could be studied as a means of allowing the automatic refinement of models and reducing their size without loss of relevant information. Though we provide a simple approach for data abstraction, namely user-defined attributes, the definition of such expressions requires that the user have a good understanding of the system so that they can choose the appropriate expressions. An automatic control flow analysis could identify the necessary abstractions. Nevertheless, the support of a theorem prover might be required, as is the case in tools supporting this feature, such as SLAM.

We intend to investigate how different coverage criteria influences the resulting models. Even though we might have complete coverage according to a particular criteria, it seems that it does not guarantee that the model to be generated will be complete. Finding the most appropriate coverage criterion (or a combination of multiple criteria) is part of our future work. 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 is to study the application of slicing to eliminate unnecessary parts of the code and allow the instrumentation and execution of a reduced version of the implementation. Using a property to be checked as the criterion to create the slices, we might be able to achieve completeness with respect to this property.

An important issue to be addressed would be the possibility of using parameters in the models we generate. In cases where the value of parameters passed to process definitions can alter their behaviour, this possibility would be essential for obtaining precise results (see the Leader Election case study in Chapter 6 and the Dining Philosophers example in Appendix B). How to automatically obtain and how to introduce these parameters in the model are questions to be considered. Thus far, corrections need to be made by hand and might not be simple.

Investigating the introduction of a refinement based on local variables is yet another possible future work. It may lead to the definition of *local system states*, providing a different level of abstraction from the current global system state. This means that the counter of a for-loop, for example, could be used to refine the specific context created by this statement so that each iteration could be distinguished when extracting the model.

Finally, we plan to improve the LTSE tool by developing a graphical environment to replace the current command-line interface. The possibility of integration with the LTSA tool will also be considered, as well as the combination with other tools that could provide support to mitigate some limitations of our tool, such as a test case generation tool.

Appendix A

TXL Java Grammar and Annotation Rules

The Java Grammar used in this work to annotate the code so as to produce the context information is the one available at http://www.txl.ca/nresources.html, which describes a TXL grammar of Java 1.1. Using this grammar, a list of new *definitions* and *redefinitions*, as allowed by the TXL language, was applied. These modifications aimed to redefine some statements so that instrumented versions of these statements would be accepted as part of the grammar when parsing and annotating the code[1].

The list of definitions and redefinitions is presented below. The definitions named as "original_⟨token⟩" are used to mark statements of type ⟨token⟩ as not parsed yet, whereas definitions named as "processed_⟨token⟩" mark statements of type ⟨token⟩ as already parsed. This prevents the TXL engine from annotating a statement more than once. The redefinitions cause the inclusion of these new definitions in the grammar.

[1]For an explanation on the construction of definitions, redefinitions and rules in TXL, refer to the TXL manual available at http://www.txl.ca/docs/TXL104ProgLang.pdf.

```
% Defines printable tokens
define printable
    [stringlit]
  | [charlit]
  | [number]
  | [expression]
  | [attribute]
end define

% Defines a list of printable tokens
define printable_list
    [printable]
  | [printable_list] '+ [printable_list]
  | [empty]
end define

% Define a print statement,
% used to annotate control
% flow statements
define print_statement
  'System '. 'err '. 'println
  '( [printable_list] ') '; [NL]
end define

% Create a user-defined comment
% to identify events
define user_statement
  '# 'action ': [printable_list] ';
end define

% Defines statements as original
% (used to recognise statements not
% yet parsed)
define original_statement
    [label_statement]
  | [empty_statement]
  | [expression_statement]
  | [if_statement]
  | [switch_statement]
  | [while_statement]
  | [do_statement]
  | [for_statement]
  | [break_statement]
  | [continue_statement]
  | [return_statement]
  | [throw_statement]
  | [synchronized_statement]
  | [try_statement]
  | [block] [NL]
  | [comment_NL]
  | [user_statement]
end define

% Defines statements as processed
define processed_statement
    [original_statement]
  | [print_statement]
  | [empty]
end define

% Redefines statements as original,
% processed or empty
redefine statement
    [original_statement]
  | [processed_statement]
end redefine

% Defines method declarations as
% original (used to recognise method
% declarations not yet parsed)
define original_method_declaration
  [NL] [repeat modifier] [type_specifier]
  [method_declarator] [opt throws]
  [method_body]
end define

% Defines method declarations
% as processed (i.e., already parsed)
define processed_method_declaration
    [original_method_declaration]
  | [empty]
end define

% Redefines method declarations
% as original, processed or empty
redefine method_declaration
    [original_method_declaration]
  | [processed_method_declaration]
end redefine

% Defines method bodies as original
% (used to recognise method
% bodies not yet parsed)
define original_method_body
    [block]  [NL][NL]
  | ';       [NL][NL]
end define

% Defines method bodies as
% processed (i.e., already parsed)
define processed_method_body
    [original_method_body]
  | [empty]
end define

% Redefines method bodies as
% original, processed or empty
redefine method_body
    [original_method_body]
  | [processed_method_body]
end redefine

% Defines declarations as original
% (used to recognise declarations
% not yet parsed)
define original_declaration
    [local_variable_declaration]
  | [class_declaration]
end define
```

```
% Defines declarations as processed
% (i.e., already parsed)
define processed_declaration
    [original_declaration]
  | [print_statement]
  | [empty]
end define

% Redefines declarations
redefine declaration_or_statement
    [original_declaration]
  | [processed_declaration]
  | [statement]
end redefine

% Attributes
define attribute
  [id]
end define

% Defines user-defined attributes
define user_attribute
  '# 'attribute ': [printable_list] '=
  [expression] '; [NL]
end define

% Defines attributes as original
% (used to recognise attributes
% not yet parsed)
define original_attribute
    [repeat modifier] [type_specifier]
    [variable_declarators] '; [NL]
  | [user_attribute]
end define
```

```
% Defines declarations as processed
% (i.e., already parsed)
define processed_attribute
    [original_attribute]
  | [empty]
end define

% Redefines a variable declaration
% so that an attribute can be
% either original or processed
redefine variable_declaration
    [original_attribute]
  | [processed_attribute]
end redefine

% Defines return statement as original
define original_return
  'return [opt expression] ';      [NL]
end define

% Defines processed return statements
define processed_return
    [original_return]
  | [block]
  | [empty]
end define

% Redefines return statement
redefine return_statement
    [original_return]
  | [processed_return]
end redefine
```

The rules applied to annotated the code based on the modified grammar (i.e., the Java grammar including our modifications) are divided into five groups: action rules, attribute rules, method rules, selection rules and repetition rules.

Action rules describe the rules to convert a user-defined action command into an action annotation. These rules are presented below.

```
% Traces user-defined events
rule trace_user_action
  replace [statement]
    RS [original_statement]
  deconstruct RS
    '# 'action ': A [printable_list] ';

  % Creates annotation
  construct ACTION [stringlit]
    "ACTION:"
  construct SEP [stringlit]
    "#"
  construct INI_MSG [printable_list]
    ACTION '+ A
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    INI_MSG '+ THIS_MSG
  construct FINAL_MSG [printable_list]
    MSG
  construct MSG_CMD [print_statement]
    System.err.println(FINAL_MSG);
  % Marks statement as processed and
  % includes annotation
  construct NS [processed_statement]
    '{
      MSG_CMD
    '}
  by
    NS
end rule
```

Attribute rules are the rules applied to collect attribute information to be used when annotating control flow statements. The rules are shown below.

```
% Collects information about static
% attributes and includes them in
% the list of attributes
rule obtain_static_attribute
  replace [field_declaration]
    RS [original_attribute]
  deconstruct RS
    'static T [type_specifier] N
    [id] D [repeat dimension] ';
  construct NAME [attribute]
    N
  import attrib_list [printable_list]

  construct STR [stringlit]
    ""
  construct ATTR_MSG [stringlit]
    STR [quote NAME]
  construct VALUE_MSG [printable_list]
    NAME
  construct NewAttribute [printable_list]
    ATTR_MSG '+ "=" '+ VALUE_MSG '+ " "

  export attrib_list
    attrib_list '+ NewAttribute
  construct NS [processed_attribute]
    'static T N D ';
  by
    NS
end rule

% Collects information about static
% attributes with modifier and
% includes them in the list of attributes
rule obtain_modifier_static_attribute
  replace [field_declaration]
    RS [original_attribute]
  deconstruct RS
    M [modifier] 'static T [type_specifier]
    N [id] D [repeat
    dimension] ';
  construct NAME [attribute]
    N

  import attrib_list [printable_list]

  construct STR [stringlit]
    ""
  construct ATTR_MSG [stringlit]
    STR [quote NAME]
  construct VALUE_MSG [printable_list]
    NAME
  construct NewAttribute [printable_list]
    ATTR_MSG '+ "=" '+ VALUE_MSG '+ " "

  export attrib_list
    attrib_list '+ NewAttribute

  construct NS [processed_attribute]
    M 'static T N D ';
  by
    NS
end rule

% Ignores constants
rule ignore_constants
  replace [field_declaration]
    RS [original_attribute]
  deconstruct RS
    'static 'final T [type_specifier] N
    [id] D [repeat dimension]
    E [equals_variable_initializer] ';

  construct NS [processed_attribute]
    'static 'final T N D E ';
  by
    NS
end rule

% Ignores constants with reversed order
% of modifiers
rule ignore_constants_2
  replace [field_declaration]
    RS [original_attribute]
  deconstruct RS
    'final 'static T [type_specifier]
    N [id] D [repeat dimension]
    E [equals_variable_initializer] ';
  construct NS [processed_attribute]
    'final 'static T N D E ';
  by
    NS
end rule
```

```
% Collects information about static
% attributes with initialisation
rule obtain_initialised_static_attribute
  replace [field_declaration]
    RS [original_attribute]
  deconstruct RS
    'static T [type_specifier] N [id]
    D [repeat dimension]
    E [equals_variable_initializer] ';
  construct NAME [attribute]
    N
  import attrib_list [printable_list]

  construct STR [stringlit]
    ""
  construct ATTR_MSG [stringlit]
    STR [quote NAME]
  construct VALUE_MSG [printable_list]
    NAME
  construct NewAttribute [printable_list]
    ATTR_MSG '+ "=" '+ VALUE_MSG '+ " "

  export attrib_list
    attrib_list '+ NewAttribute
  construct NS [processed_attribute]
    'static T N D E ';
  by
    NS
end rule

% Collects information about static
% attributes with modifier
% and with initialisation
rule obtain_initialised_modifier_static_attribute
  replace [field_declaration]
    RS [original_attribute]

  deconstruct RS
    M [modifier] 'static T [type_specifier] N
    [id] D [repeat dimension]
    E [equals_variable_initializer] ';
  construct NAME [attribute]
    N
  import attrib_list [printable_list]

  construct STR [stringlit]
    ""
  construct ATTR_MSG [stringlit]
    STR [quote NAME]
  construct VALUE_MSG [printable_list]
    NAME
  construct NewAttribute [printable_list]
    ATTR_MSG '+ "=" '+ VALUE_MSG '+ " "

  export attrib_list
    attrib_list '+ NewAttribute
  construct NS [processed_attribute]
    M 'static T N D E ';
  by
    NS
end rule

% Collects information about non-static
% attributes with modifier and includes
% them in the list of attributes
rule obtain_attribute
  replace [field_declaration]
    RS [original_attribute]
  deconstruct RS
    M [modifier] T [type_specifier] N [id] D [repeat
    dimension] ';
  construct NAME [attribute]
    N
  import attrib_list [printable_list]

  construct STR [stringlit]
    ""
  construct ATTR_MSG [stringlit]
    STR [quote NAME]
  construct VALUE_MSG [printable_list]
    NAME
  construct NewAttribute [printable_list]
    ATTR_MSG '+ "=" '+ VALUE_MSG '+ " "

  export attrib_list
    attrib_list '+ NewAttribute
  construct NS [processed_attribute]
    M T N D ';
  by
    NS
end rule

% Collects information about initialised
% non-static attributes and includes them
% in the list of attributes
rule obtain_initialised_attribute
  replace [field_declaration]
    RS [original_attribute]

  deconstruct RS
    M [modifier] T [type_specifier] N [id]
    D [repeat dimension]
    E [equals_variable_initializer]';
  construct NAME [attribute]
    N
  import attrib_list [printable_list]

  construct STR [stringlit]
    ""
  construct ATTR_MSG [stringlit]
    STR [quote NAME]
  construct VALUE_MSG [printable_list]
    NAME
  construct NewAttribute [printable_list]
    ATTR_MSG '+ "=" '+ VALUE_MSG '+ " "

  export attrib_list
    attrib_list '+ NewAttribute
  construct NS [processed_attribute]
```

```
    M T N D E ';
  by
    NS
end rule

% Obtains user-defined attributes
rule obtain_user_attribute
  replace [field_declaration]
    RS [user_attribute]

  deconstruct RS
    '# 'attribute ': N [printable_list] '=
    E [expression] ';

  import attrib_list [printable_list]

  construct NewAttribute [printable_list]
    N '+ "=" '+ '( E ') '+ " "

  export attrib_list
    attrib_list '+ NewAttribute

  by
    _
end rule
```

