Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification

Michele Sama, Sebastian Elbaum, *Member*, *IEEE*, Franco Raimondi, David S. Rosenblum, *Fellow*, *IEEE*, and Zhimin Wang

Abstract—Applications running on mobile devices are intensely context-aware and adaptive. Streams of context values continuously drive these applications, making them very powerful but, at the same time, susceptible to undesired configurations. Such configurations are not easily exposed by existing validation techniques, thereby leading to new analysis and testing challenges. In this paper, we address some of these challenges by defining and applying a new model of adaptive behavior called an Adaptation Finite-State Machine (A-FSM) to enable the detection of faults caused by both erroneous adaptation logic and asynchronous updating of context information, with the latter leading to inconsistencies between the external physical context and its internal representation within an application. We identify a number of adaptation fault patterns, each describing a class of faulty behaviors. Finally, we describe three classes of algorithms to detect such faults automatically via analysis of the A-FSM. We evaluate our approach and the trade-offs between the classes of algorithms on a set of synthetically generated Context-Aware Adaptive Applications (CAAAs) and on a simple but realistic application in which a cell phone's configuration profile changes automatically as a result of changes to the user's location, speed, and surrounding environment. Our evaluation describes the faults our algorithms are able to detect and compares the algorithms in terms of their performance and storage requirements.

Index Terms—Adaptation, context-awareness, fault detection, mobile computing, model-based analysis, model checking, ordered binary decision diagrams, symbolic verification, ubiquitous computing.

1 Introduction

The growing popularity of handheld devices, such as cell phones, PDAs, and portable consoles, and the increasing availability of infrastructures that support mobility, such as GPS satellites, WiFi networks, and Bluetooth services, together create a market for new kinds of applications that constantly monitor and react to contextual information. For example, among the top 10 awardees of Google's Android Developer Challenge conducted in 2008, five applications are heavily influenced by their environment: One application relies on user-defined conjunctions of environmental conditions to adjust a phone's configuration [17]. Another uses acceleration to determine if a collision occurred [16]. Still others use GPS location to measure how much a user has covered in a race [26] and to estimate personal carbon footprint [8]. The fifth relies on the detection of nearby users in order to establish social connections [30]. Key characteristics of these emerging *Context-Aware Adaptive Applications* (CAAAs) are that they are intensely *context-aware* and continually *adaptive* to changes in context.

- *M. Sama and D.S. Rosenblum are with the Department of Computer Science, University College London, Gower Street, London, WC1E 6BT UK. E-mail: {m.sama, d.rosenblum}@cs.ucl.ac.uk.*
- *S. Elbaum and Z. Wang are with the Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115. E-mail: {elbaum, zwang}@cse.unl.edu.*
- *F. Raimondi is with the School of Engineering and Information Sciences, Middlesex University, The Burroughs, London NW4 4BT UK. E-mail: f.raimondi@mdx.ac.uk.*

Manuscript received 26 June 2009; revised 16 Oct. 2009; accepted 21 Jan. 2010; published online 1 Mar. 2010.
Recommended for acceptance by G. Murphy and W. Schäfer.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2009-06-0168.
Digital Object Identifier no. 10.1109/TSE.2010.35.

The development and execution of CAAAs typically are supported by a *context-awareness middleware*, which employs two key components: 1) an event-driven *context manager* to collect and maintain context information that can be queried by a CAAA, and 2) an *adaptation manager* that maintains, evaluates, and applies a set of rules defining adaptive actions to take on behalf of the CAAA as context values provided by the context manager change [2], [4], [9], [10], [11], [19], [23]. Adaptation rules thus can define and automate a significant portion of a CAAA's behavior. For example, in *Locale*, one of the Android applications we referred to before, a set of rules is used to raise events that change the phone's behavior or notify external applications based on predefined situations (e.g., low battery) and on situations that a user can define (e.g., being in a meeting).

When an incorrect rule is triggered, or the correct one is not, a CAAA fails to adapt properly or behaves improperly. Again, in applications like *Locale*, failure reports often refer to issues associated with rules and their associated predicates that trigger undesired behavior (e.g., the wrong location is inferred when a rule that senses the company wireless network triggers the office profile before the rule that senses other wireless devices triggers the meeting profile) or fail to trigger appropriate behavior (e.g., a rule senses GPS, triggering the Outdoor profile, from which is not possible to make a transition to the Driving profile when a car bluetooth is later sensed).

Discovering such *adaptation faults* in CAAAs is challenging because of various confounding factors:

1. The space of rules becomes complex to analyze in the presence of shared context variables, concurrent triggering of rules, and priority ordering of rules.

2. The context variables are refreshed asynchronously at different rates by the middleware, causing artificial, transient inconsistencies between the external physical context and its internal representation within the application.
3. It is becoming increasingly common for CAAAs to let their users configure their behavior; this can lead to runtime failures due to buggy user-defined configurations.

In practice, we have observed that developers attempt to control these factors and the associated introduction of faults by constraining the rule space (e.g., disallowing disjunctions in rules), enforcing stronger priority orderings (e.g, requiring that each rule have a unique priority), restricting the adaptation actions that can be taken based on rules (e.g., disabling the power-off feature on GPS because it maybe used by other rules), and reducing the number or type of sensor information that can be considered (e.g., not letting the end user turn off particular services). In Locale, for instance, developers require users to assign priorities to rules, to express triggering conditions only as a conjunction of predicates, and to trigger a limited set of actions.

In this paper, we take a different approach that does not require such sacrifices, and instead enables the representation of CAAA behavior defined by rules in a form that is amenable to automated analysis for the detection of faults. Our approach consists of 1) a new model of CAAA behavior, 2) a set of patterns of faults that arise in the model, and 3) a family of static analysis algorithms offering different tradeoffs that detect these faults. We evaluate the approach by implementing it and assessing it through a case study of both a small but realistic CAAA and a range of synthetically generated CAAAs.

The rest of the paper is organized as follows: In Section 2, we introduce the CAAA *PhoneAdapter* to help motivate our work further. In Section 3, we present the finite-state model we use to represent the behavior of CAAAs and its encoding using different data structures. In Section 4, we describe a set of fault patterns for CAAAs and three different classes of algorithms for fault detection. We describe and discuss experimental results for these algorithms in Section 5. We discuss related work on fault detection in CAAAs in Section 6, and we conclude in Section 7.

2 An Example CAAA

In previous work, we developed a CAAA called *PhoneAdapter*, which is representative of the class of applications like *Locale* [17] that we introduced earlier in the paper, and which suffers from the kinds of faults that are peculiar to CAAAs and that our approach is able to detect [25]. 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 users selecting 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 and appointments calendar.

PhoneAdapter's adaptation rules define nine profiles:

1. *General*: the initial profile, which defines a user-specified default configuration, and which is 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 intensity and speaker volume when the user is outdoors;
6. *Jogging*: increases the backlight intensity and speaker volume and also activates vibration when the user is jogging;
7. *Driving*: connects to a car's handsfree communication system when the user is driving;
8. *DrivingFast*: diverts incoming calls to voice mail 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.

It is common in this application domain to assign a priority order to adaptation rules for safety or social reasons. For instance, in *PhoneAdapter*, 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).

Over several executions, we observed a number of nonobvious problems with *PhoneAdapter*. For instance, the profile *Sync* is never applied when the phone is adapted to *Home* or *Office*. Also, the rules that trigger adaptation to *Home* and *Office* can be satisfied simultaneously—which is possible if the user's office PC is discovered in his home or vice versa—causing nondeterministic adaptation to one of the two profiles.

But there are even more subtle problems. While the phone is in the process of adapting according to one rule, if some other rules are satisfied as well, then the phone can pass through a sequence of different profiles within the same context. This chain of adaptation causes multiple problems. In particular, the user can be annoyed by the multiple adaptations and, through the sequence of adaptations, the desired profile can become unreachable. For instance, when the user has left his office or house and has entered his car, the phone is supposed to adapt to *Driving*. However, if the Bluetooth sensor does not detect the handsfree system fast enough, the phone can adapt to *General*, and then to *Outdoor*. Then, when the user starts driving, the speed increases and the phone adapts from *Outdoor* to *Jogging*. From *Jogging* the phone cannot adapt to *Driving* even when the handsfree system finally is detected because the application cannot adapt from *Jogging* to *Driving* directly according to the rules.

It can also happen that, through a chain of adaptations, the predicates of certain rules are satisfied in such a way that they keep activating each other. For instance, from *Meeting*, when the meeting is over, the application adapts to *Office*, in which

another rule restores *Meeting*, leading to a loop because there exist particular inputs that can satisfy the necessary predicates simultaneously. In particular, the predicate $time > meeting\_start$ is always true after the meeting.

The timing of context updates can affect the triggering of rules in other ways. Since context updates occur asynchronously, the internal view of the context can become inconsistent temporarily, which causes the evaluation of rules to produce incorrect results or to trigger in a manner that violates their priorities or to become nondeterministic. For instance, if a meeting is scheduled but the user is going from his office to his car, the higher refresh rate of time relative to Bluetooth can force an adaptation to *Meeting* instead of *Driving*.

Existing analysis techniques do not differentiate predicates based on asynchronous input signals, such as GPS and Bluetooth. Such predicates could cause abnormal adaptation when updated asynchronously. Also, the space of rules becomes complex and nontrivial to analyze in the presence of shared context variables, of rules that can be concurrently triggered, and of rules with priorities. We therefore need systematic ways of discovering adaptation faults like the kinds described above. The approach we present in this paper aims to help software engineers (especially rule designers) analyze rules and detect faults in them automatically.

3 Modeling a CAAA

As described in previous sections, CAAAs typically are built using adaptation rules and, because of the complexity of the rule logic and the asynchronous updating of context variables, it is possible for the rules to embody faults. We therefore need a model of the rules that we can subject to automated analysis in order to identify the potential faults they contain. In addition, it is common practice to let users configure a CAAA partially or completely at runtime via the definition of policies and rules specified through a GUI. From this same GUI input, a model can be extracted and verified at runtime in order to give users immediate feedback on any problems with a newly specified configuration.

In this section, we describe the *Adaptation Finite-State Machine* or A-FSM, a new finite-state model that supports the analysis of potential faults in the adaptation rules of CAAAs. We also describe the derivation of two data structures that can be used to aid the analysis, State Matrices and Ordered Binary Decision Diagrams (OBDDs) [3]. The definition and derivation of the former is an extension of our previous work [25] and supports an explicit-state analysis of an A-FSM. The derivation of the latter is also an extension of our previous work [22] and supports a symbolic-state analysis of an A-FSM.

3.1 Formal Definition of A-FSMs

Context-awareness middleware lend themselves naturally to the derivation of finite-state models from the adaptation rules they support. In this section, we present formal definitions for the kind of rules that we described informally in Section 2, and then we formally define the A-FSM induced by such rules.

The *adaptation rules* for a CAAA form a set $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S} \times \mathcal{A} \times \mathbb{N}$, where $\mathcal{S}$ is a set of states characterizing the possible states of the CAAA (such as the phone profiles in *PhoneAdapter*), $\mathbb{N}$ is the set of natural numbers which represent the priorities of rules, $\mathcal{P}$ is the space of logical predicates definable over a set $\mathcal{C}$ of *propositional context variables* using conjunction, disjunction, and negation, and $\mathcal{A}$ is a set of actions that can be invoked upon entry to a state in $\mathcal{S}$ to modify the current values of propositional context variables.

