Model-Based Fault Detection in Context-Aware Adaptive Applications

Michele Sama and David S. Rosenblum
Dept. of Computer Science
University College London
London, UK
{m.sama,d.rosenblum}@cs.ucl.ac.uk

Zhimin Wang and Sebastian Elbaum
Dept. of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
{zwang,elbaum}@cse.unl.edu

ABSTRACT

Applications running on mobile devices are heavily *context-aware* and *adaptive*, leading to new analysis and testing challenges as streams of context values drive these applications to undesired configurations that are not easily exposed by existing validation techniques. We address this challenge by employing a finite-state model of adaptive behavior to enable the detection of faults caused by (1) erroneous adaptation logic, and (2) asynchronous updating of context information, which leads 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 that we detect automatically by analyzing the system's adaptation model. We illustrate our approach on a simple but realistic application in which a cellphone's configuration profile is changed automatically based on the user's location, speed and surrounding environment.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous—*rule-based programming*; D.2.4 [**Software/Program Verification**]: Model checking,Validation; D.2.5 [**Testing and Debugging**]: Diagnostics; F.1.1 [**Models of Computation**]: Automata—*finite-state models*; I.5.1 [**Models**]: Structural—*fault patterns*

General Terms

Algorithms, Design, Verification

Keywords

Adaptation, context-awareness, fault detection, hazards, mobile computing, model-based analysis, ubiquitous computing

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-59593-995-1 ...$5.00.

The growing popularity of handheld devices such as cellphones, 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 their environment. Key characteristics of these emerging *Context-Aware Adaptive Applications* (CAAAs) are that they are heavily *context-aware* and continually *adaptive* to changes in context.

The development and execution of CAAAs typically is 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 [5, 8, 9, 10, 20, 22]. Adaptation rules thus define a significant portion of the CAAA's behavior.

When an incorrect rule is triggered, or the correct one is not, a CAAA fails to adapt properly or behaves improperly. Discovering such *adaptation faults* in CAAAs is challenging because of two 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; and (2) the context variables are refreshed asynchronously at different rates by the middleware, causing artificial inconsistencies between the external physical context and its internal representation within the application.

Our work aims to detect faults in CAAAs by defining a formal, finite-state model of adaptation rules and then analyzing the model for adaptation faults. The states of the model represent equivalence classes of stable configurations of context values, while the transitions represent the satisfaction of rule predicates. Satisfaction of a predicate thus triggers an adaptive change from one state to another and may also involve the execution of associated actions. This model represents the execution of a CAAA by explicitly connecting context updates with adaptations and helps to isolate adaptation faults caused by erroneous rule predicates and asynchronous context updates. To automate the detection of inconsistences, we identified a number of *adaptation fault patterns* representing commonly occurring classes of adaptation faults, and we automatically analyze the model for their existence.

The remainder of the paper is organized as follows. Section 2 discusses related work on fault detection in CAAAs. Section 3 describes *PhoneAdapter*, a representative CAAA that we have built and tested as a motivating application for our work. Section 4 describes the finite-state model we use to represent the adaptive behavior of CAAAs. Section 5 describes the adaptation fault patterns, which we organize into two classes, *behavioral faults* and *context hazards*, and it describes the algorithms we employ to detect them. Section 6 describes results from the application of our algorithms to the *PhoneAdapter* application. Finally, Section 7 concludes the paper with a summary of our contributions and a discussion of our plans for future work.

2. RELATED WORK

Among validation techniques for context-aware mobile applications, Roman et al. define Mobile UNITY, an extension of the UNITY notation and proof logic to the verification of mobile systems [21]. 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.

Xu and Cheung propose inconsistency detection in context-aware applications whereby patterns identify conflicts among context inputs at run-time before they are fed to an application [29, 30]. The patterns are defined by engineers based on their understanding of relevant mathematical and physical laws. This work focuses mainly on verifying the correctness of the context inputs themselves. In contrast, we assume context inputs to be consistent and then evaluate them within the predicates of adaptation rules to check whether there are faults in the formulation or triggering of rules. We also consider intrinsic relationships among context variables, in particular the delays resulting from asynchronous update of context variables having different refresh rates, and we identify the adaptation faults that may arise as a result.

Several researchers from the testing community have begun to target the validation of CAAAs [17, 25, 27]. Although we share their goal of detecting faults in CAAAs, our approach is fundamentally different, employing static analysis of adaptation models, while theirs are centered primarily on test selection and runtime analysis.

Efforts for testing rule-based systems (the main adaptation mechanism used by CAAAs) have focused on the development of coverage criteria for exercising single rules or rule chains [2, 11]. In contrast, we statically analyze a set of rules to identify variables that may trigger multiple rules concurrently or multiple commutations of variables within the same rule, leading to adaptation failures.

