Int J Software Informatics, Volume 7, Issue 1 (2013), pp. 85–111 Tel: +86-10-62661040
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org

Email: ijsi@iscas.ac.cn, ijsi2007@gmail.com

Detecting Faults in Context-Aware Adaptation

Chang Xu[1,2], S.C. Cheung[3], Xiaoxing Ma[1,2], Chun Cao[1,2] and Jian Lv[1,2]

[1](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[3](Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China)

Abstract Internetware applications are context-aware. They adapt their behavior based on environmental changes. However, faulty adaptation may arise when these applications face unanticipated situations. Such adaptation faults can be difficult to detect at design time. One promising approach is to statically analyze model-based context-aware applications exhaustively for all potential faults. However, it suffers from expressiveness and precision problems. To address these limitations, we propose in this paper a dynamic adaptation model (AM) approach. AM offers increased expressive power to model complex adaptation rules, and guarantees soundness in its fault detection. In addition, AM deploys an incremental rule evaluation (IRE) technique to cater for context-aware applications, such that it can efficiently handle environmental changes in its fault detection. We evaluated AM using both simulated and real-world experiments with two context-aware applications. The experimental results confirmed that AM can detect real faults missed by existing work, and avoid numerous false warnings that were misreported otherwise.

Key words: context-aware adaptation; fault detection; incremental rule evaluation

Xu C, Cheung SC, Ma XX, Cao C, Lv J. Detecting faults in context-aware adaptation. *Int J Software Informatics*, **Vol.7, No.1 (2013): 85–111. http://www.ijsi.org/1673-7288/7/i148.htm**

1 Introduction

Internetware applications[12] are receiving increasing attention. Recent advancement in key technologies, such as wireless sensor network and radio frequency identifi- cation[4], has boosted the development and deployment of new Internetware applications. These applications continually monitor their environments and make self-adaptation for seamless integration. Such applications are also called *context-aware adaptive applications* (or CAAAs for short)[15,16]. Typical *contexts* used in CAAAs include object location, environmental noise, and any other piece of information, as long as they affect the computation in an application. One example CAAA is a PhoneAdapter application[15,16]. It can automatically mute a smart-

---

This research was partially funded by National Basic Research Program of China (973 Program 2009CB320702), National High-Tech Research & Development Program of China (863 program 2011AA010103), National Natural Science Foundation of China (61100038, 61021062), 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).

Corresponding author: Chang Xu, Email: changxu@nju.edu.cn

Received 2012-10-01; Revised 2012-12-10; Accepted 2013-01-08.

phone's ring tone and activate its vibration mode when its user is in office, and disable vibration when the user attends a meeting.

To support CAAA development and deployment, various middleware infrastructures[1,8,13,19,24,25] and application frameworks[3,5] have been studied and proposed. They commonly follow an intuitive computational model. In this model, two design concerns are well separated: (1) acquiring contexts from environments, and (2) executing adaptation based on these contexts.

This computational model allows developers or users to specify *adaptation rules* (or rules for short) that govern how their applications should react to contexts and their changes. Adaptation rules thus play an important role in deciding an application's runtime behavior. This model is simple yet powerful. However, it may be exposed to potential threats to its correctness, especially when a CAAA encounters the situations never anticipated at design time. For example, when multiple rules are triggered at the same time, a CAAA may randomly select one of them for execution. Then the application's state can become unpredictable due to this non-determinism. Besides, if one rule can be triggered without taking any new context, the concerned CAAA would have its state unstable. The state's duration would depend unpredictably on context update rate or rule execution speed. In either case, the CAAA may fail to adapt as expected.

Each CAAA may have its own criterion of deciding application-specific faults. Still, CAAAs can share some common fault patterns. For example, when properties like *determinism* (whether a CAAA is always clear about its next executed rule) and *stability* (whether a CAAA's state is always stable after each adaptation) are violated, an application would probably run in an unpredictable or unstable way, exhibiting unexpected behavior. This is usually undesirable.

To detect such adaptation faults, one promising approach is static analysis. A recent piece of work targeting this problem is adaptation finite- state machine (A-FSM)[15,16]. A-FSM first constructs a search space for all possible values to be assigned to context variables used in a CAAA's rules. It then exhaustively explores this space to find all potential faults. This approach works, but it also contains two limitations. First, A-FSM does not have sufficient expressive power to specify complex adaptation rules. As such, some recently published or pratical CAAAs cannot be precisely modeled. Second, A-FSM does not take into account the impact of variable dependency, physical constraints, and rule actions on its fault detection. As such, it may report numerous false warnings (i.e., unreal faults).