A propositional context variable is the abstract representation of some corresponding relational expression over the set $\mathcal{C}_{sensed}$ of *sensed context variables* that are used by the CAAA and updated by the underlying middleware upon the occurrence of changes in context. An action then is simply a propositional context variable that asserts an effect of a rule (such as "turn off ringtones" or "enable Bluetooth"). Thus, $\mathcal{A} \subseteq \mathcal{C}$, and only those propositional context variables that represent equality tests on sensed context variables can be used as actions. As described later in this section and in Section 4, the propositional context variables are a cornerstone of our fault-detection algorithms.

Assignments of values to propositional context variables represent a suitable input for an A-FSM and, therefore, for its associated CAAA. Note that the variables derived from different relational expressions over the same sensed context variable are not necessarily independent. For instance, if variable c_1 encodes the fact that the speed is greater than 5 km/hr and c_2 that the speed is greater than 50 km/hr, then it is not possible for c_2 to be true and c_1 to be false simultaneously. Therefore, it is possible for the rule developer to provide additional *global constraints* that eliminate such inconsistent truth assignments; the analysis algorithms then treat such global constraints as additional conjuncts of all rule predicates, effectively reducing the state space that needs to be explored.

Let $R = (S, P, S', A, N)$ be a rule in $\mathcal{R}$, with S and $S' \in \mathcal{S}$, $P \in \mathcal{P}$, $A \in \mathcal{A}$, and $N \in \mathbb{N}$. We use the notation $P(S)$ to indicate the logical result of evaluating predicate P in state S, and we say that R becomes *active* upon entry to S. The semantics of R is that whenever S is the *current state* and $P(S)$ becomes true, then the CAAA transitions or *adapts* to the *destination state* S' and invokes A upon entry to S'. In this case, P is said to have been *satisfied*, R is said to be *triggered*, and S' becomes the new current state. Suppose there exists another rule $R_2 = (S, P_2, S'_2, A_2, N_2)$ such that at some point, both $P(S)$ and $P_2(S)$ are satisfied simultaneously. If $N < N_2$, then R is triggered instead of R_2. If $N > N_2$, then R_2 is triggered instead of R. If $N = N_2$, then the choice of which rule to trigger, R or R_2, is made nondeterministically. Thus, the smaller the priority value of a rule, the higher its priority.

In general, every transition to some state S and every change to a sensed context variable within a state S requires the (re)evaluation of the predicates of the active rules of S. Note that the occurrence of a context change may change the value of a propositional context variable without triggering the satisfaction of a rule predicate and a corresponding transition to a new state. Therefore, a state may experience many possible assignments of values to the propositional context variables and many different changes to those assignments may occur within the same state before a transition out of the state occurs.

We define an *Adaptation Finite-State Machine*, or *A-FSM* for short, as the finite-state machine $M = (\mathcal{S}, \delta, S_{initial}, \mathcal{S}_{final})$ derived from a set of rules $\mathcal{R}$, where $S_{initial} \in \mathcal{S}$ is the *initial state* of the CAAA and $\mathcal{S}_{final} \subseteq \mathcal{S}$ its *final states* (which are

TABLE 1
Adaptation Rules of PhoneAdapter

Rule Name	Current States	New State	Full Predicate	Simple Predicate	Priority
ActivateOutdoor	General	Outdoor	GPS.isValid() $\wedge$ $\neg$GPS.location()=home $\wedge$ $\neg$GPS.location()=office	$A_{gps} \wedge \neg B_{gps} \wedge \neg C_{gps}$	5
DeactivateOutdoor	Outdoor	General	$\neg$ActivateOutdoor	$\neg(A_{gps} \wedge \neg B_{gps} \wedge \neg C_{gps})$	5
ActivateJogging	Outdoor	Jogging	GPS.isValid() $\wedge$ GPS.speed()> 5	$A_{gps} \wedge D_{gps}$	5
DeactivateJogging	Jogging	Outdoor	$\neg$ActivateJogging	$\neg(A_{gps} \wedge D_{gps})$	5
ActivateDriving	General, Home, Office, Outdoor	Driving	BT=car_handsfree	A_{bt}	1
DeactivateDriving	Driving	General	$\neg$ActivateDriving	$\neg A_{bt}$	1
ActivateDrivingFast	Driving	DrivingFast	GPS.isValid() $\wedge$ GPS.speed()> 70	$A_{gps} \wedge E_{gps}$	0
DeactivateDrivingFast	DrivingFast	Driving	$\neg$ActivateDrivingFast	$\neg(A_{gps} \wedge E_{gps})$	0
ActivateHome	General	Home	BT=home_pc $\vee$ (GPS.isValid() $\wedge$ GPS.location()=home)	$B_{bt} \vee (A_{gps} \wedge B_{gps})$	5
DeactivateHome	Home	General	$\neg$ActivateHome	$\neg(B_{bt} \vee (A_{gps} \wedge B_{gps}))$	5
ActivateOffice	General	Office	BT=office_pc $\vee$ BT=office_pc_* $\vee$ (GPS.isValid() $\wedge$ GPS.location()=office)	$C_{bt} \vee D_{bt} \vee (A_{gps} \wedge C_{gps})$	5
DeactivateOffice	Office	General	$\neg$ActivateOffice	$\neg(C_{bt} \vee D_{bt} \vee (A_{gps} \wedge C_{gps}))$	5
ActivateMeeting	Office	Meeting	Time$>=$meeting_start $\wedge$ BT.count()$>= 3$	$A_t \wedge E_{bt}$	4
DeactivateMeeting	Meeting	Office	Time$>=$meeting_end	B_t	4
ActivateSync	General	Sync	BT=home_pc $\vee$ BT=office_pc	$B_{bt} \vee C_{bt}$	9
DeactivateSync	Sync	General	$\neg$ActivateSync	$\neg(B_{bt} \vee C_{bt})$	9

the states that have no active rules defined for them). The transition relation $\delta \subseteq \mathcal{S} \times \mathcal{R} \times \mathcal{S}$ is defined as follows:

$$\delta = \{(S, R, S') \mid \exists R = (S, P, S', A, N) \in \mathcal{R}\}.$$

3.2 Adaptation Rules and A-FSM of PhoneAdapter

Table 1 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. Also, the table shows no actions for the rules because actions are not employed in *PhoneAdapter*.

As shown in the table, *PhoneAdapter* adapts between nine different states according to 19 different rules[1] expressed over three different sensed context variables, namely BT (Bluetooth), GPS, 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 configuration variable *home*). This relational expression is represented throughout the rules by the propositional context variable B_{gps}.

We can identify the following global constraints for *PhoneAdapter*, which account for the facts that

1. checking context via GPS first requires GPS to be on,
2. locations are mutually exclusive,
3. speeds monotonically increase, and
4. the end time of a meeting is later than its start time:

$$\neg A_{gps} \Rightarrow (\neg B_{gps} \wedge \neg C_{gps} \wedge \neg D_{gps} \wedge \neg E_{gps})$$
$$(B_{gps} \Rightarrow \neg C_{gps}) \wedge (C_{gps} \Rightarrow \neg B_{gps})$$
$$E_{gps} \Rightarrow D_{gps}$$
$$B_t \Rightarrow A_t.$$

Fig. 1 depicts the A-FSM we derive from the adaptation rules of *PhoneAdapter*, with state *General* being its initial state.

3.3 Deriving State Matrices from an A-FSM

Our initial algorithms for detecting faults in an A-FSM employ enumerative exploration of an *explicit* representation of the state space of the A-FSM. The idea behind the algorithms is to analyze the adaptation behavior of the

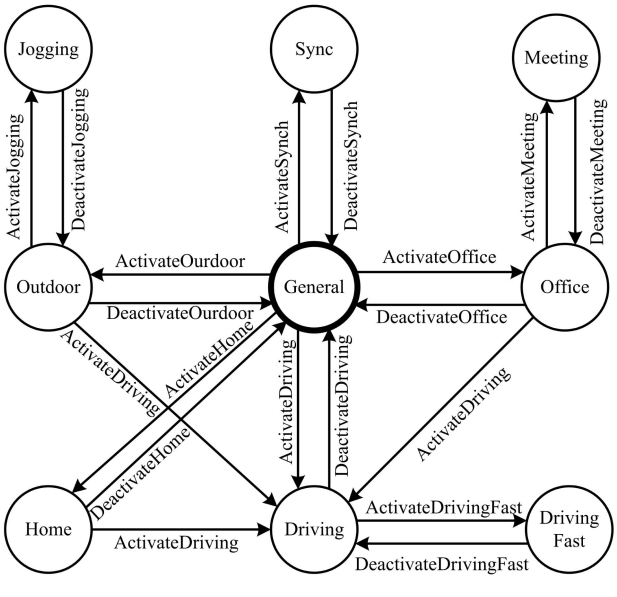


Fig. 1. A-FSM of PhoneAdapter.

1. Note that according to our definitions of $\mathcal{R}$ and M, the row named *ActivateDriving* in Table 1 actually represents four different rules, each being active in a different state; however, because the predicates, destination states, actions, and priorities of those rules are identical, we represent them in the table with a single rule name for simplicity, and we can exploit this similarity in the derived representations and algorithms.

A-FSM by simulating its execution starting from some chosen state using a chosen set of assignments of values to the propositional context variables. By repeating this process on all states and all possible assignments of values, we construct a derivative representation called a *state matrix* that is associated with each state S in M.

Conceptually, the state matrix of S enumerates *bit strings* for associated sets of rules. A bit string specifies a set of truth assignments to the set $\mathcal{C}$ of propositional context variables that can cause the predicates of the associated rules to become satisfied and the destination states of those rules to be entered. Bit strings that do not satisfy the predicates of any active rules of a state S are not included in the state matrix of S. Listing 1 depicts example entries from two state matrices for *PhoneAdapter*, the one for state *General* and the other for state *Outdoor*.

Listing 1. Example State Matrix Entries

state General:
(110000000000, [ActivateHome])
(110001000000, [ActivateHome, ActivateDriving])

state Outdoor:
(100100000000, [ActivateJogging])
(100101000000, [ActivateJogging, ActivateDriving])

Each line in the state matrix for a state S depicts a bit string of variable assignments (with 1 indicating true and 0 indicating false) along with the names of the active rules of S whose predicates become satisfied upon the variables obtaining those assignments. For the bit strings depicted here and elsewhere in the paper, the bit values are listed for the following order of propositional context variables: A_{gps}, B_{gps}, C_{gps}, D_{gps}, E_{gps}, A_{bt}, B_{bt}, C_{bt}, D_{bt}, E_{bt}, A_t, B_t. Note that while the order of variables does not matter, the same order must be used for all state matrices.

By constructing each state matrix over *all* propositional context variables, the size of the state matrix grows exponentially in the number of variables. To reduce the storage overhead of the state matrices and the execution time of some of our algorithms, we can exploit the fact that not all context variables are relevant to a particular state. We may therefore construct a *reduced state matrix* for each state that contains bits for only the relevant variables of the state (with the remaining bits essentially set to the value "don't-care"). We revisit this issue further in Section 4.2.

3.4 Deriving an OBDD from an A-FSM

In examining some of the open source CAAAs we found on the Web, we noticed that the number of states in a CAAA is generally limited, as each state typically corresponds to a different behavioral modality. On the other hand, we also noticed that the number of propositional context variables one would identify for these CAAAs can easily reach 15 to 20. The size of the state matrix and the time required to analyze it are, in the worst case, exponential in the number of variables, and so 20 variables may require too much memory to store all the state matrices.

To address this problem, we have defined algorithms that detect faults in an A-FSM through exploration of a *symbolic* representation of the state space of the A-FSM. In particular, we encode A-FSMs with OBDDs using a technique similar to those used by model checking tools [6]. The key idea is that states and transitions are encoded by means of Boolean formulas, and the Boolean formulas are then manipulated using OBDDs, which are a compact and efficient representation for Boolean formulas; typically, OBDDs allow for a reduction of several order of magnitudes in the size of a corresponding explicit-state representation.

