Algorithms for efficient symbolic detection of faults in context-aware applications

Michele Sama, Franco Raimondi, David Rosenblum, Wolfgang Emmerich
Department of Computer Science
University College London
Gower Street, London, UK
{m.sama,f.raimondi,d.rosenblum,w.emmerich}@cs.ucl.ac.uk

Abstract

Context-aware and adaptive applications running on mobile devices pose new challenges for the verification community. Current verification techniques are tailored for different domains (mostly hardware) and the kind of faults that are typical of applications running on mobile devices are difficult (or impossible) to encode using the patterns of "traditional" verification domains [9].

In this paper we present how techniques similar to the ones used in symbolic model checking can be applied to the verification of context-aware and adaptive applications. More in detail, we show how a model of a context-aware application can be encoded by means of Ordered Binary Decision Diagrams and we introduce symbolic algorithms for the verification of a number of properties.

1 Introduction

The current miniaturization trend of a number of components (such as GPS, accelerometers, barometers, Bluetooth and WiFi interfaces, light sensors and video-cameras, etc.), together with their optimization in terms of energy consumption, has made it possible to develop portable devices that can *monitor and adapt* to the environment in which they "live". Thus, for instance, it is now standard to have mobile phones that automatically adjust their brightness to the available light and many other *adaptations* can be (and are) triggered automatically by changes in the *context*.

Context-Aware Adaptive Applications (CAAAs) are the software layer responsible for these automatic changes. Typically, CAAAs read the state of the environment through a number of input channels, possibly governed by a middleware, and act on the device to execute the appropriate adaptation mechanisms [5, 10, 11, 12, 22, 24, 25]. A set of *adaptation rules* is usually included in the implementation of CAAAs to specify their actual behaviour.

The aim of this paper is to detect and identify adaptation *faults* that may arise while executing adaptation rules, such as

- The failure of a rule to trigger in spite of being supposed to do so by the designers.
- The device reaching a deadlock state.
- The device continuously looping around different configurations thus effectively making the device unusable.

The large size of the state space and the kind of properties that need to be verified make this problem challenging. In this paper we describe how the state space can be manipulated by means of Ordered Binary Decision Diagrams (OBDDs [4]) and symbolic techniques similar to the ones used in symbolic model checking [8] can then be employed in the verification step. Notice that it would not be possible to reduce our verification problem to the input problem of a model checker such as NuSMV [6] or SPIN [16], as these enable the verification of *temporal patterns* only (in the sense of [9]). As a consequence, it would not be possible to employ these tools for the verification of properties such as detecting loops and/or cycles.

The rest of the paper is organised as follows: Section 2 compares this work with recent literature. Section 3 introduces OBDDs and notions of symbolic computations. Section 5 shows how (the finite-state machine representing) a CAAA can be translated into OBDDs. An example of a CAAA is presented in Section 4. The verification algorithms are presented in Section 6. We conclude in Section 7.

2 Related work

Our work is broadly related to existing verification techniques for context-aware applications, real-time systems, sequential networks, rule-based adaptation in context-awareness middleware, and combinatorial models of inputs.

The class of system which we target perform sequential asynchronous inquiries to multiple sensors obtaining contextual information which are use to switch between adaptive behaviours. CAAAs extend the concept of context-awareness [26], in which a system reacts to external inputs, by using this reaction to feed adaptive process of self-modification [17]

Roman et al. provides Mobile extension for the UNITY notation and proof logic to the verification of mobile systems [23]. Mobile UNITY mainly focuses on mobility, while our approach is inspecting adaptations.

Validation of CAAAs has been the target of other researches [20, 28, 29]. However our approach differs from other techniques because it applies model checking to adaptation models, while other works focus on test case selection and runtime analysis.

The core of our approach is a rule based transition system. Traditional techniques for testing rule-based systems focus on predicate validation and rule chains [3, 13]. Our analysis considers multiple rules at the same time and detects interferences.

Transition systems and finite state machines (FSM) have been used extensively to represent and verify properties of systems in requirements engineering. Heitmeyer et al. use FSM models to discover inconsistencies in SCR specifications [15], and Heimdahl and Leveson use FSM models to discover inconsistencies in RSML specifications [14]. While the classes of inconsistencies that they detect are characteristic of requirements specifications, the fault patterns that we detect are characteristic of CAAAs.

