Towards Faithful Model Extraction Based on Contexts

Lucio Mauro Duarte*, Jeff Kramer, and Sebastian Uchitel

Department of Computing, Imperial College London
180 Queen's Gate, London, SW7 2AZ, UK
{lmd,jk,su2}@doc.ic.ac.uk

Abstract. Behaviour models facilitate the analysis of software systems using model-checking tools to detect errors and generate counterexamples. Such models can be generated from existing implementations using a model extraction process. This process should guarantee that an extracted model is a faithful representation of the system, so that analysis results may be trusted. This paper discusses the formal foundations of our model extraction process based on contexts. Contexts are abstractions of concrete states of a system, providing valuable information about dependencies between actions. Models are generated by a tool called LTS Extractor and can be refined to improve correctness by augmenting context information. This refinement process eliminates some false negatives and is property-preserving. Completeness of the models depends on the coverage provided by a set of traces describing behaviours of the system. We discuss the faithfulness of our models and results of two case studies.

1 Introduction

Behaviour models are abstract representations of the intended behaviours of systems [22]. They can normally be handled in situations where the real systems could not [18], and have been successfully used to uncover errors that would go undetected otherwise, such as violations of program properties [4].

In this work, we focus on the construction of behaviour models of existing systems. The process of obtaining a model from an implementation is called *model extraction* [15]. An essential requirement of this process is that the generated model should be a *faithful representation* of the system behaviour. Any analysis based on an incorrect model may mislead the developer into an erroneous understanding of how the system behaves [16].

Research has been carried out on techniques for model extraction in recent years (e.g., [6], [15], [7], [1] and [2]) and the results have been encouraging. Nevertheless, the extensive use of model extraction, and, therefore, of model checking [5] for existing systems, has been slowed down by the *model construction problem* [7]. It corresponds to finding a way of bridging the gap between the semantics of current programming languages and that of the less expressive

* Supported by CAPES (Brazil) under the grant BEX 1680-02/1.

languages used as inputs in model-checking tools. This is necessary to meet the requirement of model faithfulness.

In [8], an approach for model extraction based on the use of contexts was presented. A *context* represents an abstraction of a state of the system, composed of the identification of a block of code and the values of a set of attributes. Contexts allow the detection of relations between actions of a system and the inference of additional feasible behaviours from samples of execution [8]. Models based on contexts can be built using the LTS Extractor (LTSE), which implements most of the extraction process. Their use for verifying temporal properties of concurrent systems in the LTSA tool [19] has proved that they are good approximations of the behaviours of the systems they describe.

The aim of this paper is to discuss the formal foundations of the approach for model extraction based on contexts, focusing on factors that determine the faithfulness of models. *Completeness* of the models is shown to depend on the coverage provided by a set of traces obtained during the model extraction process, whereas *correctness* is affected by the selection of attributes used in the identification of contexts. We discuss how completeness and correctness can be improved and how to interpret property-checking results according to these characteristics of a behaviour model. We also present a description of our refinement process, which can eventually lead to a correct abstraction of the system behaviour by the addition of more attributes to contexts. This refinement process is demonstrated to be property-preserving.

This paper is organised as follows. The next section discusses basic concepts involved in our work and Sec. 3 presents our approach for model extraction using context information. In Sec. 4 we discuss the faithfulness of the extracted models and Sec. 5 describes results of two case studies. Finally, Sec. 6 compares our work to similar approaches and Sec. 7 presents the conclusions and future work.

2 Background

We build models where behaviours are described in terms of sequences of actions a system can execute. An *action* is an atomic event of the system that causes an indivisible change on the program state [19]. In this work, an action usually represents the execution of a method, but user-defined actions are also accepted, according to the format described in [9]. More specifically, we model behaviours using Labelled Transition Systems.

Definition 1 Labelled Transition System (LTS). A labelled transition system $M = (S, s_i, \Sigma, T)$ is a model where

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

In a LTS, transitions are labelled with the name 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.