Method rules apply modifications to the code to instrument method calls and method bodies. These rules are defined as presented below. Note that the rules only apply to methods containing a single return statement at the end of the method body.

```
% Ignores calls to super
% class constructor
rule ignore_super_call
  replace [statement]
    RS [original_statement]
  deconstruct RS
    ES [expression_statement]
  deconstruct ES
    'super '( _ [list argument] ') ';

  % Marks statement as processed
  construct NS [processed_statement]
    RS
  by
    NS
end rule

% Ignores calls to super class method
rule ignore_super_call2
  replace [statement]
    RS [original_statement]
  deconstruct RS
    ES [expression_statement]
  deconstruct ES
    'super '. _ [id] '( _ [list argument] ') ';

  % Marks statement as processed
  construct NS [processed_statement]
    RS
  by
    NS
end rule

% Ignores calls using 'this'
rule ignore_this_call
  replace [statement]
    RS [original_statement]
  deconstruct RS
    ES [expression_statement]
  deconstruct ES
    'this '. _ [id] '( _ [list argument] ') ';

  % Marks statement as processed
  construct NS [processed_statement]
    RS
  by
    NS
end rule

% Annotates internal method call
rule traced_int_met_call
  replace [statement]
    RS [original_statement]
  deconstruct RS
    ES [expression_statement]
  deconstruct * [reference] ES
    MET [id] '( A [list argument] ')

  % Creates annotations
  construct STR1 [stringlit]
    "CALL_ENTER:"
  construct STR2 [stringlit]
    "CALL_END:"

  construct SEP [stringlit]
    "#"

  construct INI_MSG [printable_list]
    STR1 [quote MET]
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct COMP_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
```

```
    INI_MSG '+ THIS_MSG '+ COMP_MSG

  construct INI_MSG2 [printable_list]
    STR2 [quote MET]
  construct MSG2 [printable_list]
    INI_MSG2 '+ THIS_MSG '+ COMP_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  % Updates ID information
  export counter
    counter [+ 1]

  construct FINAL_MSG [printable_list]
    MSG '+ ATTR_MSG '+ ID_MSG
  construct FINAL_MSG2 [printable_list]
    MSG2 '+ ID_MSG

  construct MSG_CMD [print_statement]
    System.err.println(FINAL_MSG);

  construct MSG_CMD2 [print_statement]
    System.err.println(FINAL_MSG2);

  % Marks statement as processed and
  % includes annotations
  construct NM [processed_statement]
    RS
  construct NS [processed_statement]
    '{
      MSG_CMD
      NM
      MSG_CMD2
    '}
  by
    NS [traced_int_met_call]
end rule

% EXTERNAL METHOD CALLS
% Annotates external method call
rule traced_ext_met_call
  replace [statement]
    RS [original_statement]
  deconstruct RS
    ES [expression_statement]
  deconstruct * [reference] ES
    COMP [id] '. MET [id] '(
    A [list argument] ')

% Creates annotations
construct STR1 [stringlit]
  "CALL_ENTER:"
construct STR2 [stringlit]
  "CALL_END:"

construct SEP [stringlit]
  "#"

construct EMPTY_STR [stringlit]
   ""
construct COMP_NAME [stringlit]
  EMPTY_STR [quote COMP]
where not
  COMP_NAME [= "Integer"]
where not
  COMP_NAME [= "System"]
where not
  COMP_NAME [= "Float"]
where not
  COMP_NAME [= "Double"]
where not
  COMP_NAME [= "Character"]
where not
  COMP_NAME [= "Thread"]
where not
  COMP_NAME [= "Math"]
where not
  COMP_NAME [= "InetAddress"]

construct INI_MSG [printable_list]
  STR1 [quote MET]
construct THIS_MSG [printable_list]
  SEP '+ this
construct COMP_MSG [printable_list]
  SEP '+ COMP
construct MSG [printable_list]
  INI_MSG '+ THIS_MSG '+ COMP_MSG

construct INI_MSG2 [printable_list]
  STR2 [quote MET]
construct MSG2 [printable_list]
  INI_MSG2 '+ THIS_MSG '+ COMP_MSG

% Includes information about values
% of attributes
import attrib_list [printable_list]
construct ATTR_MSG [printable_list]
  SEP '+ "{" '+ attrib_list '+ "}"

% Includes ID information
import counter [number]
construct empty_str [stringlit]
  ""
construct ID [stringlit]
  empty_str [quote counter]
construct ID_MSG [printable_list]
  SEP '+ ID
% Updates ID information
```

```
  export counter
    counter [+ 1]

  construct FINAL_MSG [printable_list]
    MSG '+ ATTR_MSG '+ ID_MSG
  construct FINAL_MSG2 [printable_list]
    MSG2 '+ ID_MSG

  construct MSG_CMD [print_statement]
    System.err.println(FINAL_MSG);

  construct MSG_CMD2 [print_statement]
    System.err.println(FINAL_MSG2);

  % Creates annotation of method actions
  construct ACTION [stringlit]
    "ACTION:"
  construct ACT [printable_list]
    ACTION [quote MET]
  construct MSG_ACT [printable_list]
    ACT '+ THIS_MSG
  construct MSG_ACT_PRINT [print_statement]
    System.err.println(MSG_ACT);

  % Marks statement as processed and
  % includes annotations
  construct NM [processed_statement]
    RS
  construct NS [processed_statement]
    '{
      MSG_CMD
      MSG_ACT_PRINT
      NM
      MSG_CMD2
      MSG_ACT_PRINT
    '}
  by
    NS [traced_ext_met_call]
end rule

% Annotates external method call inside
% a variable declaration
rule traced_ext_met_call2
  replace [declaration_or_statement]
    RS [original_declaration]
  deconstruct RS
    LV [local_variable_declaration]
  deconstruct * [reference] LV
    COMP [id] '. MET [id] '(
    A [list  argument] ')

  % Creates annotations
  construct STR1 [stringlit]
    "CALL_ENTER:"
  construct STR2 [stringlit]
    "CALL_END:"

  construct SEP [stringlit]
    "#"

construct EMPTY_STR [stringlit]
    ""
construct COMP_NAME [stringlit]
  EMPTY_STR [quote COMP]
where not
  COMP_NAME [= "Integer"]
where not
  COMP_NAME [= "System"]
where not
  COMP_NAME [= "Float"]
where not
  COMP_NAME [= "Double"]
where not
  COMP_NAME [= "Character"]
where not
  COMP_NAME [= "Thread"]
where not
  COMP_NAME [= "Math"]
where not
  COMP_NAME [= "InetAddress"]

construct INI_MSG [printable_list]
  STR1 [quote MET]
construct THIS_MSG [printable_list]
  SEP '+ this
construct COMP_MSG [printable_list]
  SEP '+ COMP
construct MSG [printable_list]
  INI_MSG '+ THIS_MSG '+ COMP_MSG

construct INI_MSG2 [printable_list]
  STR2 [quote MET]
construct MSG2 [printable_list]
  INI_MSG2 '+ THIS_MSG '+ COMP_MSG

% Includes information about values
% of attributes
import attrib_list [printable_list]
construct ATTR_MSG [printable_list]
  SEP '+ "{" '+ attrib_list '+ "}"

% Includes ID information
import counter [number]
construct empty_str [stringlit]
  ""
construct ID [stringlit]
  empty_str [quote counter]
construct ID_MSG [printable_list]
  SEP '+ ID
% Updates ID information
export counter
  counter [+ 1]

construct FINAL_MSG [printable_list]
  MSG '+ ATTR_MSG '+ ID_MSG
construct FINAL_MSG2 [printable_list]
  MSG2 '+ ID_MSG

construct MSG_CMD [print_statement]
  System.err.println(FINAL_MSG);
  construct MSG_CMD2 [print_statement]
    System.err.println( FINAL_MSG2);

  % Creates annotation of method actions
  construct ACTION [stringlit]
    "ACTION:"
  construct ACT [printable_list]
    ACTION [quote MET]
  construct MSG_ACT [printable_list]
    ACT '+ THIS_MSG
  construct MSG_ACT_PRINT [print_statement]
    System.err.println(MSG_ACT);

  % Marks statement as processed and
  % includes annotations
  construct NM [processed_declaration]
    RS
  construct NS [processed_statement]
    '{
      MSG_CMD
      MSG_ACT_PRINT
      NM
      MSG_CMD2
      MSG_ACT_PRINT
    '}
  by
    NS
end rule

% Annotates external method calls in
% variable assignments
rule traced_variable_assignment_ext
  replace [statement]
    RS [original_statement]
  deconstruct RS
    VAR [id] '= COMP [id] '. MET [id] '(
    A [list argument] ') ';

  % Creates annotations
    construct STR1 [stringlit]
      "CALL_ENTER:"
    construct STR2 [stringlit]
      "CALL_END:"
    construct SEP [stringlit]
      "#"

    construct EMPTY_STR [stringlit]
      ""
    construct COMP_NAME [stringlit]
      EMPTY_STR [quote COMP]
    where not
      COMP_NAME [= "Integer"]
    where not
      COMP_NAME [= "System"]
    where not
      COMP_NAME [= "Float"]
    where not
      COMP_NAME [= "Double"]
    where not
      COMP_NAME [= "Character"]
    where not
      COMP_NAME [= "Thread"]
    where not
      COMP_NAME [= "Math"]
    where not
      COMP_NAME [= "InetAddress"]


    construct INI_MSG [printable_list]
      STR1 [quote MET]
    construct THIS_MSG [printable_list]
      SEP '+ this
    construct COMP_MSG [printable_list]
      SEP '+ COMP
    construct MSG [printable_list]
      INI_MSG '+ THIS_MSG '+ COMP_MSG

    construct INI_MSG2 [printable_list]
      STR2 [quote MET]
    construct MSG2 [printable_list]
      INI_MSG2 '+ THIS_MSG '+ COMP_MSG

    % Includes information about values
    % of attributes
    import attrib_list [printable_list]
    construct ATTR_MSG [printable_list]
      SEP '+ "{" '+ attrib_list '+ "}"

    % Includes ID information
    import counter [number]
    construct empty_str [stringlit]
      ""
    construct ID [stringlit]
      empty_str [quote counter]
    construct ID_MSG [printable_list]
      SEP '+ ID
    % Updates ID information
    export counter
      counter [+ 1]

    construct FINAL_MSG [printable_list]
      MSG '+ ATTR_MSG '+ ID_MSG
    construct FINAL_MSG2 [printable_list]
      MSG2 '+ ID_MSG

    construct MSG_CMD [print_statement]
      'System '. 'err '. 'println '(
      FINAL_MSG ') ';

    construct MSG_CMD2 [print_statement]
      'System '. 'err '. 'println '(
      FINAL_MSG2 ') ';

    % Marks statement as processed and
    % includes annotations
    construct NM [processed_statement]
      RS
    construct NS [processed_statement]
      '{
        MSG_CMD
        NM
        MSG_CMD2
      '}
    by
    NS
end rule

% TRACE INSIDE METHODS
% Annotates method bodies
rule traced_method
  replace [method_declaration]
    RS [original_method_declaration]
  deconstruct RS
    M [repeat modifier] T [type_specifier]
    Decl [method_declarator]
    E [opt throws] Body [method_body]

  deconstruct Decl
    MET [method_name] '(
    P [list formal_parameter] ')

  % Avoids annotating main method
  construct EMPTY_STR [stringlit]
    ""
  construct MET_NAME [stringlit]
    EMPTY_STR [quote MET]
  where not
    MET_NAME [= "main"]
  export MET

  construct TYPE_NAME [stringlit]
    EMPTY_STR [quote T]
  where
    TYPE_NAME [= "void"]

  construct ND [processed_method_declaration]
    RS
  by
    ND [methodAnnotation]
end rule

% Includes messages inside method bodies
rule methodAnnotation
  replace [method_body]
    RS [original_method_body]

  import MET [method_name]

  % Creates annotation
  construct STR1 [stringlit]
    "MET_ENTER:"
  construct STR2 [stringlit]
    "MET_END:"
  construct SEP [stringlit]
    "#"
  construct THIS_MSG [printable_list]
    SEP '+ this

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  % Updates ID information
  export counter
    counter [+ 1]

  construct MSG [printable_list]
    STR1 [quote MET]
  construct MSG2 [printable_list]
    MSG '+ THIS_MSG '+ ATTR_MSG '+ ID_MSG
  construct ENTER_MSG [print_statement]
    System.err.println(MSG2);

  construct MSG3 [printable_list]
    STR2 [quote MET]
  construct MSG4 [printable_list]
    MSG3 '+ THIS_MSG '+ ID_MSG
  construct EXIT_MSG [print_statement]
    System.err.println(MSG4);

  % Creates annotation of method actions
  construct ACTION [stringlit]
    "ACTION:"
  construct ACT [printable_list]
    ACTION [quote MET]
  construct MSG_ACT [printable_list]
    ACT '+ THIS_MSG
  construct MSG_ACT_PRINT [print_statement]
    System.err.println(MSG_ACT);

  deconstruct RS
    '{ D [repeat declaration_or_statement] '}

  % Includes annotation in method body
  construct NewBody [processed_method_body]
    '{
      ENTER_MSG
      MSG_ACT_PRINT
      '{
        D
      '}
      EXIT_MSG
      MSG_ACT_PRINT
    '}
  by
    NewBody
end rule

% Annotates methods with return statement
```

```
rule traced_method_with_return
  replace [method_declaration]
    RS [original_method_declaration]
  deconstruct RS
    M [repeat modifier] T [type_specifier]
     Decl [method_declarator]
    E [opt throws] Body [method_body]

  deconstruct Decl
    MET [method_name] '(
    P [list formal_parameter] ')

  % Avoids annotating main method
  construct EMPTY_STR [stringlit]
    ""
  construct MET_NAME [stringlit]
    EMPTY_STR [quote MET]
  where not
    MET_NAME [= "main"]
  export MET

  construct TYPE_NAME [stringlit]
    EMPTY_STR [quote T]
  where not
    TYPE_NAME [= "void"]

  construct ND [processed_method_declaration]
    RS
  by
    ND [methodAnnotation2] [traced_return]
end rule

% Includes messages in method bodies with
% return statement
rule methodAnnotation2
  replace [method_body]
    RS [original_method_body]

  import MET [method_name]

  % Creates annotation
  construct STR1 [stringlit]
    "MET_ENTER:"
  construct SEP [stringlit]
    "#"
  construct THIS_MSG [printable_list]
    SEP '+ this

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID

  construct MSG [printable_list]
    STR1 [quote MET]
  construct MSG2 [printable_list]
    MSG '+ THIS_MSG '+ ATTR_MSG '+ ID_MSG
  construct ENTER_MSG [print_statement]
    System.err.println(MSG2);

  % Creates annotation of method actions
  construct ACTION [stringlit]
    "ACTION:"
  construct ACT [printable_list]
    ACTION [quote MET]
  construct MSG_ACT [printable_list]
    ACT '+ THIS_MSG
  construct MSG_ACT_PRINT [print_statement]
    System.err.println(MSG_ACT);

  deconstruct RS
    '{ D [repeat declaration_or_statement] '}

  % Includes annotation in method body
  construct NewBody [processed_method_body]
    '{
      ENTER_MSG
      MSG_ACT_PRINT
      '{
        D
      '}
    '}
  by
    NewBody
end rule

% Annotates return statement inside a
% method body
rule traced_return
  replace [return_statement]
    RS [original_return]
  deconstruct RS
    'return RV [expression] ';

  import MET [method_name]

  construct STR [stringlit]
    "MET_END:"
  construct SEP [stringlit]
    "#"
  construct THIS_MSG [printable_list]
    SEP '+ this

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
```

```
    SEP '+ ID
  % Updates ID information
  export counter
    counter [+ 1]

  construct MSG [printable_list]
    STR [quote MET]
  construct MSG2 [printable_list]
    MSG '+ THIS_MSG '+ ID_MSG
  construct EXIT_MSG [print_statement]
    System.err.println(MSG2);

  % Creates annotation of method actions
  construct ACTION [stringlit]
    "ACTION:"
  construct ACT [printable_list]
    ACTION [quote MET]
  construct MSG_ACT [printable_list]
    ACT '+ THIS_MSG
  construct MSG_ACT_PRINT [print_statement]
    System.err.println(MSG_ACT);

  construct PR [processed_return]
    RS

  construct NR [processed_return]
    '{
      EXIT_MSG
      MSG_ACT_PRINT
      PR
    '}
  by
    NR
end rule

% Annotates the main method
rule traced_main_method
  replace [method_declaration]
    RS [original_method_declaration]
  deconstruct RS
    M [repeat modifier] T [type_specifier]
    Decl [method_declarator]
    E [opt throws] Body [method_body]

  deconstruct Decl
    MET [method_name] '(
    P [list formal_parameter] ')

  % Checks whether it is the main method
  construct EMPTY_STR [stringlit]
    ""
  construct MET_NAME [stringlit]
    EMPTY_STR [quote MET]
  where
    MET_NAME [= "main"]

  export MET

  construct ND [processed_method_declaration]
    RS
  by
    ND [methodMainAnnotation]
end rule

% Includes commands inside main method
rule methodMainAnnotation
  replace [method_body]
    RS [original_method_body]

  import MET [method_name]

  % Creates annotation
  construct STR [stringlit]
    "END"

  construct MSG_END_PRINT
  [processed_method_body]
    { System.err.println(STR); }

  construct TM
  [processed_method_declaration]
    public void run () MSG_END_PRINT

  construct TD [processed_declaration]
    Thread myShutdownThread =
    new Thread ( ) { TM };

  construct RM [processed_statement]
    Runtime.getRuntime().addShutdownHook
    (myShutdownThread);

  deconstruct RS
    '{ D [repeat declaration_or_statement] '}

  % Includes annotation in method body
  construct NewBody [processed_method_body]
    '{
      '{
        D
      '}
      TD
      RM
    '}
  by
    NewBody
end rule
```