When designing sequential digital circuits in which multiple signals are input to a network of logic gates, engineers must avoid *hazards* and *races*, which may produce incorrect outputs. Unger [26] and Hauck [12] summarize timing problems in sequential circuits and describe techniques for predicting and correcting them. Our work is based in part on the insight that processing of context inputs induces similar kinds of hazards and races in CAAAs, and we provide an appropriate formulation of such faults using our adaptation models. In particular, in our work we detect faults in which the choice of adaptation rules to trigger and the order in which to trigger them depends on how long a context input value holds.

Timing problems in real-time systems have been analyzed by using methods based on specialized finite-state models. Alur and Dill propose a timed automata model that incorporates time constraints for specifying real-time systems [1]. Timed automata have been utilized as well by several test case generation techniques, which exploit timing constraints specified between actions or events [7, 15, 16, 18]. In a similar way, we apply constrains not only to time, but to all kinds of context inputs utilized by the adaptation rules.

The work by Nilsson and Offut [19] uses patterns of potential faults to detect missed scheduling of sporadic and periodic time-critical tasks. Our approach employs a similar idea, in which the different refresh rates of asynchronously updated context variables may trigger incorrect decisions in an application's adaptation behavior. We analyze the impact of asynchronous updates on the evaluation and triggering of adaptation rules, allowing us to determine where faults can occur.

Finite-state models have been used extensively to represent and verify properties of systems. Some work similar to our own has been done in the context of requirements engineering. 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 instability faults (described below in Section 5.1.3) and context hazards (described below in Section 5.2). Finally, we also note that model checkers use finite-state representations extensively to model concurrent systems and verify their temporal properties [6], and static checkers for metaprogramming languages use such models to detect potential vulnerabilities in generated code [28]. Similarly, our analysis operates on a finite-state model, but we have extended it to incorporate context information and have tailored the analysis to focus on properties that are of particular relevance to CAAAs.

3. AN EXAMPLE CAAA

In this section we present *PhoneAdapter*, an application that suffers from the kinds of faults peculiar to CAAAs that our approach is able to detect.

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.

PhoneAdapter's adaptation rules define nine profiles:

1. *General*: the initial profile, which defines a user-speci-

fied 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 the car's handsfree communication system when the user is driving;
8. *DrivingFast*: diverts calls when the user is driving fast;
9. *Sync*: periodically synchronizes personal information on the phone with the user's home or office PC when the phone is not in use and the PC is discovered via Bluetooth.

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

Over several executions, we observed a number of non-obvious 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 the home location, 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, 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 contradictory rules are satisfied and 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. For instance, 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. For instance, if a meeting is scheduled but the user is going from the 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 non-trivial 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. Our approach aims to help software engineers (especially rule designers) analyze rules and detect faults in them automatically.

4. MODELING A CAAA

In order to detect adaptation faults in CAAAs, we need a suitable abstract model of their adaptive behavior. Fortunately, the style of rule specification used in typical context-awareness middleware [3, 5, 8, 9, 10, 20] and the style we have been using in our own work lends itself naturally to the derivation of a finite-state model that is suitable for fault detection, such as the ones used to detect faults in SCR specifications [14]. This section presents formal definitions of our adaptation rules and the finite-state model we derive from them, while Section 5 describes the algorithms we use to analyze the model for faults.

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

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. The set $\mathcal{C}$ thus represents the set of semantically different relational expressions over sensed context variables that are used to define the rule predicates. As described later in this section and in Section 5, the propositional context variables are a cornerstone of our fault-detection algorithms.

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 new 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 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.

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 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:

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

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.[1] Also, the table does not show the actions of rules, since they play no role in fault detection.

As shown in the table, *PhoneAdapter* adapts between nine different states according to 19 different rules expressed over three different sensed context variables, namely BT (Bluetooth), GPS and Time, which are monitored via 12 propositional context variables representing the 12 different relational expressions in which the sensed context variables are used. For example, one such relational expression is *GPS.location()=home*, which tests whether the location sensed by the phone's GPS sensor corresponds to the user's home location (stored in variable *home*). This relational expression is represented throughout the rules by the propositional context variable B_{gps}. Figure 1 depicts the A-FSM we derive from the adaptation rules of *PhoneAdapter*, with state *General* being its initial state.