In this paper, we present a novel adaptation model (AM) approach to address these two limitations. We base AM on existing A-FSM, and improve it by both increasing its model's expressive power and avoiding reporting false warnings in fault detection. In addition, we deploy an incremental rule evaluation (IRE) technique to enhance AM's runtime efficiency, so that it can be used for practical context-aware applications, which are subject to continual environmental changes.

The remainder of this paper is organized as follows. Section 2 presents our motivating example for explaining 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 its adaptation model, to fault detection algorithm, and to runtime rule evaluation technique. Section 4 evaluates

AM and compares it to A-FSM using a simulated stock tracking application and a real-world self-controlling robot-car application. Finally, Section 5 discusses the related work, and Section 6 concludes this paper.

2 Motivating Example and Problem Analysis

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

2.1 Stock tracking application

Our example was adapted from our pilot study of an RFID-enabled stock tracking application in an international paper company (RFID stands for radio frequency identification).

The application controls a forklift to transport RFID-tagged paper boxes from the loading bay of a warehouse to its storage bay, as illustrated in Fig. 1. RFID gates are installed at these two bays to collect contexts (e.g., RFID codes and detection locations of the paper boxes) in this stock tracking process. *Missing readings* can occur because RFID codes of some paper boxes may not be successfully detected at any time. This problem can be alleviated by increasing RFID antennae's transmission power or altering their orientations. However, this treatment may instead cause RFID codes of some paper boxes to be accidentally detected by nearby irrelevant RFID gates. These wrongly detected RFID codes are called *cross readings*. Missing readings and cross readings are very common in practical RFID deployments[7,23].

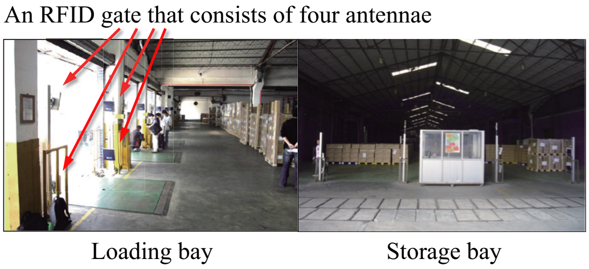

Figure 1. Loading bay (left) and storage bay (right) in a warehouse

The stock tracking application runs in a finite-state machine style, as illustrated in Fig. 2. It consists of a set of states and adaptation rules connecting them. The application starts with its initial "loading" state, at which a forklift is controlled to load a pallet of paper boxes at a loading bay gate. Each paper box is attached with an RFID tag, which carries a unique RFID code to distinguish this box from others. When a paper box is passing by an RFID gate, its associated RFID code would be read and recorded. After the box loading is complete, the application proceeds to the "transporting" state. At this state, the forklift is controlled to transport a pallet of paper boxes from the loading bay to the storage bay, where these boxes are to be unloaded. Box unloading would cause the application to enter the "unloading_1" state, which is then followed by the "unloading_2" state (i.e., two unloading steps). During these two steps, RFID codes of all paper boxes are read again to check whether there is any cross reading or missing reading problem. The checking is conducted with respect to two constraints: (1) During the transportation, RFID codes of these paper boxes should not be read by nearby irrelevant RFID gates; (2) The two sets of RFID codes collected at the loading bay and unloading bay must match. If any cross reading occurs, it would be discarded at the "cross_reading" state. If any missing reading occurs, it would be recovered at the "missing_reading" state. After the checking is complete, the application proceeds to the "returning" state. At this state, the forklift is controlled to come back to the loading bay for its next transportation task. If there is no such task, the application would enter the "energy_saving" state after a period of time, and put all RFID gates into sleep for saving energy, until it receives its next transportation task.

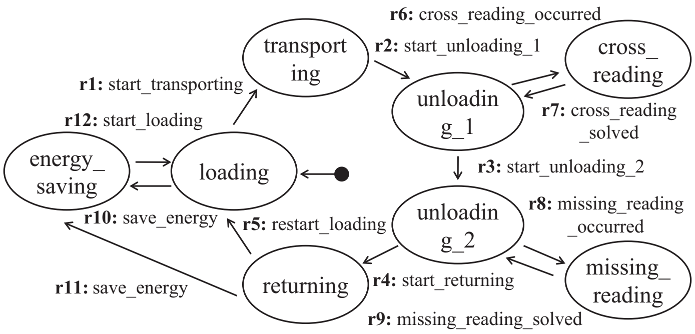

Figure 2. Stock tracking application
(its state transition diagram with 8 states and 12 rules)

In summary, the stock tracking application implements the following three functional features:

1. **Normal workflow.** The application runs as a complete work loop for transporting paper boxes from the loading bay to the storage bay. This feature covers "loading", "transporting", "unloading_1", "unloading_2", and "returning" five states. These states associate five adaptation rules **r1-5**.