Selection rules are related to the annotation of selection statements, i.e., if-statements and switch-statements. The rules are presented below.

```
% Includes trace information
% in if commands
rule traced_if
  replace [statement]
    RS [original_statement]
  deconstruct RS
    'if '( E [expression] ') S [statement]
    C [opt else_clause]

  % Creates annotations
  construct SEL [stringlit]
    "SEL_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct AUX [stringlit]
    LPAR [quote E] [+ RPAR]
  construct STR [printable_list]
    SEL '+ AUX '+ SEP
  construct EXP [expression]
    '( E ')
  construct INI_MSG [printable_list]
    STR '+ EXP
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    INI_MSG '+ THIS_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  % Updates ID information
  export counter
    counter [+ 1]

  construct END_SEL [stringlit]
    "SEL_END:"
  construct STR2 [stringlit]
    END_SEL [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    System.err.println(COMP_MSG);
  construct END_MSG_CMD [print_statement]
    System.err.println( END_MSG);

  % Marks statement as processed and
  % includes annotations
  construct NI [processed_statement]
    'if '( E ') S C
  construct NS [processed_statement]
    '{
      MSG_CMD
      NI
      END_MSG_CMD
    '}
  by
    NS
end rule

% Includes trace information in switch
% commands
rule traced_switch
  replace [statement]
    RS [original_statement]
  deconstruct RS
    'switch '( E [expression] ') '{
    A [repeat switch_alternative] '}

  % Exports condition to be used to annotate
  % case and default clauses
  export cond [expression]
    E

  import counter [number]
  export sid [number]
    counter
  % Updates ID information
  export counter
    counter [+ 1]

  % Marks statement as processed
  construct NS [processed_statement]
    'switch '( E ') '{ A '}
  construct NT [processed_statement]
    NS
  by
    NT [delete_breaks] [traced_case]
    [traced_default]
end rule

% Eliminates original break commands in
% the switch
rule delete_breaks
  replace [statement]
    RS [original_statement]
  deconstruct RS
    _ [break_statement]
by
  % empty
end rule
```

```
% Includes trace information in case clauses
rule traced_case
  replace $ [switch_alternative]
    RS [switch_alternative]
  deconstruct RS
    'case C [expression] ':
    D [repeat declaration_or_statement]

  % Imports condition of switch command
  import cond [expression]

  % Creates annotations
  construct SEL [stringlit]
    "SEL_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct EXP_MSG [stringlit]
    _ [quote cond]
  construct AUX [stringlit]
    LPAR [+ EXP_MSG] [+ RPAR]
  construct STR [stringlit]
    SEL [+ AUX] [+ SEP]
  construct EXP [expression]
    cond
  construct INI_MSG [printable_list]
    STR '+ EXP
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    INI_MSG '+ THIS_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import sid [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote sid]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  construct END_SEL [stringlit]
    "SEL_END:"
  construct STR2 [stringlit]
    END_SEL [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    System.err.println(COMP_MSG);
  construct END_MSG_CMD [print_statement]
    System.err.println(END_MSG);

  % Creates new break statement
  construct B [processed_statement]
    'break ';
  % Constructs new case clause with
  % annotations
  construct ND
  [repeat declaration_or_statement]
    MSG_CMD
    '{
      D
    '}
    END_MSG_CMD
    B
  construct NA [switch_alternative]
    'case C ': '{
                    ND
                 '}
  by
    NA
end rule

% Includes trace information in default
% clauses
rule traced_default
  replace $ [switch_alternative]
    RS [switch_alternative]
  deconstruct RS
    'default ':
    D [repeat declaration_or_statement]

  % Imports condition of switch command
  import cond [expression]

  % Creates annotations
  construct SEL [stringlit]
    "SEL_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct EXP_MSG [stringlit]
    _ [quote cond]
  construct AUX [stringlit]
    LPAR [+ EXP_MSG] [+ RPAR]
  construct STR [stringlit]
    SEL [+ AUX] [+ SEP]
  construct EXP [expression]
    cond
  construct INI_MSG [printable_list]
    STR '+ EXP
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    INI_MSG '+ THIS_MSG
  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  % Updates ID information
  export counter
    counter [+ 1]

  construct END_SEL [stringlit]
    "SEL_END:"
  construct STR2 [stringlit]
    END_SEL [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    System.err.println(COMP_MSG);
  construct END_MSG_CMD [print_statement]
    System.err.println(END_MSG);

  export SW_EXP_MSG [print_statement]
    END_MSG_CMD
  % Constructs new default clause with
  % annotations
  construct ND
  [repeat declaration_or_statement]
    MSG_CMD
    '{
      D
    '}
    END_MSG_CMD
  construct NA [switch_alternative]
    'default ': '{
                 ND
               '}
  by
    NA
end rule

% Annotates selection statements in
% ternary assignments
rule traced_ternary_assignment
  replace [statement]
    RS [original_statement]
  deconstruct RS
    VAR [id] '= '( E [conditional_expression]
    ') C [conditional_choice] ';
deconstruct C
  '? T [expression] ':
  F [conditional_expression]

% Creates annotations
  construct SEL [stringlit]
    "SEL_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct AUX [stringlit]
    LPAR [quote E] [+ RPAR]
  construct STR [printable_list]
    SEL '+ AUX '+ SEP
  construct EXP [expression]
    '( E ')
  construct INI_MSG [printable_list]
    STR '+ EXP
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    INI_MSG '+ THIS_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  % Updates ID information
  export counter
    counter [+ 1]

  construct END_SEL [stringlit]
    "SEL_END:"
  construct STR2 [stringlit]
    END_SEL [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    'System '. 'err '. 'println
    '( COMP_MSG ') ';
  construct END_MSG_CMD [print_statement]
    'System '. 'err '. 'println
    '( END_MSG ') ';

  % Marks statement as processed and
  % includes annotations
  construct NT [processed_statement]
    VAR '= '( E ') C ';
  construct NS [processed_statement]
    '{
      MSG_CMD
      NT
      END_MSG_CMD
    '}
  by
    NS
end rule
```

Finally, *repetition rules* annotate repetition statements, i.e., while-statements, do-statements and for-statements. These rules as described below.

```
% Includes trace information in
% while-statements
rule traced_while
  replace [statement]
    RS [original_statement]
  deconstruct RS
    'while '( E [expression] ')
    S [statement]

  % Creates annotations
  construct LOOP [stringlit]
    "REP_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct AUX [stringlit]
    LPAR [quote E] [+ RPAR]
  construct STR [stringlit]
    LOOP [+ AUX]
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    STR '+ THIS_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  % Updates ID information
  export counter
    counter [+ 1]

  construct END_LOOP [stringlit]
    "REP_END:"
  construct STR2 [stringlit]
    END_LOOP [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    System.err.println(COMP_MSG);
  construct END_MSG_CMD [print_statement]
    System.err.println(END_MSG);

  % Marks statement as processed and
  % includes annotations
  construct NB [processed_statement]
    '{
      MSG_CMD
      S
      END_MSG_CMD
    '}
  construct NW [processed_statement]
    'while '( E ') NB

  construct NS [processed_statement]
    '{
      NW
    '}
  by
    NS
end rule

% Includes trace information in do-statements
rule traced_do
  replace [statement]
    RS [original_statement]
  deconstruct RS
    'do S [statement] 'while '(
    E [expression] ') ';

  % Creates annotations
  construct LOOP [stringlit]
    "REP_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct AUX [stringlit]
    LPAR [quote E] [+ RPAR]
  construct STR [stringlit]
    LOOP [+ AUX]
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    STR '+ THIS_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  % Updates ID information
  export counter
    counter [+ 1]

  construct NOT_COMP_MSG [printable_list]
    NOT_MSG '+ ATTR_MSG '+ ID_MSG
  construct END_LOOP [stringlit]
    "REP_END:"
  construct STR2 [stringlit]
    END_LOOP [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    System.err.println(COMP_MSG);
  construct END_MSG_CMD [print_statement]
    System.err.println(END_MSG);

  % Marks statement as processed and
  % includes annotations
  construct NB [processed_statement]
    '{
      MSG_CMD
      S
      END_MSG_CMD
    '}
  construct NO [processed_statement]
    'do NB 'while '( E ') ';
  construct NS [processed_statement]
    '{
      NO
    '}
  by
    NS
end rule

% Includes trace information in
% for-statements
rule traced_for
  replace [statement]
    RS [original_statement]
  deconstruct RS
    'for '( FI [for_init] FE [for_expression]
    FU [for_update] ') S
    [statement]
  deconstruct FE
    E [expression] ';

  % Creates annotations
  construct LOOP [stringlit]
    "REP_ENTER:"
  construct SEP [stringlit]
    "#"
  construct LPAR [stringlit]
    "("
  construct RPAR [stringlit]
    ")"
  construct AUX [stringlit]
    LPAR [quote E] [+ RPAR]
  construct STR [stringlit]
    LOOP [+ AUX]
  construct THIS_MSG [printable_list]
    SEP '+ this
  construct MSG [printable_list]
    STR '+ THIS_MSG

  % Includes information about values
  % of attributes
  import attrib_list [printable_list]
  construct ATTR_MSG [printable_list]
    SEP '+ "{" '+ attrib_list '+ "}"
  construct PART_MSG [printable_list]
    MSG '+ ATTR_MSG

  % Includes ID information
  import counter [number]
  construct empty_str [stringlit]
    ""
  construct ID [stringlit]
    empty_str [quote counter]
  construct ID_MSG [printable_list]
    SEP '+ ID
  construct COMP_MSG [printable_list]
    PART_MSG '+ ID_MSG
  % Updates ID information
  export counter
```

```
    counter [+ 1]

  construct END_LOOP [stringlit]
    "REP_END:"
  construct STR2 [stringlit]
    END_LOOP [+ AUX]
  construct END_MSG [printable_list]
    STR2 '+ THIS_MSG '+ ID_MSG
  construct MSG_CMD [print_statement]
    System.err.println(COMP_MSG);
  construct END_MSG_CMD [print_statement]
    System.err.println(END_MSG);

  % Marks statement as processed and
  % includes annotations
  construct NB [processed_statement]
    '{
      MSG_CMD
      S
      END_MSG_CMD
    '}
  construct NF [processed_statement]
    'for '( FI FE FU ') NB
  construct NS [processed_statement]
    '{
      NF
    '}
  by
    NS
end rule
```