3 Preliminaries

3.1 Binary Decision Diagrams

Ordered Binary Decision Diagrams have been particularly successful in the last two decades because they offer, on average, a much more compact representation of Boolean functions with respect to other canonical forms (e.g., conjunctive/disjunctive normal forms).

A *Boolean variable* x is a variable whose value is either 0 or 1. A *Boolean function* of n Boolean variables is a function $f : \{0,1\}^n \rightarrow \{0,1\}$. *Boolean formulae* can be seen as Boolean functions. For instance, the Boolean formula $x_1 \wedge (x_2 \vee x_3)$ can be seen as the Boolean function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$.

A rooted, directed graph G can be associated to every Boolean function $f(x_1, \ldots, x_n)$ by imposing an ordering on the variables $x_1, \ldots, x_n$, and by reducing the graph (in the sense explained below) [4]. The graph G is called the *Ordered Binary Decision Diagrams* of f. For instance, the reduced graph associated with the Boolean function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$ is depicted in Figure 1 (b),

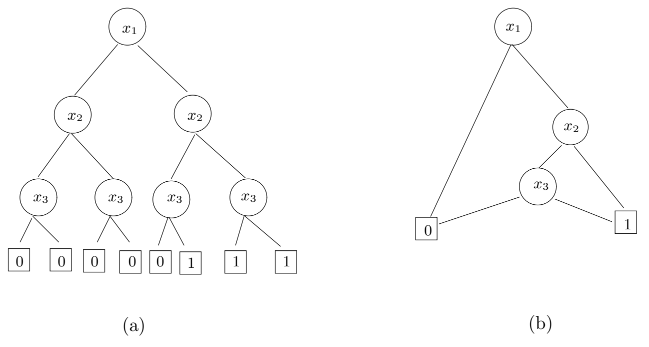

Figure 1. OBDD example for $f = x_1 \wedge (x_2 \vee x_3)$.

by "simplifying" the graph depicted in Figure 1 (a). Formally, a graph is reduced by iteratively eliminating the vertexes which are the root of two isomorphic subgraphs, and by merging isomorphic subgraphs. A graph is said to be *reduced* if it contains no isomorphic subgraphs and no vertexes v and v' such that the sub-graphs rooted at v and v' are isomorphic. We assume here that the left child of a vertex corresponds to the choice of the value 0 (i.e., *false*) for the variable preceding it, while the right child correspond to the choice of the value 1 (i.e., *true*). Thus, the leftmost path of Figure 1 (a) corresponds to an assignment of 0 to all variables and, consequently, to the value 0 to the expression $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$.

It is shown in [4] that, given a fixed ordering of the Boolean variables $x_1, \ldots, x_n$, the reduced graph of any Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is unique (i.e., OBDDs are a *canonical* representation for Boolean functions).

Boolean operators can be applied to Boolean functions; for instance the disjunction operator $\vee$ can be applied to two Boolean functions f_1 and f_2 to obtain a third Boolean function $f_3 = f_1 \vee f_2$.

3.2 Symbolic computations

The use of OBDDs to represent states and transitions has been proposed by [21]. The key idea here is to represent states (and set of states) as Boolean formulae which, in turn, can be encoded as OBDDs. As an example, consider the set of states $\mathcal{S} = \{\mathcal{S}_\infty, \mathcal{S}_\in, \mathcal{S}_\ni\}$ and the relation $\mathcal{R} = \{(\mathcal{S}_\infty, \mathcal{S}_\in), (\mathcal{S}_\in, \mathcal{S}_\ni), (\mathcal{S}_\ni, \mathcal{S}_\infty)\}$ (i.e. a simple loop). Let $N = \lceil log_2|\mathcal{S}|\rceil$; in our example $N = 2$. Each element $S \in \mathcal{S}$ is associated with a vector of Boolean variables $\overline{x} = (x_1, \ldots, x_N)$, i.e., each element of S is associated with a tuple of $\{0,1\}^N$. Each tuple $\overline{x} = (x_1, \ldots, x_N)$ is then identified with a Boolean formula, represented by a conjunction of literals, i.e., a conjunction of variables or

State	Boolean vector	Boolean formula
S_1	$(1,1)$	$x_1 \wedge x_2$
S_2	$(1,0)$	$x_1 \wedge \neg x_2$
S_3	$(0,1)$	$\neg x_1 \wedge x_2$

Table 1. Example of Boolean encoding.

their negation[1]. It is assumed that the value 0 in a tuple corresponds to a negation. The encoding of the states in our example is given in Table 1.