The Boolean formulas encoding an A-FSM are expressed over a number of variables defined to represent the states and transitions of the A-FSM. The number of Boolean variables required to encode the states $\mathcal{S}$ of an A-FSM M is $K_S = \lceil log_2|\mathcal{S}| \rceil$; for instance, for the A-FSM of *PhoneAdapter*, $K_S = \lceil log_2|9| \rceil = 4$. Let $\overline{s} = (s_1, \ldots, s_{K_S})$ be the vector of K_S Boolean variables encoding $\mathcal{S}$; for instance, the vector $(0, 0, 0, 0)$ can be used to encode the first state (corresponding to the Boolean formula $\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4$), the vector $(0, 0, 0, 1)$ the second state, and so on. We introduce K_S more variables to encode the destination state of a transition by means of a vector $\overline{s'} = (s'_1, \ldots, s'_{K_S})$. We introduce $K_R = |\mathcal{R}|$ variables in a vector $\overline{r} = (r_1, \ldots, r_{K_R})$ to identify the different rules in $\mathcal{R}$; for the A-FSM of *PhoneAdapter* there are four such variables.[2] Finally, we introduce $K_C = |\mathcal{C}|$ variables in a vector $\overline{c} = (c_1, \ldots, c_{K_C})$ to represent the truth values of the Boolean propositional context variables and K_C more variables in a vector $\overline{c'} = (c'_1, \ldots, c'_{K_C})$ to represent actions; for the A-FSM of *PhoneAdapter* there are $2 * |K_C| = 24$ such variables.

Note that we do not use Boolean variables to encode priority since priority is handled as part of the derivation algorithm, as explained below.

The Boolean variables introduced above support the encoding of a *state activation BDD* that characterizes the evolution of the CAAA from a particular state as the transitions in that state are triggered. We identify the space of feasible inputs by computing GcBDD, a BDD containing the conjunction of all of the global constraints. Inputs outside of GcBDD violate at least one constraint and are unfeasible or unreachable. Algorithm 1 computes the state activation BDD as follows: For each priority level (Line 3), we compute the disjunction (Line 10) of the Boolean formulas encoding the active rules for the state at the priority level (Lines 7-9), where the formula encoding a rule is the conjunction of the formulas encoding the rule identity, the destination state, the *trigger inputs* and *future assignments*. Then, for each rule (Line 5), the trigger input BDD is computed by constraining the predicate of the current role with the negation of the predicates of higher priority rules (Line 7), and the future assignments BDD is obtained by applying the rules action to the trigger inputs (Line 8). The activiation BDD for the current priority level (Line 10) is the disjunction of the individual rule activation BDDs (Line 9). Note that for each priority level we also compute the conjunction of all rule predicates at that level (Line 6) and then save that conjunction for exclusion at lower priority levels (Line 12). The complexity of this algorithm is $O(P*|\mathcal{R}|)$, where P is the number of priority levels.

Algorithm 1. State Activation BDD Generation
Input: S: a state of the A-FSM;
GcBDD: global constraint BDD.
Output: StateActivationBDD: state activation BDD.
1: BDD Activation $= 0$

2. This is an example of how we can exploit the similarity of the four different rules labeled *ActivateDriving* in Table 1, effectively reducing 19 rules to 16 rules represented by four variables.

```
2: BDD exclusion = 0
3:   for i = VALUE_OF_HIGHEST_PRIORITY to
     VALUE_OF_LOWEST_PRIORITY do
4:     BDD exclusionAtPriority = 0
5:     for each Rule R ∈ S.getActiveRulesAtPriority(i) do
6:       exclusionAtPriority = exclusionAtPriority ∧
         R.getPredicate()
7:       BDD triggerInput=(R.getPredicate() ∧ ¬exclusion)
         ∧ GcBDD
8:       BDD futureAssignments
         R.applyActionToPredicate(triggerInput)
9:       BDD ruleActivation =
         triggerInput ∧ R.getEncoding() ∧
         R.getDestState().getFutureEncoding() ∧
         futureAssignments
10:      activation = activation ∨ ruleActivation
11:    end for
12:    exclusion = exclusion ∨ exclusionAtPriority
13: end for
```

In the rest of this paper, we use the following notation to represent BDDs:[3] $\langle V_i{:}0\rangle$ means that the variable V_i must be false to satisfy the enclosing BDD, while $\langle V_i{:}1\rangle$ means V_i must be true. We indicate a conjunction by placing multiple variable assignments within the same pair of angle brackets and disjunction by a sequence of angle-bracketed variable assignments. Thus, $\langle V_i{:}0, V_j{:}0\rangle$ indicates that both V_i and V_j must be false to satisfy the BDD, while $\langle V_i{:}0\rangle\langle V_j{:}0\rangle$ means that at least one of V_i and V_j must be false to satisfy the BDD.

The state activation BDD for state *Sync* of *PhoneAdapter* can be represented using this notation as follows:

$$\langle 0{:}0, 9{:}0, 16{:}0, 17{:}0, 18{:}0, 19{:}0, 20{:}1, 21{:}1, 22{:}1, 23{:}1, 24{:}0, 32{:}0\rangle$$

In this and all subsequent examples for *PhoneAdapter*, $i:1$ means that variable number i is true and $i:0$ means that variable number i is false. Variables 0-11 represent the propositional context variables. The current state is encoded with variables 12-15, which are absent here because, as mentioned above, a state activation BDD does not explicitly encode its associated state. The destination state is encoded in variables 16-19, and rules are encoded in variables 20-23. Thus, the state activation BDD of *Sync* encodes fact that when the predicate $\langle 0{:}0, 9{:}0\rangle$ is satisfied, rule $\langle 20{:}1, 21{:}1, 22{:}1, 23{:}1\rangle$ will adapt the A-FSM to state $\langle 16{:}0, 17{:}0, 18{:}0, 19{:}0\rangle$. Variables 24-35 represent the context after the action has been applied. Note that in this example their value is the same as variable 0-11 because *PhoneAdapter* does not employ any action.

The conjunction of a state activation BDD and a BDD encoding an assignment of values to the propositional context variables produces a *contextual BDD* encoding the adaptations triggered by that input from the state associated with the state activation BDD.

Finally, the *global activation BDD* $\mathcal{B}$ encodes all the states and transitions of the A-FSM M and is derived by first conjoining each state activation BDD with the encoding of its associated state, and then computing the disjunction of the resulting state-specific BDDs over all states S:

$$\mathcal{B} = \bigvee_{S \in \mathcal{S}} (S.\text{getEncoding}() \wedge S.\text{getStateActivationBDD}()).$$

3. This is the notation used by the open source library JavaBDD [15] to print BDDs as strings.

4 Detecting Faults in a CAAA

In this section, we describe the patterns of faults that we can identify within an A-FSM, and we provide algorithms for their automated detection. We present three kinds of detection algorithms: 1) *enumerative, explicit-state* algorithms that identify faults through analysis of the state matrices constructed for each state of an A-FSM, 2) *symbolic* algorithms that identify faults through analysis of the OBDDs derived from the A-FSM, and 3) *hybrid* algorithms that perform verification on the state activation BDDs of each individual state independently without using a global activation BDD.

In order to detect adaptation faults in a CAAA, we analyze its rules and derive an A-FSM to identify behavioral faults, which are faults in the logic of and relationships between rule predicates. We first describe the different patterns of behavioral faults in detail in Section 4.1. We then describe their enumerative algorithms in Section 4.2 and the symbolic and hybrid algorithms in Section 4.3.

4.1 Patterns of Behavioral Faults in a CAAA

The detection of behavioral faults is driven by the requirement that the rules of a CAAA and its A-FSM satisfy the following properties:

- **Determinism:** For each state in the A-FSM and each possible assignment of values to propositional context variables in that state, there is at most one rule that can be triggered. If the rules of a CAAA violate the Determinism property, we say that the rules contain a *Nondeterministic Activation fault*, a pattern of faults characterized by the presence of multiple active rules in the same state and with the same priority whose predicates can be satisfied by the same set of context updates. This kind of fault happens because unrelated predicates might not be mutually exclusive.
- **Rule Liveness:** For each state in the A-FSM 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. If the rules of a CAAA violate the Rule Liveness property, then we say that the rules contain a *Dead Predicate fault*, a pattern of faults characterized by the presence of an unsatisfiable predicate in the set of active rules of some state.
- **State Liveness:** For each state in the A-FSM, 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. If the rules of a CAAA violate the State Liveness property, then *all* of the active rules of the state contain a Dead Predicate fault, and we say that the set of rules contains a *Dead State fault*, a pattern of faults characterized by a deadlock state.
- **Stability:** The state of an A-FSM should not be dependent on the length of time a propositional context variable holds its value. A CAAA suffers from stability problems when a set of context updates can produce a sequence of adaptations such that the choice of which state ends the sequence depends on the duration with which some updated context

variable holds its value. More specifically, we say that the rules of a CAAA contain an *Adaptation Race fault* when the rules allow an indefinite number of adaptations to occur while some propositional context variable holds its value. If the adaptations form a cycle with an indefinite number of iterations, then we say that the rules contain an *Adaptation Cycle fault*. Often these patterns of behavior may produce multiple adaptations that merely annoy the user. Nevertheless, races can be dangerous because, if the affected variable holds its value long enough, then the CAAA will adapt to the last state of the race, but otherwise, if interrupted by an asynchronous update, the choice of last state will be random.

- **Reachability:** For every state, it is possible to reach the state from the initial state via some sequence of adaptations. If a state of the CAAA is not reachable (through any sequence of adaptations), we say that the rules contain an *Unreachable State fault*.

4.2 Fault Detection: Enumerative Algorithms

In this section, we present a set of algorithms based on the State Matrix representation described in Section 3.3 to detect the faults mentioned above.

4.2.1 Detection of Nondeterministic Activations

To detect this pattern of fault, we define an algorithm that explores each state matrix in the A-FSM and identifies the existence of bit strings that satisfy the predicates of multiple rules, also taking into account the priorities of the rules. Algorithm 2 detects this pattern of faults. For each state, if there are predicates of multiple highest-priority rules that can be satisfied in that state for the same bit string, then the adaptation is nondeterministic. In Line 2 of the algorithm, *S*.getStateMatrix() returns the local state matrix for state S, which contains a list of pairs (*bit string*, *satisfied rules*) for S. In Line 4, *S*.getSatisfiedRules(bitString) returns all of the highest priority satisfied rules for the current bit string. Finally, in Lines 5-7, if there is more than one such highest-priority rule, then the affected rules and state are reported along with the current bit string, which can be used to diagnose and eliminate the discovered fault.

Algorithm 2. Nondeterministic Activation Detection (Enumerative)
Input: M: an A-FSM.
Output: faultsVector: vector of detected faults.

```
1: for each State S in M do
2:   stateMatrix = S.getStateMatrix()
3:   for each BitString bitString ∈ stateMatrix do
4:     Vector rules = S.getSatisfiedRules(bitString)
5:     if rules.length() > 1 then
6:       faultsVector = faultsVector + {S, rules, bitString}
7:     end if
8:   end for
9: end for
```

Note that nondeterministic inputs could also be detected at the time the state matrix is derived from the A-FSM by requiring that each input satisfy the predicate of at most one rule. However, since the state matrices are used by several algorithms, we decided to separate their generation from their analysis.

For each reported pair of bit string and affected state, its Nondeterministic Activation fault can be eliminated in three ways: 1) by reformulating the predicates of the affected rules in such a way that at most one is satisfied by the bit string, 2) by splitting the affected state into multiple states, with the affected rules associated with different states, and 3) by assigning different priorities to the affected rules.