As in any TXL program, a *main function* is defined to determine what rules are applied and in what order. The main function for the rules presented before is shown below.

```
% Escapes quotes inside strings
#pragma -esc "\"

% MAIN FUNCTION
function main
  % List of attributes used in annotations
  export attrib_list [printable_list]
    ""

  % Counter used to created a block ID
  export counter [number]
    0

  replace [program]
    P [program]

  by P
      % Attributes
      [ignore_constants]
      [ignore_constants_2]
      [obtain_static_attribute]
      [obtain_modifier_static_attribute]
      [obtain_initialised_static_attribute]
      [obtain_initialised_modifier_
      static_attribute]
      [obtain_attribute]
      [obtain_initialised_attribute]
      [obtain_user_attribute]

      % Method Calls
      [ignore_super_call]
      [ignore_super_call2]
      [ignore_this_call]
      [traced_int_met_call]
      [traced_ext_met_call]
      [traced_ext_met_call2]

      % User-defined actions
      [trace_user_action]

      % Selection Statements
      [traced_if]
      [traced_switch]

      % Method bodies
      [traced_method_with_return]
      [traced_method]
      [traced_main_method]

      % Repetition Statements
      [traced_while]
      [traced_do]
      [traced_for]

      % Assignment Statements with method calls
      [traced_variable_assignment_ext]
      [traced_ternary_assignment]
end function
```

The order of application of each rule has been defined so that the execution of one rule does not interfere in the execution of the following rules. The execution of this main function causes an original Java source code to be replaced by an instrumented version where statements have been annotated as defined in the applied rules.

Appendix B

Additional Case Studies

We have validated our approach through a number of case studies involving single- and multi-threaded systems. Here we present the results from three selected case studies.

B.1 Traffic Lights Control System

This case study focuses on a simple traffic lights control system, whose code is presented in Figure B.1. The system receives inputs from a controller indicating which lights must be on - `GREEN` or `RED` - and finishes its execution when it receives a signal `END`. The value of attribute `isGreen` is used to indicate the current state of the lights. The change in colour is performed by method `changeColour`.

This system should preserve a property that states that the system alternates between red lights and green lights. This can be expressed using fluents as shown below:

```
fluent GREEN_ON = <greenLights,redLights> initially 0
fluent RED_ON = <redLights,greenLights> initially 1
assert ONE_GREEN = [](GREEN_ON-> X !greenLights)
assert ONE_RED = [](RED_ON -> X !redLights)
assert CORRECT_LIGHTS = [](ONE_GREEN && ONE_RED)
```

```
public class TrafficLights {
  private static final int GREEN = 0;
  private static final int RED = 1;
  private static final int END = 2;
  private boolean isGreen;

  public TrafficLights () {
    isGreen = false;
    Controller c = new Controller ();
    int opt = −1;
    do {
      opt = c.nextSignal ();
      switch (opt) {
        case GREEN:
          if (!isGreen)
            greenLights ();
          break;
        case RED:
          if (isGreen)
            redLights ();
      }
    } while (opt != END);
  }

  private void greenLights () {
    isGreen = true;
    changeColour ("green");
  }

  private void redLights() {
    isGreen = false;
    changeColour ("red");
  }

  private void changeColour
    (String newColour) {
    /* Changes lights colour */
    ...
    System.out.println (newColour);
  }

  public static void main (String args[])
  {
    TrafficLights t =new TrafficLights();
  }
}
```

Figure B.1: Traffic lights control system code.

For this experiment, we used the following parameters:

- Alphabet $\Sigma = \{$`greenLights`,`redLights`,`changeColour`$\}$;
- System state $P = \emptyset$
- Set of traces

  T1= $\langle$`greenLights changeColour redLights changeColour greenLights changeColour redLights changeColour`$\rangle$

Using these parameters, the model shown in Figure B.2 was generated. The model produced by the LTSE tool was made deterministic in the LTSA tool to create this graphical representation.

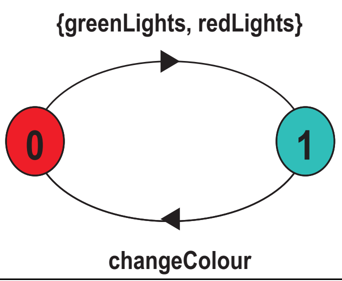


Figure B.2: LTS model of the traffic lights control system.

Checking property `CORRECT_LIGHTS` against this model results in the generation of a violation reported by the LTSA tool:

```
Trace to property violation in CORRECT_LIGHTS:
    greenLights    GREEN_ON
    changeColour   GREEN_ON
    greenLights    GREEN_ON
```

This error trace shows that the model allows the lights to turn to a colour they are already set to. However, inspecting the code, it is possible to see that such a behaviour is not permitted. This restriction is controlled by the value of attribute `isGreen`, which determines the value of the control predicates that allow the access to the methods that change the lights colour. Therefore, the value of this attribute should be taken into account when creating the model.

We applied the refinement process using the updated system state $P = \{\texttt{isGreen}\}$ to build the model this time. The deterministic graphical representation of the refined model is presented in Figure B.3.

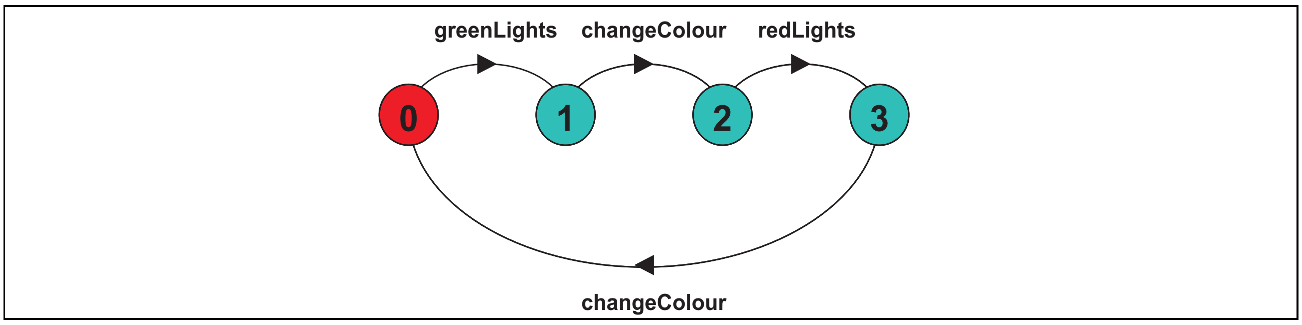


Figure B.3: Refined model of the traffic lights control system.

Note that now the model allows only the alternated execution of actions `greenLights` and `redLights`. A new check of the property confirms that it is not violated by the model.

B.2 Cruise Control System

An automobile cruise control system is controlled by three buttons: `on`, `off` and `resume`. Pressing `on` when the car engine is working causes the system to record the current speed and enables the system that keeps the car at that speed. The same speed is maintained until the car is accelerated, the speed is reduced pressing the brake or `off` is pressed, disabling the control system. If `resume` is then pressed, the system is re-enabled and increases or decreases the speed to set it to the previously recorded speed.

The source code and the models for each component of the system are presented in [MK06]. A property specification and results of checking it against the composed model are also discussed. Therefore, this a good example to validate our approach by analysing the results of using our models in lieu of the manually created models.

We opted for replacing only one of the components of the system, the cruise controller (class `Controller`), which is called from the user interface on events `on`, `off`, `resume`, `accelerate`, `brake`, `engineOn` and `engineOff`. It reacts to these events by enabling or disabling the control system and recording the cruise speed.

For this experiment, we used the following parameters:

- Alphabet $\Sigma = \{$`engineOn,engineOff,accelarator, brake,on,off,resume, enableControl,disableControl,clearSpeed, recordSpeed`$\}$;
- System state $P = \{$`controlState`$\}$
- Set of traces

  T1= ⟨`engineOn clearSpeed engineOff engineOn clearSpeed engineOff`⟩

  T2= ⟨`engineOn clearSpeed accelerator brake accelerator brake accelerator engineOff`⟩

  T3= ⟨`engineOn clearSpeed accelerator on recordSpeed enableControl brake disableControl on recordSpeed enableControl accelerator` `disableControl brake on recordSpeed enableControl off disableControl engineOff`⟩

T4= ⟨`engineOn clearSpeed accelerator on recordSpeed enableControl accelerator disableControl resume enableControl brake disableControl resume enableControl off disableControl accelerator resume enableControl off disableControl engineOff`⟩

T5= ⟨`engineOn clearSpeed on recordSpeed enableControl accelerator disableControl on recordSpeed enableControl brake disableControl engineOff engineOn clearSpeed accelerator on recordSpeed enableControl off disableControl resume enableControl engineOff`⟩

T6= ⟨`engineOn clearSpeed accelerator on recordSpeed enableControl off disableControl accelerator resume enableControl off disableControl brake on recordSpeed enableControl accelerator disableControl on recordSpeed enableControl brake disableControl resume enableControl brake disableControl engineOff`⟩

T7= ⟨`engineOn clearSpeed accelerator on recordSpeed enableControl engineOff engineOn clearSpeed accelerator brake accelerator on recordSpeed enableControl off disableControl resume enableControl off disableControl engineOff`⟩

The test cases used to produce the traces were chosen based on a desired safety property `CRUISESAFETY` presented in [MK06], which states that the `Controller` relinquishes control of the speed as soon as the `brake`, the `accelerator` or the button `off` is pressed.

The logs generated with these parameters were used in the LTSE tool to create the FSP description of the `Controller`. The LTSA tool realised the conversion from the FSP description into its graphical representation. Figure B.4 shows the model that is the deterministic version of the model derived from the FSP description. Note that, in the original model, the $FINAL$ state had been inserted to represent that the system was interrupted. Because this termination was not part of the behaviour of the system, we chose to ignore transitions leading to the $FINAL$ state. This option did not affect the general behaviour of the model.

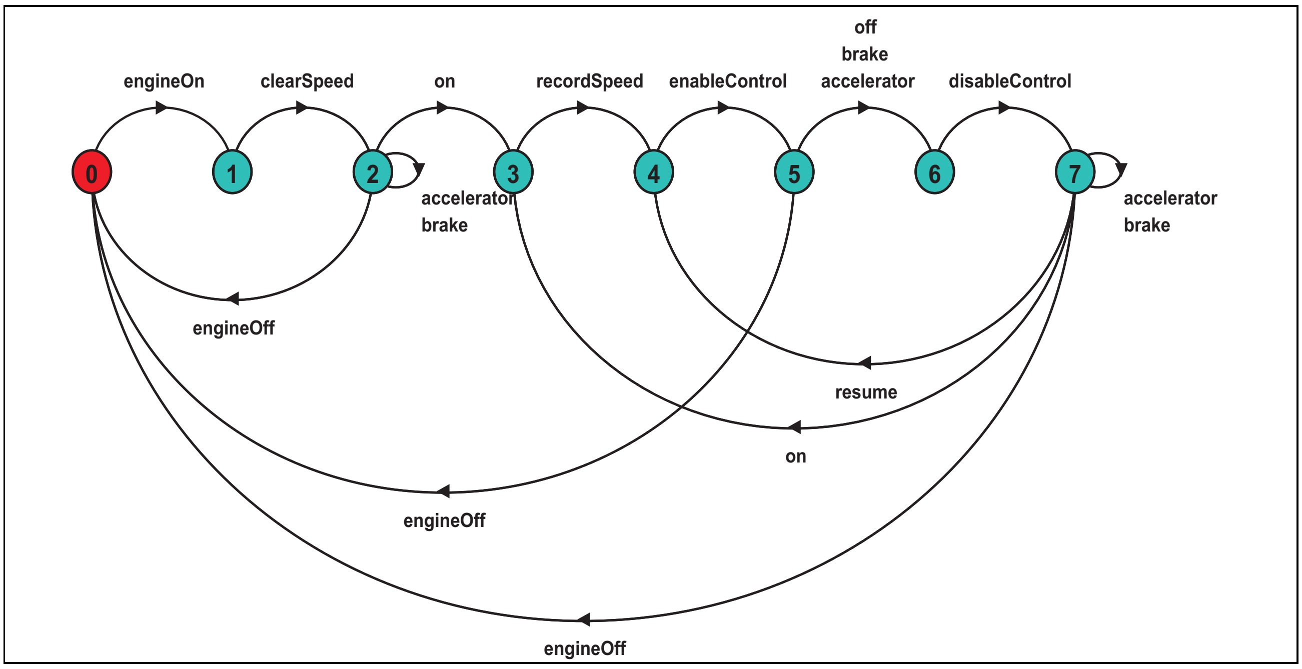


Figure B.4: LTS model of the cruise controller.