Sets of states are encoded by taking the disjunction of the Boolean formulae encoding the single states. For instance, the set of states $\{S_1, S_3\}$ from the example in Table 1 is encoded by the Boolean formula $f = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_2)$.

A new set of "primed" variables $(x'_1, \ldots, x'_N)$ is introduced to encode the relation between two states $S, S' \in \mathcal{S}$. In particular, if $S\mathcal{R}S'$ holds, then S is encoded using the non-primed variables, S' is encoded using the primed variables, and the relation $S\mathcal{R}S'$ is expressed as a Boolean formula by taking the conjunction of the encoding for S and S'. The whole relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is encoded as a Boolean formula by taking the disjunction of all the transition steps. In our example, the transition relation is encoded by the following Boolean formula f_R:

$$f_R(x_1, x_2, x'_1, x'_2) = ((x_1 \wedge x_2) \wedge (x'_1 \wedge \neg x'_2)) \vee \\ ((x_1 \wedge \neg x_2) \wedge (\neg x'_1 \wedge x'_2)) \vee ((\neg x_1 \wedge x_2) \wedge (x'_1 \wedge x'_2))$$

3.3 Modelling CAAAs

We model a CAAAs by means of a set of states $\mathcal{S}$; for instance, the profile of a mobile phone (e.g., *Meeting* or *Office*) is a possible state, see Section 4. CAAAs evolve from an initial state $S_{initial} \in \mathcal{S}$ in accordance to a set of *adaptation rules*. We model adaptation rules by means of a "transition" relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S} \times \mathcal{A}$, where $\mathcal{P}$ is a set of Boolean formulae called *predicates* that can be built starting from a set $\mathcal{C}$ of *propositional context variables* and $\mathcal{A}$ is a set of actions (such as "enable Bluetooth").

Let $R = (S, P, S', A)$ be a rule in $\mathcal{R}$. By slight abuse of notation, we denote by $P(S)$ the result of evaluating the Boolean expression P in state S: this evaluation is performed by evaluating the propositional context variables in S and then computing the result when these are substituted in P. If $P(S)$ evaluates to true, then we say that R becomes *active*. Intuitively, this means that whenever S is the current state and $P(S)$ is true, then the CAAA performs a transition to S' and executes A upon entering in S'.

[1]By slight abuse of notation, the same symbols $x_i (i \in \{1, \ldots, N\})$ are used to denote Boolean variables in a vector, and atomic propositions in logical formulae.

4 A concrete example

In this section we present *PhoneAdapter*, a typical application that can be verified using our approach.

The application uses contextual information to adapt a phone's configuration *profile*. Phone profiles are settings that determine a phone's behavior, such as display intensity, ring tone volume, vibration, and Bluetooth discovery. Instead of requiring users to select a profile manually, the application is driven by a set of adaptation rules, each of which specifies a predicate whose satisfaction automatically triggers the activation of an associated profile. The selected profile prevails until a more suitable one is chosen through the triggering of other rules. The rule predicates are expressed over context readings from Bluetooth and GPS sensors on the phone plus the phone's internal clock.

PhoneAdapter's adaptation rules define nine profiles:

1. *General*: the initial profile (which defines a user-specified default configuration), and the profile applied by default when the phone's sensors are unable to detect any activity related to one of the remaining profiles;
2. *Home*: increases the ring tone volume and removes vibration when the user is at home;
3. *Office*: mutes the ring tone and activates vibration when the user is in his office;
4. *Meeting*: mutes the ring tone and disables vibration when the user is in a meeting;
5. *Outdoor*: increases the backlight and speaker volume when the user is outdoors;
6. *Jogging*: increases the backlight and speaker volume and also activates vibration when the user is jogging;
7. *Driving*: connects to the car's handsfree communication system when the user is driving;
8. *DrivingFast*: diverts calls when the user is driving fast;
9. *Sync*: periodically synchronizes personal information on the phone with the user's home or office PC when the phone is not in use and the PC is discovered via Bluetooth.

Some profiles are more important than others for safety or social reasons, so it is possible to sort the rules with a weak priority order that determines their evaluation order. In this scenario, high priority is given to rules related to *DrivingFast* and *Driving*, medium priority to rules related to *Meeting*, *Home*, *Outdoor*, *Jogging*, and *Office*, and low priority to rules related to *Sync* (since synchronization can be performed after other activities have been accounted for).