Algorithm 2 has worst-case complexity $O(|\mathcal{S}| * 2^{|\mathcal{C}|})$ since, for each state, it potentially enumerates all possible assignments of values to the propositional context variables. The remaining algorithms assume that the set of rules is deterministic; therefore, Nondeterministic Activation faults must be eliminated before applying them.

4.2.2 Detection of Dead Predicates and Dead States

In terms of state matrices, a Dead Rule fault is indicated by the absence of bit strings that satisfy the predicate of some rule. If, for a state S, all of its active rules are dead, then there exists a Dead State fault. Algorithm 3 checks, for each state, whether the predicates for all rules are satisfiable for at least one bit string. The algorithm iterates over all states and, for each state, executes two loops, the first one identifying the live rules and the second one reporting any remaining dead rules. Like Algorithm 2, this algorithm considers each state independently of the others, and so Line 2 retrieves the state matrix for state S. Line 3 initializes a collection with all the rules active in the current state. Line 9 removes the associated rule of each bit string in the state matrix, and there should be only one such rule since, as mentioned in Section 4.2.1, the algorithm assumes that any Nondeterministic Activation faults have been eliminated. Line 12 reports any rules that are not satisfied after searching through the state matrix. If all of the rules in a state are not satisfied, then in Line 15, the current state is reported as being dead. Algorithm 3 potentially explores all of the bit strings for the propositional context variables and thus has worst-case complexity $O(|\mathcal{S}| * (2^{|\mathcal{C}|} + |\mathcal{R}|))$.

Algorithm 3. Dead Predicate and Dead State Detection (Enumerative)
Input: M: an instance of A-FSM.
Output: faultsVector: vector of detected faults.

```
 1: for each State S in M do
 2:     StateMatrix stateMatrix = S.getStateMatrix()
 3:     Set untriggered = S.getActiveRules()
 4:     for each BitString bitString ∈ stateMatrix do
 5:     if untriggered == {} then
 6:         break
 7:     end if
 8:     Rule R = S.getSatisfiedRules(bitString)
          {Deterministic: R is unique}
 9:         untriggered = untriggered − R
10:   end for
11:   for each Rule R ∈ untriggered do
12:         faultsVector = faultsVector + {S, R}
            {Dead Rule}
13:   end for
14:   if untriggered == S.getActiveRules() then
```

```
15:            faultsVector = faultsVector + {S}
               {Dead State}
16:      end if
17:  end for
```

4.2.3 Detection of Adaptation Races and Cycles

If an A-FSM is deterministic, it is possible to search for Adaptation Races and Cycles by looking for *paths* of transitions among multiple states whose active rules contain predicates that are satisfied by the same bit string. Thus, in order to detect these faults, it is necessary to consider all propositional context variables, not just the subset relevant to the active rules of a single state.

Algorithm 4 checks, for each state S, whether a particular bit string in the state matrix of S can trigger a path of at least two transitions out of the state. Note that this algorithm is not local, in the sense that it iterates over paths through multiple states. Therefore, as mentioned in Section 3.3, the algorithm must consider the propositional context variables relevant to *all* of the states in the A-FSM, and thus it constructs a state matrix over all such variables (Line 2). Line 3 selects the next bit string to be searched. In Lines 4-5, the variable rvector is set up to store the affected rules of any detected Adaptation Race or Adaptation Cycle, while svector is set up to store the affected states. In Line 6, the variable isCycle is used to differentiate between Adaptation Races and Adaptation Cycles, and it is also used to force the algorithm to terminate. Lines 7-8 find the destination state for the highest-priority rule of the current bit string and store it in variable destState. Lines 10-15 set isCycle true because, after at least one iteration of the innermost enclosing loop, at least one repeated state has been detected at that point. Line 18 looks for highest-priority rules whose source state is destState and whose predicate is satisfied on the *same* bit string under consideration, thus indicating the presence of an Adaptation Race or Cycle. Line 19 updates the bit string with the appropriate corresponding actions. If a sequence of two or more states is detected after searching the bit strings for all active rules of the current state, then Lines 22-28 report the rules that form Adaptation Races and Adaptation Cycles, along with the bit strings that cause them.

Algorithm 4. Adaptation Race and Cycle Detection (Enumerative)
Input: M: an instance of A-FSM.
Output: faultsVector: vector of detected faults.

```
 1: for each State S in M do
 2:   StateMatrix stateMatrix = S.getStateMatrix().toGlobal()
 3:   for each BitString bitString ∈ stateMatrix do
 4:     Vector rvector = {} {explored rules}
 5:     Vector svector = {} {reached states}
 6:     Boolean isCycle = false
 7:     Rule R = S.getSatisfiedRules(bitString)
        {Deterministic: R is unique}
 8:     State destState = R.getDestState()
 9:     while destState != null ||destState ∉ svector do
10:       if destState ∈ svector then
11:         isCycle = true
12:         rvector = revector + R
13:         svector = svector + destState
14:         break
15:       end if
16:       rvector = revector + R
17:       svector = svector + destState
18:       Rule R1 = destState.getSatisfiedRules(bitString)
          {Deterministic: R1 is unique}
19:       destState = R1.getDestState()
20:     end while
21:     if svector.length() > 2 then
22:       if isCycle then
23:         faultsVector = faultsvector + {S, rvector,
            "cycle", bitString}
24:       else
25:         faultsVector = faultrVector + {S, rvector,
            "race", bitString}
26:       end if
27:     end if
28:   end for
29: end for
```

The worst-case complexity of Algorithm 4 is $O(|\mathcal{S}|^2 * 2^{|\mathcal{C}|})$ since it potentially explores all bit strings for paths potentially containing all states of the A-FSM.

4.2.4 Detection of Unreachable States

Detecting which states are reachable and which are not requires the exploration of the state machine starting from the initial state and then iteratively applying all adaptations to states that have not been reached earlier in the exploration.

Algorithm 5 starts with a collection of unexplored states containing all the states (Line 2) and a collection of states to explore that contains only the initial state (Line 1). In each iteration, in Line 5, we explore consecutively all the states in the current collection of states to explore by analyzing the state matrix and, in Line 13, by adding to the collection of states to explore any additional states contained in the state matrix under consideration. Additionally, in Line 6, we remove the current state from the collection of unreached states.

Algorithm 5. Unreachable State Detection (Enumerative)
Input: M: an A-FSM.
Output: faultsVector: vector of detected faults.

```
 1: Vector next = M.getInitialState()
 2: Vector toExplore = {}
 3: Vector unreached = M.getStates()
 4: while next ! = {} do
 5:   for each State S ∈ next do
 6:     unreached = unreached − S
 7:     StateMatrix stateMatrix = S.getStateMatrix()
 8:     for each BitString bitString ∈ stateMatrix do
 9:       R = S.getSatisfiedRules(bitString)
          {Deterministic: R is unique}
10:       toExplore = toExplore + R.getDestState() ∩
          unreached
11:     end for
12:   end for
13:   next = toExplore
14:   toExplore = {}
15: end while
16: for each State S ∈ unreached do
```

17: faultsVector = faultsVector + S
18: **end for**

Intuitively, in the first iteration, we explore the initial state, in the second one all of the states at distance one from the initial state, and in the ith iteration, we explore all of the states with distance $i-1$ from the initial state. The total number of iterations is equal to the maximum among the lengths of the shortest paths from the initial state to each other state. Since each state is explored at most once, the worst-case complexity of the algorithm is $O(|\mathcal{S}| * 2^{|\mathcal{C}|})$.

We can modify this algorithm to account for the possibility that Algorithm 3 does not report any Dead Rule faults. In such a situation, this algorithm simply can loop on all rules within each iteration for a state, which would reduce the complexity to $O(|\mathcal{S}| * |\mathcal{R}|)$.

4.3 Fault Detection: OBDD-Based Algorithms

We conjecture that OBDD-based approaches may scale better as the size of the A-FSM increases because, instead of considering all inputs, states, and rules, such approaches group all solutions symbolically using OBDDs. The disadvantage of this approach is that creating, using, and decoding example solutions from an OBDD all have a computational cost. In this section, we describe our two classes of OBDD-based algorithms: 1) *fully symbolic* algorithms that use a single OBDD derived from the whole A-FSM and 2) *hybrid* algorithms that visit each state individually and verify it symbolically. The idea behind the hybrid algorithms is that since the number of states is relatively small and since each state produces a different adaptive behavior, it makes sense to analyze individual states separately.

For both classes of algorithms, we detect faults by manipulating the OBDDs via the standard operations of conjunction (in which one OBDD is used to constrain the behavior represented by another), disjunction (to merge the behaviors represented by two OBDDs), subtraction (to remove the bhavior represented by one OBDD from another OBDD), existential quantification (to eliminate from an OBDD a set of variables that are irrelevant for a particular analysis task), and swapping (to convert occurrences of one class of variables within an OBDD to another and vice versa, which typically corresponds to executing a transition in the state machine from which the OBDD was derived). Such operations may result in an OBDD representing the value false (i.e., 0), which in our algorithm often indicates the existence of a fault.

4.3.1 Detection of Nondeterministic Activations

Fully symbolic approach. Algorithm 6 implements the detection of Nondeterministic Activations using a fully symbolic approach. This algorithm isolates satisfying inputs for all states (Line 4), detects which rules are satisfied by those, and if there is more than one, it reports a fault (Lines 13-22). The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|^3)$.[4]

Algorithm 6. Nondeterministic Activation Detection (Symbolic)
Input: AFSM M: an A-FSM encoded using OBDDs.
Output: Vector faults: vector of detected faults.
1: faults = {}
2: BDD gactivation = M.getGlobalActivation()
3: gactivation = gactivation.exist(M.getDestStateVars(), M.getActionVars())
{gActivation now contains states rules and the PCV assignments}
4: **for** each StateBDD $S \in M$ **do**
5: BDD activationInState = gactivation $\wedge$ S
6: activationInState = activationsInState.exist(M.getStateVars())
{activationInState contains rules and the PCV assignments for all the activations in state S}
7: **for** each RuleBDD $R \in S$.getActiveRules() **do**
8: BDD ruleActivation = activationInState $\wedge$ R
9: ruleActivation = ruleActivation.exist(M.getRuleVars())
{ruleActivation contains the PCV assignments}
10: BDD activationInStateForrule = activationInState $\wedge$ ruleActivation
{activationInStateForRule characterizes all the inputs triggering R in S}
11: Set faultyRules = {}
12: BDD faultyInput = 0
13: **for** each RuleBDD $R1 \in S$.getActiveRules() - {R} **do**
14: BDD overlap = activationInStateForRule $\wedge R1$
{overlap contains TCV assignments satisfying both R and $R1$ in S}
15: **if** overlap != 0 **then**
16: faultyInput = faultyInput $\vee$ overlap
17: faultyRule = faultyRule + $R1$
18: **end if**
19: **end for**
20: **if** (faultyRules.size() > 1) **then**
21: faults = faults + {S, faultyRules, faultyInput};
22: **end if**
23: **end for**
24: **end for**

Hybrid approach. Algorithm 7 explores all pairs of rules (Line 4 and Line 6) and generates a fault report if their predicates can be satisfied at the same time (Line 10). Note that if the predicates of three or more rules are satisfied, then the algorithm generates a solution for each of them. The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|^2)$.