This model is very similar to the one presented in [MK06] when actions `clearSpeed`, `recordSpeed`, `enableControl`, `disableControl` are hidden. The only difference is that we did not produce traces where the buttons were pressed but the system ignored these events, such as pressing the button `on` when the engine was off.

For the model checking process, we composed our model of the `Controller` with those of the other components of the system as they were described in [MK06]. Even though there was the mentioned difference between our model and the one proposed in [MK06], we obtained the same results. As expected, the property `CRUISESAFETY` was verified not to be violated when the components of the system were composed. Nevertheless, a progress check provided by the LTSA tool showed the problem described in [MK06], involving the cruise control system not being disabled when the engine was switched off. Hence, when the car engine was turned back on, the car would accelerate automatically to the last recorded speed. The error trace obtained with our model in the composition showed exactly the described problem, as can be seen below:

```
Progress violation for actions:
    {accelerator, brake, engineOff, engineOn, off, on, resume}
Trace to terminal set of states:
    engineOn
    on
    tau
    engineOff
    engineOn
Cycle in terminal set:
    speed
    setThrottle
     zoom
Actions in terminal set:
    {setThrottle, speed, zoom}
```

In this case, the problem was twofold: firstly, the system allowed this dangerous situation to happen; and secondly, the property specification did not include a check of this possible undesired behaviour. To correct this, we applied the necessary corrections to the implementation, to prevent the system from remaining on once the engine was turned off, and to the property specification, to guarantee that this check was now included. These changes resulted in the creation of a model, which, when composed to the other components, generated no violations during the verification process. This model is shown in Figure B.5.

B.3 Dining Philosophers

The dining philosophers problem is a widely used example to demonstrate how a deadlock situation may be difficult to identify. The problem is defined as follows: five philosophers are seated around a table, sharing a plate of spaghetti. Each philosopher alternates moments of thinking and moments of eating. In order to eat the spaghetti, a philosopher needs two forks.

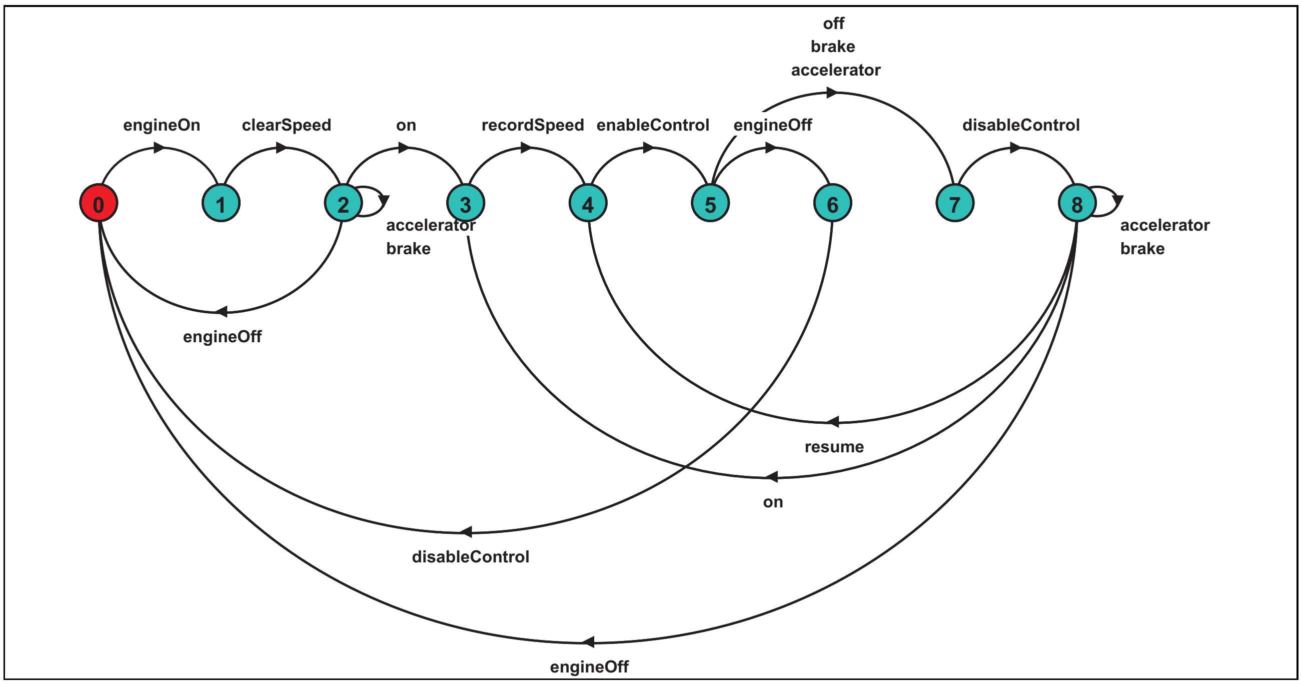


Figure B.5: Fixed model of the cruise controller.

However, only five forks are available for use, each one placed in between two philosophers. Therefore, each philosopher eats using the fork to his immediate right and the fork to his immediate left.

For this example, we would like to check the absence of deadlocks. As stated in the definition above, forks are resources shared by philosophers. Hence, forks are passive entities, whereas philosophers are active entities.

We used the source codes provided in [MK06] to obtain the context information. The code of the `Philosopher` was modified to include some user-defined actions with the purpose of marking events not represented by method calls, such as the philosopher sitting down to eat and rising to think. Besides that, we used user-defined actions to identify which fork - right or left - the actions `get` and `put` referred to. Note that this is not possible using just method calls, as they would only tell us the name of the method called but not the name used in the program as a reference to the instance being activated. The modified code of the `Philosopher` from [MK06] is shown in Figure B.6.

No test cases were selected; rather, we executed the program and monitored the events of

```
class Philosopher extends Thread {
  private int identity;
  private PhilCanvas view;
  private Diners controller;
  private Fork left;
  private Fork right;

  Philosopher(Diners ctr, int id,
              Fork l, Fork r) {
    controller = ctr;
    view = ctr.display;
    identity = id;
    left = l;
    right = r;
  }

  public void run() {
    try {
      while (true) {
        //thinking
        view.setPhil
          (identity,view.THINKING);
        sleep(controller.sleepTime());
        //hungry
        view.setPhil
          (identity,view.HUNGRY);
        #action:"sitdown";
        right.get();
        #action:"right.get";
        //got right chopstick
        view.setPhil
          (identity,view.GOTRIGHT);
        sleep(500);
        left.get();
        #action:"left.get";
        //eating
        #action:"eat";
        view.setPhil
          (identity,view.EATING);
        sleep(controller.eatTime());
        right.put();
        #action:"right.put";
        left.put();
        #action:"left.put";
        #action:"arise";
      }
    }
    catch (InterruptedException e) {}
  }
}
```

Figure B.6: Modified Philosopher code.

interest. Thus, the system was executed and actions performed by each instance of Fork and each instance of Philosopher were recorded in the logs.

We separated the alphabets of either component (i.e., we had one filter file for either component) so that the calls to methods `get` and `put` by the `Philosopher` instances were not included in the `Philosopher` model. Each trace represented the events produced by one instance of its respective component (all traces were the same for component `Fork`). Therefore, in this experiment we used the parameters presented below:

- Alphabets

  $\Sigma_{\texttt{Fork}} = \{\texttt{get,put}\}$

  $\Sigma_{\texttt{Philosopher}} = \{\texttt{left.get, left.put,right.get,right.put,sitdown,eat,arise}\}$

- System state $P = \{\texttt{taken}\}$

- Set of traces of `Fork` instances: all traces are equal to this one

  T= ⟨`get put get put get put get put get put get put get put get put get put get put get put get put get put get put get put get`⟩

- Set of traces of `Philosopher` instances:

  T1= ⟨`sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown`⟩

  T2= ⟨`sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat`⟩

  T3= ⟨`sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get`⟩

  T4= ⟨`sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise`⟩

  T5= ⟨`sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat right.put left.put arise sitdown right.get left.get eat`⟩

It is important to mention that no deadlock situation was observed during the collection of traces. Hence, from the observed executions of instances of `Fork` and `Philosopher`, deadlocks did not occur in the system.

Figure B.7 and Figure B.8 present the models extracted, respectively, for the `Fork` component and for the `Philosopher` component. We made both models deterministic and eliminated transitions to state $FINAL$, as they were produced by us abruptly terminating the execution and did not affect the understanding of the behaviour of the components.

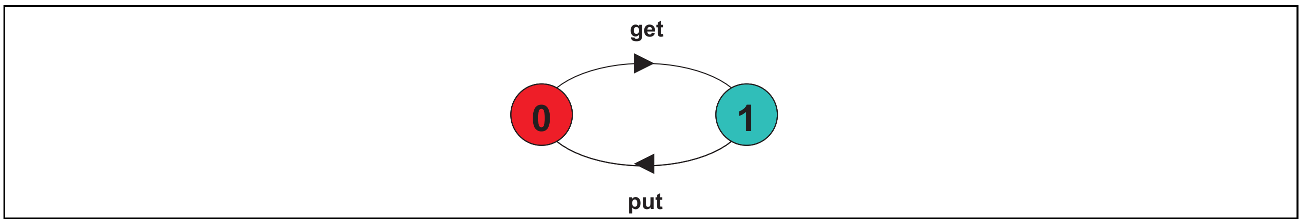


Figure B.7: Model of the Fork component.

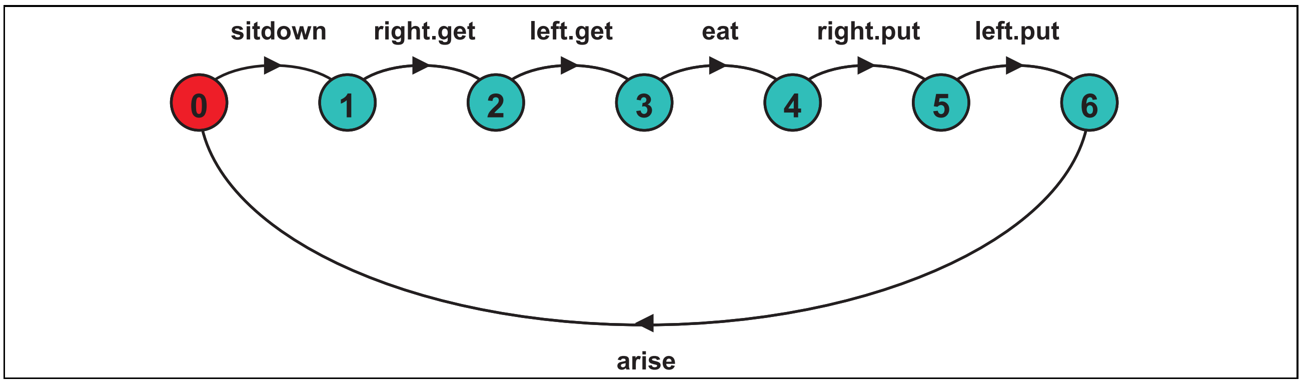


Figure B.8: Model of the Philosopher component.

In order to represent the existence of multiple instances of components and allow interaction between forks and philosophers, we used the composed model specification presented below:

```
||Diners(N=5) = forall [i:0..N-1]
  (phil[i]:Phil || {phil[i].left, phil[((i-1)+N)%N].right}::Fork).
```

The FSP operator `::` represents that the philosophers share the forks, so that actions `get` and `put` of `Fork` instances can synchronise with actions `left.get`, `right.get`, `left.put` and `right.put` of `Philosopher` instances. For a complete description of the FSP operators and semantics, refer to [MK06].

Using the presented models, composed as described above, we checked the composition for potential deadlocks. Using the LTSA tool, we obtained the following error trace:

```
Trace to DEADLOCK:

    phil.0.sitdown

    phil.0.right.get

    phil.1.sitdown

    phil.1.right.get

    phil.2.sitdown

    phil.2.right.get

    phil.3.sitdown

    phil.3.right.get

    phil.4.sitdown

    phil.4.right.get
```

The error trace was the same obtained by the authors using manually created models in [MK06]. It showed that a deadlock situation was possible because all philosophers might simultaneously get the fork to their immediate right and, after that, none of them would be able to get hold of the fork to their left, thus resulting in the blocking of the whole program.

As suggested in [MK06], we used the fixed source code presented in Figure B.9 to solve the problem, where philosophers were divided into two groups depending on their identity being even or odd. Philosophers with even identities take first the fork to their left, whereas philosophers with odd identities take first the fork to their right.

To produce the model of the `FixedPhilosopher` we used the system state $P = \{\texttt{even}\}$ to try to separate the behaviour of each group of philosophers. The resulting (deterministic) model is presented in Figure B.10.