The following definitions apply to LTS models following Definition 1. A *behaviour* is a finite sequence of actions $\pi = \langle a_1 \dots a_n \rangle$ such that $a_1, \dots, a_n \in \Sigma$. The set $L(M) = \{\pi_1, \pi_2, \dots\}$ of all behaviours of M is called its *language*.

For a state $s \in S$, $E(s) = \{a \in \Sigma \mid \exists s' \in S \cdot (s, a, s') \in T\}$ represents the non-empty finite set of actions *enabled* in s . A *path* $\lambda = \langle s_1, a_1, s_2, a_2, s_3, \dots \rangle$ is a sequence of alternating states $s_1, s_2, s_3, \dots \in S$ and actions $a_1, a_2, \dots \in \Sigma$ labelling transitions connecting these states, such that, for $i \geq 1$, for every transition $t_i = (s_i, a, s_{i+1})$ composing λ , $t_i \in T$. A path always starts and ends with a state. We use $\Lambda(M)$ to denote the set of all paths of M .

An LTS 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. This means that, ideally, $L(M) = L(Prog)$, where $L(Prog)$ represents the language of $Prog$. However, the faithfulness of a model is affected by its completeness and correctness.

Definition 2 Completeness. M is complete w.r.t. $Prog$ iff $L(Prog) \subseteq L(M)$.

Definition 3 Correctness. M is correct w.r.t. $Prog$ iff $L(M) \subseteq L(Prog)$.

Completeness, therefore, is related to language containment, whereas correctness is related to the absence of invalid behaviours. If the model is not complete, then *false positives* may occur, i.e., properties might be checked to hold in the model even though they are violated by the system. If the model is not correct, then the model might violate a property not violated by the system, representing a *false negative*.

3 Contexts

As described in [9], we build models combining a *control component*, which indicates the execution point where the system is at, and a *data component*, representing the state of the system in terms of values of program variables. The control component is obtained based on the control flow graph of the implementation of the system.

Definition 4 Control Flow Graph (CFG). 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),
- $q_i = (bc_i, \text{true}) \in Q$, 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.

As for the data component, we adopt the values of attributes (system state). 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 $v = \{val(p_1), \dots, val(p_n)\}$ represents one possible combination of values of attributes $p_1, \dots, p_n \in P_{Prog}$. The set $V(P_{Prog}) = \{v_1, \dots, v_n\}$ is composed of all possible system states of $Prog$, such that $v_1 = \emptyset$, representing the beginning of the execution, when values of attributes are yet unknown. The finite set $V(P) \subseteq V(P_{Prog})$ represents all possible combinations of values of attributes $p_1, \dots, p_n \in P$, where $P \subseteq P_{Prog}$.

We define a context as the conjunction of a control component with a data component, thus representing an abstract state of the actual system.

Definition 5 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 $P \subseteq P_{Prog}$.*

The information collected to identify contexts is denominated *context information*. Using this information, our model extraction approach builds LTS models from Java source code. This is done in the steps discussed next, which, except for the first one, are all implemented by the LTSE¹.

3.1 Information Gathering

We obtain context information from the system through the instrumentation of the source code. Annotations are included for each block of code (selection and repetition statements, method calls and method bodies) and action (methods). The result of executing the instrumented code is the creation of a set of traces, which are recorded in log files.

For a given program $Prog$, a recorded trace t , produced by the execution of the instrumented $Prog$, is a finite sequence $\langle CA_1, AA_1, CA_2, \dots, AA_n, CA_n \rangle$, where $CA_1, CA_2, \dots, CA_n$ are *context annotations*, which contain context information, and $AA_1, \dots, AA_n$ are *action annotations* describing the actions that happened between two consecutive contexts. We use $Tr(Prog)$ to denote the set of all traces that $Prog$ 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 files in F_{Prog} , such that $Tr(F_{Prog}) \subseteq Tr(Prog)$.

As an example, we use the code of the buffer component presented in [13], which is part of a producer-consumer system and has a storage capacity of two elements. An action `halt` is used by the producer to signal that it has finished its operations on the buffer. An exception is generated whenever the consumer attempts to get a new element from the buffer but it is empty and the producer has stopped.

