Dynamic Fault Detection in Context-aware Adaptation

Chang Xu[1,2], S.C. Cheung[3], Xiaoxing Ma[1,2], Chun Cao[1,2], Jian Lu[1,2]
[1]State Key Lab for Novel Software Technology
[2]Department of Computer Science and Technology
Nanjing University, Nanjing, Jiangsu, China
{changxu, xxm, caochun, lj}@nju.edu.cn
[3]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Kowloon, Hong Kong, China
scc@cse.ust.hk

ABSTRACT
Internetware applications are context-aware and adaptive to their environmental changes. Faulty adaptation may arise when these applications face unexpected situations. Such adaptation faults can be difficult to detect at design time. The recent Adaptation Finite-State Machine (A-FSM) approach proposes to statically analyze model-based context-aware applications for adaptation faults. However, this approach may suffer expressiveness and precision problems. To address these limitations, we propose an Adaptation Model (AM) approach. As compared with A-FSM, AM offers increased expressive power to model complex rules, and guarantees soundness in fault detection. Besides, AM deploys an efficient rule evaluation technique to cater for context-aware applications that are subject to continual environmental changes. We evaluated our AM approach using both simulated and real-world experiments with two applications. The experimental results confirmed that AM can detect real faults missed by A-FSM, and avoid false positives that were misreported otherwise.

Categories and Subject Descriptors
D.2.4 [**Software/Program Verification**]: Validation; F.1.2 [**Models of Computation**]: Alternation and non-determinism.

General Terms
Performance, Design, Experimentation, Verification.

Keywords
Context-aware adaptation, fault detection, incremental rule evaluation.

1. INTRODUCTION
With recent advancement of Internetware [10] technologies such as wireless sensor networks (WSNs) and radio frequency identification (RFID), new kinds of applications that continually monitor environments for seamless integration are receiving increasing attention. These Internetware applications, often called *Context-aware Adaptive Applications* (CAAAs) [13][14], can adapt their behavior to changes of their environmental contexts. Typical examples of such *contexts* include object location, environmental noise, or other pieces of information that can affect the computation of CAAAs. For example, a PhoneAdapter application [13][14] mutes a smartphone's ring tone and activates its vibration when its user is in his office, and disables vibration when the user attends a meeting.

Various middleware infrastructures [1][7][11][18] and application frameworks [3][4] have been proposed to support the development and deployment of CAAAs. Most of them share an intuitive computational model, in which two concerns are well separated: (1) acquiring contexts from environments, and (2) executing adaptation based on these contexts.

This computational model enables users to specify *adaptation rules* that govern how their applications react to context changes. Adaptation rules thus play an important role of deciding application behavior. However, this model, although simple yet powerful, may expose potential threats to its correctness if a CAAA encounters at runtime the situations never expected at design time. For example, if multiple rules are triggered at the same time and the CAAA randomly executes one of them, its state may become unpredictable due to this non-determinism. Or, if some rules can be triggered without taking any new contexts, the CAAA's state would become unstable and its current state's duration would depend on its context update rate and rule execution speed. As a result, the CAAA may fail to adapt as expected at design time.

Although each CAAA has its own criteria to decide application-specific faults, many CAAAs share common fault patterns. For example, when properties like *determinism* (whether a CAAA is always clear about which rule to execute at any time) and *stability* (whether a CAAA's state is independent of its context update rate and rule execution speed) are violated, a CAAA would run in an unpredictable or unstable way.

A recent piece of work has proposed an Adaptation Finite-State Machine (A-FSM) approach to detect such adaptation faults by static analysis [13][14]. A-FSM detects adaptation faults by exhaustively exploring the space constructed by all possible value assignments to context variables used in a CAAA's rules. However, A-FSM contains two inherent limitations: (1) A-FSM does not have sufficient expressive power to specify complex adaptation rules in recently published CAAAs; (2) A-FSM does not take into account the impact of variable dependency, physical constraints, and rule actions on its fault detection results, and may report false positives (i.e., unreal faults).

Therefore we present our new Adaptation Model (AM) approach to address these two limitations. We base our work on A-FSM and improve it by increasing its model's expressive power and avoiding reporting unreal faults. In addition, we deploy an incremental rule evaluation technique to enhance AM's runtime effi-

Internetware'12, October 30–31, 2012, Qingdao, Shandong, China.

Figure 1. Loading bay (left) and storage bay (right).

ciency so that it can be used for practical context-aware applications, which are subject to continually changing environments.

The remainder of this paper is organized as follows. Section 2 presents a motivating example to explain A-FSM's limitations in modeling CAAAs and detecting faults, and analyzes the challenges of addressing these limitations. Section 3 introduces our AM approach in detail, from adaptation model, to fault detection algorithm, and to runtime rule evaluation. Section 4 evaluates our AM approach and compares it to A-FSM using a simulated stock tracking application and a real-world self-controlling mini-car system. Finally, Section 5 discusses related work and Section 6 concludes this paper.

2. MOTIVATING EXAMPLE

In this section, we present a motivating example to illustrate our target problem. The example exhibits interesting context-aware features found in typical CAAAs.

2.1 Stock Tracking Application

Our motivating example was originated from one pilot study of an RFID-enabled stock tracking application in a paper company.

The application uses a forklift to transport RFID-tagged paper boxes from the loading bay to the storage bay of a warehouse (Figure 1). RFID gates are installed at these bays to collect contexts (including the RFID codes and locations of paper boxes) for stock tracking. *Missing readings* occur when the RFID codes of some paper boxes cannot be successfully detected. Although the problem can be partially alleviated by increasing the antennas' transmission power or altering the antennas' positions, this may instead cause the RFID codes of some boxes to be accidentally detected by their neighboring RFID gates. These wrongly detected codes are called *cross readings*. Missing and cross readings are common to RFID deployments [6]. In our pilot study, such readings are continually detected at runtime and fed to the application in terms of context changes and later be repaired.

The process of the stock tracking application is described by a finite-state machine in Figure 2. The application starts with the initial "loading" state, in which one forklift loads a pallet of paper boxes at a gate of the loading bay. Each paper box has an RFID tag with a unique code that identifies this box. The code is read when the box passes an RFID gate. The application then reaches the "transporting" state, in which the forklift transports the pallet from the loading bay to the storage bay, where the boxes on the pallet are unloaded. Pallet unloading causes the application to enter the "unloading_1" state, which is followed by the "unloading_2" state (i.e., two steps). During the two unloading steps, RFID codes of the boxes are read again to check whether there are

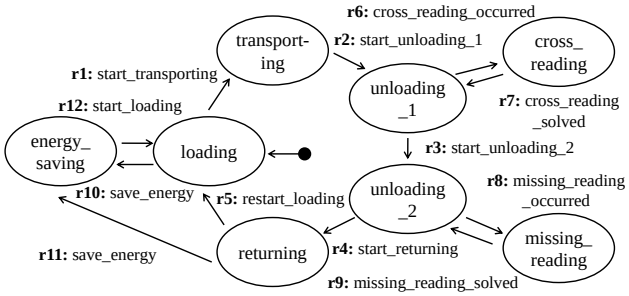

Figure 2. Stock tracking application (its state transition diagram with 8 states and 12 rules).

any cross readings or missing readings. The checking is based on two consistency constraints: (1) During the transportation, the RFID codes of these boxes should not be read by irrelevant RFID gates; (2) The two sets of box readings collected during loading and unloading must match. Cross readings are discarded in the "cross_reading" state, and missed readings are recovered in the "missing_reading" state. After the box unloading and checking, the application enters the "returning" state, and the forklift goes back to the loading bay for the next loading. If there are no available transportation activities for long time, the application would enter the "energy_saving" state and put the RFID gates into sleep mode (detecting RFID codes in a longer period) for saving energy until receiving new tasks.