In order to simplify the detection of faults in an A-FSM M, 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* and 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 target states of those rules to be entered. Bit strings that do not satisfy the predicates of any active rules are not included in the state matrix of S. In implementing the analysis of M, we can use various projections of the state matrices to dramatically reduce the number of variable assignments analyzed and consequently the analysis time. Listing 1 depicts example entries from two state matrices for *PhoneAdapter*, the one for state *General* and the one for state *Outdoor*.

[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 and priorities of those rules are identical, we represent them in the table with a single rule name for simplicity.

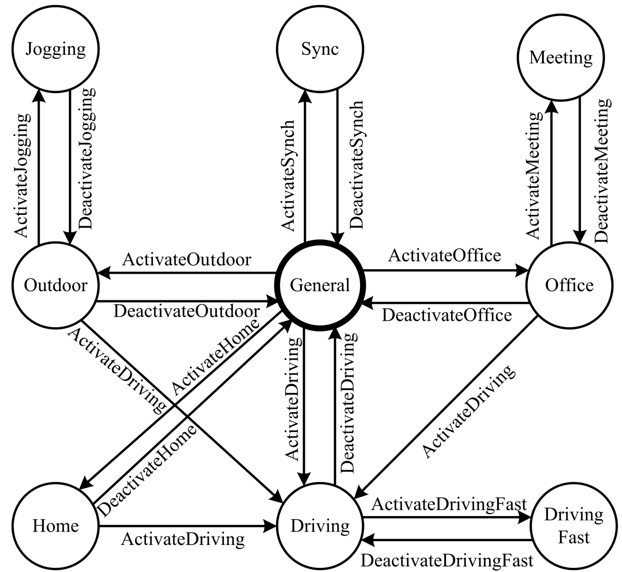

Figure 1: A-FSM of PhoneAdapter.

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 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.

In summary, given a set of adaptation rules $\mathcal{R}$ used to define the context-aware adaptive behavior of a CAAA, we derive an associated A-FSM M, each of whose states has an associated state matrix. The next section describes algorithms that use $\mathcal{R}$, M and the state matrices to identify adaptation faults in the CAAA.

5. DETECTING FAULTS IN A CAAA

In order to detect adaptation faults in a CAAA, we analyze its rules and derived A-FSM for two families of faults, (1) *behavioral faults*, which are faults in the logic of and relationships between rule predicates, and (2) *context hazards*, which are faults that arise due to the asynchronous nature of context updates and the varying refresh rates at which they occur. In this section we describe these two families of

Table 1: Adaptation Rules of PhoneAdapter

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

faults in detail and describe algorithms for their detection. Then in Section 6 we describe the results we obtained from applying the algorithms to *PhoneAdapter*.

5.1 Behavioral Fault Detection

The detection of behavioral faults is driven by the requirement that the rules 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.
- **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.
- **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.
- **Stability:** The state of an A-FSM is not dependent on the length of time a propositional context variable holds its value.
- **Reachability:** For every state, it is possible to reach the state from the initial state via some sequence of adaptations.

Each of these properties gives rise to one or more *fault patterns* characterizing the situations in which the associated property is violated. We discuss these properties in detail below and present algorithms for their detection, and then in Section 6 we present examples of their occurrence in *PhoneAdapter*. Due to space constraints, and because violations of the Reachability property seem relatively rare, we discuss that property no further in this section.

Algorithm 1 NondeterministicActivationDetection

Input: M: an A-FSM.
Output: *faultsVector*: vector of detected faults.

1: **for** each state S in M **do**
2: *stateMatrix*[] = mergeBitStrings(S.getStateMatrix())
3: **for** each *bitString* $\in$ *stateMatrix*[] **do**
4: *rules*[] = S.getSatisfiedRules(*bitString*)
5: R[] = getHighestPriorityRule(*rules*[])
6: **if** size(R[]) > 1 **then**
7: *faultsVector*.add({S, R[], *bitString*})
8: **end if**
9: **end for**
10: **end for**
11: **return** *faultsVector*

5.1.1 Determinism Property

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 with the same priority whose predicates can be satisfied by the same set of context updates, generating nondeterministic adaptations.

To detect such faults, we define an algorithm that explores each state matrix in the A-FSM and identifies the existence of bit strings for which the predicates of multiple rules are satisfiable, taking into account also the priorities of the rules. Algorithm 1 detects this pattern of faults. For each state, if there are predicates of multiple highest-priority rules that can be satisfied in that state on the same bit string, then the adaptation is nondeterministic. In Line 2 of the algorithm, *getStateMatrix*(S) returns the state matrix for state S, which contains a list of (*bit string*, *satisfied rules*) pairs for S. Because the algorithm considers each state independently of the others, the analysis can be performed on *merged bit strings*, further reducing the analysis time, and so the function *mergeBitStrings* on Line 2 performs this merging. In particular, whenever a state matrix contains a pair of bit strings for the same set of rules differ-

ing only in their value for one propositional context variable, it means that the predicates of those rules do not depend on the value of that variable. Therefore, the two bit strings are merged, and the value of the affected variable is replaced by $*$ to indicate "don't-care"; this merging is applied repeatedly until no more merging is possible. In Line 4, *getSatisfiedRules*(*S*,*bitString*) returns the satisfied rules for the current bit string. In Line 5, the algorithm returns the subset of the satisfied rules having the highest priority. Finally, in Lines 6–8, 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.