We created test cases considering the number of operations each of these components carries out on the buffer. Below, we show the traces obtained after instrumenting the code of the buffer as described in [8]. Due to lack of space, we

¹ Available at <http://www.doc.ic.ac.uk/~lmd/ltse>

only present the sequence of actions produced in each trace. For the format of complete recorded traces, refer to [9].

$T1 = \langle \text{put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ halt} \rangle$
 $T2 = \langle \text{put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ halt} \text{ halt_exception} \rangle$
 $T3 = \langle \text{put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ put}.1 \text{ get}.0 \text{ put}.1 \text{ halt} \rangle$

The number after the action name describes the number of elements in the buffer as a result of the execution of the operation.

3.2 Context Identification

The context information collected from traces is recorded in a *context table* $CT = \{c_1, \dots, c_n\}$, where $c_1, \dots, c_n$ are entries of the table. Each entry is assigned a *context ID* (CID), which is a unique sequential numeric identifier, and contains a set of values of attributes defining a context, including an identification of the executed block of code (BID), and the evaluation of the control predicate associated with this block. The CT is initialised with an *initial context*, which represents the beginning of the execution.

The LTSE tool constructs the CT by reading each annotation from the log files, identifying contexts and comparing each context found to every context already recorded. A context C is identified as in the CT if, when compared to 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 same BID and the same value of the control predicate. When the context is not in the CT yet, a new entry is created to store it, which is assigned a new CID. Table 1 shows part of the CT of the buffer example. Note that no attributes were selected to identify contexts in this example.

The result of the context identification phase is the generation of a set of context traces. A *context trace* $ctr = \langle s_1, a_1, s_2, \dots, a_n, s_n \rangle$ is a finite sequence of abstract states $s_1, s_2, \dots, s_n$ that correspond to CIDs, such that, for $1 \leq j \leq n$, for every CID _{j} there exists a state s_j which represents that context, and actions $a_1, \dots, a_n \in Act$. It describes the contexts the system went through during the execution, according to context annotations, and the actions that happened in between them, defined by action annotations. As an example, this is part of the context trace generated based on the trace T1: $\langle 0 \ 1 \ 2 \ 3 \ 4 \ \text{put}.1 \ 5 \ 6 \ 7 \ \text{get}.0 \ 2 \ \dots \rangle$. Note that some states might not be connected by actions. In this case, we use an empty action ϵ to represent a transition that is always enabled.

3.3 LKS Creation

As previously stated, we use context information to generate LTS models. However, in order to use values of attributes during the construction of the models, we need an intermediate structure which can deal with both actions and states. We have adopted Labelled Kripke Structures as our intermediate structure.

Table 1. Part of the CT for the buffer component

CID	Predicate	Val	Attribs	BID
0		true		-1
1	put	true		9
2	(usedSlots == 0)	true		5
3	get	true		8
4	(halted)	false		6
5	(usedSlots == SIZE)	false		7
...	...	...	...	...

Definition 6 Labelled Kripke Structure (LKS). A Labelled Kripke Structure $K = (S, s_i, P, \Gamma, \Sigma, T)$ 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 values of 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 [2] in that, instead of propositions, which are always of boolean type, we use attributes to label states. Because of that, in our case, the state-labelling function Γ labels every state with the values of every attribute in P . Moreover, we use a singleton set of initial states to guarantee conformance to Definition 1 when creating the LTS model. This also reflects the fact that the initial state of our models represents the initial context, which is unique.