Rule Name	Current States	New State	Full Predicate	Simple Predicate
ActivateOutdoor	General	Outdoor	GPS.isValid() and !GPS.location()=home and !GPS.location()=office	A_{gps} and $!B_{gps}$ and $!C_{gps}$
DeactivateOutdoor	Outdoor	General	!ActivateOutdoor	!(A_{gps} and $!B_{gps}$ and $!C_{gps}$
ActivateJogging	Outdoor	Jogging	GPS.isValid() and GPS.speed()>5	A_{gps} and D_{gps}
DeactivateJogging	Jogging	Outdoor	!ActivateJogging	!(A_{gps} and D_{gps})
ActivateDriving	General, Home, Office, Outdoor	Driving	BT=car_handsfree	A_{bt}
DeactivateDriving	Driving	General	!ActivateDriving	$!A_{bt}$
ActivateDrivingFast	Driving	DrivingFast	GPS.isValid() and GPS.speed()>70	A_{gps} and E_{gps}
DeactivateDrivingFast	DrivingFast	Driving	!ActivateDrivingFast	!(A_{gps} and E_{gps})
ActivateHome	General	Home	BT=home_pc or (GPS.isValid() and GPS.location()=home)	B_{bt} or (A_{gps} and B_{gps})
DeactivateHome	Home	General	!ActivateHome	!(B_{bt} or (A_{gps} and B_{gps}))
ActivateOffice	General	Office	BT=office_pc or BT=office_pc_* or (GPS.isValid() and GPS.location()=office)	C_{bt} or D_{bt} or (A_{gps} and C_{gps})
DeactivateOffice	Office	General	!ActivateOffice	!(C_{bt} or D_{bt} or (A_{gps} and C_{gps}))
ActivateMeeting	Office	Meeting	Time>=meeting_start and BT.count()>=3	A_t and E_{bt}
DeactivateMeeting	Meeting	Office	Time>=meeting_end	B_t
ActivateSync	General	Sync	BT=home_pc or BT=office_pc	B_{bt} or C_{bt}
DeactivateSync	Sync	General	!ActivateSync	!(B_{bt} or C_{bt})

Table 2. Adaptation Rules of PhoneAdapter

Table 2 presents the set of adaptation rules we defined for *PhoneAdapter*. For convenience, the table depicts names we use later in the text to refer to specific rules, and it depicts rule predicates both in their simplified form expressed over propositional context variables, and in their fully expanded form expressed over sensed context variables. In some cases a rule name is used in place of a full predicate, meaning that the full predicate is the same as that of the named rule.[2] Also, the table does not show the actions of rules, since they play no role in our example.

As shown in the table, *PhoneAdapter* adapts between nine different states according to 19 different rules expressed over three different sensed context variables, namely BT (Bluetooth), GPS (or other form of location such as AGPS, based on cell id, or using WiFi mapping) and Time, which are monitored via 12 propositional context variables representing the 12 different relational expressions in which the sensed context variables are used. For example, one such relational expression is *GPS.location()=home*, which tests whether the location sensed by the phone's GPS sensor corresponds to the user's home location (stored in variable *home*). This relational expression is represented throughout the rules by the propositional context variable B_{gps}.

Figure 2 depicts the (labelled) transition system associated with the adaptation rules of *PhoneAdapter*, with state *General* being its initial state.

[2]Note that according to our definitions of $\mathcal{R}$ and M, the row named *ActivateDriving* in Table 2 actually represents four different rules, each being active in a different state; however, because the predicates and priorities of those rules are identical, we represent them in the table with a single rule name for simplicity.

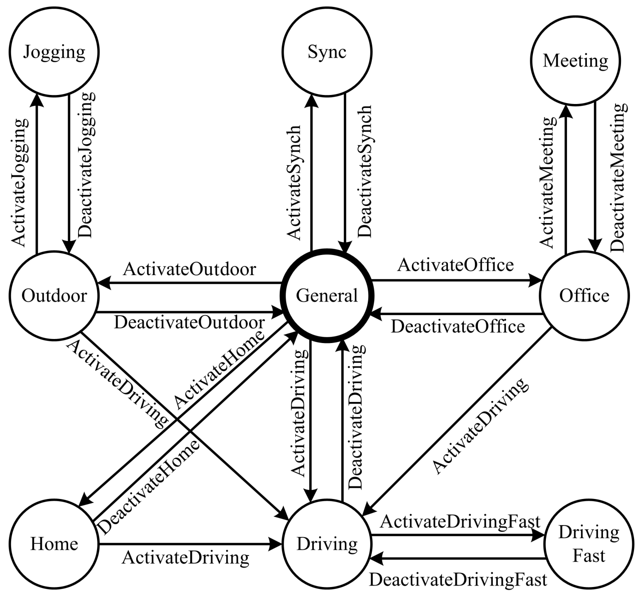

Figure 2. The transition system for PhoneAdapter.