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.

The remaining algorithms in this section and in Section 5.2 assume that the set of rules is deterministic; therefore, Nondeterministic Activation faults must be eliminated before applying the remaining algorithms.

Algorithm 2 DeadPredicateDetection

Input: M: an instance of A-FSM.
Output: *faultsVector*: vector of detected faults.

```
1: for each state S in M do
2:    stateMatrix[] = mergeBitStrings(S.getStateMatrix())
3:    rules[] = S.getRules()
4:    markRules(rules[], "dead")
5:    for each bitString ∈ stateMatrix do
6:       if isAllMarkedNotDead(rules[]) then
7:          break
8:       end if
9:       rules[] = S.getSatisfiedRules(bitString)
10:      R = getHighestPriorityRule(rules[])
11:      markRules(R, "not dead")
12:   end for
13:   for each R ∈ rules[] do
14:      if isDead(R) then
15:         faultsVector.add({S, R, "dead"})
16:      end if
17:   end for
18: end for
19: return faultsVector
```

5.1.2 Liveness Properties

Once the Nondeterministic Activation faults are eliminated from the rules, algorithms can be applied to check that the Liveness properties are satisfied. 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. Furthermore, if the rules violate the State Liveness property, then *all* the active rules of the state have Dead Predicate faults, and we say that the rules contain a *Dead State* fault, a pattern of faults characterized by the presence of a deadlocked state. Note that final states have no active rules and thus cannot suffer from Dead State faults.

In terms of state matrices, a Dead Predicate fault is indicated by the absence of bit strings that cause the predicate of some active rule for the state to be satisfied. Algorithm 2 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 1, this algorithm considers each state independently of the others, and so Line 2 uses the merged bit strings from the state matrix. Line 4 marks all active rules for the current state as being "dead" initially. Lines 9–11 mark the highest-priority rule for each bit string as being "not dead"; there should be only one such rule since, as mentioned in Section 5.1.1, the algorithm assumes that any Nondeterministic Activation faults have been eliminated. Finally, Lines 14–16 report any rules that remain dead after searching through the state matrix.

5.1.3 Stability Property

A sequential digital circuit is said to be in a *metastable* state when the circuit remains in the state for an indefinite period of time during a series of changes in the input [26]. Similarly, a CAAA suffers from metastability 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. In this case the rules of the CAAA violate the Stability property. 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 an *Adaptation Cycle* fault.

In general, these patterns of behavior may not always be considered faulty and instead 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 the choice of last state will be random.

In terms of A-FSMs and state matrices, if an A-FSM is deterministic, it is possible to search for Adaptation Cycles and Adaptation Races by looking for *paths* of transitions among multiple states whose active rules contain predicates that are satisfiable on 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 specific state.

Algorithm 3 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. 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 also is used to force the algorithm to terminate. Lines 7–9 find the destination state for the highest-priority rule of the current bit string and store it in variable *destState*. Lines 11–16 set *isCycle* true because,

Algorithm 3 RaceCycleDetection

Input: M: an instance of A-FSM.
Output: *faultsVector*: vector of detected faults.

```
1: for each state S in M do
2:   stateMatrix[] = mergeBitStrings(S.getStateMatrix())
3:   for each bitString ∈ stateMatrix do
4:     rvector = {} // explored rules
5:     svector = {} // reached states
6:     isCycle = false
7:     rules[] = S.getSatisfiedRules(bitString)
8:     R = getHighestPriorityRule(rules[])
9:     destState = R.getDestState()
10:    while destState != null || !destState ∈ svector do
11:      if destState ∈ svector then
12:        isCycle = true
13:        rvector.add(R)
14:        svector.add(destState)
15:        break
16:      end if
17:      rvector.add(R)
18:      svector.add(destState)
19:      rules[] = destState.getSatisfiedRules(bitString)
20:      R = getHighestPriorityRule(rules[])
21:      destState = R.getDestState()
22:    end while
23:    if size(svector)>2 then
24:      if isCycle then
25:        faultsVector.add({S, rvector, "cycle", bitString})
26:      else
27:        faultsVector.add({S, rvector, "race", bitString})
28:      end if
29:    end if
30:  end for
31: end for
32: return faultsVector
```

after at least one iteration of the innermost enclosing loop, at least one repeated state has been detected at that point. Lines 19–20 look 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 Cycle or Adaptation Race. 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 23–29 report the rules that form Adaptation Races and Adaptation Cycles along with the bit strings that cause them.