Our mapping from context traces collected from *Prog* to an LKS involves translating concrete states of *Prog* (information from context annotations) into abstract states of K . Let $CFG_{Prog} = (Q, q_i, Act, \Delta)$ be the CFG of *Prog* and $V(P_{Prog})$ be the set of possible system states. 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 $v \in V(P_{Prog})$. We use $\Theta(Prog) = \{\theta_1, \theta_2, \dots\}$ to denote the set of all possible concrete states of *Prog* and $\Omega \subseteq \Theta(Prog) \times Act \times \Theta(Prog)$ to represent the transition relation between them. The mapping from concrete to abstract states is described below:

- Every concrete state $\theta = (q, v) \in \Theta(Prog)$, where $v = \{val(p_1), \dots, val(p_n)\} \in V(P)$ for $P = \{p_1, \dots, p_n\} \subseteq P_{Prog}$, is modelled by an abstract state $s \in S$, such that $\Gamma(s) = v$, where s is derived from a CID appearing in the context traces generated by *Prog*. This abstract state includes only the values of attributes in the selected set P . Hence, s may represent a set of concrete states $\Theta(Prog)_s = \{\theta_1, \dots, \theta_n\}$, where $\Theta(Prog)_s \subseteq \Theta(Prog)$. These concrete states are indistinguishable when the information used for comparison is restricted to system states containing 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 this 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 $\Theta(Prog)_s = \{\theta_1, \dots, \theta_n\}$, such that $\Theta(Prog)_s \subseteq \Theta(Prog)$, where, for $1 \leq i \leq n$, $\theta_i = (q_i, \{v_i\} \cap V(P))$. Abstract state s' models a set of concrete states $\Theta(Prog)_{s'} = \{\theta'_1, \dots, \theta'_m\}$, such that $\Theta(Prog)_{s'} \subseteq \Theta(Prog)$, where, for $1 \leq j \leq m$, $\theta'_j = (q'_j, \{v'_j\} \cap V(P))$. Let $a \in \Sigma$ be an action. A transition $(s, a, s') \in T$ exists iff there exists a concrete transition $(\theta, a, \theta') \in \Omega$ such that $\theta \in \Theta(Prog)_s$ and $\theta' \in \Theta(Prog)_{s'}$.

Note that the LTSE 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 traces and, subsequently, produce an LTS model from it. Transition labels in the LKS model 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.

3.4 Mapping the LKS into an LTS Model

Essentially, an LKS is an LTS where states are labelled with values of attributes using a state-labelling function. Therefore, an LTS $M = (S', s'_i, \Sigma', T')$ can be obtained from an LKS $K = (S, s_i, P, \Gamma, \Sigma, T)$ simply by ignoring the values of the state labels of K . In this state-label elimination (SLE) process, every state $s' \in S'$ corresponds to a state $s \in S$, such that s' is the same as s but without its label, i.e., $\Gamma(s') = \Gamma(s) \setminus P$. The alphabet and the transition relation do not change after the mapping. Hence, $\Sigma' = \Sigma$ and $T' = T$.

If we associate propositions with actions [11,17], LTL formulas [20] can be defined on behaviours of a model. Hence, a model K satisfies an LTL property ϕ over Σ iff, for all $\pi \in L(K)$, $\pi \models \phi$.

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

Therefore, this mapping is property-preserving when we consider LTL properties that only refer to actions in Σ . Note 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.

The generated LTS can be visualised using the LTSA tool. This tool also allows the specification of LTL properties over actions and supports the checking of such properties against extracted models to detect possible violations.

For a complete description of the model extraction process and of the formal proofs of Theorem 1 and the theorems presented in the next section, refer to [9].

4 Model Faithfulness

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. 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 generation of traces.

Correctness of the models depends essentially on the selection of the attributes to form the system state, used to define contexts. An empty set defines the most abstract model. By including more attributes to the set, the level of abstraction of the generated model can be decreased. Therefore, changing the attributes in the system state directly affects the correctness of the model.

Ideally, if a property ϕ holds in a behaviour model M , then it should also hold in the program $Prog$ represented by M . Nevertheless, this cannot always be guaranteed unless M is complete and correct. If a model is complete, then if a property ϕ holds in the model, it is guaranteed to hold in the system, irrespective of the model being correct or not. However, if the model is incorrect, detected violations can be real or just false negatives. If the model is correct but incomplete, then the absence of violations in the model does not necessarily mean that the property holds in the system. This only ensures that behaviours in $L(M) \cap L(Prog)$ preserve the property. Behaviours in $L(Prog) \setminus L(M)$ cannot be guaranteed to not violate the property.