The application consists of 8 states and 12 rules, by which it supports the following three functional features:
(1) **Normal workflow**, which runs as a complete work loop to transport any given paper boxes from the loading bay to the storage bay. This feature covers "loading", "transporting", "unloading_1", "unloading_2", and "returning" five states and their five associated rules **r1-5**.
(2) **Exception handling**, which detects cross readings and missing readings for paper boxes and resolves them at runtime. This feature covers "unloading_1", "cross_reading", "unloading_2", and "missing_reading" four states and their four associated rules **r6-9**.
(3) **Energy-awareness**, which switches the application between the normal working mode and energy-saving mode. This feature covers "energy_saving", "loading", and "returning" three states and their three associated rules **r10-12**.

2.2 Application Modeling

As aforementioned, adaptation rules play an important role in defining an application's behavior upon context changes. A modeling language that has sufficient expressive power to define conditions upon context changes is necessary. In the stock tracking application, contexts include readings from pressure sensors (installed at the loading bay and storage bay to indicate whether a forklift has arrived or left) and RFID readings (for tracking locations of pallets and paper boxes). The application needs to check these contexts and their relationships to decide its adaptive behavior. A modeling language based on propositional logic (e.g., the one used in A-FSM) may not suffice for this purpose.

A-FSM defines logical rules using propositional context variables and three logical operators "and", "or", and "not". For example, one can define a rule "(flArv(R_{stor})) and (pallet(T_{stor}))" using two propositional context variables "flArv(R_{stor})" and "pallet(T_{stor})" to check whether one pressure sensor ("R" for short) reports the

arrival of a forklift at the storage bay as well as its associated pallet ("*T*" for short) being found there. We observe that rules in propositional logic are based on *current values* of context variables, e.g., the preceding rule uses "flArv(R_{stor})" and "pallet(T_{stor})" to show whether the forklift is *currently* at the storage bay and whether the pallet's *current* location is the storage bay, respectively. We call these rules *simple rules*.

In the stock tracking application, some rules require *multiple values* of certain context variables in a spatial or temporal dimension. For example, the rule for missing reading checking may be specified as "$\exists B_{load}$ ($\forall B_{stor}[t]$ (not (matched(B_{load}, B_{stor}))))". This rule checks the existence of a box that, after its RFID reading ("*B*" for short) has been collected at the loading bay, is no longer detectable at the storage bay ("B_{stor} [*t*]" represents the set of box RFID readings collected later at the storage bay in a time interval *t*). Specifying such rules needs the support of first-order logic that contains universal and existential quantifiers. We name such a modeling language a *first-order logic based language* and the rules thus expressed *complex rules*.

Besides the stock tracking application, we note that the requirement for modeling complex rules is common in recently published CAAAs. For example, a ConChat application [12] checks whether *all* people from the same group are in a certain meeting room or *any* of them is in another specific room like 3234. Another context-aware communication application [4] checks whether its user is occupied in a meeting or phone call in *any* of possible time slots. In these rules, first-order logic is commonly used to check a set of contexts restricted by spatial or temporal conditions.

2.3 Modeling Language

We propose the following first-order logic based language [18] to specify adaptation rules for CAAAs:

rule := $\forall var[t]$ (*rule*) | $\exists var[t]$ (*rule*) | (*rule*) and (*rule*) | (*rule*) or (*rule*) | (*rule*) implies (*rule*) | not (*rule*) | *func*(*var*, *var*, …).

As the language syntax shows, a rule is defined recursively. By universal or existential quantifiers, a rule defines context variables (*var*) that can take arbitrary values (not just Boolean values). A context variable can be optionally restricted by a time interval (*t*). When the interval is present, the variable binds to a set of values collected within this time interval; otherwise, it binds to the latest value only. By this, the language supports both complex rules and simple rules. In addition, users can also define domain- or application-specific functions (*func*). To illustrate the language's expressive power, we specify all the 12 rules of the stock tracking application using this language in the following.

(1) Normal workflow (normal priority):

r1: "start_transporting": $\exists R_{load}$ (flGone(R_{load})).

r2: "start_unloading_1":
($\exists R_{stor}$ (flArv(R_{stor}))) and ($\exists T_{stor}$ (pallet(T_{stor}))).

r3: "start_unloading_2": $\exists C_{stor}$ (proceed(C_{stor})).

r4: "start_returning": $\exists R_{stor}$ (flGone(R_{stor})).

r5: "restart_loading": $\exists R_{load}$ (flArv(R_{load})).

Context variables R_{load} and R_{stor} report the latest status for two pressure sensors ("*R*" for short) installed at the loading bay and the storage bay, respectively. Functions "flGone" and "flArv" check whether the forklift has left or arrived at the two sensors, respectively. Another context variable T_{stor} reports the most recent pallet RFID reading ("*T*" for short) at the storage bay, and function "pallet" checks whether this reading belongs to the pallet being used by the forklift. Finally, context variable C_{stor} contains the latest control data ("*C*" for short) that indicates the completion of box RFID reading at the storage bay and that the application can proceed to check missing readings (by function "proceed").

We consider one rule for example. According to Rule **r2**, when the application detects that the forklift has arrived at the storage bay (flArv(R_{stor}) = true) and its pallet also appears there (pallet(T_{stor}) = true), the application can enter the "unloading_1" state.

(2) Exception handling (high priority):

r6: "cross_reading_occurred":
$\exists B_{stor}$ ($\exists B_{gate}[-t]$ (matched(B_{stor}, B_{gate}))).

r7: "cross_reading_solved":
$\forall B_{stor}$ (not ($\exists B_{gate}[-t]$ (matched(B_{stor}, B_{gate})))).

r8: "missing_reading_occurred":
$\exists B_{load}$ ($\forall B_{stor}[t]$ (not (matched(B_{load}, B_{stor})))).

r9: "missing_reading_solved":
$\forall B_{load}$ ($\exists B_{stor}[t]$ (matched(B_{load}, B_{stor}))).

In the second feature, Rules **r6-9** are all complex rules, which cannot be expressed in a propositional logic based language.

Rule **r6** checks whether any collected paper box RFID reading ("B_{stor}" for short) at the storage bay belongs to the set of box RFID readings ("B_{gate}" for short) collected earlier by another irrelevant gate within a past time interval *t*. The value of *t* is set to a reasonable transportation time by the forklift between the loading bay and the storage bay such that any irrelevant readings of the boxes being transported are considered *cross readings*. Rule **r7** inverses **r6**'s condition to ensure that all cross readings have been handled properly before the application moves back to the "unloading_1" state. Rules **r8** and **r9** work similarly. Rule **r8** checks whether there is a box no longer detectable at the storage bay after its RFID reading has been collected at the loading bay ("B_{load}" for short). Rule **r9** inverses **r8**'s condition to ensure that all missing readings have been handled properly. As these rules show, our modeling language has the expressive power to specify rules that refer to a set of historical contexts ([*-t*]) or future contexts ([*t*]) as required in a real-world situation.