5.2 Context Hazard Detection

Even if a set of rules for a CAAA satisfies the desired behavioral properties described in Section 5.1, they still may suffer from faults related to the asynchronous way in which context variables are updated. We can treat the effects of delays in asynchronous updates to context variables as *hazards*, similar to those found in sequential digital circuits. Thus, hazards in a CAAA arise not as a result of the logic of the rules, but as a result of the way physical changes to context propagate to the evaluation of rule predicates.

In previous work we described how the layered architecture of CAAAs gives rise to four different views of the context, two of which are important for detection of context

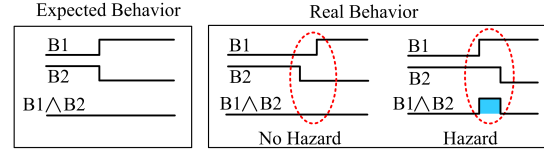

Figure 2: A Static 0-Hazard in an AND-Gate.

hazards—the *physical context* and the *sensed context* [24]. The physical context is the context as it exists physically in the environment of a CAAA. The sensed context is the discretization of continuous physical context values that results when a context-awareness middleware periodically reads the values from sensors and stores them in the set $\mathcal{C}_{sensed}$ of sensed context variables described in Section 4. For instance, in a physical context in which a user is driving a car with his phone actively paired to a Bluetooth handsfree system, the pairing would be represented by a periodically sensed context value such as "BT 00:01:A6:23:FD paired".

Whenever multiple changes occur to the physical context of the CAAA, the internal representation of the sensed context will be inconsistent with the physical context, or *stale*, until *all* the relevant sensed context variables variables have been refreshed. Evaluating rule predicates on a stale sensed context exposes the system to hazards, namely to incorrect or unexpected adaptations. Such faults exist due to three related reasons: (1) the predicates of rules are re-evaluated every time a sensed context variable is updated; (2) the sensed context variables are updated asynchronously according to different refresh rates; and (3) synchronizing the updates of the sensed context variables is difficult (because typically they are updated by different sensor-specific run-time libraries) and undesirable (because of the resulting degradation in performance).

For a given predicate, the occurrence of a hazard depends on the *commutation order* of the predicate's constituent propositional context variables. We illustrate this in Figure 2 with the simple case of an AND-gate that takes two inputs, B1 and B2, which are initially 0 and 1, respectively, thus producing an output of 0. Suppose the inputs undergo a 2-commutation to the values 1 and 0, respectively, producing an output of 0 again. If B2 commutes first, then the output of the gate does not change. However, if B1 commutes first, then the output transiently has the value 1 until B2 commutes, thereby exposing the hazard.

5.2.1 Hold, Activation, and Priority Inversion Hazards

In the adaptation rules of a CAAA, we can identify three different patterns of faults:

- A *Hold Hazard* occurs when the rules adapt to a new state in a situation when the current state should prevail instead. This is similar to the *static 0-hazard* depicted in Figure 2. From the user's point of view this fault produces an unwanted adaptation.
- An *Activation Hazard* occurs when the rules adapt to a new state before all relevant variables have commuted during a commutation of multiple variables, and the new state is different from what is expected.
- A *Priority Inversion Hazard* is similar to an Activation

Hazard and occurs when the rule that triggers has a lower priority than the one that should have triggered. Activation Hazards and Priority Inversion Hazards are similar to a *dynamic hazard* in sequential digital circuits [26], where a different output is produced before the evaluation is completed. From the user perspective the system performs an incorrect adaptation and incorrect actions.

Static hazards in sequential digital circuits are eliminated by introducing delays (such as a double negation) into a specific signal path [26]. A solution for the adaptation rules of CAAAs could be to introduce delays in the invocation of the actions of every triggered rule in order to make sure the destination state of the triggered rule holds. Unfortunately, this would apply a fixed delay even to safe commutations that do not create problems. Therefore, we instead focus on identifying which commutations of propositional context variables may lead to a hazard and then compute the smallest delay that will avoid it.