Algorithm 7. Nondeterministic Activation Detection (Hybrid)
Input: AFSM M: an A-FSM encoded using OBDD.
Output: Vector faults: vector of detected faults.
1: faults = {}
2: **for** each StateBDD $S \in M$ **do**
3: Set unexploredRules = S.getActiveRules()
4: **for** RuleBDD $R1 \in S$.getActiverules() **do**
5: unexploredRules = unexploredRules - {$R1$}
6: **for** RuleBDD $R2 \in$ exploredRules **do**
7: **if** $R1$.getPriority() != $R2$.getPriority() **then**

4. The complexity of the exist() operation in Line 3 of Algorithm 6 is exponential in the number of variables quantified and thus linear in the number of rules.

```
 8:          continue;
             {Skips different priorities}
 9:      end if
10:      BDD overlap = R1.getPredicate() ∧
         R2.getPredicate()
         {only overlaps belonging to the activation BDD
         are faults}
11:      BDD fault = overlap ∧ S.getActivation() ∧
         R1.getEncoding()
12:      if fault != 0 then
13:        faults = faults + {(S, R1, R2, fault)}
14:      end if
15:    end for
16:  end for
17:end for
```

4.3.2 Detection of Dead Rules and Dead States

Fully symbolic approach. Algorithm 8 starts from the global activation BDD of the input A-FSM (Line 3), which encodes all of the information about which rules can be triggered. The algorithm then iterates over states and rules (Line 4 and Line 8). If a state does not exist in the global activation BDD (Line 6), then the state and all its active rules are dead and are added to the fault set in Lines 6-10. Otherwise the decoding adds as a fault each single rule that is active in the state at the current iteration but does not appear in the global activation BDD. The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|)$.

Algorithm 8. Dead Rule and Dead State Detection (Symbolic)
Input: AFSM M: an A-FSM encoded using OBDD.
Output: Vector deadStates, deadRules: vectors of detected faults.

```
 1: deadStates = {}
 2: deadRules = {}
 3: BDD gActivation = M.getGlobalActivation();
 4: for each StateBDD S ∈ M do
 5:   BDD stateActivation = gActivation ∧ S
 6:   if stateActivation == 0 then
 7:     deadStates = deadStates + {S}
 8:     for each RuleBDD R ∈ S.getActiveRuled() do
 9:       deadRules = deadRules + {(S, R)}
10:     end for
11:   else
12:     for each RuleBDD R ∈ S.getActiveRules() do
13:       BDD ruleActivation = stateActivation ∧ R
14:       if ruleActivation == 0 then
15:         deadRules = deadRules + {(S, R)}
16:       end if
17:     end for
18:   end if
19: end for
```

Hybrid approach. Algorithm 9 starts from the activation of each state in Line 3, checking for dead states in Lines 5-9. If the state is live, then the algorithm decodes each single rule from the state activation BDD (Lines 11-16). Like its fully symbolic counterpart, the complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|)$, but over a smaller data structure.

Algorithm 9. Dead Rule and Dead State Detection (Hybrid)
Input: AFSM M: an A-FSM encoded using OBDDs.
Output: Vector deadStates, deadRules: vectors of detected faults.

```
 1: deadStates = {}
 2: deadRules = {}
 3: for each StateBDD S ∈ m do
 4:   BDD stateActivation = S.getActivation();
 5:   if stateActivation == 0 then
 6:     deadStates = deadStates + {S}
 7:     for each RuleBDD R ∈ S.getActiveRules() do
 8:       deadRules = deadRules + {S, R}
 9:     end for
10:   else
11:     for each RuleBDD R ∈ S.getActiveRules() do
12:       BDD ruleActivation = statActivation ∧ R;
13:       if ruleActivation == 0 then
14:       deadRules = deadRules +{(S, R)}
15:       end if
16:     end for
17:   end if
18: end for
```

4.3.3 Detection of Adaptation Races and Cycles

Fully symbolic approach. Algorithm 10 swaps present and destination states (Line 4), applies the action (Line 5), and checks whether there exists a destination state that is not "stable" in the sense that it contains no active rule that can be triggered on the same input that triggered a transition to that destination state (Lines 3-5). This is encoded in the BDD in Line 6, which has to be decoded to extract the faults. The complexity of this algorithm is $O(|\mathcal{S}| + |\mathcal{R}| + (|\mathcal{S}| * |\mathcal{R}|))$.

Algorithm 10. Adaptation Race and Cycle Detection (Symbolic)
Input: AFSM M: an A-FSM encoded using OBDDs.
Output: Vector faults: vector of detected faults.

```
 1: faults = {}
 2: BDD gactivation = M.getGlobalActivation()
 3: BDD futureActivation =
    g.activation.exist(M.getStateVars(). M.getRuleVars(),
    M.getPcvVars()))
 4: futureActivation = future.Activation.swap
    (M.getActionVars(), M.getStateVars())
 5: futureActivation = futureActivation.swap
    (M.getActionVars(), M.getStateVars())
 6: BDD fault = gActivation ∧ futureActivation
 7: for each StateBDD s ∈ M do
 8:   for each ruleBDD rule ∈ S.getActiveRules() do
 9:     StateBDD D = R.getDestState()
10:     BDD stateRuleDestFilter = S ∧ R ∧ D
11:     BDD faultInstance = fault ∧ stateRuleDestFilter
12:     if faultInstance != 0 then
13:       faults = faults + {(S, R, D)}
14:     end if
15:   end for
16: end for
```

Hybrid approach. Algorithm 11 iterates over the set of all states (Line 2) and for each state checks whether any of the rules (Line 4) is not stable in the destination state (Line 10). The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}| * (|\mathcal{S}| + |\mathcal{R}|))$.

Algorithm 11. Adaptation Race and Cycle Detection (Hybrid)
Input: AFSM M: an A-FSM encoded using OBDD.
Output: Vector faults: vector of detected faults.

```
 1: faults = {}
 2: for each StateBDD S ∈ M do
 3:     BDD activation = S.getActivation()
 4:     for each RuleBDD R ∈ S.getActiveRules() do
 5:         BDD ruleActivation = activation ∧ R
 6:         ruleActivation = activation.exist
            (M.getDestStateVars(),
            M.getRuleVars(), M.getStateVars())
 7:         ruleActivation = ruleActivation.swap
            (M.getActionVars(), M.getPcvVars())
            {compares with the destination activation}
 8:         StateBDD D = R.getDestState()
 9:         BDD futureActivation = D.getActivation()
10:         BDD faultInstance = futureActivation ∧
            ruleActivation
11:         if faultInstance != 0 then
12:             faults = faults + {(S, R, D)}
13:         end if
14:     end for
15: end for
```

4.3.4 Detection of Unreachable States

Fully symbolic approach. Algorithm 12 starts from the initial state (Line 2) and iteratively removes new states from the set of unexplored states (Line 6) until nothing is left to explore. Then, the algorithm iterates over all of the states and checks whether any state is in the set of unexplored states (Line 10). This algorithm has a worst-case complexity of $O(|\mathcal{S}| * (|\mathcal{S}| + |\mathcal{R}| + 2^{|\mathcal{C}|}) + |\mathcal{S}|)$.

Algorithm 12. Unreachable State Detection (Symbolic)
Input: AFSM M: an A-FSM encoded using OBDD.
Output: Vector faultsVector: vector of detected faults.

```
 1: faultsVector = {}
 2: BDD toExplore = M.getInitialState().getActivation()
 3: BDD unexplored = 1
    {All the states}
 4: BDD gActivation = M.getGlobalActivation();
 5: while toExplore != 0 do
 6:     unexplored = unexplored − toExplore
 7:     toExplore = gActivation ∧ toExplore
 8:     toExplore = toExplore.exist(M.getStateVars()
        M.getRuleVars(), M.getPcvVars())
 9:     toExplore = toExplore.swap(M.getDestStateVars(),
        M.getStateVars())
10:     toExplore = toExplore ∧ unexplored
11: end while
12: for StateBDD S ∈ M do
13:     BDD faultInstance = unexplored ∧ S
        {If a state exists then it is unreachable}
14:     if faultInstance ! = 0 then
15:         faults = faults + {S}
16:     end if
17: end for
```

Hybrid approach. Similarly to Algorithm 12, Algorithm 13 starts from a set of states to explore (Line 2) initialized with the initial state and in Lines 8-13 iteratively explores all the states reachable from the states explored in the current iteration. In contrast to Algorithm 12, this algorithm loops using the A-FSM instead of the BDDs and only uses the BDDs to check which states are reached in each iteration in Line 9. The complexity of this Algorithm is $O(|\mathcal{S}|)$.

Algorithm 13. Unreachable State Detection (Hybrid)
Input: M: an A-FSM encoded using OBDDs.
Output: Vector unreached: vector of detected faulty states.

```
 1: unreached = {M.getStates()}
 2: Set toExplore = {M.getInitialState()}
 3: Set toExploreNext = {}
 4: while toExplore ! = 0 do
 5:     unreached = unreached - toExplore
 6:     for each StateBDD S ∈ toExplore do
 7:         BDD activation = S.getActivation()
 8:         for each StateBDD K ∈ unreached do
 9:             BDD reached = activation ∧
                K.getDestStateEncoding()
10:             if reached ! = 0 then
11:                 toExploreNext = + K
12:             end if
13:         end for
14:     end for
15:   toExplore = toExploreNext
16:   toExploreNext = {}
17: end while
```

5 Study

In this section, we explore three research questions:

- RQ1: How effective are the analysis algorithms in detecting faults? To begin answering this question, we use our algorithms to analyze *PhoneAdapter*, the CAAA described in Section 2 and modeled in Section 3. In Section 5.2, we provide a summary of the faults found, highlight some of the interesting faults and their impact, and describe the reports provided by the algorithms.
- RQ2: How do the algorithms scale as the complexity of the A-FSM increases? In Section 5.3, we again use *PhoneAdapter* and also a suite of synthetic A-FSMs to measure the performance of the algorithms as we increase the number of states, rules, and variables.
- RQ3: What is the memory consumption behavior of the algorithms in limited memory environments such as would be found in end-user devices? To answer this question, in Section 5.4, we use *PhoneAdapter* in a series of configurations with small and gradually decreasing amounts of memory to measure the threshold under which execution aborts or performance degrades excessively.

TABLE 2
Faulty Input Configurations Reported for PhoneAdapter

State	Nondeterministic Adaptations	Dead Predicates	Adaptation		Unreachable States
			Races	Cycles	
General	37	1	45	13	0
Outdoor	3	0	135	23	0
Jogging	0	0	97	19	0
Driving	0	0	36	13	0
DrivingFast	0	0	58	19	0
Home	0	0	76	19	0
Office	0	0	29	1	0
Meeting	0	0	32	1	0
Sync	0	0	27	5	1

5.1 Study Infrastructure

As mentioned before, our primary analysis artifact is *PhoneAdapter*, which is implemented on top of ContextNotifier, a J2ME rule-based adaptation framework and middleware for CAAAs [23] and targeted for deployment on the Nokia N95 cell phone. We ran the application and its adaptation rules within TestingEmulator, an emulator we have built for CAAAs [24]. Our implementation of *PhoneAdapter* has 9 states and 19 rules, and it uses 12 propositional context variables.

To further assess the algorithms' performance, in addition to *PhoneAdapter*, we generated a set of synthetic A-FSMs of increasing complexity with various numbers of states, rules, and variables. To make the A-FSMs more realistic, we constrained the generation process as follows:

1. All states are the destination state of at least one rule (with the exception of the initial state), which avoids having unreachable states that are trivially detected. The number of active rules in each state is between one and eight, following what we have observed in practice in publicly available tools.
2. The active rules of each state all have different priorities in order to guarantee that the generated A-FSM is deterministic since determinism is a prerequisite for most of the algorithms.
3. The number of variables used in each rule predicate is less than a specified maximum, and all variables are used in at least one predicate. In the experiment, we imposed a maximum of five variables per predicate in order to generate predicates of a complexity that represents what we have observed in practice.
4. The variables used in each predicate are composed using a combination of negation, conjunction, and disjunction according to the following probabilities: Each variable has a 50 percent probability of being negated, and conjunctions or disjunctions are selected with a 50 percent probability.