4.1 Model Refinement

False negatives can usually be eliminated from the model using an abstraction refinement process. In our approach, this process corresponds to the addition of more attributes to the system state, thus decreasing the level of abstraction and improving correctness.

In a refinement process, an original model is said to be an *abstraction* of a refined model, as it includes just part of the information included in its refined version. In [2], the following definition is presented for an abstraction relation considering LKS models:

Definition 7 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. 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 $\lambda = \langle s_1, a_1, \dots \rangle \in \Lambda(K)$ there exists a path $\lambda' = \langle s'_1, a'_1, \dots \rangle \in \Lambda(K_A)$ 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 $\{true, false\}$.

Theorem 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 a set of traces $Tr(F_{Prog})$, collected from F_{Prog} during the CT construction, and a set of attributes $P_A \subseteq P_{Prog}$. If $Tr(F_{Prog})$ 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 $K \sqsubseteq K_A$.*

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

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

Therefore, our refinement process between LKS models preserves LTL properties that consider only actions of the alphabet of the more abstract model. As a consequence, given that there is a property-preserving relation between two LKS models built with different sets of attributes, where one set is a subset of the other, and that the mapping from an LKS to an LTS model is also property-preserving, the generated LTS models have a property-preserving relation between them, which is also a refinement.

We will use again the buffer component mentioned in the previous section as an example. Following the mappings described before, and based on the traces collected, the LTSE generated the model presented in Fig. 1, where state E represents the final state. Note that it incorrectly allows action `get.0` to happen repeatedly, even when the buffer is empty.

Fig. 1. LTS model of the buffer

Following our refinement approach, we attempt to remove this invalid behaviour by adding attributes to the system state. The problem seems to be connected with the fact that the model, at this level of abstraction, does not consider the status of the buffer. That is, the model does not show the behaviour of the

buffer depending on the quantity of stored elements. Therefore, we add attribute `usedSlots` to the system state. This attribute controls the number of elements currently in the buffer.

Using the new system state to generate a model of the buffer results in the LTS shown in Fig. 2. This model does not include the possibility of a `get.0` happening when the buffer is empty.

Fig. 2. Refined LTS model of the buffer

4.2 Improving Completeness

It is possible to improve completeness by adding new traces to the model. The addition of new traces increases the coverage of observed situations and may reveal unknown behaviours, which may violate the property being checked. One possible way of selecting 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.

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 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 to check a given property.

For instance, consider again the example discussed before. Even though the model in Fig. 2 seems a correct abstraction of the behaviour of the buffer, since it does not contain infeasible behaviours, it is incomplete. Note that, after the first occurrence of `put.1`, we reach state 1, where only actions `halt` and `get.0` are enabled. Therefore, the model does not permit the producer to store more than one element in the buffer at all times.

The absence of this behaviour does not affect the correctness of the model, but imposes a restriction that is not real. We improve completeness by adding a new trace. A delay for the initialisation of the consumer was introduced so that

the producer could use the whole capacity of the buffer before the first attempt by the consumer to remove an element from it. This generates the new trace:

$T4 = \langle \text{put.1 put.2 get.1 put.2 get.1 put.2 get.1 get.0 halt} \rangle$

The addition of this trace leads to the construction of the model presented in Fig. 3, which includes the possibility of executing a second **put** before a **get**. Because we did not change the system state, this model preserves the correctness of the previous one.

Fig. 3. More complete LTS model of the buffer

Note that the use of contexts not only allows us to combine multiple traces, as in this example, but also may result in the inclusion of additional behaviours to the model. These behaviours, though not observed in the individual traces, may be inferred based on the identification of similar contexts. Alternative paths may be included in the model even if these paths appear in different traces, provided that the context traces derived from them contain some common contexts. For instance, the sequence of actions $\langle \text{put.1 put.2 get.1 get.0} \rangle$ is a valid behaviour present in the model of Fig. 3 that does not appear in any of the traces. Hence, completeness may be automatically improved by the LTSE based on identified contexts, even without the addition of new traces.