Algorithm 4 HazardDetection

Input: M: an instance of A-FSM.
Output: *faultsVector*: vector of detected faults.

```
1: for each state S in M do
2:   stableAssignments = complementBitStrings(S.getStateMatrix())
3:   for each bitString in stableAssignments do
4:     for i = 2 to S.numVar() do
5:       indexList = permutation(S.getVars(), i)
6:       for each sequence in indexList do
7:         rvector = {} // selected rule in each step
8:         u = bitString
9:         hazard = null
10:        for j = 0 to i - 1 do
11:          u.flipBitAtIndex(sequence[j])
12:          rules[] = S.getSatisfiedRules(u)
13:          R = getHighestPriorityRule(rules[])
14:          if isEmpty(rules[]) && (j != i - 1) then
15:            break // reached stable assignment
16:          else if (R != null) && !(rvector.contains(R)) then
17:            if R.isHigherPriorityRule(rvector) then
18:              hazard = "Priority Inversion"
19:            end if
20:            rvector.add(R)
21:          else if (j == i - 1) then
22:            if isEmpty(rules[]) then
23:              hazard = "Hold" // may override
24:            else if !R ∈ rvector.prefix(R) && hazard != "Priority Inversion" then
25:              hazard = "Activation"
26:            end if
27:          end if
28:        end for
29:        faultsVector.add({hazard, S, bitString, sequence, rvector})
30:      end for
31:    end for
32:  end for
33: end for
34: return faultsVector
```

For a given state, we define a *stable assignment* as an assignment of values to the propositional context variables that satisfies none of the predicates of the active rules of the state. Correspondingly, an *unstable assignment* is an assignment that satisfies some predicate. We define a *critical path* as any sequence of commutations that starts with a stable assignment and has one or more intervening unstable assignments. If a critical path ends in a stable assignment for the given state, then we have a Hold Hazard. If the critical path ends in an unstable assignment, then we need to check for a Priority Inversion Hazard. Otherwise we have an Activation Hazard. We only consider critical paths in which each variable commutes at most once, since a critical path with multiple commutations of the same variable can be subdivided into multiple critical paths with single commutations. In addition, we assume that multiple propositional context variables associated with the same underlying sensed concrete variable are updated simultaneously whenever the sensed concrete variable is updated. We can relax this assumption by accounting for any implementation delays in the updating of the propositional context variables.

Algorithm 4 is applied in each state S and searches for hazards beginning from stable assignments, which Line 2 identifies as the set of all bit strings satisfying *no* predicates of active rules of S. Lines 4–31 explore all commutations from length two to the number of propositional context variables, with the loop variable i indicating the current length to consider. Line 5 generates the set of permutations of length i of indexes into the current bit string, indicating the different variables and their orderings to consider for commutations. For each permutation, Lines 6–30 sequentially commute the variables according to the current permutation and look for hazards under that setting. Line 15 discards a stable path that has been processed already in a previous shorter path. Lines 16–20 detect Priority Inversion Hazards when a rule R is discovered with higher priority than rules found before. Lines 21–27 detect Hold Hazards and Activation Hazards. A Hold Hazard is present when there is an adaptation (indicated by *!isEmpty(rules[])* in Line 22) between stable assignments. Line 29 reports all detected critical paths and their hazard category.

5.2.2 Fixing a Hazard

For a detected critical path, a simple solution to prevent its associated hazard would be to introduce a delay until all the variables of the critical path have been updated by the underlying middleware. Underestimating this delay may not eliminate the hazard, while overestimating it may make the application inefficient. In any case, in the typical situation, different sensed concrete variables have different refresh rates, and so additional commutations of a variable that already commuted may occur before any introduced delay has elapsed.

To address this problem, for each unstable assignment reachable from a given initial stable assignment in a given state, we can calculate the *minimum safe delay*, defined as the smallest interval of time starting from the time at which the first variable of the assignment commuted, after which the assignment is hazard-free. If during the unsafe period another variable commutes, then a new minimum safe delay must be recomputed for the resulting assignment. For assignments not affected by hazards this delay is zero. For other assignments, it is the maximum over all hazards from

Algorithm 5 MinimumSafeDelays

Input: *faultsVector*: vector of detected faults.
Output: *delaysVector*: {state, bitString, subPath, delay}.

```
1: delaysVector = {}
2: for each fault in faultsVector do
3:    path = fault.getCriticalPath()
4:    for i = 0 to path.size() - 1 do
5:       subPath[] = path.getSubPath(0, i)
6:       delay = 0
7:       for each Variable v in path do
8:          ContextVariable cv = v.getContext()
9:          if !(cv ∈ subPath.getContexts()) then
10:            t = cv.getRefreshRate()
11:            delay = max(delay, t)
12:         end if
13:      end for
14:      if delay > delaysVector.get(subPath) then
15:         delaysVector.add({fault.getState(),
            fault.getBitString(), subPath, delay})
16:      end if
17:   end for
18: end for
19: return delaysVector
```

which the assignment must be protected. Algorithm 5 generates the minimum safe delays for a given set of hazards, such as those reported by Algorithm 4. For each critical *path*, Lines 3–5 select a *subPath* of the current length i. Lines 7–13 extract the set of context variables corresponding to context that may commute in *path* but that have not commuted in *subPath*. The slowest refresh rate of the remaining context is stored in Lines 10–11. Since the same *subPath* can be obtained from multiple *path*s, Lines 14–16 store only the slowest. Engineers can use Algorithm 5's output to force waits that prevents hazards.