5 Encoding a CAAA using OBDDs

In this section we show how a CAAA can be encoded using OBDDs, using a technique similar to the one presented in Section 3.

The number of Boolean variables required to encode states of $\mathcal{S}$ is $N = \lceil log_2|\mathcal{S}|\rceil$; for instance, for the example in the previous section $N = \lceil log_2|9|\rceil = 4$. Let $\overline{v}$ be the vector of N Boolean variables encoding $\mathcal{S}$; in addition, we introduce N more variables to encode the "destination" state in a transition by means of a vector $\overline{v'} = (v_1, \ldots, v_N)$. We also introduce $M = |\mathcal{C}|$ variables that represent the truth value of the Boolean propositional context variables (in the example presented in the previous section there are 12 such variables) encoded by the Boolean vector $\overline{c} = (c_1, \ldots, c_M)$. Finally, we introduce $O = \lceil log_2|\mathcal{A}|\rceil$ variables to encode actions.

The representation above allows for the encoding of rules in $\mathcal{R}$: let $R = (S, P, S', A)$ be a rule in $\mathcal{R}$. Its Boolean representation is given by $\overline{v} \wedge \overline{P(S)} \wedge \overline{v'} \wedge \overline{a}$, where $\overline{v}$ and $\overline{v'}$ are, respectively, the Boolean representation of states S and S', $\overline{P(S)}$ is a Boolean formula representing the predicate P in which the appropriate variables from $\mathcal{C}$ have been substituted, and $\overline{a}$ is the Boolean encoding for the action A. For instance, if S is represented by the Boolean vector $(1, 1, 1, 1)$, S' by the Boolean vector $(1, 1, 1, 0)$, $P(S)$ by the Boolean expression $c_1 \wedge c_3 \wedge \neg c_6$ (these represent the context variables as they are used in $P(S)$), and A by the Boolean vector $(1, 0)$, then $\overline{R}$ is the following Boolean expression:

$$(v_1 \wedge v_2 \wedge v_3 \wedge v_4) \wedge (v'_1 \wedge v'_2 \wedge v'_3 \wedge \neg v'_4) \wedge (c_1 \wedge c_3 \wedge \neg c_6) \wedge (a_1 \wedge \neg a_2)$$

The set of rules $\mathcal{R}$ is encoded by a Boolean formula as well by taking the disjunction of all the rules $R_i \in \mathcal{R}$, i.e.:

$$\overline{\mathcal{R}} = \bigvee_{R_i \in \mathcal{R}} \overline{R_i}$$

All the Boolean formulae mentioned here can be represented by means of OBDDs thus providing a compact representation and enabling symbolic computations. In particular, given the Boolean formula $\overline{\mathcal{R}}$ encoding rules, we compute the set of states reachable from the initial state by means of Algorithm 1. In the following, we use the notation from [27], which provides a C/C++ library for the efficient manipulation of OBDDs (notice that BDD and OBDD are synonyms in this library).

The set of states reachable from S_{init} is needed for the verification algorithms presented in the next section. This set is obtained using Algorithm 1; the idea here is to compute iteratively the set of reachable states by starting from the set of initial states (line 1), then computing a transition step (line 8) and repeating until there is no change (i.e. a fix-point has been reached). The function "exists($\overline{v}$,*next*)" is used to quantify a vector of Boolean variables in a BDD. Formally, given a Boolean function $f(x_1, \ldots, x_n)$, the operation $\exists x_i.f(x_1, \ldots, x_n)$ is defined as the application of the disjunction operator to the composition of f with a constant function, i.e., $\exists x_i.f(x_1, \ldots, x_n) = f_{x_i=0}(x_1, \ldots, x_n) \vee f_{x_i=1}(x_1, \ldots, x_n)$. This definition is extended to the quantification over a vector by taking the disjunction of all the possible combinations of truth assignments to the vector. An efficient implementation for this