(3) Energy-awareness (high priority):

r10/11: "save_energy": ($\exists R_{load}$ (flGone(R_{load})))
and (not ($\exists T_{load}[-t]$ (pallet(T_{load})))).

r12: "start_loading":
($\exists R_{load}$ (flArv(R_{load})) or ($\exists T_{load}$ (pallet(T_{load})).

The third feature contains two complex rules **r10-11** and one simple rule **r12**. Rule **r10/11** enables the application to enter the "energy_saving" state when the forklift has left (maybe used somewhere else) and the pallet is taken away for long time (already timeout). Rule **r10/11** applies to both the "loading" and "returning" states. The last Rule **r12** brings the application back to the normal working mode if any condition is no longer satisfied.

2.4 Fault Detection Precision

According to A-FSM's study and its experimental results with the PhoneAdapter application, there are some fault patterns common to typical CAAAs, such as violation to an application's determinism or stability. We name them *non-determinism fault* and *instability fault*, respectively. These faults can be identified by A-

FSM's static analysis. However, A-FSM has made two assumptions that may affect its fault detection precision.

Assumption 1: Using a propositional logic based language. A-FSM assumes that any CAAA should be modeled using a propositional logic based language. By doing so, the number of context variables used in a rule becomes *fixed* and they can only take either true or false. This makes possible for A-FSM to statically explore all possible value assignments to these context variables. To detect non-determinism faults, A-FSM tries all value assignments in each state and sees whether under any of them, more than one rule would be triggered. To detect instability faults, A-FSM tries all value assignments in each state and sees whether any rule is triggered, and after executing this rule, whether another rule is already triggered without taking any new context changes.

Due to the use of a propositional logic based language, A-FSM models and detects faults in simple rules only. As such, its fault detection results may not be *complete*. This implies that A-FSM may introduce *false negatives* (i.e., no faults are reported but there actually are some).

For example, in the stock tracking application, a non-determinism fault may occur at the "loading" state, where two Rules **r1** and **r10** are triggered at the same time. This happens when the forklift stays at the loading bay but the pallet has been taken away for some time (the right part of Rule **r10** becomes true), and then the forklift is also taken away and the application would face a nondeterministic situation because Rules **r1** and **r10** are both triggered (Rule **r1** and the left part of Rule **r10** become true). However, A-FSM would never detect it because Rule **r10** is a complex rule a propositional logic based language cannot model.

Assumption 2: Not considering dynamic information. A-FSM is a static analysis approach, which does not consider the impact of variable dependency, physical constraints, and rule actions on its fault detection, and therefore its detection results may be *imprecise*. This is because: (1) Variable dependency and physical constraints enforce specific relationships among contexts such that context variables cannot take arbitrary values at runtime; (2) Rule actions introduce dynamic context changes which static analysis is not aware of.

For example, A-FSM would detect an instability fault at the "energy_saving" state when the value assignment to context variables is as follows: pallet(T_{load}) = true, flArv(R_{load}) = true, flGone(R_{load}) = true, pallet(T_{stor}) = false, flArv(R_{stor}) = true, flGone(R_{stor}) = false, and proceed(C_{stor}) = false. When this fault occurs, the application would make a state transition from "energy_saving" to "loading" (since flArv(R_{load}) = true) and then to "transporting" (since flGone(R_{load}) = true) without taking any new contexts. The activity of collecting RFID readings for paper boxes may or may not be skipped, depending on the rule execution speed and context update rate. However, such a fault will never occur in practice. This is because the associated value assignment is invalid due to the variable dependency between flArv(R_{load}) and flGone(R_{load}): they cannot both take true at the same time.

It could be argued that such variable dependency can be studied in advance and also modeled as rules for checking during fault detection. However, there are three concerns. First, variable dependency rules can be a lot and there may be no automated way to derive a complete and precise set of them for a given CAAA. Second, modeling these variable dependency rules may require the expressive power beyond that of a propositional logic based language can offer. Third, physical constraints and rule actions may add at runtime unpredictable context changes, which cannot be modeled in advance.

For example, a physical constraint like "the forklift cannot jump from the loading bay to the storage bay or back suddenly" requires that "pallet(T_{load}) = true" cannot be followed immediately by a context change that demands "pallet(T_{stor}) = true". Exploring a complete set of such constraints needs extensive knowledge on mathematical and physical laws about the real world and seems never to be an easy task. Besides, the application may modify contexts at the "cross reading" and "missing reading" states and this would update the values of relevant context variables. A static analysis approach like A-FSM would fail to incorporate such dynamic information into its fault detection process and has to suffer from the *imprecision* problem.

2.5 Challenges

According to our preceding analysis, the expressive power of a propositional logic based language and the nature of a static analysis based approach make fault detection precision impaired. However, a migration from a propositional logic based language to a first-order logic based language is not straightforward.

First, for a rule that is specified by a first-order logic based language like Rule **r8** "$\exists B_{load}$ ($\forall B_{stor}[t]$ (not (matched(B_{load}, B_{stor}))))" in the stock tracking application, each context variable in the rule can take arbitrary values (not only true or false) and it may refer to a set of contexts (e.g., $B_{stor}[t]$) rather than only one value. Not only what contexts can become the values for such a context variable is not clear (decided at runtime), but also how many of them can be in this set is no longer knowable in advance (decided by the time interval t). As such, statically exploring all possible value assignments to the context variables used in a rule becomes impossible. Then, the idea of exhaustively exploring a rule's space to detect potential adaptation faults for CAAAs is no longer applicable to first-order logic specified rules.

Second, since a static analysis of space of a first-order logic specified rule is impossible to conduct, one may migrate from static analysis to dynamic analysis, i.e., detecting adaptation faults at runtime. This can address the concern that there is no way of automatically deriving a complete and precise set of variable dependency and physical constraints for a given CAAA. However, detecting faults at runtime requires high efficiency because otherwise, the CAAA's responsiveness to context changes would be impaired, denying its original purpose of being context-aware.

Regarding these two challenges, we propose our new AM approach to detect CAAA adaptation faults in Internetware environments. Our ideas are as follows:

(1) Completeness: Deploy our first-order logic based language to model CAAA rules. It supports both simple and complex rules. It also allows domain- or application-specific functions.

(2) Precision: Detect faults at runtime to avoid false positives. Our AM approach detects CAAA's adaptation faults dynamically and must report real faults.

(3) Efficiency: Detect faults in an incremental way. Our AM approach reuses previous fault detection results and efficiently handles context changes at runtime.

3. THE AM APPROACH

In this section, we present our Adaptation Model (AM) approach. We first introduce its underlying model to describe how a CAAA runs with state transitions. We then propose an algorithm based on this model to detect adaptation faults at runtime. Finally, we discuss how we minimize the impact of runtime fault detection on a CAAA's responsiveness to context changes.