6. FINDING FAULTS IN PHONEADAPTER

This section describes our results from applying our fault detection algorithms to *PhoneAdapter*, the CAAA described in Section 3 and modeled in Section 4. *PhoneAdapter* is implemented on top of ContextNotifier, a J2ME rule-based adaptation framework and middleware for CAAAs [22], and targeted to deploy on the Nokia N95 cellphone. We ran the application and its adaptation rules within TestingEmulator, an emulator we have built for CAAAs [23]. The implemented *PhoneAdapter* has nine states and 19 rules, and it utilizes 12 propositional context variables. For this analysis, we set refresh rates in the TestingEmulator of one millisecond for time, 10 seconds for GPS, and 60 seconds for Bluetooth.

Table 2 summarizes the results. The first column shows the nine states in the A-FSM. The second column shows the number of propositional context variables used by the predicates associated with each state. The remaining columns present the results of applying the four fault detection algorithms to *PhoneAdapter*.

6.1 Detecting Nondeterministic Adaptations

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

The column "Nondet. Adaptation" in Table 2 shows the number of assignments of propositional context variables analyzed by Algorithm 1 and the number of assignments that induce nondeterministic activations. For example, 7 propositional context variables are used in state *General* and all 128 (2^7) assignments are analyzed. 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*.

To eliminate these faults, we assigned distinct priorities to the affected rules in line with the behavior desired for *PhoneAdapter*.

6.2 Detecting Dead Predicates

This pattern of faults consists of predicates that cannot be satisfied by any assignments for 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 the number of assignments to propositional context variables analyzed by Algorithm 2 and the one fault it detected in rule *ActivateSync* within state *General*. The number of assignments analyzed is a subset of the number considered by the previous algorithm since one assignment satisfying a predicate is enough to show that that the predicate is not dead. By examining the state matrix associated with this state, we note that it is possible for predicates of rule *ActivateSync* to be satisfied by certain assignments, but such assignments also satisfy *ActivateOffice* and *ActivateHome*, which have higher priority, and thus *ActivateSync* is always preempted and never triggered.

6.3 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.

As shown in the column "Adaptation Races and Cycles" of Table 2, Algorithm 3 analyzes a larger number of assignments than the previous algorithms because it explores paths of transitions among multiple states. The algorithm detected 535 races and 113 cycles. There are several different races producing fluctuations in the states that merely may disturb the user temporarily. For example, if a user starts to drive and accelerates quickly, the application may or may not reach *DrivingFast* (in which it will divert calls), 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 is re-detected, because 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 produced by rules *ActivateMeeting* and *DeactivateMeeting* when the state is *Office* and $Time >= meeting_end$.

Table 2: Faults Detected in PhoneAdapter

State	Vars.	Nondet. Adaptation		Dead Predicates		Adaptation Races and Cycles			Context Hazards			
		Assignments	Faults	Assignments	Faults	Assignments	Race	Cycle	Paths	Hold	Activ.	Prior.
General	7	128	37	128	1	3968	45	13	14085	0	11	3182
Outdoor	5	32	3	17	0	3968	135	23	161	0	0	52
Jogging	2	4	0	1	0	3072	97	19	2	0	0	0
Driving	3	8	0	7	0	2560	36	13	16	2	2	4
DrivingFast	2	4	0	2	0	3072	58	19	2	0	0	0
Home	4	16	0	9	0	2816	76	19	104	8	0	13
Office	7	128	1	65	0	2848	29	1	82634	1828	368	2164
Meeting	1	2	0	2	0	2048	32	1	0	0	0	0
Sync	2	4	0	1	0	1024	27	5	2	2	0	0

6.4 Detecting Context Hazards

This class of faults corresponds to sequences of asynchronous changes to propositional context variables that introduce unwanted adaptations or unexpected states.

Executing Algorithm 4 results in many critical paths, or sequences of changes to propositional context variables leading to hazards. The column "Context Hazards" of Table 2 shows the number of paths considered and the 7636 paths that induce the three patterns of context hazards. Most of the hazards are associated with paths starting in *Office* and *General*, and Priority Inversion Hazards are the most common of the three.