5 Case Studies

Our approach has been applied to a variety of sequential and concurrent systems. Here, we discuss the results of two case studies. Detailed information can be found in [9].

5.1 Single-Lane Bridge

The first case study was based on the Single-Lane Bridge problem described in [19]. Though this system was quite simple, it helped us apply and evaluate our approach in a concurrent system. Moreover, manually created models are

presented in [19], allowing the comparison of those models to our automatically generated models.

The traces generated in this example were a result of selecting options allowed by the interface of the system. We executed the system with one, two and three cars moving in either direction. We extracted the models for each component of the system and used model parallel composition [19] to generate a global model, which was checked against a property specification defined in [19]. A false negative was detected during this procedure, which was eliminated using the refinement process. The results of the analysis using the refined model in the LTSA tool confirmed those found by the authors.

5.2 Bully Algorithm

The Bully Algorithm [10] 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 priority is higher than any of those of the processes still alive, then it becomes the leader.

For this case study, we used an implementation of the Bully Algorithm available on the Internet². In order to reduce the complexity of the model to be generated and concentrate on the election procedure, we chose to analyse only the components involved in the process of electing a leader.

Election members were modelled using a parallel composition of models of six components, where each component represented the behaviour of a local thread. The models of each member were then also composed to generate a model of the entire system.

An interface was provided by the implementation which included operations *start*, *fail*, *recover* and *close* on election members. 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 *start*, *F* represents *fail*, *R* represents *recover*, *C* represents *close* and the numbers between brackets define the priorities of the members executing the operations³:

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 the values of these

² <http://www.cs.queensu.ca/~huang/cisc833/BullyElection.pdf>

³ Note that priority 1 is the highest priority and priority 3 is the lowest.

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.

A safety property was specified for the algorithm [9] stating that there could only be one leader at all times. We checked the property using the safety check provided by the LTSA tool. The detection of some false negatives led us to refine the models, resulting in the complete elimination of those invalid behaviours.

Even though the composite model could not be entirely generated due to lack of memory, it was still possible to check the property, as the safety check of the LTSA may not need to generate the whole composition to detect violations. Checking the property against the model with two members, the tool detected a violation.

The error trace indicated that if communication between members is too slow, the system might reach a state where there is more than one leader. 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 it 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.

6 Related Work and Discussion

Techniques based only on traces, such as [6,21], share with our approach the dependence on the samples of execution to achieve completeness. However, they do not provide means of refining models to improve correctness. Moreover, though the work presented in [21] describes an incremental approach, the increase of completeness of an existing model usually causes the decrease of correctness. This is a consequence of the lack of information about how to combine different traces without creating infeasible behaviours. Context information provides us with the support for such an operation and, thus, improvement of completeness does not affect the correctness of the models.

Some techniques guarantee completeness by obtaining the complete CFG of the system [1,14,3]. As expected, this results in an over-approximated abstraction of the system, which can yield a number of false negatives, but which guarantees the absence of false positives. They rule out false negatives by applying an automatic refinement process based on predicate abstraction [12].

Based on the context information obtained from traces, we can use only partial control flow information to build our models. One could imagine our LTS models

as partial representations of CFGs, as they contain only the sequences of actions defined by the behaviours described in the traces. This guarantees that every behaviour included in the model is a feasible behaviour at a certain level of abstraction, defined by the system state.

Though we do not provide automatic refinement, our refinement process has proved to successfully eliminate false negatives. This process is simple and, unlike the aforementioned related work, does not require the support of a theorem prover. However, the process still lacks well-defined heuristics as to how to select attributes to be used to refine models.

7 Conclusions and Future Work

Our model extraction process generates models based on traces containing context information. The completeness of these models depends on the quantity and quality of the observed behaviours and can be improved with the inclusion of more behaviours. The correctness of the models is affected by the set of attributes used as the system state. Improvement of correctness can be achieved by the addition of more attributes to contexts, thus ruling out false negatives. This refinement process is property-preserving provided that the properties only predicate over actions of the more abstract model.