Our algorithm implementations are Java 6-compliant.[5] For the symbolic and hybrid algorithms, we relied on the JavaBDD library version 2.0 and on its default Java implementation of the OBDD library [15].

In addition to running the algorithms on *PhoneAdapter* to count and compare the number of detected faults, we also ran two performance experiments with multiple executions of the different algorithms. The first experiment was designed to answer RQ2, to measure the scalability of the algorithms. The second experiment was designed to answer RQ3, to measure the algorithms' behavior in the presence of a limited amount of main memory (i.e., JVM heap). Both of the experiments were executed on an Intel i7 920 equipped with 6 GB of DDR3 RAM running Ubuntu 9.04 with the OpenJDK 6 at 64 bit. We set a time-out of 30 minutes for all analyses. For the first experiment, we configured the Java virtual machine with a heap size of 4 GB. For the second experiment, we decreased the JVM heap over the range from 32 MB down to 2 MB.

5. The full code for our implementation and the models used in the evaluation are available online [21].

5.2 Detecting Faults in PhoneAdapter

Table 2 summarizes the number of faulty input configurations found in *PhoneAdapter*. The first column shows the nine states of the A-FSM. The remaining columns present the number of faulty configurations found when we apply the fault detection algorithms. In general, each of the three set of algorithms detects the same faulty configurations but reports them in a different fashion, as described in greater detail below.

5.2.1 Detecting Nondeterministic Adaptations

This pattern of faults appears when predicates of multiple rules with the same priority and active in the same state can be satisfied by the same assignments to the propositional context variables.

By applying the nondeterministic adaptation algorithms (Algorithms 2, 6, and 7), we obtain the results shown in column "Nondeterministic Adaptations" in Table 2. Most of the nondeterministic adaptations were found in state *General*. The analysis for this state discovered 37 different assignments to propositional context variables for GPS and Bluetooth that simultaneously satisfy the predicates for rules *ActivateOffice*, *ActivateHome*, and *ActivateOutdoor*.

Although the different types of algorithms detect the same faults, the way they report it varies enough to warrant presenting further details. Note that for brevity, we employ the data structures as presented in Sections 3.3 and 3.4, but more user-friendly reports can be generated by expanding these structures.

The enumerative Algorithm 2 returns a list of all the variable assignments and corresponding rules that can lead to a nondeterministic adaptation fault. The following is a sample report:

```
=========================================
Nondeterministic Adaptations [Enumerative]:
=========================================
============== State: General ==============
101**0100*** [ActivateOffice, ActivateHome]
100**0010*** [ActivateOffice, ActivateOutdoor]
100**0110*** [ActivateOffice, ActivateHome,
              ActivateOutdoor]
[...]
```

The symbolic Algorithm 6 organizes the results in a similar but more compact way, by returning an OBDD describing the fault. An OBDD is returned for each rule containing nondeterministic activation faults, along with an indication of the interfering rules. Such OBDDs encode both inputs and rules because the nondeterminism exists among a set of rules. The following is a sample fault report:

```
=========================================
Nondeterministic Adaptations [Symbolic]:
=========================================
============== State: General ==============
[ActivateOutdoor, ActivateHome, ActivateOffice]
     OBDD = <0:0, 3:0, 4:1, 6:0, 9:1, 11:0,
               20:0, 21:1, 22:1, 23:0>
             <0:1, 3:0, 4:1, 6:0, 9:0, 11:0,
               20:0, 21:0, 22:1, 23:0>
             <0:1, 3:0, 4:1, 6:0, 9:1, 11:0,
               20:0, 22:1, 23:0>
[ActivateHome, ActivateOutdoor, ActivateOffice]
    OBDD = <...>
[ActivateOffice, ActivateOutdoor, ActivateHome]
    OBDD = <...>
============== State: Outdoor ==============
[DeactivateOutdoor,ActivateJogging]
    OBDD = <...>
[ActivateJogging,DeactivateOutdoor]
    OBDD = <...>
```

The hybrid Algorithm 7 does not return a single group of interfering rules, but instead detects pairs of such rules. While the same fault can involve more than one pair, the OBDDs containing the faults are simpler because they contain only the assignments of propositional context variables that cause the faults. Faults are reported as in the following example:

```
=========================================
Nondeterministic Adaptations [Hybrid]:
=========================================
============== State: General ==============
[ActivateOutdoor, ActivateHome]
   OBDD = <0:1, 3:0, 4:1, 11:0>
[ActivateOutdoor, ActivateOffice]
   OBDD = <3:0, 4:1, 9:1, 11:0>
[ActivateHome, ActivateOffice]
   OBDD = <0:0, 3:1, 4:1, 5:0, 9:1>
          <0:0, 3:1, 4:1, 5:1, 9:0, 11:1>
          <0:0, 3:1, 4:1, 5:1, 9:1>
          <0:1, 4:0, 9:1><0:1, 4:1, 5:0, 9:1>
          <0:1, 4:1, 5:1, 9:0, 11:1>
          <0:1, 4:1, 5:1, 9:1>
============== State: Outdoor ==============
[DeactivateOutdoor, ActivateJogging]
    OBDD = <2:1, 3:0, 4:1, 11:1>
           <2:1, 3:1, 4:1>
```

For the same detected faults in *PhoneAdapter*, Algorithm 7 generates four OBDDs with an average of 5.25 internal nodes, while Algorithm 6 reports five OBDDs with an average of 20.2 internal nodes. Thus, the latter are roughly four times as large as the former.

Developers interested in understanding which rules are interfering can benefit from the output produced by the symbolic Algorithm 6 because it groups interfering rules. The enumerative Algorithm 2 shows faulty inputs more clearly but may be too verbose. Algorithm 7 lies somewhat between these extremes.

One common way to eliminate these faults is to assign distinct priorities to the affected rules in line with the behavior desired for PhoneAdapter. In our case, we decreased ActivateOutdoor priority to 6 and increased ActivateOffice priority to 4.

5.2.2 Detecting Dead Predicates

This pattern of faults consists of predicates that cannot be satisfied by any assignments to the propositional context variables, or predicates that could be satisfied but are always preempted by predicates of rules with higher priority.

The column "Dead Predicates" of Table 2 shows that one fault was detected in rule *ActivateSync* in state *General*. By examining this state, we note that it is possible for predicates of the rule *ActivateSync* to be satisfied by certain assignments, but such assignments also satisfy *ActivateOffice* and *ActivateHome*, which has higher priority, and thus *ActivateSync* is never triggered.

All three algorithms for identifying dead predicate faults report these errors in the same format: affected state and rule. As this pattern of fault is caused by the absence of satisfying inputs, there is no error trace to report apart from the actual state and rule.

5.2.3 Detecting Adaptation Races and Cycles

Faults associated with races and cycles result from assignments that induce sequential or cyclic adaptations where the last state of the sequence depends on how long an assignment holds.

The algorithms identified many races that produce fluctuations in the states and may disturb the user temporarily. For example, if a user starts to drive and accelerates quickly, the application may or may not reach *DrivingFast* (a state in which all calls are diverted), depending on whether the high speed is maintained long enough to enable the transition from *General* to *Driving* and then to *DrivingFast*.

There are also races generating unwanted behaviors from which the application cannot recover quickly. For instance, while in *Driving*, if Bluetooth loses the connection with the handsfree system, the phone will adapt through *General* to *Outdoor* and then *Jogging*, from where it is impossible to reactivate *Driving* even if the handsfree system is redetected. (The adaptation rules are defined in such a way that the rule that activates *Driving* never triggers while *Jogging* is active.) Finally, all the detected cycles are

TABLE 3
Performance Results for the Three Classes of Algorithms (milliseconds)

	Enumerative					Symbolic					Hybrid				
PhoneAdapter	82	3	1	207	2	16.9	5.9	0.9	10.5	1.3	13.1	0.2	0.8	8.9	0.8
(10,40,10)	12	6.5	4	13085	8.3	24.3	19.3	1.4	26.4	5.5	13.45	0.15	0.6	10.55	0.5
(10,40,15)	327.3	49.1	39.5	127442	131.8	130.4	109.6	9.5	240.4	37.1	18.6	0.4	1.3	17.2	0.7
(10,40,20)	OUT OF MEMORY					955.2	1033.0	158.0	2654.5	369.4	24.8	0.7	2.0	11.0	1.0
(10,40,25)	OUT OF MEMORY					13522.0	19179.9	2384.8	48032	6939.3	25.1	0.5	1.2	13.0	0.9
(10,40,30)	OUT OF MEMORY					OUT OF MEMORY					26.7	0.3	1.5	10.7	0.9
(10,40,35)	OUT OF MEMORY					OUT OF MEMORY					28.0	1.6	2.8	9.9	1.8
(10,40,40)	OUT OF MEMORY					OUT OF MEMORY					35.3	0.6	5.3	15.7	2.6
(10,45,15)	TIMEOUT					157.2	130.0	14.5	314.6	61.0	22.4	0.4	2.0	12.6	1.1
(10,60,20)	OUT OF MEMORY					4343.4	2616.7	492.4	12099	1366.1	51.7	0.9	3.9	19.8	1.6
(10,75,25)	OUT OF MEMORY					OUT OF MEMORY					54.4	2.2	16.6	29.5	2.6
(10,90,30)	OUT OF MEMORY					OUT OF MEMORY					71.2	2.0	43.9	73.7	4.3
(10,30,10)	11.0	5.4	2.7	349.5	4.6	18.2	15.9	1.8	22.3	3.7	15.7	0.3	0.6	6.2	0.8
(15,45,15)	204.1	41.6	23.6	46718	39.6	223.0	153.6	13.5	336.6	54.0	20.6	1.1	0.9	14.9	1.1
(20,60,20)	OUT OF MEMORY					7737.4	4607.9	374.0	11207	1469.2	33.5	0.2	1.0	6.7	1.7
(25,75,25)	OUT OF MEMORY					OUT OF MEMORY					34.3	1.3	3.1	11.4	5.5
(30,90,30)	OUT OF MEMORY					OUT OF MEMORY					35.7	0.6	3.6	11.4	6.3
(35,105,35)	OUT OF MEMORY					OUT OF MEMORY					39.7	0.7	3.4	10.8	8.4
(40,120,40)	OUT OF MEMORY					OUT OF MEMORY					47.6	4.1	4.9	11.3	17.2
(45,135,45)	OUT OF MEMORY					OUT OF MEMORY					66.3	0.9	2.8	15.5	8.6
(100,300,100)	OUT OF MEMORY					OUT OF MEMORY					169.7	1.5	6.3	18.9	33.3
(200,600,200)	OUT OF MEMORY					OUT OF MEMORY					1024.6	2.6	16.8	43.9	191.8

MG: model generation, ND: Nondeterministic Activation, DR: Dead Rule, AR: Adaptation Race, US: Unreachable State.

produced by the rules *ActivateMeeting* and *DeactivateMeeting* when the state is *Office* and $Time >= meeting\_end$.