3.1 Adaptation Model

We present an adaptation model (AM) to specify a rule-based CAAA. An AM is a finite-state machine that contains a set of *states S* and a set of *transitions* (called *adaptation rules*) *R*. Rules are defined as: $R \subseteq S \times C \times S \times A \times N$, in which *C* is a set of rule conditions specified in our first-order logic based language, *A* is a set of actions that may modify the values of context variables, and *N* is a set of natural numbers presenting rule priorities.

A rule *r* is a tuple of five attributes (*s*, *c*, *s*', *a*, *n*). If the CAAA is at state *s* and condition *c* is evaluated to true, then rule *r* is said *triggered*. All rules (such as *r*) starting from state *s* are considered *active* for *s*. If multiple active rules are triggered, then only one of them can be selected for execution, depending on their priorities (*n*). When rule *r* is executed, the CAAA conducts action *a* and moves to a new state *s*'.

A CAAA's adaptation model is defined as: $AM = (S, R, s_0, S_f, V, s_c)$. *S* and *R* are the set of states and set of adaptation rules, respectively. State $s_0 \in S$ is the CAAA's initial state, and $S_f \subseteq S$ is the set of its final states, at any one of which the CAAA stops. *V* and s_c capture the runtime information for an *AM*, in which *V* represents the value assignment for all context variables $\wp(CTX\_VAR \times \wp(CTX\_VAL))$, and $s_c \in S$ is the CAAA's current state. In an AM, a context variable can be mapped to either a single context value (for simple rules) or a set of context values (for complex rules).

3.2 Fault Detection Algorithm

We focus on two types of fault in CAAAs, *non-determinism fault* and *instability fault*, because of their popularity [13][14]. As earlier examples show, non-determinism fault violates the property that for each state in an AM and each possible value assignment to context variables at that state, there is at most one active rule that can be triggered. Instability fault violates the property that an AM's state is not dependent on the context update rate or rule execution speed. When an instability fault occurs, a set of context changes 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. This would cause the CAAA to run in an unstable way (called *adaptation race*) or in an infinite loop (called *adaptation cycle*) [13][14].

Figure 3 gives our fault detection algorithm that is composed of three parts. Part 1 (`detectFaults`) is invoked when a new context change *c* is received. The CAAA's adaptation model *M* would evaluate all active rules against *c* to see whether any rule is triggered (Lines 1-3 of Part 1). Part 2 (`checkNondeterminism`) is then invoked to collect non-determinism faults if any, and select one triggered rule *r* for execution (Lines 1-9 of Part 2). If such a rule *r* exists, Part 1 continues to check adaptation race or cycle faults (Lines 4-7 of Part 1). The check forms a loop, in which Part 3 (`checkRaceCycle`) executes the selected rule *r* and collects resulting context changes (Line 3 of Part 3), then

```
Algorithm (Part 1): detectFaults
Input: AM M, context change c
Output: faults F
 1: M.evaluateRules(c)
 2: F = {}
 3: r = checkNondeterminism(M, F)
 4: if r != null then
 5:    state = M.getState()
 6:    cycle = false
 7:    R = checkRaceCycle(M, r, F, cycle)
 8:    if size(R) > 1 then
 9:       if cycle then
10:          F.add({"cycle", state, R})
11:       else
12:          F.add({"race", state, R})
13:       end if
14:    end if
15: end if
16: return F
```

```
Algorithm (Part 2): checkNondeterminism
Input: AM M, faults F
Output: triggered rule r, faults F
 1: rules[] = M.getSatisfiedRules()
 2: if size(rules[]) > 1 then
 3:    F.add({"nondeterminism", M.getState(), rules[]})
 4:    r = selectRandom(rules[])
 5: else if size(rules[]) == 1 then
 6:    r = rules[0]
 7: else
 8:    r = null
 9: end if
10: return r
```

```
Algorithm (Part 3): checkRaceCycle
Input: AM M, triggered rule r, faults F, boolean cycle
Output: race rules R, faults F, boolean cycle
 1: R = {r}
 2: while r != null && !cycle do
 3:    changes[] = M.executeRule(r)
 4:    M.evaluateRules(changes[])
 5:    r = checkNondeterminism(M, F)
 6:    if r != null then
 7:       if r ∈ R then
 8:          cycle = true
 9:       end if
10:       R.add(r)
11:    end if
12: end while
13: return R
```

Figure 3. Fault detection algorithm.

reevaluates active rules against these new changes and checks new non-determinism faults (Lines 4-5 of Part 3), and finally selects a new rule for execution if any. The loop keeps repeated until no more rules can be triggered or the selected rules have already formed a cycle (Line 2 of Part 3). If the loop keeps repeated, this indicates an *adaptation race fault*, in which no new context changes are received but the CAAA keeps triggering itself and making continual adaptations. This adaptation process may also form an infinite *adaptation cycle*, in which the same state and rule sequences repeat themselves forever.

3.3 Incremental Rule Evaluation (IRE)

Our fault detection algorithm works at runtime and its time complexity is a key issue. In the algorithm, when a context change is received, function `evaluateRules` is invoked to check whether this context change would cause any active rules to be triggered. The check needs to reevaluate all active rules. Later when a triggered rule is selected for execution, new context changes may be produced as a result of the rule's action. These new context changes have to be examined again by function `evaluateR-`

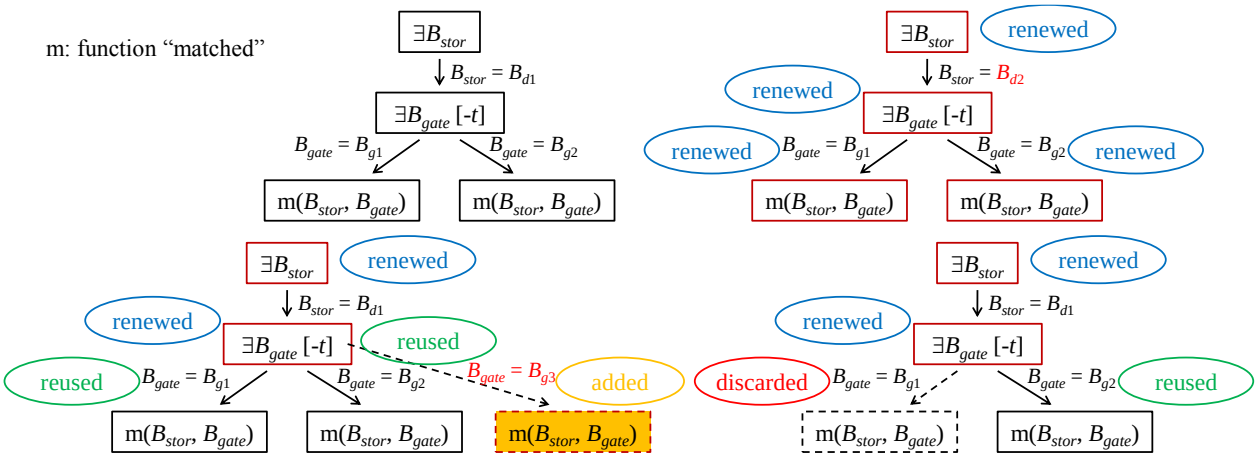

Figure 4. Top-left: Rule r6's RET; top-right: context update, bottom-left: context addition, bottom-right: context deletion.