Though the model clearly shows two distinct behaviours, unlike the model described in [MK06], our automatically generated FSP specification does not include parameterised process definitions. Hence, we do not have instances with two distinct behaviours, but instances that can exhibit either behaviour. For this reason, the checking of the composition using this new model still resulted in the detection of a deadlock situation. Not even the use of attribute `identity`

```
class FixedPhilosopher extends Thread {
  private int identity;
  PhilCanvas view;
  Diners controller;
  Fork left;
  Fork right;
  private boolean even = false;

  FixedPhilosopher(Diners controller,
                   int identity,
                   Fork left, Fork right) {
    this.controller = controller;
    this.view = controller.display;
    this.identity = identity;
    if (identity%2 == 0)
      even = true;
    else
      even = false;
    this.left = left;
    this.right = right;
  }

  public void run() {
    while (true) {
      try {
        //thinking
        view.setPhil
          (identity,view.THINKING);
        sleep(controller.sleepTime());
        //hungry
        view.setPhil
          (identity,view.HUNGRY);
        #action:"sitdown";
        //get forks
        if (identity%2 == 0) {
          left.get();
          #action:"left.get";
          view.setPhil
            (identity,view.GOTLEFT);
        }
        else {
          right.get();
          #action:"right.get";
          view.setPhil
            (identity,view.GOTRIGHT);
        }
        sleep(500);
        if (identity%2 == 0) {
          right.get();
          #action:"right.get";
          view.setPhil
            (identity,view.GOTRIGHT);
        }
        else {
          left.get();
          #action:"left.get";
          view.setPhil
            (identity,view.GOTLEFT);
        }
        //eating
        view.setPhil
          (identity,view.EATING);
        #action:"eat";
        sleep(controller.eatTime());
        right.put();
        #action:"right.put";
        left.put();
        #action:"left.put";
        #action:"arise";
      }
      catch (InterruptedException e) {}
    }
  }
}
```

Figure B.9: Fixed Philosopher code.

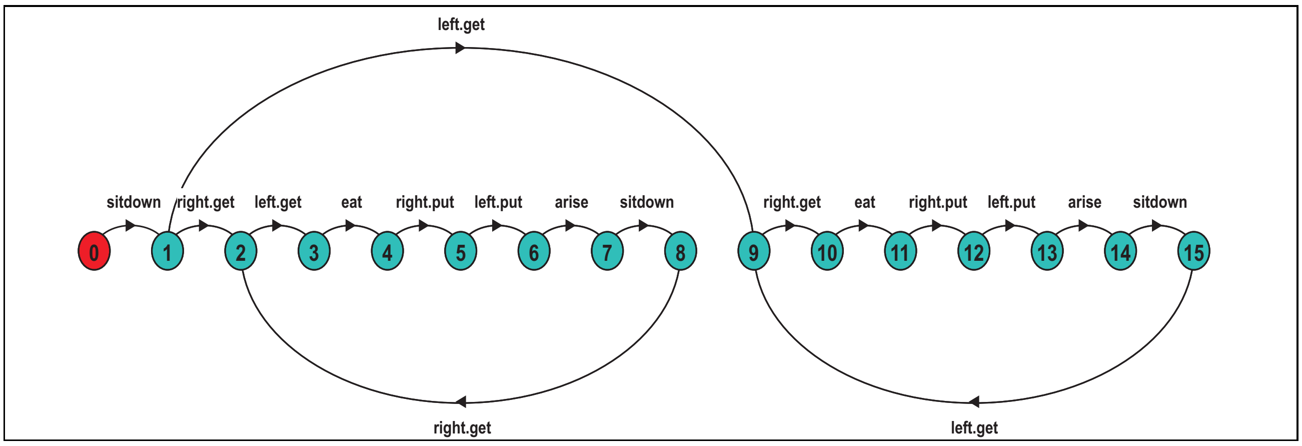


Figure B.10: Model of the `FixedPhilosopher`.

in the system state could make any difference in this case, since it did not help us part the model into two, depending on the identity of the instance.

In a situation like this, the model has to be modified by hand, which is not always simple and might require too many changes in the original model. In this particular case, the change would be simply adding a parameterised choice at the point where the behaviour to be adopted is selected, so that each instance of philosopher would be defined as even or odd at the beginning of the process definition.