2. **Exception handling.** The application checks for cross readings and missing readings in collected RFID codes for paper boxes, and fixes these problems if necessary. This feature covers "unloading_1", "cross_reading", "unloading_2", and "missing_reading" four states. These states associate four adaptation rules **r6-9**.

3. **Energy-awareness**. The application can switch between its normal working mode and energy-saving mode. This feature covers "energy_saving", "loading", and "returning" three states. These states associate three adaptation rules **r10-12**.

2.2 Application modeling and rule types

As mentioned, adaptation rules play an important role in specifying an application's behavior upon context changes. An expressive modeling language is thus required for defining rule conditions based on context changes. In our stock tracking application, its contexts mainly include two types: (1) pressure sensor readings (these sensors are installed at the loading bay and storage bay, and their reading values indicate whether a forklift has arrived or left a bay); (2) RFID readings (for tracking the pallets and paper boxes being transported). The application needs to check these contexts and their changes so as to decide its behavior. For this stock tracking application, we aruge that a language based on propositional logic (e.g., the one used in A-FSM) does not suffice for this modeling purpose.

A-FSM's modeling language defines adaptation rules by logical formulae. These formulae use propositional context variables and three logical operators ("and", "or", "not") connecting them. For example, one can specify a rule like "(flArv(r_S)) and (pallet(t_S))" using two propositional context variables "flArv(r_S)" and "pallet(t_S)" to check whether one pressure sensor (r_S) reports the arrival of a forklift at the storage bay as well as its associated pallet (t_S) being found there. Such propositional context variables return "true" or "false" only. We observe that the rules expressed in propositional logic are based only on *current values* of the concerned context variables. For example, the preceding rule uses "flArv(r_S)" and "pallet(t_S)" 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 such rules *simple rules.*

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

The stock tracking application includes both simple and complex rules. In fact, complex rules are quite common in modeling recently published or practical CAAAs. For example, a ConChat application[14] checks whether *all* members of a group are in a specific meeting room, or *any one* of them is in another room. Another context-aware communication application[5] checks whether its user is occupied in *any one* of all possible time slots. In these rules, first-order logic is commonly used to retrieve every context from a set that is restricted by spatial or temporal conditions.

2.3 Modeling language and application specification

We adopt the following first-order logic based language to specify adaptation rules in CAAAs:

$$\begin{aligned} \textit{rule} := {} & \forall v \in S[t]\ (\textit{rule})\ |\exists v \in S[t]\ (\textit{rule})\ |(\textit{rule}) \text{ and } (\textit{rule})\ | \\ & (\textit{rule}) \text{ or } (\textit{rule})\ |(\textit{rule}) \text{ implies } (\textit{rule})\ |\text{not } (\textit{rule})\ | \\ & \textit{func}(v,\ v,\ \ldots). \end{aligned}$$

By using this language, an adaptation rule is specified recursively. By universal or existential quantifier, a rule defines a context variable (v) that can take any arbitrary value from a set ($S[t]$). The set can be optionally restricted by an interval t. When the interval is there, the set contains all contexts collected within this interval; otherwise, the set contains the latest context only. By this, the language supports both simple and complex rules. Developers or users can also define application-specific functions (*func*). To illustrate the use of this language for modeling CAAAs, we specify all 12 adaptation rules in the stock tracking application.

2.3.1 Specification: normal workflow

The whole application specification consists of three parts. The first part covers the "normal workflow" feature. It contains five adaptation rules **r1-5**:

r1("start_transporting"):

$\exists r_L \in R_{load}$ (flGone(r_L)).

r2("start_unloading_1"):

($\exists r_S \in R_{stor}$ (flArv(r_S))) and ($\exists t_S \in T_{stor}$ (pallet(t_S))).

r3("start_unloading_2"):

$\exists c_S \in C_{stor}$ (proceed(c_S)).

r4("start_returning"):

$\exists r_S \in R_{stor}$ (flGone(r_S)).

r5("restart_loading"):

$\exists r_L \in R_{load}$ (flArv(r_L)).

In these five rules, context variables r_L and r_S report the latest status for two pressure sensors installed at the loading bay and storage bay, respectively. Functions "flGone" and "flArv" check whether the forklift has left or just arrived according to sensor readings, respectively. Another context variable t_S reports the most recent RFID reading for the pallet carrying paper boxes at the storage bay, and function "pallet" checks whether this reading belongs to the pallet being used by the forklift. Finally, context variable c_S contains the latest controlling signal that indicates whether or not the paper box RFID reading has completed at the storage bay and the application can proceed to the next checking phase (by function "proceed").