`ules` to see whether any new rules are triggered. As such, this function may be called multiple times in one run. To study its complexity at runtime, we conducted initial experiments to investigate the percentage of time spent on this function against the total context processing time. We found that the percentage could be up to 99%. This result suggests that reducing the time complexity for function `evaluateRules` is the key to improving the overall efficiency of our runtime fault detection.

3.3.1 *Rule Evaluation Tree (RET)*

We propose an incremental rule evaluation (IRE) technique to address this concern. IRE builds on our previous work on partial constraint checking [18] and extends it to support efficient handling of three types of context change and enhanced reuse of both data structures and evaluation results.

IRE represents each rule in a tree structure (called *rule evaluation tree* or RET). Each layer in a tree corresponds to a nested level of the associated rule. For example, we consider Rule **r6** "$\exists B_{stor}$ ($\exists B_{gate}$[-t] (matched(B_{stor}, B_{gate})))". Suppose that at the time when we evaluate this rule, context variable B_{stor} takes the value of B_{d1} and context variable B_{gate} takes its value from the set of {B_{g1}, B_{g2}}, which have been collected within the past t time. Then Rule **r6** can be represented by an RET as shown in Figure 4 (top-left diagram). The RET spans itself by listing all possible value assignments from currently available contexts. A post-order traversal (i.e., visiting each tree node and evaluating its truth value using its associated value assignment) to the RET would return the truth value of Rule **r6**. Intermediate truth values (called intermediate *evaluation results*) are recorded in each node for later reuse.

3.3.2 *Handling Context Changes*

We consider three types of context change: (1) *Context update*, which modifies the current value of a context variable like B_{stor} in Rule **r6**; (2) *Context addition*, which adds a new value for a context variable like B_{gate} in Rule **r6**; (3) *Context deletion*, which removes a previous value for a context variable like B_{gate}. Context update changes work for simple rules that are specified by a propositional logic based language. Handling such context changes supports the case that the current value of a context variable is replaced by the latest value. Context addition and deletion changes work for complex rules that are specified by a first-order logic based language. Handling such context changes supports the case that a context variable represents a set of contexts restricted by spatial or temporal conditions and this set is altered by inserting new contexts (received from environments) or removing previous contexts (obsolete due to certain timeliness requirements).

We outline our ideas in the following:
(1) Reuse: Previous evaluation results are reused if they are still useful (both node structures and evaluation results are reused);
(2) Renew: Previous evaluation results are renewed if they have become obsolete (node structures are reused but evaluation results need update);
(3) Discard: Previous evaluation results are discarded if they are no longer useful (both node structures and evaluation results are discarded);
(4) Add: New evaluation results are added if necessary (both node structures and evaluation results are newly created).

Context update change. Suppose that a context update change modifies context variable B_{stor}'s value from B_{d1} to B_{d2} as shown in Figure 4 (top-right diagram). All nodes in the RET are reusable but their evaluation results need update. IRE would first locate the node that contains context variable B_{stor} and then update its branch "$B_{stor} = B_{d1}$" to "$B_{stor} = B_{d2}$". Since all nodes in this branch are affected by this update, their evaluation results need renewal. Finally, the node "$\exists B_{stor}$" renews its own evaluation result.

Context addition change. Suppose that a context addition change adds a new context B_{g3} to the context set of {B_{g1}, B_{g2}} for context variable B_{gate} as shown in Figure 4 (bottom-left diagram). IRE would first locate the node that contains context variable B_{gate} and then add one branch to it. The new branch corresponds to the value assignment of $B_{gate} = B_{g3}$ and a new node "matched(B_{stor}, B_{gate})" (rightmost one) is created for this new assignment. IRE would then evaluate this node as it is completely new, but both the node structures and previous evaluation results of the other two "matched(B_{stor}, B_{gate})" nodes (two left ones) can be fully reused. Finally, for the two top nodes "$\exists B_{gate}$[-t]" and "$\exists B_{stor}$", their nodes are reusable but IRE would renew their evaluation results as their children have updated evaluation results.

Context deletion change. Suppose that a context deletion change removes a previous context B_{g1} from the context set of {B_{g1}, B_{g2}} for context variable B_{gate} as shown in Figure 4 (bottom-right diagram). IRE would first locate the node that contains context vari-

able B_{gate} and then remove the branch corresponding to the value assignment $B_{gate} = B_{g1}$ from it. All nodes on this branch are discarded, but the nodes and evaluation results on the "$\exists B_{gate}[\text{-}t]$" node's other branches are still reusable. Finally, IRE would renew the evaluation results for the two top nodes "$\exists B_{gate}[\text{-}t]$" and "$\exists B_{stor}$" because their children have updated evaluation results, but their node structures keep unchanged and thus still reusable.

3.3.3 Time Complexity

Let the number of nodes in an RET be n and one node visit be the unit time. Due to the reuse of node structures and evaluation results, the time complexity is O(1)~O(n) for handling a context update or addition change and is always O(1) for handling a context deletion change. Without IRE, one has to perform complete recreation and reevaluation for each affected rule, no matter which type of context change it is. The time complexity is always O(n).

When IRE reaches its worst-case complexity, the spent time is still much less than the time required by a non-incremental technique. This is because IRE handles affected nodes only, whereas a non-incremental technique has to recreate and reevaluate whole RETs. We shall illustrate the significant performance difference between the two techniques in later evaluation.

3.4 Comparison

We compare our AM approach with A-FSM as follows. Given a CAAA that is specified in our AM model, we project it on simple rules. The resulting model is named AM_{simple} model. We apply A-FSM to this model and the set of its detected faults is named $\text{Faults}_{simple,\ \text{A-FSM}}$. We then apply our AM approach to the AM model directly, and we name the set of its detected faults $\text{Faults}_{all,\ \text{AM}}$. We further divide $\text{Faults}_{all,\ \text{AM}}$ into two subsets: $\text{Faults}_{simple,\ \text{AM}}$ and $\text{Faults}_{other,\ \text{AM}}$. The former contains those faults that are related to simple rules only and the latter is defined as: $\text{Faults}_{all,\ \text{AM}} - \text{Faults}_{simple,\ \text{AM}}$.

We study the following three parts of faults in the comparison:

(1) Real faults: $\text{Faults}_{simple,\ \text{A-FSM}} \cap \text{Faults}_{simple,\ \text{AM}}$. This part gives the faults related to simple rules only and they must be real faults. This is because they are detected by A-FSM using static analysis and also confirmed by our AM approach in runtime detection.

(2) Potential false positives: $\text{Faults}_{simple,\ \text{A-FSM}} - \text{Faults}_{simple,\ \text{AM}}$. This part gives the faults related to simple rules and they are potentially false positives. This is because these faults are detected by A-FSM but not confirmed by our AM approach. As A-FSM has not taken into account the impact of variable dependency, physical constraints, and rule actions, these faults are potentially unreal and need further analysis.