Algorithm 1 Reachable States

Input: the CAAAs encoded using OBDDs.
Output: *reach*: reachable states (OBDD).

```
1: BDD q,reach,next;
2: q = $\overline{S_{init}}$;
3: reach = bddZero();
4: next = bddZero();
5: while q != reach do
6:     reach = q;
7:     next = q;
8:     next = next * $\overline{\mathcal{R}}$;
9:     next = exists($\overline{v}$,next);
10:    next = exists($\overline{c}$,next);
11:    next = exists($\overline{a}$,next);
12:    next = next.swapVariables($\overline{v}$, $\overline{v'}$);
13:    q = q + next;
14: end while
15: return reach
```

function is provided by [27]. Notice that the OBDD resulting from the quantification over a vector $\overline{v}$ does *not* depend on $\overline{v}$, i.e., the OBDD *next* at line 11 is a function of $\overline{v'}$ only because all the other variables have been quantified. Finally, line 12 converts a "next" state to a "current" state (i.e. in terms of $\overline{v}$), and *next* now encodes (as an OBDD) the set of states reachable with one transition from the set of reachable states computed in previous iterations. In line 13 this set is added to the set of reachable states and the loop is repeated. The termination of the loop is guaranteed by the fact that the construction is monotonically increasing and the set of reachable states is finite.

6 Verification Algorithms

In this section we describe a family of faults which we name *behavioural faults*. These are faults caused by errors in the logic of and in the relationship between rules. We have identified the following properties that may be violated:

- **Reachability:** each state is reachable from the initial state via some sequence of adaptations.
- **Determinism:** For each state in a CAAA and each possible assignment of values to propositional context variables in that state, there is at most one rule that can be triggered.
- **State Liveness:** For each state in a CAAA, if the state contains any active rules (and thus is not a final state), then at least one of the active rules has a satisfiable predicate.

- **Rule Liveness:** For each state in a CAAA and each of its active rules, there is at least one assignment of values to propositional context variables that satisfies the predicate of the rule.
- **Stability:** The state of a CAAA is not dependent on the length of time a propositional context variable holds its value.

We discuss these properties in detail below and we present symbolic algorithms for the detection of violations of these properties, using the example presented in Section 4 where needed to clarify the presentation.

6.1 Reachability property

The violation of this property can identify faults at design level and it is easily verified by comparing the OBDD for reachable states *reach* computed by means of Algorithm 1 and the OBDD encoding the set $\mathcal{S}$: if the two OBDDs are not (syntactically) equal, then there exists at least a state of $\mathcal{S}$ which is not in *reach*. The set of states that are not reachable is encoded by the following OBDD:

$$unreach = \overline{\mathcal{S}} \wedge (\neg reach)$$

The set of states corresponding to this OBDD (if not empty) can be printed using a number of utility functions from [27].

6.2 Determinism properties

If the rules of a CAAA violate the determinism property we say that the rules contain a non-deterministic activation. This violation pattern is characterized by the presence of multiple active rules in the same state whose predicates can be satisfied the same set of context variables, generating non-deterministic adaptations.

Algorithm 2 finds the set of predicates that allow for a transition to two different states from the same starting state, represented by means of the OBDD *faults*. This set is computed by iterating over all the states and checking the predicates that are enabled in that state, encoded by means of the OBDD *rules*: this latter OBDD is computed by quantifying over actions and "next" states (see lines 3 and 4). For each predicate in this set the algorithm checks the size of the BDD encoding the successor state from $\overline{v}$ (lines 6 and 7). If this is greater than one (line 8), then the definition of the original CAAA contains two rules that can be satisfied at the same time in the same state.

6.3 Liveness properties

If the rules of a CAAA violate the Rule Liveness property, then the rule contains a *dead predicate*, i.e., in some state there is a predicate which is not satisfiable. Furthermore, if *none* of the predicates in a state are satisfiable, then no transition is possible from that state, i.e., the state is a deadlock state.