Appendix C

Bully Algorithm Source Code

Source code of an election member of the Bully Algorithm case study including the necessary user-defined actions and user-defined attributes.

```java
import java.net.*;
import java.io.*;
import java.util.Vector;

/*******************************************************************
 This class is the implementation of an experiment console that
 initializes a distributed election program and interacts with the user
 to simulate failure and recovery of various election member processes.

 @author Freeman Yufei Huang
 @date Mar.28,2001
 *******************************************************************/
public class ElectionMember extends Thread {
  private InetAddress consoleHost;
  private int consolePort = 3833;
  private int controlPort = 4833;
  private int memberPort = 5833;
  private InetAddress localHost;
  private int priority;
  private DatagramSocket consoleClient;
  private DatagramSocket memberSocket;
  private BufferedReader reader=new BufferedReader(
                                    new InputStreamReader(System.in));
  private static final int COMMAND_LEN = 16;
  private static final int START_MSG_LEN = 512;
  private static final int TIME_OUT = 500;            // in ms
  private static final int MODULA = 20;
  private static final int PRINT_INTERVAL = 3000;     //in ms
  private static final int SLEEP_INTERVAL = 5;        // in ms
  private String status = "down";
  private Member coordinator = new Member ();
  private Vector definition;
  private Vector members = new Vector ();
  private Member halted;
  #attribute:"members"=(members.size ());

  public static void main(String[] args){
    (new ElectionMember()).start();
  }

  public ElectionMember() {
    super("ElectionMember");
    String consoleName = new String ();
    try {
      String localAddress =
        InetAddress.getLocalHost ().getHostAddress ();
      System.out.print("Host name of the console["+localAddress+"]: ");
      consoleName = reader.readLine().trim();
      if (consoleName.equals(""))
        consoleName = localAddress;
      System.out.print("Port number of the console[3833]: ");
      String portChar=reader.readLine().trim();
      if (!portChar.equals(""))
        consolePort = Integer.parseInt(portChar);
      System.out.print("Local port number listening to the
                        console[4833]: ");
      portChar=reader.readLine().trim();
      if (!portChar.equals(""))
        controlPort = Integer.parseInt(portChar);
      System.out.print("Local port number for election[5833]: ");
      portChar=reader.readLine().trim();
      if (!portChar.equals(""))
        memberPort = Integer.parseInt(portChar);
    }
    catch (Exception exception) {
      System.out.println("\nException: user input error.\n");
      System.exit(1);
    }
    try { consoleHost = InetAddress.getByName(consoleName); }
    catch (UnknownHostException exception) {
      System.out.println("\nException: host for console does not
                          exist.\n");
      System.exit(1);
    }
    try {
      localHost = InetAddress.getLocalHost();
      memberSocket = new DatagramSocket(memberPort, localHost);
      consoleClient = new DatagramSocket(controlPort, localHost);
      System.out.println("\nSecret communication with console at "+
                         consoleClient.getLocalAddress().getHostAddress()+":"+
                         consoleClient.getLocalPort()+".");
      System.out.println("Group coordination and election at "+
                         memberSocket.getLocalAddress().getHostAddress()+":"+
                         memberPort+".");
    }
    catch (IOException exception) {
      System.out.println("\nException: local IP or ports not available.\n");
      System.exit(1);
    }
    catch (SecurityException exception) {
      System.out.println("\nException: security violation.\n");
      System.exit(1);
    }
  }

  public void run() {
    byte[] outMsg = ("register"+memberPort).getBytes();
    DatagramPacket outPacket = new DatagramPacket(outMsg, outMsg.length,
                                                  consoleHost, consolePort);
    try { consoleClient.send(outPacket); } catch(IOException exception) {}
    byte[] inMsg = waitForReply();
    if (!(new String(inMsg)).startsWith("registered")) {
      System.out.println("\nException: registration to console timeout.\n");
      System.exit(1);
    }
    System.out.println("Registration to console succeeded.
                          Now wait for start command.\n");
    inMsg = new byte[START_MSG_LEN];
    String command = null;
    DatagramPacket inPacket = new DatagramPacket(inMsg, inMsg.length);
    ByteArrayInputStream fromByte = new ByteArrayInputStream(inMsg);
    ObjectInputStream in = null;
    while (command == null || !command.equalsIgnoreCase("start")) {
      inPacket.setLength(START_MSG_LEN);
      try {
        consoleClient.receive(inPacket);
        in = new ObjectInputStream(fromByte);
        command = (String)in.readObject();
        members = (Vector)in.readObject();
      }
      catch (Exception exception) {
        System.out.println("Exception: error when reading the start command.");
        continue;
      }
    }
    Member member;
    for (int i=0; i<members.size(); i++) {
      member = (Member) members.get(i);
      if (member.equals(localHost.getHostAddress (), memberPort)) {
        priority = member.priority;
        System.out.println ("-> Priority = " + priority);
        break;
      }
    }
    try {
      in.close();
      fromByte.close();
    }
    catch(IOException exception) {}
    System.out.println("Start command with member list received.
                        Now start experiment...");
    Experiment experiment = new Experiment();
    #action:"startExperiment";
    experiment.start();
    while(!status.equalsIgnoreCase("normal") &&
          !status.equalsIgnoreCase("coord")) {
      try { sleep(SLEEP_INTERVAL); } catch(Exception exception) {}
    }
    #action:"statusSet";
    outPacket = new DatagramPacket(status.getBytes(), status.length(),
                                   consoleHost, consolePort);
    try { consoleClient.send(outPacket); } catch(IOException exception) {}
    boolean close = false;
    while (!close) {
      inPacket = new DatagramPacket(new byte[COMMAND_LEN], COMMAND_LEN);
      try { consoleClient.receive(inPacket); } catch(IOException exception) {}
      command = (new String(inPacket.getData())).trim();
      if (command.equalsIgnoreCase("sleep")) {
        if (!status.equalsIgnoreCase("down") && experiment != null) {
          System.out.println("Got command to fail.
                              stoping group coordination...");
          #action:"closeExperiment";
          experiment.close();
          memberSocket.close();
          status = "down";
        }
      }
      else
        if (command.equalsIgnoreCase("wakeup")) {
          if (status.equalsIgnoreCase("down")) {
            System.out.println("Got command to recover.
                                restarting group coordination...");
            try { memberSocket = new DatagramSocket(memberPort, localHost); }
            catch(SocketException exception) {
              System.out.println("\nException: local IP or port not available.\n");
```

```
                System.exit(1);
              }
              experiment = new Experiment();
              #action:"startExperiment";
              experiment.start();
              while(!status.equalsIgnoreCase("normal") &&
                    !status.equalsIgnoreCase("coord")) {
                try { sleep(SLEEP_INTERVAL); } catch(Exception exception) {}
              }
              #action:"statusSet";
            }
          }
          else
            if (command.equalsIgnoreCase("close")) {
              System.out.println("Got command to close.
                                   Shutting down completely...");
              if (experiment != null && experiment.isAlive()) {
                #action:"closeExperiment";
                experiment.close();
              }
              status = "close";
            }
        outPacket = new DatagramPacket(status.getBytes(), status.length(),
                                       consoleHost, consolePort);
        try { consoleClient.send(outPacket); } catch(IOException exception) {}
        if (command.equalsIgnoreCase("close"))
          close = true;
      }
      try {
        #action:"closeExperiment";
        experiment.close();
        reader.close();
        memberSocket.close();
        consoleClient.close();
      }
      catch (IOException exception) {}
    }

    public byte[] waitForReply() {
      byte[] inMsg = new byte[COMMAND_LEN];
      Receiver receiver = new Receiver(inMsg);
      receiver.start();
      long start = System.currentTimeMillis();
      long time = 0;
      while(receiver.isAlive() && time < TIME_OUT) {
        time = System.currentTimeMillis() - start;
      }
      if (receiver.isAlive()) receiver.stop();
        return inMsg;
```

```
  }

  protected class Receiver extends Thread {
    private byte[] inMsg;

    public Receiver(byte[] inMsg) {
      super("Receiver");
      this.inMsg = inMsg;
    }

    public void run() {
      DatagramPacket inPacket = new DatagramPacket(inMsg, inMsg.length);
      try { consoleClient.receive(inPacket); } catch(IOException exception) {}
    }
  }

  private PrintClient printout = null;
  private Printer printer = null;
  private ElectThread election = null;
  private CoordThread coordthread = null;
  private MonitorThread monitor = null;
  private CoordTimeout coordtimeout = null;
  private MessageManager mm = null;

  protected class Experiment extends Thread {
    private boolean stop;

    public Experiment() { super("Experiment"); }

    public void run() {
      #action:"startExperiment";
      monitor = new MonitorThread();
      #action:"startMonitor";
      monitor.start();
      mm = new MessageManager();
      #action:"startMM";
      mm.start();
      status = "down";
      election = new ElectThread();
      #action:"startElection";
      election.start();

      while(!status.equalsIgnoreCase("normal") &&
            !status.equalsIgnoreCase("coord")) {
        try {  sleep(SLEEP_INTERVAL); } catch(Exception exception) {}
      }

      #action:"statusSet";

      printout = new PrintClient();
      printout.start();
    }

    public void close() {
      #action:"closeExperiment";
      printout.close();
      if (election != null) {
        #action:"closeElection";
        election.close();
      }
      if (coordthread != null) {
        #action:"closeCoord";
        coordthread.close();
      }
      #action:"closeMM";
      mm.close();
      #action:"closeMonitor";
      monitor.close();
      if (printer != null) {
        printer.close();
      }
    }
  }

  protected class MessageManager extends Thread {
    private boolean stop;
    #attribute:"members"=(members.size ());

    public MessageManager() {
      super("MessageManager");
      stop = false;
    }

    public void run() {
      #action:"startMM";
      DatagramPacket inPacket;
      String inMsg = null;
      while(!stop) {
        inPacket = new DatagramPacket(new byte[START_MSG_LEN],START_MSG_LEN);
        try { memberSocket.receive(inPacket); }
        catch(IOException exception) { break; }
        String senderName = inPacket.getAddress().getHostAddress ();
        int senderPort = inPacket.getPort();
        inMsg = (new String(inPacket.getData())).trim();
        Member member = null;
        for (int i = 0; i < members.size (); i++) {
          member = (Member) members.get(i);
          if (member.hostName.equals (senderName) &&
              (member.port == senderPort))
            break;
        }
        if(inMsg.equalsIgnoreCase("printed")) {
          while(!printout.msgConsumed()) {
            try { sleep(SLEEP_INTERVAL/4 + 1); }
            catch(Exception exception) {}
          }
          printout.setMsg(inMsg);
        }
        else
          if(inMsg.startsWith("printout") &&
             printer != null && printer.isAlive()) {
            while(!printer.msgConsumed()) {
              try {  sleep(SLEEP_INTERVAL/4 + 1); }
              catch(Exception exception) {}
            }
            printer.setMsg(inPacket);
          }
          else
            if(inMsg.equalsIgnoreCase("IAmNormal") &&
               coordthread!=null && coordthread.isAlive()) {
              #action:"msgIAmNormal["+member.priority+"]["+priority+"]";
              while(!coordthread.msgConsumed()) {
                try {  sleep(SLEEP_INTERVAL/4 + 1); }
                catch(Exception exception) {}
              }
              #action:"receivedMemberIsNormal["+member.priority+"]
                      ["+priority+"]";
              coordthread.setMsg(inMsg);
            }
            else
              if(inMsg.equalsIgnoreCase("IAmUp") &&
                 coordtimeout != null && coordtimeout.isAlive()) {
                while(!coordtimeout.msgConsumed()) {
                  try {  sleep(SLEEP_INTERVAL/4 + 1); }
                  catch(Exception exception) {}
                }
                coordtimeout.setMsg(inMsg);
              }
              else
                if(inMsg.equalsIgnoreCase("IAmUp") &&
                   election != null && election.isAlive()) {
                  #action:"msgIAmUp["+member.priority+"]["+priority+"]";
                  while(!election.msgConsumed()) {
                    try {  sleep(SLEEP_INTERVAL/4 + 1); }
                    catch(Exception exception) {}
                  }
                  #action:"receivedCandidateIsUp["+member.priority+"]
```

```
                          ["+priority+"]";
                    election.setMsg(inPacket);
                  }
                  else
                    if(inMsg.equalsIgnoreCase("AreYouUp") ||
                       inMsg.equalsIgnoreCase("AreYouNormal") ||
                       inMsg.equalsIgnoreCase("EnterElection") ||
                       inMsg.equalsIgnoreCase("SetCoord") ||
                       inMsg.startsWith("NewState")) {

                      if (inMsg.startsWith("NewState"))
                        #action:"msgNewState["+member.priority+"]
                                ["+priority+"]";
                      else
                        #action:"msg"+inMsg+"["+member.priority+"]
                                ["+priority+"]";
                      while(!monitor.msgConsumed()) {
                        try {  sleep(SLEEP_INTERVAL/4 + 1); }
                        catch(Exception exception) {}
                      }
                      if (inMsg.startsWith ("NewState"))
                        #action:"receivedRequestNewState["+member.priority+"]
                                 ["+priority+"]";
                      else
                        #action:"receivedRequest"+inMsg+"["+member.priority+"]
                                ["+priority+"]";
                      monitor.setMsg(inPacket);
                    }
                    else
                      if((inMsg.equalsIgnoreCase("InElection") ||
                          inMsg.equalsIgnoreCase("CoordSet") ||
                          inMsg.equalsIgnoreCase("StateUpdated")) &&
                          election!=null&& election.isAlive()) {
                        #action:"msg"+inMsg+"["+member.priority+"]
                                ["+priority+"]";
                        while(!election.msgConsumed()) {
                          try {  sleep(SLEEP_INTERVAL/4 + 1); }
                          catch(Exception exception) {}
                        }
                        #action:"receivedMember"+inMsg+"["+member.priority+"]
                                ["+priority+"]";
                        election.setMsg(inPacket);
                      }
      }
      #action:"stopMM";
    }

    public void close() { stop = true; }
}
```

```java

  protected class PrintClient extends Thread {
    private String inMsg = null;
    private boolean stop = false;

    public PrintClient() { super("PrintClient"); }

    public void run() {
      int count = 0;
      long startWaiting;
      boolean timeout;
      int counter = 0;

      while (!stop && counter == 0) {
        while((!status.equalsIgnoreCase("normal") &&
               !status.equalsIgnoreCase("coord"))||
               (election != null && election.isAlive()) ||
               (coordtimeout != null && coordtimeout.isAlive())) {
          try { sleep(TIME_OUT/2); } catch(Exception exception) {}
        }
        try { sleep(PRINT_INTERVAL); } catch(Exception exception) {}
        send(coordinator, "printout" + count);
        timeout = true;
        startWaiting = System.currentTimeMillis();
        while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
          if (inMsg != null && inMsg.equals("printed")) {
            inMsg = null;        //consume the incoming message
            timeout = false;
            count = (count + 1) % MODULA;
            break;
          }
        }
        if(!stop && timeout &&  !isCoord() &&
            status.equalsIgnoreCase("normal") &&
            ((coordtimeout == null) || !coordtimeout.isAlive())) {
          coordtimeout = new CoordTimeout();
          coordtimeout.start();
        }
      }
    }

    public void close() { stop = true; }

    public void setMsg(String msg) { inMsg = msg; }

    public boolean msgConsumed() { return (inMsg == null); }
  }

  protected class Printer extends Thread {
    private DatagramPacket inPacket = null;
    private boolean stop = false;

    public Printer() { super("Printer"); }

    public void run() {
      Member member = null;
      String senderName;
      int senderPort;
      int count = 0;

      while (!stop && status.equalsIgnoreCase("coord")) {
        while(!stop && inPacket == null) {
          try { sleep(SLEEP_INTERVAL); } catch(Exception exception) {} }
          if(stop) {
            inPacket = null;
            return;
          }
          senderName = inPacket.getAddress().getHostAddress();
          senderPort = inPacket.getPort();

          for (int i=0; i<members.size(); i++) {
            member = (Member)members.get(i);
            if (member.equals(senderName, senderPort)) break;
          }
          try {
            count = Integer.parseInt(
              (new String(inPacket.getData())).substring(8).trim());
          }
          catch (NumberFormatException exception) {
            inPacket = null;
            continue;
          }
          inPacket = null;
          System.out.println("Here is the printout message for " +
                             senderName + ":" + senderPort + ", #"
                             + count );
          send(member, "printed");
      }
    }

    public void close() { stop = true; }

    public void setMsg(DatagramPacket inPacket) {
      this.inPacket = inPacket;
    }

    public boolean msgConsumed() { return (inPacket == null); }
  }

  protected class ElectThread extends Thread {
    private DatagramPacket inPacket = null;
    private String status;
    private int priority;
    #attribute:"members"=(members.size ());

    public ElectThread() { super("ElectThread"); }

    public void run() {
      #action:"startElection";
      String inMsg = null;
      Member member = null;
      String senderName;
      int senderPort;
      long startWaiting;
      for(int i=0; i<members.size(); i++) {
        member = (Member)members.get(i);

        if(member.priority>=priority) { continue; }

        #action:"sendAreYouUp["+priority+"]["+member.priority+"]";
        send(member, "AreYouUp");
        startWaiting = System.currentTimeMillis();
        while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
          if (inPacket != null) {
            senderName = inPacket.getAddress().getHostAddress();
            senderPort = inPacket.getPort();
            inMsg = (new String(inPacket.getData())).trim();
            inPacket = null;
            if (member.equals(senderName, senderPort) &&
                inMsg.equalsIgnoreCase("IAmUp")) {
              #action:"candidateIsUp["+member.priority+"]
                        ["+priority+"]";
              #action:"stopElection";
              return;
            }
          }
        }
      }

      #action:"noCandidateUp";
      System.out.println("Election ongoing, I am the coord candidate...");
      status = "election";
      for(int i=0; i<members.size(); i++) {
        member = (Member)members.get(i);
        if (member.equals(localHost.getHostAddress(), memberPort))
          halted = member;
        else
          if (member.priority<priority)
            member.status = "down";
          else {
            #action:"sendEnterElection["+priority+"]["+member.priority+"]";
            send(member, "EnterElection");
            System.out.println("Send EE to "+member);
            member.status = "down";
            startWaiting = System.currentTimeMillis();
            while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
              if (inPacket != null) {
                senderName = inPacket.getAddress ().getHostAddress();
                senderPort = inPacket.getPort();
                inMsg = (new String(inPacket.getData())).trim();
                inPacket = null;
                if (member.equals(senderName, senderPort) &&
                    inMsg.equalsIgnoreCase("InElection")) {
                  #action:"memberInElection["+member.priority+"]
                           ["+priority+"]";
                  member.status = "normal";
                  System.out.println("Get EE back");
                  break;
                }
              }
            }
            if (member.status.equals ("down"))
              #action:"timeout["+member.priority+"]";
          }
      }

      coordinator = halted;
      coordinator.status = "coord";
      status = "reorgan";
      boolean timeout = true;
      for(int i=0; i<members.size(); i++) {
        member = (Member)members.get(i);
        if (member.status.equals("normal")) {
          #action:"sendSetCoord["+priority+"]["+member.priority+"]";
          send(member, "SetCoord");
          System.out.println("Send SC to "+member);
          startWaiting = System.currentTimeMillis();
          while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
            if (inPacket != null) {
              senderName = inPacket.getAddress().getHostAddress();
              senderPort = inPacket.getPort();
              inMsg = (new String(inPacket.getData())).trim();
              inPacket = null;
              if (member.equals(senderName, senderPort) &&
                  inMsg.equalsIgnoreCase("CoordSet")) {
                #action:"memberCoordSet["+member.priority+"]
```

```
                        ["+priority+"]";
                    timeout = false;
                    System.out.println("Get SC back");
                    break;
                }
              }
            }
            if (timeout) {
              #action:"timeout["+member.priority+"]";
              #action:"startNewElection";
              election = new ElectThread();
              election.start();
              return;
            }
            timeout = true;
          }
        }

        definition = members;
        ByteArrayOutputStream toByte = new ByteArrayOutputStream();
        try {
          ObjectOutputStream out = new ObjectOutputStream(toByte);
          out.writeObject(definition);
          out.flush();
          out.close();
          toByte.close();
        }
        catch (IOException exception) {
          System.out.println("Exception: error when writing new state.");
          System.exit(1);
        }
        String outMsg = new String ();
        outMsg = "NewState" + (new String(toByte.toByteArray()));
        for(int i=0; i<members.size(); i++) {
          member = (Member)members.get(i);
          if (member.status.equals("normal")) {
            #action:"sendNewState["+priority+"]["+member.priority+"]";
            send(member, outMsg);
            System.out.println("Send NS to "+member);
            startWaiting = System.currentTimeMillis();
            while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
              if (inPacket != null) {
                senderName = inPacket.getAddress().getHostAddress();
                senderPort = inPacket.getPort();
                inMsg = (new String(inPacket.getData())).trim();
                inPacket = null;
                if (member.equals(senderName, senderPort) &&
                    inMsg.equalsIgnoreCase("StateUpdated")) {
                  #action:"memberStateUpdated["+member.priority+"]
```

```
                              ["+priority+"]";
                      timeout = false;
                      System.out.println("Get NS back");
                      break;
                    }
                  }
                }
                if (timeout) {
                  #action:"timeout["+member.priority+"]";
                  #action:"startNewElection";
                  election = new ElectThread();
                  election.start();
                  return;
                }
                timeout = true;
              }
            }
            status = "coord";
            #action:"statusSet";
            #action:"coordStatus";
            if (printer==null || !printer.isAlive()) {
              printer = new Printer();
              printer.start();
            }
            System.out.println("I am elected. accepting printing requests...");
            if (coordthread==null || !coordthread.isAlive()) {
              coordthread = new CoordThread();
              #action:"startCoord";
              coordthread.start();
            }
            #action:"stopElection";
          }

          public void close() { stop(); }

          public void setMsg(DatagramPacket inPacket) {
            this.inPacket = inPacket;
          }

          public boolean msgConsumed() {return (inPacket == null); }
        }

        protected class CoordThread extends Thread {
          private String inMsg;
          private boolean stop = false;

          public CoordThread() { super("CoordThread"); }

          public void run() {
```

```
    #action:"startCoord";
    Member member;
    long startWaiting;
    boolean intime;
    boolean first = true;
    boolean inElection = false;

    while (!stop) {
      while(!status.equalsIgnoreCase("coord")) {
        try { sleep(TIME_OUT); } catch(Exception exception) {}
      }

      if (first || inElection) {
        #action:"coordStatus";
        first = false;
        inElection = false;
      }

      try { sleep(TIME_OUT); } catch(Exception exception) {}

      for (int i=0; i<members.size(); i++) {
        member = (Member) members.get(i);

        if (member.equals(coordinator)) { continue; }

        #action:"sendAreYouNormal["+priority+"]["+member.priority+"]";
        send(member, "AreYouNormal");
        intime = false;
        startWaiting = System.currentTimeMillis();
        while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
          if (inMsg != null && inMsg.equalsIgnoreCase("IAmNormal")) {
            #action:"memberIsNormal["+member.priority+"]["+priority+"]";
            inMsg = null;
            intime = true;
            break;
          }
        }

        if ((member.status.equalsIgnoreCase("normal") && !intime) ||
            (!member.status.equalsIgnoreCase("normal") && intime)) {
          if(!stop && (election == null || !election.isAlive())) {
            election = new ElectThread();
            #action:"startElection";
            election.start();
            inElection = true;
          }
        }
      }
    }
```

```
      #action:"stopCoord";
    }

    public void close() { stop = true; }

    public void setMsg(String msg, int p) { inMsg = msg; }

    public boolean msgConsumed() { return (inMsg == null); }
  }

  protected class MonitorThread extends Thread {
    private DatagramPacket inPacket = null;
    private boolean stop = false;

    public MonitorThread() { super("MonitorThread"); }

    public void run() {
      #action:"startMonitor";
      String inMsg = null;
      Member member = null;
      String senderName;
      int senderPort;
      while (!stop) {
        while(!stop && inPacket == null) {
          try { sleep(SLEEP_INTERVAL); }
          catch(Exception exception) {}
        }

        if(stop) { return; }

        senderName = inPacket.getAddress().getHostAddress ();
        senderPort = inPacket.getPort();
        for (int i=0; i<members.size(); i++) {
          member = (Member)members.get(i);
          if (member.equals(senderName, senderPort)) { break; }
        }
        inMsg = (new String(inPacket.getData())).trim();
        if (inMsg.equalsIgnoreCase("AreYouUp")) {
          #action:"requestAreYouUp["+member.priority+"]["+priority+"]";
          #action:"sendIAmUp["+priority+"]["+member.priority+"]";
          send(member, "IAmUp");
        }
        else
          if (inMsg.equalsIgnoreCase("AreYouNormal")) {
            #action:"requestAreYouNormal["+member.priority+"]
                    ["+priority+"]";
            #action:"sendIAmNormal["+priority+"]["+member.priority+"]";
            send(member, "IAmNormal");
          }
        else
          if (inMsg.equalsIgnoreCase("EnterElection")) {
            status = "election";
            #action:"requestEnterElection["+member.priority+"]
                      ["+priority+"]";
            System.out.println("Election ongoing, "+member+"
                                  is the candidate.");
            if (printer != null) { printer.close(); }
            if (coordthread!=null) {
              #action:"closeCoord";
              coordthread.close();
            }
            if (election != null) {
              #action:"closeElection";
              election.close();
            }
            halted = member;
            #action:"sendInElection["+priority+"]["+member.priority+"]";
            send(member, "InElection");
          }
          else
            if (inMsg.equalsIgnoreCase("SetCoord")) {
              #action:"requestSetCoord["+member.priority+"]
                      ["+priority+"]";
              if (status.equalsIgnoreCase("election") &&
                  halted.equals(member)) {
                coordinator = member;
                status = "reorgan";
              }
              #action:"sendCoordSet["+priority+"]["+member.priority+"]";
              send(member, "CoordSet");
            }
            else
              if (inMsg.startsWith("NewState")) {
                #action:"requestNewState["+member.priority+"]
                        ["+priority+"]";
                if (coordinator.equals(member) &&
                    status.equalsIgnoreCase("reorgan")) {
                  definition =
                    getNewState((new String(
                      inPacket.getData())).substring(8).getBytes());
                  if (definition==null) {
                    System.out.println("Exception: error when
                                        reading new state.");
                  }
                  else {
                    #action:"sendStateUpdated["+priority+"]
                            ["+member.priority+"]";
                    send(member, "StateUpdated");
```

```
                        members = definition;
                        System.out.println(member+" is elected.
                                           Go on printing...");
                        status = "normal";
                        #action:"statusSet";
                        #action:"normalStatus";
                      }
                    }
                  }
          inPacket = null;
        }
        #action:"stopMonitor";
      }

      public void close() { stop = true; }

      public void setMsg(DatagramPacket inPacket) {
        this.inPacket = inPacket;
      }

      public boolean msgConsumed() {  return (inPacket == null); }
    }

    public Vector getNewState(byte[] inMsg) {
      Vector newDef = null;
      ByteArrayInputStream fromByte = new ByteArrayInputStream(inMsg);
      ObjectInputStream in = null;
      try {  in = new ObjectInputStream(fromByte); }
      catch (Exception exception) { return null; }
      try {  newDef = (Vector)in.readObject(); }
      catch (Exception exception) { return null; }
      try {
        in.close();
        fromByte.close();
      }
      catch(IOException exception) {}
      return newDef;
    }

    protected class CoordTimeout extends Thread {
      private String inMsg;

      public CoordTimeout() { super("CoordTimeout"); }

      public void run() {
        send(coordinator, "AreYouUp");
        long startWaiting = System.currentTimeMillis();
        while(System.currentTimeMillis() - startWaiting < TIME_OUT) {
          if (inMsg != null && inMsg.equalsIgnoreCase("IAmUp")) {
```

```
          inMsg = null;       //consume the incoming message
          return;
        }
      }
      if(election == null || !election.isAlive()) {
        election = new ElectThread();
        election.start();
      }
    }

    public void setMsg(String msg) { inMsg = msg; }

    public boolean msgConsumed() { return (inMsg == null); }
  }

  public void send(Member member, String msg) {
    InetAddress destHost = null;
    try {  destHost = InetAddress.getByName(member.hostName); }
    catch(UnknownHostException exception) { return; }
    byte[] outMsg = msg.getBytes();
    DatagramPacket outPacket =
       new DatagramPacket(outMsg, outMsg.length, destHost, member.port);
    try { memberSocket.send(outPacket); }
    catch(IOException exception) {}
  }

  public boolean isCoord() {
    return coordinator.equals(localHost.getHostAddress(), memberPort);
  }
}
```