(3) False negatives: $\text{Faults}_{simple,\ \text{AM}} - \text{Faults}_{simple,\ \text{A-FSM}} \cup \text{Faults}_{other,\ \text{AM}}$. This part gives the faults missed by A-FSM but reported by our AM approach. They are real faults because they have been observed. They have been missed by A-FSM due to various reasons (e.g., some rule actions have modified contexts at runtime or some faults relate to complex rules).

We study these three parts of faults for the comparison of our AM and existing A-FSM in the following evaluation.

4. EVALUATION

In this section, we conduct simulated experiments for the stock tracking application and real-world experiments for a self-controlling mini-car system. We study two research questions:

(1) Effectiveness: *Compared to A-FSM, how effective can our AM approach be in detecting CAAA faults?*

(2) Efficiency: *How efficient does our AM approach support runtime fault detection in CAAAs?*

4.1 Stock Tracking Application

We set up simulated stock tracking scenarios based on parameters from our engineers according to practical environments. Each forklift transportation took 20-60 seconds. In each transportation, the forklift carried 50 paper boxes, each of which was attached with an RFID tag for tracking. The cross reading rate and missing reading rate were both set to 5%. Besides, a forklift might be delayed in its return trip to the loading bay for other uses due to resource sharing. The probability of such events was set to 10% and the delay thus incurred could be up to 40 seconds.

The experiments were conducted on a PC with an Intel® Pentium® 4 3.2GHz CPU and 2GB RAM. The operating system is Windows XP Professional SP3. We implemented both A-FSM and AM in Java (Oracle/Sun JRE 7) and integrated them in our Cabot context middleware [18] as plug-in services. The stock tracking application ran on top of Cabot.

4.1.1 Effectiveness

The AM model for the stock tracking application contains 12 rules as mentioned earlier. The corresponding AM_{simple} model applicable to A-FSM contains 6 simple rules **r1-5** and **r12**. A-FSM did not detect any non-determinism faults but detected 164 adaptation race faults and 12 adaptation cycle faults (both belong to instability faults) on the AM_{simple} model. On the other hand, AM detected faults on the AM model directly. AM detected 1 non-determinism fault and 2 adaptation race faults only. Seemingly A-FSM has detected much more faults but we are interested in their quality. We measure the three parts of faults as below.

(1) Real faults: There is only one adaptation race fault detected by both approaches. The fault shows that an adaptation race occurred at the "energy_saving" state when the "start_loading" rule was executed and was immediately followed by the triggering of the "start_transporting" rule. This happened when a forklift and its pallet were taken for other uses at the "energy_saving" state and the pallet came back earlier than the forklift. As a result, the application's state transited from "energy_saving" to "loading" and then quickly to "transporting". The activity of reading out the RFID tags of paper boxes might be skipped at the "loading" state.

(2) Potential false positives: A-FSM's detected other 163 adaptation race faults and all 12 adaptation cycle faults are potential false positives. After our analysis, all of them were invalidated by variable dependency or physical constraints. First, 132 adaptation race and 12 cycle faults violate the variable dependency among four context variables $\text{flArv}(R_{load})$, $\text{flGone}(R_{load})$, $\text{flArv}(R_{stor})$ and $\text{flGone}(R_{stor})$ (e.g., $\text{flArv}(R_{load})$ and $\text{flGone}(R_{load})$ cannot both take true). Then, the remaining 31 adaptation race faults violate four physical constraints "pallet/forklift cannot be at two places at the same time", "pallet and forklift must go together", "forklift cannot jump from the loading bay to the storage bay or back suddenly" and "paper box unloading cannot take zero time".

(3) False negatives: There are two faults (1 non-determinism fault and 1 adaptation race fault) detected by AM but missed by A-FSM. The non-determinism fault occurred at the "loading" state,

Table 1. Effectiveness comparison.

Faults (#)	Nondeterminism	Adaptation race	Adaptation cycle
A-FSM	0 (1 missing)	164 (1 missing; 163 unreal)	12 (all unreal)
AM	1	2	0

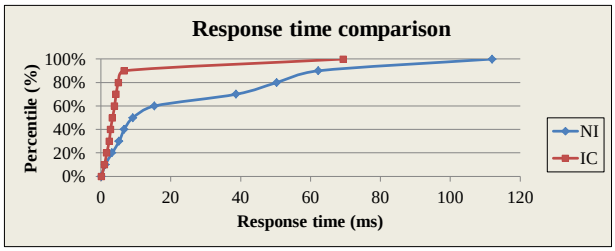

Figure 5. Response time comparison.

where both rules "save_energy" and "start_transporting" were triggered. This happened when the forklift stayed at the loading bay and the pallet had been taken away for some time, and then the forklift was also taken away and the application would face a non-deterministic situation because both rules were triggered. The adaptation race fault occurred at the "unloading_1" state when the application moved to the "unloading_2" state where some missing readings were present, and then the application would immediately enter the "missing_reading" state. The preparation for missing reading checking might be skipped at the "unloading_2" state.

Table 1 summarizes these effectiveness comparison results.

4.1.2 Efficiency

Processing time. We measure AM's efficiency with and without the IRE technique (named IC and NI, respectively). We simulated forklift transportation 50 times and measured IC's and NI's total context processing time. NI took 209,261 ms to process all contexts, whereas IC took only 43,413 ms (20.7%).

Response time. We study the application's response time in Figure 5. *Response time* measures the interval between an application receives a context and it takes adaptive action. For NI, 80% response times are less than 50.4 ms, 90% are less than 62.3 ms, and the average is 23.0 ms. For IC, the three values are greatly reduced to 5.0 ms, 6.7 ms, and 4.1 ms, respectively.

Scalability. We also compare IC's and NI's scalability and study its impact on rule triggering. In order not to change the application semantics, we kept all rules and contexts unchanged. We set up a scale factor F (from 1 to 10) to control the number of transportations running in parallel. The number of contexts to be processed in unit time would thus increase and the interval between contexts would decrease accordingly.

From Figure 6, NI's occupied time increases quickly with F's growth. *Occupied time* measures the percentage of context processing time against the total time allowed. IC's occupied time grows much slower. We observe that starting from F = 5, NI's occupied time becomes close to 100% and its rule satisfaction number decreases quickly instead (down to below 60% as compared to IC's value at F = 10). When this number decreases, rules cannot be triggered in time and this would affect the application's responsiveness. IC's rule satisfaction number keeps quite stable.

We owe AM's efficiency to its IRE technique. Our measurement

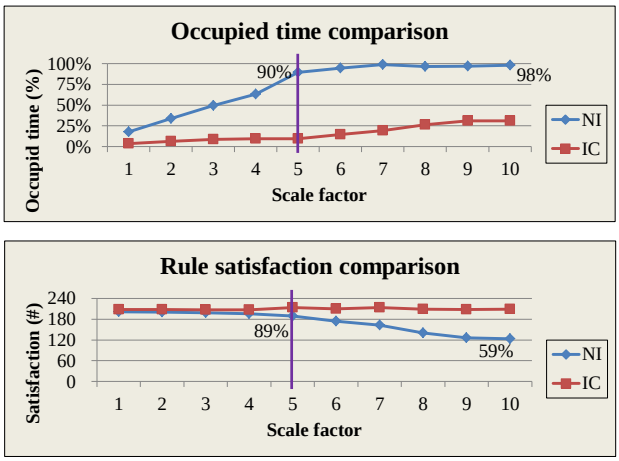

Figure 6. Scalability comparison.

showed that NI created 43,993,249 tree nodes in one transportation on average, whereas IC created only 903,829 nodes (2.0%). IC realized this by reusing 42,524,948 nodes (96.7%) and renewing 564,472 nodes (1.3%). The reuse of node structures and evaluation results helped AM run much more efficiently at runtime.