As shown in the columns "Adaptation Races/Cycles" of Table 2, these are the most common faults in *PhoneAdapter* and in our experience, some of the hardest to detect without some form of automated support. With the enumerative Algorithm 4, faults are reported as a list of inputs and activations, for instance:

```
=========================================
Adaptation Races and Cycles [Enumerative]:
=========================================
============== State = General ============
*****010**** Race (Driving,DeactivateDriving)
              ->(General,ActivateSynch)
              -> Sync
[...]
000****11111 Cycle (Outdoor,DeactivateOutdoor)
              ->(General,ActivateOffice)
              ->(Office,ActivateMeeting)
              ->(Meeting,DeactivateMeeting)
              -> Office
[...]
```

Note that Algorithm 4 explores all sequences to identify races and cycles, and for each race or cycle it reports every assignment that can trigger it. The other two algorithms, the symbolic Algorithm 10 and the hybrid Algorithm 11, report the state where a race begins, the rule causing the race, and to which destination state and with which assignment, all in the form of OBDD. *PhoneAdapter* has 15 different sets of this kind. Developers interested in knowing *if* races exist in a CAAA will find this compact output more useful than the one produced by Algorithm 4. On the other hand, developers wanting to understand *how* the CAAA will race or cycle will find the error report of Algorithm 4 more suitable.

```
=========================================
Adaptation Races and Cycles [Hybrid]:
=========================================
race: (Driving,DeactivateDriving) -> General
OBDD = <0:0, 3:0, 4:0, 6:0, 9:1, 16:0, 17:1,
        18:1, 19:0, 20:0, 21:1, 22:1, 23:0>
       <0:0, 3:0, 4:1, 5:0, 6:0, 9:0, 10:0,
        11:0, 16:1, 17:0, 18:0, 19:0, 20:0,
        21:0, 22:0, 23:0>
       [...]
race: (Outdoor,DeactivateOutdoor) -> General
OBDD = <0:0, 3:0, 4:0, 6:0, 9:1, 16:0, 17:1,
        18:1, 19:0, 20:0, 21:1, 22:1, 23:0>
        <0:0, 3:0, 4:1, 5:0, 6:0, 9:1, 11:1,
        16:0, 17:1, 18:1, 19:0, 20:0, 21:1,
        22:1, 23:0>
        <0:0, 3:0, 4:1, 5:1, 6:0, 11:1, 16:0,
        17:1, 18:1, 19:0, 20:0, 21:1, 22:1,
        23:0>
        [...]
race: (General,ActivateOffice) -> Office
OBDD =< [...]>

[...]
```

5.2.4 Detecting Unreachable States

States are unreachable when all of the rules of which they are a destination state are dead. Running this algorithm on an A-FSM without dead rules always returns an empty set of faults. As shown in the last column of Table 2, state *Sync* is unreachable because the only rule with state *Sync* as destination state, *ActivateSync*, is a dead rule. All three algorithms simply return a list of unreachable states.

5.3 Comparing the Algorithms' Performance

Table 3 reports the performance of the algorithms running on the A-FSM of *PhoneAdapter* and the randomly generated CAAAs. The size of the random CAAAs is reported in the first column as a triple specifying the number of states, rules, and variables. Performance times are reported in milliseconds, represent the average over 10 runs, and include the

TABLE 4
Comparison of OBDD Complexity (PhoneAdapter)

	State Activation OBDD (state average)	Global Activation OBDD (total)
Nodes	24.2	935
Paths	12.2	6008

time both detect faults and (where applicable) to decode the generated OBDD. For each algorithm, we measured the time required to generate the model in memory, which is shown in the columns labelled "MG." For the enumerative algorithms, the "MG" value corresponds to the time to create the matrix. For the symbolic and hybrid algorithms, the value corresponds to the time to compute the state activation OBDDs, plus, in the fully symbolic case, the global activation OBDD.

The studied A-FSMs are meant to expose the performance of the algorithms under various configurations. In our study, we first manipulated the number of variables while keeping the number of rules and states constant. Second, we manipulated the number of variables and rules while maintaining the number of states constant. Third, we manipulated all three factors.

Overall, we find that the enumerative algorithms do not scale to larger CAAAs and are generally slower than the others, especially in the detection of races and cycles. The symbolic algorithms scale only slightly better than the enumerative ones. The hybrid algorithms are the fastest, which was unexpected since it is normally assumed that symbolic approaches are more efficient than ones including any type of enumeration like the hybrid one. Further study reveals two reasons that make the hybrid approach faster for our particular domain.

First, the OBDDs manipulated by the hybrid algorithms are generally smaller than the OBDDs manipulated by the symbolic algorithms, so even if the number of loops in the hybrid algorithms is greater, the smaller size of the OBDDs makes the algorithms faster overall. We can measure the complexity of the state activation OBDDs by the number of nodes and the number of paths that satisfy them [3]. Table 4 reports the average number of nodes and paths for the state activation OBDDs of *PhoneAdapter* (with the average taken over all states) and compares those with the total number of nodes and paths of the global activation OBDD. We note that the symbolic algorithms deal with an OBDD of almost two orders of magnitude larger than the hybrid ones. This is due to the fact that, in the worst case, the complexity of an OBDD increases exponentially in the number of variables used.

Second, while a fully symbolic approach can be slightly faster in detecting faults, it returns *all* of the faults encoded as a single OBDD, which then must be decoded; this decoding process is a further bottleneck of the fully symbolic approach. In contrast, with the hybrid algorithms all of the faults are reported as soon as they are found, and despite the fact that the number of iterations is greater, the size of the OBDDs is smaller, and there is no decoding required. More generally, we note that we evaluated our algorithms only on a single CPU core. Better results can be obtained with the hybrid approach by exploring each state in a dedicated thread where possible.

5.4 Executing Algorithms in a Limited Memory Environment

In the experiments described thus far, we let the algorithms use all of the memory at our disposal (4 GB). To better understand the trade-offs between space and speed, we next explored the performance of the algorithms on *PhoneAdapter* in more constrained memory settings. Such settings are also interesting in practice since more CAAAs are enabling end users of limited memory devices to redefine adaptation rules, which can also be faulty and targets of our analysis.

Table 5 summarizes our findings. We observe that, for all algorithms, as memory decreases, the time required to generate the model increases slightly, in part because the garbage collector is invoked more often. All algorithms can run successfully on *PhoneAdapter* with 24 MB of memory. However, with less than 18 MB, symbolic algorithms to detect races and unreachable states (Algorithms 10 and 12) run out of memory. Since the symbolic algorithms use a data structure that is an order of magnitude bigger than the one used by the hybrid algorithms, it was expected that they would have run out of memory sooner. If the available memory is decreased to 16 MB, then hybrid Algorithm 11 (the only hybrid algorithm that operates on multiple states) runs out of memory as well. If the memory is further reduced, all symbolic algorithms fail. With the exception of Algorithms 11 and 4, all the other hybrid and enumerative algorithms manage to run with just 2 MB of memory. Overall, the enumerative approach with its small data structures seems to be a better fit for constrained settings. This result seems to contradict the one reported in Table 3 in which the enumerative approach had a largest footprint. We note, however, that the configurations used in this study are small enough that an adaptation state can be represented by all algorithms. Instead, the peak of memory consumption occurs the analysis of such representations.

TABLE 5
Performance while Testing PhoneAdapter with Limited Memory (milliseconds)

	Enumerative					Symbolic					Hybrid				
Memory (MB)	MG	ND	DR	AR	US	MG	ND	DR	AR	US	MG	ND	DR	AR	US
32	24.6	0.6	0.4	66.5	1.1	24.8	16.25	0.85	21.2	1.15	19.8	0.25	0.65	30.3	0.7
24	26.2	0.8	0.3	67.7	1.3	20.7	23.8	0.8	17.25	1.2	23.1	0.25	0.65	31.55	0.8
18	32.3	0.7	0.6	80.6	0.7	31.6	15.4	0.85	OOM	OOM	24.3	0.3	0.65	71.45	0.75
16	42.0	1.6	0.4	83.2	0.6	28.45	31.6	0.85	OOM	OOM	29.05	0.45	0.7	OOM	0.7
12	63.4	0.55	0.9	87.3	0.6	41.25	OOM	OOM	OOM	OOM	34.2	0.15	0.75	OOM	0.75
8	269.5	1.2	0.65	OOM	1.45	54.4	OOM	OOM	OOM	OOM	44.85	0.4	0.7	OOM	0.7
4	268.1	1.65	0.55	OOM	1.65	55.6	OOM	OOM	OOM	OOM	47.55	0.35	0.55	OOM	0.9
2	276.0	1.6	0.6	OOM	1.7	51.15	OOM	OOM	OOM	OOM	48.7	0.15	0.65	OOM	0.65

MG: model generation, ND: Nondeterministic Activation, DR: Dead Rule, AR: Adaptation Race, US: Unreachable State.

We finalize the performance study by qualifying our findings. We observe that today's CAAAs are mostly of the size of *PhoneAdapter* and that simpler algorithms like the enumerative ones may suffice. Still, the growing complexity of CAAAs we are witnessing indicates that the application of more efficient algorithms will become increasingly important. At the same time, we must acknowledge that the quality of the fault reports and their corresponding impact on the developer isolating and fixing the faults are critical performance aspects that still need to be evaluated.

6 Related Work

Our work is broadly related to research efforts associated with the analysis of finite-state machines and rules, and with some more recent domain-specific efforts aimed at the validation and verification of CAAAs.

6.1 Analysis of FSMs and Rules

One of the most common underlying adaptation mechanisms used in CAAAs, and our own chosen representation for CAAAs, is a finite-state machine. In general, finite-state models have been used extensively to represent and verify system properties. In the context of requirements engineering, for example, Heitmeyer et al. use finite-state models to discover inconsistencies in SCR specifications [14] and Heimdahl and Leveson use finite-state models to discover inconsistencies in RSML specifications [13]. While the classes of inconsistencies that they detect are characteristic of requirements specifications, the fault patterns that we detect are characteristic of CAAAs. Thus, although there are some similarities between our fault patterns and their classes of inconsistencies, certain classes appear to arise only in CAAAs, notably the instability faults described in Section 4.2.3.

Finite-state representations are also used by other verification systems, such as model checkers, to verify temporal properties of concurrent systems [5], and by static checkers in general to identify likely faults (such as code vulnerabilities [29]). Similarly, our analysis starts with a finite-state model, which we have extended to incorporate context information and whose analysis we have tailored to focus on properties that are of particular relevance to CAAAs. We utilize multiple analysis techniques, each of which employs its own specific model derived from our extended finite-state model plus its own support data structures.

Efforts for analyzing rule-based systems, not necessarily based on finite-state machines, have focused on the development of coverage criteria to assess and guide the testing activity as it exercises rules or rule chains [1], [12]. In contrast, our analyses are static and operate on a model to identify variables that may trigger multiple rules concurrently or multiple commutations of variables within the same rule, which constitute adaptation faults.

6.2 Testing and Verification of CAAAs

In the last few years, several domain-specific analysis techniques have emerged targeting the CAAAs.

Some of these efforts aimed at adapting existing testing techniques to address the validation challenges of CAAAs, including the larger input space, the continuous input streams that drive them, and the inconsistencies that arise from noisy contextual information [18], [27], [28]. Although we share those other approaches' goal of detecting faults in CAAAs, our approach is fundamentally different, employing static analysis of the adaptation models to check for the properties specified in Section 3, while the other, testing-based efforts are centered primarily on test generation and runtime analysis of the application. As such, our analysis will be able to detect all instances of the faults that violate the specified properties, but it may also report false positives as the model may overapproximate the behavior of the application. Existing testing approaches, on the other hand, will miss some faults but will only report true positives.

A second focus area for analysis techniques of CAAAs has been the verification of mobility properties. Roman et al. defined Mobile UNITY, an extension of the UNITY notation and proof logic to the verification of mobile systems [20]. Given mobile applications specified in Mobile UNITY and associated specified properties, Mobile UNITY is able to verify the application against the specified properties. This work mainly focuses on verifying the mobility aspects of the application, whereas our approach is concerned with discovering faults in an application's context-awareness and adaptation behavior.