One example of a Hold Hazard occurs within *Home*, when GPS becomes disconnected and the home PC is not detected quickly enough via Bluetooth. In this situation, the phone will adapt from *Home* to *General* because there is no context indicating that the user is at home.

We also detected several Activation Hazards. For example, when GPS indicates that a user is at the office, when Bluetooth shows that more than three people are nearby, and when it is time for a meeting, then the rule *ActivateMeeting* will be triggered and the phone will adapt to *Meeting*. Then when GPS becomes disconnected and fewer than three people are detected nearby, even though rule *DeactivateOffice* defined for state *Office* is satisfied, the phone will remain in state *Meeting* because that rule is not active within *Meeting*.

Finally, we detected many Priority Inversion Hazards. For example, when GPS senses a user's speed of more than 5 miles per hour before the car's handsfree device is detected, the phone will adapt to *Jogging* instead of *Driving*, even through the priority of rule *ActivateDriving* is higher than the priority of rule *ActivateJogging*. This problem can be avoided by forcing a wait for Bluetooth to detect the car's handsfree system, which will enable the phone to adapt to *Driving*.

6.5 Determining Minimum Safe Delays

Once we identified the paths that may induce context hazards, we calculated the minimum safe delay for each path in order to achieve hazard-free adaptation of the system with minimum delays. The application of Algorithm 5 to *PhoneAdapter* reports a minimum delay for detected hazards that is, on average, 75% shorter than the delay produced by simply waiting for a commutation of all context variables to occur.

7. CONCLUSION AND FUTURE WORK

In this paper we have described three contributions to the validation of modern software applications. First, we have defined a formal model of a key complex behavioral characteristic, namely *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 algorithms for automatically detecting occurrences of the fault patterns, enabling software engineers to increase the quality and robustness of their applications. We illustrated 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.

In future work we plan to apply *constraint propagation* over variables that are affected by physical laws (e.g., time always increasing, battery level usually decreasing) in order to reduce the number of possible paths to explore in an A-FSM, which is crucial for faults affecting multiple states. To improve the efficiency of the rules' state space exploration we also plan to use a symbolic representation of the state space (such as binary decision diagrams [4]) and to explore the possibility of encoding our fault patterns as properties that could be checked by model checkers. We also will extend our approach to accommodate end users' needs by introducing interactive "wizards" for improving adaptation rules and eliminating their faults. Finally, given how error-prone the use of rule-based adaptation can be for CAAAs, it will be worthwhile to investigate alternative approaches to support adaptation.

8. ACKNOWLEDGEMENTS

This work was supported in part by the US National Science Foundation (NSF) under CAREER Award 0347518, and by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/D077273/1, EP/E006191/1 and EP/F013442/1. David Rosenblum holds a Wolfson Research Merit Award from the Royal Society.

9. REFERENCES

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

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

[3] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Proc. Second*

IEEE Annual Conference on Pervasive Computing and Communications, pages 361–365, March 2004.

[4] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[5] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, October 2003.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programmming Languages and Systems*, 8(2):244–263, 1986.

[7] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1023–1038, November 2002.

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

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

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

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

[12] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83:69–93, January 1995.

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

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

[15] T. Higashino, A. Nakata, K. Taniguchi, and A. R. Cavalli. Generating test cases for a timed I/O automaton model. In *Proc. International Workshop on Testing Communicating Systems: Method and Applications*, pages 197–214, September 1999.

[16] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Proc. 11th International SPIN Workshop*, pages 109–126, April 2004.

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

[18] B. Nielsen and A. Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5(1):59–77, November 2003.

[19] R. Nilsson and J. Offutt. Automated testing of timeliness: A case study. In *Proc. International Workshop on Automation of Software Test*, page 11, May 2007.

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

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

[22] M. Sama and D. Rosenblum. ContextNotifier. http://code.google.com/p/contextnotifier/. Last accessed 7 March 2008.

[23] M. Sama and D. Rosenblum. TestingEmulator. http://code.google.com/p/testingemulator/. Last accessed 7 March 2008.

[24] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. Multi-layer faults in the architectures of mobile, context-aware adaptive applications: A position paper (Short Paper). In *Proc. ICSE 2008 International Workshop on Software Architectures and Mobility*, pages 47–50, May 2008.

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

[26] S. H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.

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

[28] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology*, 16(4):article 14, 2007.

[29] C. Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proc. Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 336–345, September 2005.

[30] C. Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *Proc. International Conference on Software Engineering*, pages 292–301, May 2006.