4.2 Self-controlling Mini-car System

We then apply our AM approach to a real-world self-controlling mini-car system. The mini-car is developed on Cirrus Logic EDB9302 Board (ARM920T) with a TelosB mote for wireless communication. The car is equipped with eight ultrasonic sensors at four orientations (two sensors at each orientation). The two rear wheels of the car are programmable for different speeds and the two front wheels are equipped with two speed sensors.

A mini-car console program runs on the car. It collects contexts (ultrasonic and speed sensor data) periodically (every 350 ms) and reports them to a PC console program running on a PC. The PC console program analyzes collected contexts, evaluates user-specified adaptation rules, and sends back decisions to guide the car to move. The PC console program aims at guiding the car to explore an unknown map. It supports *automatic speed adjustment* (high-speed mode HS and low-speed mode LS) and *automatic obstacle avoidance* (normal movement mode M and obstacle avoidance mode A). The combinations of the four modes form a state transition diagram as shown in Figure 7.

The application starts with the initial "Stop" state, where it moves to one of three different states, depending on its contexts. For example, it goes to the "HS-M" state if no obstacle is detected nearby; otherwise, it goes to the "LS-A" or "LS-M" state, depending on whether the detected nearby obstacle is very close or not. The application's state keeps changing when surrounding obstacle conditions change for the purpose of exploring the map efficiently and at the same time keeping safety.

The application contains 5 states and 12 rules. All rules are complex ones because they check all ultrasonic sensors' reported data within a recent time interval to decide the obstacle condition for each orientation. For example, the "lsm_2_hsm" rule drives the car from state "LS-M" to "HS-M" when all sensors report the "clear" condition recently: $\forall OC_{all}[-t]$ (clear(OC_{all})). Another rule "lsa_2_hsa" enables the car from state "LS-A" to "HS-A" if

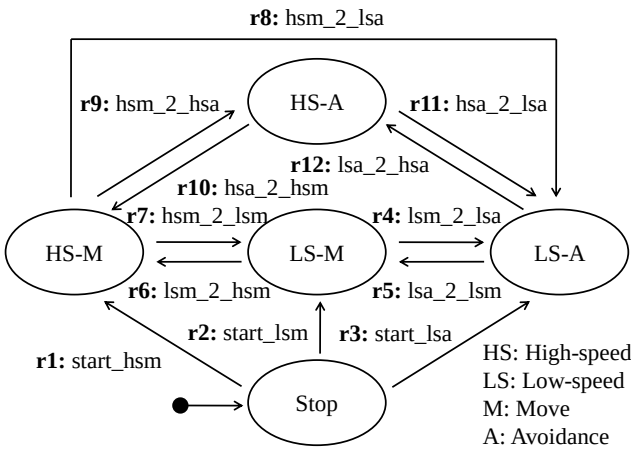

Figure 7. State transition diagram of the mini-car application.

no sensor reports the "dangerous" condition recently and the obstacle avoidance function is not so frequently called recently: (not ($\exists OC_{all}[\text{-}t]$ (danger(OC_{all})))) and (not ($\exists$HS[-t] (frequent(HS))))). Besides, when the car gets stuck (judged from speed sensors), the application would clear all contexts, resets its state to "Stop", and then restart itself. A-FSM is not applicable to this application due to the presence of complex rules and runtime context changes (A-FSM assumes that rule actions do not modify contexts).

We ran the mini-car in a 5.0m × 3.5m office environment that contains desks, chairs, and some cabinets. The application ran on a PC with the same hardware and software configurations as explained earlier. We ran the mini-car 10 times, each taking 2.5 minutes due to battery limitation.

AM detected a total of 6 distinct non-determinism (D1-6) and 8 distinct adaptation race faults (R1-8) with various occurrences, which differed greatly. As shown in Figure 8, the distribution of non-determinism (ND) faults vary from 3.1-36.3% and that of adaptation race (AR) faults vary from 0.3-44.7%. This shows that some faults are really hard to detect. The hardest non-determinism fault (5 occurrences) happened at the "HS-M" state where three rules "hsm_2_lsm", "hsm_2_lsa", and "hsm_2_hsa" were all triggered. The hardest adaptation race fault (2 occurrences) happened also at the "HS-M" state where rules "hsm_2_hsa" and "hsa_2_lsa" were triggered in turn. We found that both faults happened only when the two front ultrasonic sensors reported inconsistent obstacle conditions ahead (e.g., one reported that "the car is approaching some obstacle but there is still some distance" and the other reported that "the obstacle is very close and the situation is dangerous"). We understand that we may not be able to easily control real-world contexts in practice and this explains why it is so difficult to observe these faults at runtime. Traditional coverage information (e.g., state reachability, liveness and rule liveness [13][14]) may not be so useful because we have measured that all states and rules have been covered 100%. We plan to further study how to detect such hard faults in our future work.

Finally, we note that the overhead of our AM approach is very small. For the mini-car application, its occupied time is only as high as 3.0%. For response time, 80% of them are less than 1.1 ms, 90% are less than 2.3 ms, and the average time is 1.0 ms. It shows that our runtime fault detection only incurred negligible time in addition to normal application execution.

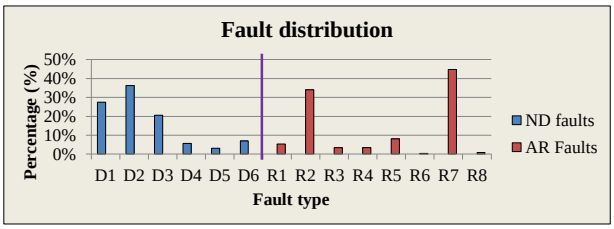

Figure 8. Fault distribution in the mini-car application.

5. RELATED WORK

Context-aware computing is receiving increasing attention. Applications become context-aware by perceiving environmental changes and making seamless adaptation. To support such applications, various middleware infrastructures [1][7][11][18] and application frameworks [3][4] have been proposed to assist application development and deployment. Many interesting applications have been proposed, such as highway collision avoidance system [7], Roaming Jigsaw [11], ConChat [12] and Call Forwarding [16].

Many developers have assumed that: (1) contexts can be *correctly* received from environments, and (2) adaptations can be *correctly* applied once triggering conditions are satisfied. Recent researchers have observed from practice that contexts are often noisy, inaccurate, and easily obsolete. This often causes applications to run in an unexpected way if they have not considered unreliable contexts adequately. Deshpande et al. [2] proposed to use threshold smoothing to filter out unreasonable contexts. Our previous work [18] presented a constraint checking approach to identify correlated contexts whose inconsistency is difficult to judge by threshold. To repair contexts, Jeffery et al. [6] proposed a probabilistic approach to fix missing RFID contexts. Our previous work [17] complemented it in a heuristic way to enforce general contexts to satisfy consistency constraints.