Bibliography

[ABL02] G. Ammons, R. Bodìk, and J. R. Larus. Mining Specifications. In *ACM Symp. on Principles of Programming Languages*, pages 4–16, Portland, USA, January 2002.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining Approximations in Software Predicate Abstraction. *Lecture Notes in Computer Science*, 2988:388–403, January 2004.

[BDFSV99] R. Barbuti, N. De Francesco, A. Santone, and G. Vaglini. Selective mu-calculus and Formula-Based Equivalence of Transition Systems. *Journal of Computer and System Sciences*, 59:537–556, 1999.

[BPSH05] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic Analysis of Java Applications for Multithreaded Antipatterns. In *Workshop on Dynamic Analysis*, pages 1–7, May 2005.

[BR00] T. Ball and S.K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN*, pages 113–130, 2000.

[BR02] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *ACM Symposium on Principles of Programming Languages*, pages 1–3, Portland, OR, USA, January 2002.

[CC77] P. M. Cousot and R. Cousot. Automatic Synthesis of Optimal Invariant Assertions. In *ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, Rochester, USA, August 1977.

[CCG$^+$04] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[CCO02] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A Finite State Verification Technique for Software Systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[CCO$^+$04] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. *Lecture Notes in Computer Science*, 2999:128–147, April 2004.

[CCO$^+$05] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent Software Verification with States, Events and Deadlocks. *Formal Aspects of Computing Journal*, 17(4):461–483, December 2005.

[CDH$^+$00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society Press.

[CDK05] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 4th edition edition, 2005.

[CDMS02] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology, Special Issue on Source Code Analysis and Manipulation*, 44(13):827–837, October 2002.

[CGJ$^+$03] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, September 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, USA, 1999.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *CAV'96*, volume 1102, pages 394–397, New Brunswick, NJ, USA, July 1996.

[CW96] E.M. Clarke and J.M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[CW98] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[DAC98] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *FMSP '98: Proceedings of the Second Workshop on Formal methods in Software Practice*, pages 7–15, New York, NY, USA, March 1998. ACM, ACM Press.

[Dam03] D. Dams. Comparing Abstraction Refinement Algorithms. *Electronic Notes in Theoretical Computer Science*, 89(3):405–416, July 2003.

[DH99] M. B. Dwyer and J. Hatcliff. Slicing Software for Model Construction. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 105–118, San Antonio, USA, January 1999.

[ECGN01] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[Ern03] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, USA, May 2003.

[ES96] E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1-2):105–131, 1996.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[GL94] O. Grumberg and D.E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[GM82] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[GM03] D. Giannakopoulou and J. Magee. Fluent Model Checking for Event-Based Systems. In *ESEC/FSE*, pages 257–266, Helsinki, Finland, September 2003.

[God03] P. Godefroid. Software Model Checking: The Verisoft Approach. Bell Labs Technical Memorandum ITD-03-44189G, Bell Laboratories, Lucent Technologies, August 2003.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. *Lecture Notes in Computer Science*, 1254:72–83, June 1997.

[GSVV04] S. Gradara, A. Santone, M.L. Villani, and G. Vaglini. Model Checking Multithreaded Programs by Means of Reduced Models. *Electronic Notes in Theoretical Computer Science*, 110:55–74, 2004.

[HD01] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. *Lecture Notes in Computer Science*, 2154:39–58, 2001.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. In *FASE*, pages 80–95, Grenoble, France, April 2002.

[HJMS02] T.A. Henzinger, R. Jahla, R. Majumdar, and G. Sutre. Lazy Abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, OR, USA, January 2002. ACM Press.

[HJMS03] T.A. Henzinger, R. Jahla, R. Majumdar, and G. Sutre. Software Verification with BLAST. *Lecture Notes in Computer Science*, 2648:235–239, 2003.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, N.J, 1985.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Hol01] G.J. Holzmann. From Code to Models. In *ACSD*, pages 3–10, Newcastle upon Tyne, UK, June 2001.

[HP00] K. Havelund and T. Pressburguer. Model Checking Java Programs Using Java PathFinder. *Intl Journal on Software Tools for Technology Transfer*, 2(4):366–381, March 2000.

[HS99] G.J. Holzmann and M.H. Smith. A Practical Method for Verifying Event-Driven Software. In *International Conference on Software Engineering*, pages 597–607, Los Angeles, USA, May 1999.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[HVV03] H. Hansen, H. Virtanen, and A. Valmari. Merging State-Based and Action-Based Verification. *Third International Conference on Application of Concurrency to System Design (ACSD'03)*, pages 150–156, 2003.

[JD96] D. Jackson and C.A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[JD00] D. Jackson and C.A. Damon. Software Analysis: A Roadmap. In *Intl Conf. on Software Engineering*, pages 133–145, Limerick, Ireland, June 2000. ACM Press.

[Kel76] R.M. Keller. Formal Verification of Parallel Programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[KGC04] D. Kroening, A. Groce, and E.M. Clarke. Counterexample Guided Abstraction Refinement via Program Execution. *Lecture Notes in Computer Science*, 3308:224–238, November 2004.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KP89] S. Katz and D. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 489–507, London, UK, 1989.

[LMC01] M. Leuschel, T. Massart, and A. Currie. How to Make FDR Spin: LTL Model Checking of CSP by Refinement. *Lecture Notes in Computer Science*, 2021:99–118, March 2001.

[LMP06] D. Lorenzoli, L. Mariani, and M. Pezze. Inferring State-Based Behavior Models. In *WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 25–32, New York, NY, USA, 2006. ACM Press.

[LNS00] K.R.M. Leino, G. Nelson, and J.B. Saxe. ESC/Java User's Manual. Technical Report 2000-002, Compaq Systems Research Center, Palo Alto, California, October 2000.

[Lud03] J. Ludewig. Models in Software Engineering - An Introduction. *Journal on Software and System Modeling*, 2(1):5–14, February 2003.

[Mar05] L. Mariani. *Behavior Capture and Test: Dynamic Analysis of Component-Based Systems.* PhD, Università degli Studi di Milano Bicocca, 2005.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

[MG96] G. Malcolm and J.A. Goguen. Proving Correctness of Refinement and Implementation. Technical Monography PRG 114, Oxford University, 1996.

[Mil71] R. Milner. An Algebraic Definition of Simulation Between Programs. In British Computer Society, editor, *2nd IJCAI*, pages 481–489, September 1971.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus.* Cambridge University Press, New York, NY, USA, 1999.

[MK06] J. Magee and J. Kramer. *Concurrency: State Models and Java Programming.* Wiley and Sons, 2nd edition, 2006.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[NE02] J.W. Nimmer and M.D. Ernst. Automatic Generation of Program Specifications. In *Intl Symp. on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 2002.

[Par81] D.M.R. Park. Concurrency and Automata on Infinite Sequences. *Lecture Notes in Computer Science*, 104:167–183, March 1981.

[Pat06] R. Patton. *Software Testing*. Sams, 2nd edition, 2006.

[Pel01] D.A. Peled. *Software Reliability Methods*. Texts in Computer Science. Springer-Verlag, 2001.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[RS02] T. Robschink and G. Snelting. Efficient Path Conditions in Dependence Graphs. In *International Conference on Software Engineering*, pages 478–488, Orlando, Florida, USA, April 2002.

[SC96] Sane A. Sefika, M. and R.H. Campbell. Monitoring Compliance of a Software System with Its High-Level Design Models. In *International Conference on Software Engineering*, pages 387–396, Berlin, Germany, March 1996.

[TAC04] J. Tan, G.S. Avrunin, and L.A. Clarke. Heuristic-Based Model Refinement for FLAVERS. In *International Conference on Software Engineering*, pages 635–644, Washington, DC, USA, May 2004.

[Tip95] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[UKM03] S. Uchitel, J. Kramer, and J. Magee. Behaviour Model Elaboration Using Partial Labelled Transition Systems. In *ESEC/FSE*, pages 19–27, Helsinki, Finland, September 2003.

[vG01] R.J. van Glabbeek. *The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes*. Elsevier Science, Amsterdam, The Netherlands, 2001.

[VHB$^+$03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Journal of Automated Software Engineering*, 10(2):203–232, 2003.