The third emerging analysis area for CAAAs has been inconsistency detection, which is increasingly relevant in the presence of more and richer sensors. Xu and Cheung have proposed inconsistency detection in context-aware applications whereby patterns identify conflicts among context inputs at runtime before they are fed to an application [32], [33]. The patterns are defined by engineers based on their understanding of relevant mathematical and physical laws. Their work focuses mainly on verifying the runtime correctness of the context inputs themselves. In contrast, our analysis evaluates the adaptation predicates of the rules to check whether there are faults in their formulation. We also consider mathematical and physical laws, but we capture them in global constraints to identify intrinsic relationships among context variables, which are used to prune infeasible combinations of variables explored in the analysis.

Last, the requirements engineering community has also started to address the problem of inconsistent and missing context readings in adaptive environments. Domain- specific languages like RELAX [31] enable the specification of uncertainty so that, for example, an adaptation rule can include *ApplyOffice = AS CLOSE AS POSSIBLE TO location(office)*. If adopted, such specifications languages would facilitate the definition of more precise rules, but methodologies for their verification remain to be investigated.

7 Conclusion and Future Work

In this paper, we have described several contributions to the validation of modern software applications. First, we have defined a formal model of a key complex behavioral characteristic, namely, *context-driven adaptation*, of an increasingly large and important class of computing applications, namely CAAAs. Second, we have identified a large set of patterns of frequently occurring adaptation faults that are not easily detected by existing validation techniques. Third, we have defined three classes of algorithms for automatically detecting occurrences of the fault patterns, enabling software engineers to increase the quality and robustness of their applications. Fourth,

we have quantitatively compared the three classes of algorithms and have discussed their merits and drawbacks.

We illustrated and evaluated these contributions on a small but realistic CAAA called *PhoneAdapter*, which exhibits many of the fault patterns we have identified and whose faults are automatically detected by our algorithms, and we also have evaluated our algorithms on a range of synthetically generated CAAAs.

While we defined our A-FSM initially to validate CAAAs, we have observed that the model can be applied more generally to other domains that can benefit from the use of FSMs, whereby domain-specific fault patterns can be identified and detected via a context-oriented characterization of the faults. Indeed, in parallel work, we have applied the A-FSM model to context-aware adaptive Web-Service composition in an effort to detect deadlocked or otherwise faulty compositions [7].

In future work, we plan to apply *constraint propagation* over variables that are affected by physical laws such as those captured in the global constraints described in Section 3.1 and illustrated in Section 3.2 (such as the fact that time is always increasing, and battery level is usually decreasing). This will allow us to reduce the number of possible paths to explore in an A-FSM, which is crucial for faults affecting multiple states. We also will extend our approach to accommodate end users' needs by introducing interactive "wizards" for improving adaptation rules and eliminating their faults.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) under CAREER Award 0347518, by the United Kingdom Engineering and Physical Sciences Research Council (EPSRC) for project UbiVal under grants EP/D077273/1, EP/E006191/1, and EP/F013442/1, by the Royal Society under a Wolfson Research Merit Award for David Rosenblum, and by the European Commission Framework Programme 6 in project PLASTIC (IST-2005-026955). This paper is a revised version of an earlier paper presented at the ACM SIGSOFT 2008 International Symposium on the Foundations of Software Engineering (FSE 2008) [25]. This new paper presents a new data structure and algorithms for the symbolic analysis of the A-FSM model presented in the FSE 2008 paper. An early version of this symbolic support was presented at the International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS 2008) [22].

REFERENCES

[1] V. Barr, "Applications of Rule-Base Coverage Measures to Expert System Evaluation," *Proc. Nat'l Conf. Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conf.*, pp. 411-416, July 1997.

[2] G. Biegel and V. Cahill, "A Framework for Developing Mobile, Context-Aware Applications," *Proc. Second IEEE Int'l Ann. Conf. Pervasive Computing and Comm.*, pp. 361-365, Mar. 2004.

[3] R.E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293-318, 1992.

[4] L. Capra, W. Emmerich, and C. Mascolo, "Carisma: Context-Aware Reflective Middleware System for Mobile Applications," *IEEE Trans. Software Eng.*, vol. 29, no. 10, pp. 929-945, Oct. 2003.

[5] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 2, pp. 244-263, 1986.

[6] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*. MIT Press, 1999.

[7] J. Cubo, M. Sama, F. Raimondi, and D.S. Rosenblum, "A Model to Design and Verify Context-Aware Adaptive Service Composition," *Proc. IEEE Int'l Conf. Services Computing*, Sept. 2009.

[8] "Track Your Mobile Carbon Footprint. Reduce and Offset It. Inspire Others to Do the Same," http://www.ecorio.org/, Oct. 2009.

[9] P. Fahy and S. Clarke, "CASS—Middleware for Mobile Context-Aware Applications," *Proc. MobiSys Workshop Context Awareness*, pp. 304-308, June 2004.

[10] J. Floch, "Theory of Adaptation," Deliverable D2.2, MADAM Project, http://www.ist-music.eu/MUSIC/madam-project/madam-deliverables/techreportreference.2007-04-13.0451108510, 2006.

[11] T. Gu, H.K. Pung, and D.Q. Zhang, "A Middleware for Building Context-Aware Mobile Services," *Proc. IEEE Vehicular Technology Conf.*, pp. 2656-2660, May 2004.

[12] U.G. Gupta, "Automatic Tools for Testing Expert Systems," *Comm. ACM*, vol. 5, pp. 179-184, May 1998.

[13] M.P. Heimdahl and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Trans. Software Eng.*, vol. 22, no. 6, pp. 363-377, June 1996.

[14] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, pp. 231-261, 1996.

[15] JavaBDD, Version 2.0, http://javabdd.sourceforge.net/, June 2009.

[16] Life360: Live Confidently, http://www.life360.com/, Oct. 2009.

[17] Locale, http://www.twofortyfouram.com/, June 2009.

[18] H. Lu, W.K. Chan, and T.H. Tse, "Testing Context-Aware Middleware-Centric Programs: A Data Flow Approach and an RFID-Based Experimentation," *Proc. Int'l Symp. Foundations of Software Eng.*, pp. 242-252, Nov. 2006.

[19] A. Ranganathan and R.H. Campbell, "A Middleware for Context-Aware Agents in Ubiquitous Computing Environments," *Proc. ACM/IFIP/USENIX Int'l Middleware Conf.*, pp. 143-161, June 2003.

[20] G.-C. Roman, P.J. McCann, and J.Y. Plun, "Mobile UNITY: Reasoning and Specification in Mobile Computing," *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 3, pp. 250-282, July 1997.

[21] M. Sama, "CAAA Verifier," http://code.google.com/p/caaaverification/, 2010.

[22] M. Sama, F. Raimondi, D.S. Rosenblum, and W. Emmerich, "Algorithms for Efficient Symbolic Detection of Faults in Context-Aware Applications," *Proc. First Int'l Workshop Automated Eng. of Autonomous and Run-Time Evolving Systems*, pp. 1-8, Sept. 2008.

[23] M. Sama and D. Rosenblum, ContextNotifier, http://code.google.com/p/contextnotifier/, June 2009.

[24] M. Sama and D. Rosenblum, TestingEmulator, http://code.google.com/p/testingemulator/, June 2009.

[25] M. Sama, D.S. Rosenblum, Z. Wang, and S. Elbaum, "Model-Based Fault Detection in Context-Aware Adaptive Applications," *Proc. 16th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 261-271, Nov. 2008.

[26] Softrace, http://www.softrace.net/, Oct. 2009.

[27] T. Tse, S. Yau, W. Chan, H. Lu, and T. Chen, "Testing Context-Sensitive Middleware-Based Software Applications," *Proc. Int'l Computer Software and Applications Conf.*, pp. 458-465, Sept. 2004.

[28] Z. Wang, S. Elbaum, and D.S. Rosenblum, "Automated Generation of Context-Aware Tests," *Proc. Int'l Conf. Software Eng.*, pp. 406-415, May 2007.

[29] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," *ACM Trans. Software Eng. and Methodology*, vol. 16, no. 4 article 14, 2007.

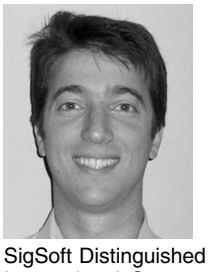
[30] The Mobile App for Nightlife, http://wertago.com/, Oct. 2009.

[31] J. Whittle, P. Sawyer, N. Bencomo, B.H. Cheng, and J.-M. Bruel, "RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems," *Proc. 17th IEEE Int'l Conf. Requirements Eng.*, pp. 79-88, 2009.

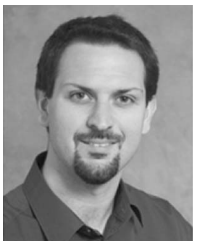
[32] C. Xu and S.C. Cheung, "Inconsistency Detection and Resolution for Context-Aware Middleware Support," *Proc. Joint 10th European Software Eng. Conf. and 13th ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 336-345, Sept. 2005.

[33] C. Xu, S.C. Cheung, and W.K. Chan, "Incremental Consistency Checking for Pervasive Context," *Proc. Int'l Conf. Software Eng.*, pp. 292-301, May 2006.

Michele Sama received the BSc and MSc degrees from the University of Bologna. He is a PhD student at University College London. His research interests include testing and architectures for mobile and real-time systems. He is funded under a grant from the EPSRC for the UbiVal project. He has also been an intern in software test engineering at Google, where he developed tools for test automation of Java ME applications.



Sebastian Elbaum received the systems engineering degree from the Universidad Catolica de Cordoba, Argentina, and the PhD degree in computer science from the University of Idaho. He is an associate professor at the University of Nebraska-Lincoln. His research aims to improve software dependability through testing, monitoring, and analysis. He is the recipient of the US National Science Foundation (NSF) Career award, an IBM Innovation award, and two ACM SigSoft Distinguished Paper awards. He was the program chair for the International Symposium of Software Testing and Analysis, program cochair for the Symposium of Empirical Software Engineering and Measurement, coeditor for the *Information and Software Technology* Journal, and he is an associate editor of the *ACM Transactions on Software Engineering and Methodology*. He is a cofounder of the EUSES Consortium to support end-user programmers and the E2 Software Engineering Group at UNL. He is a member of the IEEE.



Franco Raimondi received the BSc and MSc degrees in physics from the University of Milan and the PhD degree in computer science from University College London in 2006. He is currently a senior lecturer at Middlesex University, London. His research interests include model checking for extensions of temporal logics, multiagent systems, and modal logics. He is the author of various publications in software engineering, AI, and formal methods. He is actively involved in the organization of various conferences and workshops.

David S. Rosenblum is professor of software systems and head of Software Systems Engineering at the University College London and director of the EPSRC-funded project UbiVal. His research interests are in architectures and testing of large-scale software systems. He was general chair of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007), and he is an associate editor of the *ACM Transactions on Software Engineering and Methodology* (ACM TOSEM). In 2002, he received the ICSE Most Influential Paper award for his ICSE 1992 paper on assertion checking, and in 2008, he received the first ACM SIGSOFT Impact Paper award with Alexander L. Wolf for their ESEC/FSE 1997 paper on Internet-scale event notification. He is a chartered engineer, a fellow of the IEEE, and chair of the ACM Special Interest Group in Software Engineering (ACM SIGSOFT).

Zhimin Wang received the PhD degree in computer science from the University of Nebraska-Lincoln. He is a software test engineer at Microsoft Corporation. His research interests include program analysis and testing.