Results of case studies applying our approach have demonstrated its usefulness for property checking. The refinement process has proved to effectively eliminate false negatives. Though completeness is not always possible to obtain, aiming to include only the necessary behaviours for checking a certain property reduces the possibility of false positives.

As future work, we intend to investigate techniques for the automatic selection of test cases based on a property specification. This would facilitate the identification of which behaviours can affect the property in order to choose an appropriate test suite. This investigation will also enhance our knowledge on how much results of an analysis using our models can improve and/or complement previous analyses based on testing outcomes.

Another possible path to be followed 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 slice, we might be able to achieve completeness with respect to this property.

A definition of heuristics for the selection of attributes used as refinements will also be studied. These heuristics would possibly allow the implementation of an automatic refinement process. Though we have already identified that attributes used in control predicates are more likely to produce the expected results during the refinement process, we still need to find a more formal definition of the influence of these attributes regarding the checking of properties.

Acknowledgments. We thank Freeman Huang for providing the source code for the Bully Algorithm case study.

References

1. Ball, T., Rajamani, S.K.: The SLAM Project: Debugging System Software via Static Analysis. In: POPL, Portland, OR, USA, January 2002, pp. 1–3 (2002)
2. Chaki, S., Clarke, E.M., Ouaknine, J., et al.: State/Event-Based Software Model Checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
3. Chaki, S., Clarke, E.M., Groce, A., et al.: Modular Verification of Software Components in C. IEEE TSE 30(6), 388–402 (2004)
4. Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28(4), 626–643 (1996)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
6. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM ToSEM 7(3), 215–249 (1998)
7. Corbett, J.C., Dwyer, M.B., Hatcliff, J., et al.: Bandera: Extracting Finite-State Models from Java Source Code. In: ICSE, Limerick, Ireland, June 2000, pp. 439–448 (2000)
8. Duarte, L.M., Kramer, J., Uchitel, S.: Model Extraction Using Context Information. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 380–394. Springer, Heidelberg (2006)
9. Duarte, L.M.: Behaviour Model Extraction using Context Information. Ph.D. thesis, Imperial College London, University of London (November 2007)
10. Garcia-Molina, H.: Elections in a Distributed Computing System. IEEE Trans. on Computers C-31(1), 48–59 (1982)
11. Giannakopoulou, D., Magee, J.: Fluent Model Checking for Event-Based Systems. In: ESEC/FSE, Helsinki, Finland, September 2003, pp. 257–266 (2003)
12. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
13. Havelund, K., Pressburguer, T.: Model Checking Java Programs Using Java PathFinder. STTT 2(4), 366–381 (2000)
14. Henzinger, T.A., Jahla, R., Majumdar, R., et al.: Lazy Abstraction. In: POPL, Portland, OR, USA, January 2002, pp. 58–70 (2002)
15. Holzmann, G.J., Smith, M.H.: A Practical Method for Verifying Event-Driven Software. In: ICSE, Los Angeles, USA, May 1999, pp. 597–607 (1999)
16. Jackson, D., Damon, C.A.: Software Analysis: A Roadmap. In: ICSE, Limerick, Ireland, June 2000, pp. 133–145 (2000)
17. Leuschel, M., Massart, T., Currie, A.: How to Make FDR Spin: LTL Model Checking of CSP by Refinement. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 99–118. Springer, Heidelberg (2001)
18. Ludewig, J.: Models in Software Engineering - An Introduction. SoSyM 2(1), 5–14 (2003)
19. Magee, J., Kramer, J.: Concurrency: State Models and Java Programming, 2nd edn. Wiley and Sons, Chichester (2006)
20. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York (1992)
21. Mariani, L.: Behavior Capture and Test: Dynamic Analysis of Component-Based Systems. Ph.D. thesis, Università degli Studi di Milano Bicocca (2005)
22. Uchitel, S., Kramer, J., Magee, J.: Behaviour Model Elaboration Using Partial Labelled Transition Systems. In: ESEC/FSE, Helsinki, Finland, September 2003, pp. 19–27 (2003)