Algorithm 2 Non-determinism detection

Input: the CAAAs encoded using OBDDs.
Output: *faults*: fault states (OBDD).

```
1:  BDD rules, next
2:  for each state $\overline{v} \in \mathcal{S}$ do
3:      rules = exists($\overline{v'}$,$\overline{\mathcal{R}} \wedge \overline{v}$);
4:      rules = exists($\overline{a}$,rules);
5:      for each predicate $\overline{p} \in$ rules do
6:          next = $\overline{p} \wedge \overline{\mathcal{R}} \wedge \overline{v}$
7:          next = exists($\{\overline{a}, \overline{v}, \overline{v'}\}$,next);
8:          if size(next $> 1$) then
9:              faults.add($\overline{p}$);
10:         end if
11:     end for
12: end for
13: return faults
```

Algorithm 3 Rule liveness detection

Input: the CAAAs encoded using OBDDs.
Output: *faults*: fault states (OBDD).

```
1:  BDD rules, next
2:  for each state $\overline{v} \in \mathcal{S}$ do
3:      next = exists($\{\overline{a}, \overline{c}\}$,$\overline{\mathcal{R}} \wedge \overline{v}$);
4:      for each state $\overline{v'} \in$ next do
5:          rules = $\overline{v} \wedge \overline{\mathcal{R}} \wedge \overline{v'}$
6:          rules = exists($\{\overline{a}, \overline{v}, \overline{v'}\}$,next);
7:          if rules == bddFalse() then
8:              faults.add($\overline{v}, \overline{v'}$);
9:          end if
10:     end for
11: end for
12: return faults
```

Algorithm 3 detects the set of states (current,next) such that a dead rule exists between them. This is done by iterating over the set of states (line 2) and then over the set of states reachable from each state (line 4): if a predicate that cannot be satisfied was defined for these two states then the pair of states is added to a vector of faults. In line 7, un-satisfiability of a predicate is checked by comparing the OBDD encoding the predicate to the zero OBDD (i.e., false, represented by the built-in function "bddFalse()").

Similarly, Algorithm 4 detects the states in which *all* the rules are unsatisfiable: in this case for each state all the enable rules are obtained by quantifying over the next states and actions (line 3). If this set is not satisfiable, the state is added to the set of faulty state. As above, un-satisfiability is checked by comparing the OBDD to the zero OBDD.

Algorithm 4 State liveness detection

Input: the CAAAs encoded using OBDDs.
Output: *faults*: fault states (OBDD).

```
1: BDD rules, next
2: for each state $\overline{v} \in \mathcal{S}$ do
3:     rules = exists($\{\overline{a}, \overline{v}, \overline{v'}\}$,$\overline{v} \wedge \overline{\mathcal{R}}$);
4:     if rules == bddFalse() then
5:         faults.add($\overline{v}, \overline{v'}$);
6:     end if
7: end for
8: return faults
```

6.4 Stability properties

A CAAA suffers from *stability problems* when a set of context variables can produce a sequence of adaptations and when the final state of the sequence may depend on the length of time with which some context variables hold their value. In general, we say that the rules of a CAAA contain an *Adaptation Race* fault when the rules allow an indefinite number of adaptation rules to occur. If the adaptations form a cycle, then we say that the rules contain an *Adaptation Cycle* fault.

In general, these pattern of behaviour may not always be considered faulty; however, they may produce multiple adaptations which, in the best case, could annoy the user. Moreover, races can be dangerous because the CAAA will adapt to the last state of the race if the affected variables hold their values long enough, but otherwise the final state could be random.

As an example, consider the states Office and Meeting in Table 2 and the transition rules ActivateMeeting and DeactivateMeeting. In particular, the predicate Time$>=$MeetingStart is true for *any* time greater than MeetingStart, in particular even after the end of the meeting. Therefore, an adaptation cycle is executed here at the end of the meeting, when the profile wrongly loops between Office and Meeting.

Algorithm 5 presents a method to compute the predicates which trigger adaptations of length greater than 1. First, all the predicates are extracted from the rules (line 2); for each predicate, we compute the set *from* of states from where a transition is triggered by the predicate and the set *to* of states reachable with that predicate (line 4 and 5). We then take the intersection of these two sets (line 6) and, if it is not null (line 7), then we found a state (the state *to*) which is at the same time a starting and a final state for the same predicate, i.e., at least an adaptation of length 2 exists for the predicate.

Algorithm 5 Metastability detection

Input: the CAAAs encoded using OBDDs.
Output: *faults*: fault predicates (OBDD).

```
1: BDD rules, from, to
2: rules = exists($\{\overline{a}, \overline{v}, \overline{v'}\}$,$\overline{\mathcal{R}}$);
3: for each predicate $\overline{p} \in$ rules do
4:     from = exists($\{\{\overline{a}, \overline{v'}\}$, $\overline{\mathcal{R}} \wedge \overline{p}$);
5:     to = exists($\{\{\overline{a}, \overline{v}\}$, $\overline{\mathcal{R}} \wedge \overline{p}$);
6:     overlap = from $\wedge$ to.swapVariables($\overline{v}, \overline{v'}$);
7:     if (overlap) != bddFalse() then
8:         faults.add($\overline{p}$);
9:     end if
10: end for
11: return faults
```

Under our assumption, we consider faulty all the adaptations requiring more than one transition step but our algorithm could be easily extended to detect cycles only.