While researchers are making efforts to improve the reliability of contexts, applications may still suffer unexpected faults at runtime when non-determinism and adaptation races or cycles arise. Insuk et al. [5] tolerated non-determinism as long as simultaneously triggered adaptations do not conflict with each other in semantics. Sama et al. [13][14] set up a stricter criterion by disallowing any occurrences of non-determinism and adaptation races or cycles. They realized this by statically exploring all application space. Lacking dynamic information makes their approach possibly report false positives as we analyzed earlier. This concern is also echoed by Capra et al. [1] who believed that removing adaptation conflicts should be conducted dynamically.

Researchers from software testing communities detected faults for context-aware applications by focusing on new testing coverage criteria for data flows affected by context changes [8][9] and test case generation incorporating contexts into context-aware program points [15]. We in this paper aim at improving A-FSM's expressive power and fault detection precision. We also enhance the algorithm's efficiency to make it useful for practical applications. This complements our previous efforts on studying the correlations between errors and failures in CAAAs [19].

6. CONCLUSION

In this paper, we have carefully reviewed the existing A-FSM approach for detecting adaptation faults in context-aware adaptive

applications (CAAAs), and pointed out two places for improvement. We have proposed our AM approach and explained how it can be used to model complex rules and improve fault detection precision by removing false positives. Our approach also supports runtime fault detection by an efficient rule evaluation technique.

We have evaluated our AM approach and compared it to A-FSM using our two applications. Both applications adapt to environments based on contexts and adaptation rules. One concern is why we do not use other published CAAAs in the evaluation. There are two reasons: (1) The stock tracking application is derived from a practical RFID application scenario with real parameters observed by our engineers; (2) The mini-car application runs on a real hardware platform, which collects real-world, noisy contexts from environments. By this, we try to make the experiments realistic and measure AM's effectiveness and efficiency in practice.

Detected non-determinism faults can be solved by rule priority adjustment but solving adaptation race or cycle faults need more analysis. If the quality of reported faults is so low such that most faults are false positives, then a great deal of analysis effort would be wasted. Our AM approach helps by reporting real faults only. In addition, AM also reports the occurrence for each distinct fault. This provides additional information on which faults are more frequent and thus more urgent. We note that this information is collected at runtime and thus reflects real fault conditions that are different from those collected by a static analysis approach.

An interesting remaining issue is how to enhance our AM approach's capability to detect those hard faults that rarely happen at runtime. We are going along this line.

ACKNOWLEDGMENTS

This research was funded by National Basic Research Program (973 Program 2009CB320702), National High-Tech Research & Development Program (863 program 2011AA010103), National Natural Science Foundation (61100038, 61021062) of China, and by Research Grants Council (612210, 612309) of Hong Kong. Chang Xu was also supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

REFERENCES

[1] Capra, L., Emmerich, W. and Mascolo, C. CARISMA: Context-aware Reflective Middleware System for Mobile Applications. *IEEE Trans. on Software Engineering 29*, 10 (Oct), 2003, 929-945.

[2] Deshpande, A., Guestrin, C. and Madden, S. Using Probabilistic Models for Data Management in Acquisitional Environments. In *Proc. the 2nd Biennial Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, Jan 2005, 317-328.

[3] Dey, A.K., Abowd, G.D. and Salber, D. A Context-based Infrastructure for Smart Environments. In *Proc. the 1st International Workshop on Managing Interactions in Smart Environments*, Dublin, Ireland, Dec 1999, 114-128.

[4] Henricksen, K. and Indulska, J. A Software Engineering Framework for Context-aware Pervasive Computing. In *Proc. the 2nd IEEE Conf. on Pervasive Computing and Communications*, Orlando, Florida, USA, Mar 2004, 77-86.

[5] Insuk, P., Lee, D. and Hyun, S.J. A Dynamic Context-conflict Management Scheme for Group-aware Ubiquitous Computing Environments. In *Proc. the 29th Annual International Computer Software and Applications Conf.*, Edinburgh, UK, Jul 2005, 359-364.

[6] Jeffery, S.R., Garofalakis, M. and Frankin, M.J. Adaptive Cleaning for RFID Data Streams. In *Proc. the 32nd International Conf. on Very Large Data Bases*, Seoul, Korea, Sep 2006, 163-174.

[7] Julien, C. and Roman, G.C. EgoSpaces: Facilitating Rapid Development of Context-aware Mobile Applications. *IEEE Trans. on Software Engineering 32*, 5 (May), 2006, 281-298.

[8] Lu, H., Chan, W.K. and Tse, T.H. Testing Context-aware Middleware-centric Programs: A Data Flow Approach and a RFID-based Experimentation. In *Proc. the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Portland, Oregon, USA, Nov 2006, 242-252.

[9] Lu, H., Chan, W.K. and Tse, T.H. Testing Pervasive Software in the Presence of Context Inconsistency Resolution Services. In *Proc. the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, 61-70.

[10] Lu, J., Ma, X., Huang, Y., Cao, C., and Xu, F. Internetware: A Shift of Software Paradigm. In *Proceedings of the 1st Asia-Pacific Symposium on Internetware*, 2009.

[11] Murphy, A.L., Picco, G.P. and Roman, G.C. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Trans. on Software Engineering and Methodology 15*, 3 (July), 2006, 279-328.

[12] Ranganathan, A., Campbell, R.H., Ravi, A. and Mahajan, A. ConChat: A Context-aware Chat Program. *IEEE Pervasive Computing 1 (3)*, Jul-Sep 2002, 51-57.

[13] Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., and Wang, Z. Context-aware Adaptive Applications: Fault Patterns and Their Automated Identification. *IEEE Trans. on Software Engineering 36*, 5 (Sep/Oct), 2010, 644-661.

[14] Sama, M., Rosenblum, D.S., Wang, Z. and Elbaum, S. Model-based Fault Detection in Context-aware Adaptive Applications. In *Proc. the 16th ACM SIGSOFT International Symp. on the Foundations of Software Engineering*, Atlanta, GA, USA, Nov 2008, 261-271.

[15] Wang, Z., Elbaum, S. and Rosenblum, D.S. Automated Generation of Context-aware Tests. In *Proc. the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, 406-415.

[16] Want, R., Hopper, A., Falcao, V. and Gibbons, J. The Active Badge Location System. *ACM Trans. on Information Systems 10*, 1 (Jan), 1992, 91-102.

[17] Xu, C., Cheung, S.C., Chan, W.K. and Ye, C. Heuristics-based Strategies for Resolving Context Inconsistencies in Pervasive Computing Applications. In *Proc. the 28th International Conf. on Distributed Computing Systems*, Beijing, China, Jun 2008, 713-721.

[18] Xu, C., Cheung, S.C., Chan, W.K. and Ye, C. Partial Constraint Checking for Context Consistency in Pervasive Computing. *ACM Trans. on Software Engineering and Methodology 9*, 3 (Jan), 2010, Article 9, 1-61.

[19] Xu, C., Cheung, S.C., Ma, X., Cao, C., and Lu, J. ADAM: Identifying Defects in Context-aware Adaptation. *The Journal of Systems & Software*, 2012, forthcoming.