7 Discussion and conclusion

In this paper we have presented how CAAAs can be represented by means of OBDDs and we have introduced in detail a number of algorithms to verify a set of properties symbolically: determinism of transitions, liveness properties, stability of states. Some of these properties cannot be encoded using traditional temporal patterns in the sense of [9], therefore traditional model checkers could only be used to verify part of the properties presented in this paper. In particular, it is not possible to verify that a property holds infinitely often in a model by means of temporal model checkers: this is the reason for the introduction of fairness conditions outside the model in model checkers such as NuSMV [6] (see details in [7]). Moreover, most temporal model checkers do not have the notion of labelled transitions, and therefore it would not be possible to check a property such as the Rule liveness property. We are investigating the applicability of some extensions of temporal logic with actions [2, 19] to our scenario.

We are currently evaluating our approach using a concrete implementation based on [27]. Preliminary experimental results are encouraging and we plan to present a more detailed evaluation at the workshop.

Various directions are worth exploring starting from the algorithms presented here: we plan to define an input language for CAAAs to define rules and properties, with the idea of implementing a model checker for CAAAs. In addition, we plan real-time extensions of our algorithms using quantitative approaches as in [1, 18].

Acknowledgements

The work described in this paper builds on previous work with Sebastian Elbaum and Zhimin Wang of the University of Nebraska-Lincoln.

This work was partially supported by the EPSRC under grant EP/D077273/1 for project UbiVal and by the EU project PLASTIC. David Rosenblum holds a Wolfson Research Merit Award from the Royal Society.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, April 1994.

[2] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[3] V. Barr. Applications of rule-base coverage measures to expert system evaluation. In *Proc. National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, pages 411 – 416, July 1997.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[5] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):29–945, October 2003.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: An open-source tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364. Springer-Verlag, 2002.

[7] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical report, Pittsburgh, PA, USA, 1994.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York, 1998. ACM Press.

[10] P. Fahy and S. Clarke. CASS—Middleware for mobile context-aware applications. In *Proc. MobiSys Workshop on Context Awareness*, pages 304–308, June 2004.

[11] J. Floch. Theory of adaptation. Deliverable D2.2, MADAM Project, available from http://www.ist-music.eu/MUSIC/madam-project/madam-deliverables/techreportreference.2007-04-13.0451108510, 2006. Last accessed 7 March 2008.

[12] T. Gu, H. K. Pung, and D. Q. Zhang. A middleware for building context-aware mobile services. In *Proc. IEEE Vehicular Technology Conference*, pages 2656– 2660, May 2004.

[13] U. G. Gupta. Automatic tools for testing expert systems. *Communications of the ACM*, 5:179–184, May 1998.

[14] M. P. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[16] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Eng.*, 23(5):279–295, 1997.

[17] iscid. http://www.iscid.org/encyclopedia/adaptive_system.

[18] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[19] A. Lomuscio and F. Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS'06)*, pages 161–168, Hakodake, Japan, 2006. ACM Press.

[20] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: A data flow approach and an RFID-based experimentation. In *Proc. International Symposium on Foundations of Software Engineering*, pages 242–252, November 2006.

[21] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[22] A. Ranganathan and R. H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, pages 143–161, June 2003.

[23] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250 – 282, July 1997.

[24] M. Sama and D. Rosenblum. ContextNotifier. http://code.google.com/p/contextnotifier/.

[25] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. Multi-layer faults in the architectures of mobile, context-aware adaptive applications: A position paper (Short Paper). In *Proc. ICSE 2008 International Workshop on Software Architectures and Mobility*, May 2008. Short Paper, to appear.

[26] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[27] F. Somenzi. CUDD: CU decision diagram package - release 2.4.1. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`, 2005.

[28] T. Tse, S. Yau, W. Chan, H. Lu, and T. Chen. Testing context-sensitive middleware-based software applications. In *Proc. International Computer Software and Applications Conference*, pages 458–465, September 2004.

[29] Z. Wang, S. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *Proc. International Conference on Software Engineering*, pages 406–415, May 2007.