We discuss one example. According to Rule **r2**, when the application detects that the forklift has arrived at the storage bay (flArv(r_S) = true) and its pallet also appears there (pallet(t_S) = true), this rule is triggered and executed. As a result, the application enters the "unloading_1" state.

2.3.2 Speficiation: exception handling

The second part covers the "exception handling" feature. It contains four adaptation rules **r6-9**:

r6("cross_reading_occurred"):

$\exists b_S \in B_{stor}$ ($\exists b_G \in B_{gate}[-t]$ (matched(b_S, b_G))).

r7("cross_reading_solved"):

$\forall b_S \in B_{stor}$ (not ($\exists b_G \in B_{gate}[-t]$ (matched(b_S, b_G)))).

r8 ("missing_reading_occurred"):

$\exists b_L \in B_{load}$ ($\forall b_S \in B_{stor}[t]$ (not (matched(b_L, b_S)))).

r9 ("missing_reading_solved"):

$\forall b_L \in B_{load}$ ($\exists b_S \in B_{stor}[t]$ (matched(b_L, b_S))).

For this feature, all rules **r6-9** are complex rules. They cannot be specified in a propositional logic based language like the one used in A-FSM.

Rule **r6** checks whether any paper box RFID reading collected at the storage bay matches any paper box RFID readings collected earlier by another irrelevant RFID gate within a past interval t. The value of t can be set to reasonable transportation time between the loading bay and storage bay, such that any matched RFID reading is considered *cross reading*. Rule **r7** reverses rule **r6**'s condition to make sure that all cross readings have been properly handled. Rules **r8** and **r9** work in a pair similarly. Rule **r8** checks whether there is any paper box no longer detectable at the storage bay, after its RFID reading has been successfully collected at the loading bay. Rule **r9** reverses rule **r8**'s condition to make sure that all such missing readings have been properly handled. As these rules show, our modeling language is expressive in specifying the rules that refer to both historical contexts (when using $[-t]$) and future contexts (when using $[t]$).

2.3.3 Specification: energy-awareness

The third part covers the "energy-awareness" feature. It contains three adaptation rules **r10-12**, in which **r10** and **r11** are the same:

r10/11 ("save_energy"):

$(\exists r_L \in R_{load}$ (flGone(r_L))) and (not $(\exists t_L \in T_{load}[-t]$ (pallet(t_L)))).

r12 ("start_loading"):

$(\exists r_L \in R_{load}$ (flArv(r_L)) or $(\exists t_L \in T_{load}[-t]$ (pallet(t_L))).

This 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 a period of time (already timeout). Rule **r10/11** applies to both the "loading" and "returning" states. The last Rule **r12** brings the application back to its normal working mode if either of the above two condition is no longer satisfied.

2.4 Fault detection precision analysis

Within the scope of this paper, we focus on two adaptation faults commonly found in CAAAs[15,16]. They are *non-determinism fault* (violating the determinism property) and *instability fault* (violating the stability property). As mentioned, the existing A-FSM approach can detect such faults by static analysis. However, we also note that A-FSM has made two implicit assumptions that have affected its fault detection precision.

2.4.1 Assumption 1: using a propositional logic based specification language

A-FSM supports only those CAAAs whose adaptation rules are specified by a propositional logic based language. By doing so, the number of context variables used in a rule is *fixed*, and at the same time they can only take "true" or "false" as its value. This is important and necessary, because otherwise A-FSM cannot exhaustively explore all possible value assignments made to these context variables. When detecting non-determinism faults, A-FSM enumerates all value assignments in each state and examines whether any of them would trigger more than one rule at the same time. When detecting instability faults, A-FSM enumerates all value assignments in each state, and examines when any rule is triggered and executed, whether another rule is already triggered without taking any new context changes.

Since A-FSM adopts a propositional logic based specification language, it models and detects faults in simple rules only. As such, its fault detection results may not be *complete*. This implies that A-FSM can introduce *false negatives* (i.e., no fault is reported but there actually exists).

Take our stock tracking application as example. A non-determinism fault may occur at the "loading" state, where two rules **r1** and **r10** can be triggered at the same time. This happens when the forklift stays at the loading bay but the pallet has been taken away for a period of time, causing the right part of rule **r10** evaluated to "true". Now the forklift is also taken away, and then the application would face a nondeterministic situation. This is because two rules **r1** and **r10** are both triggered (rule **r1** and the left part of rule **r10** are both evaluated to "true"). However, A-FSM cannot detect it because **r10** is a complex rule and A-FSM cannot even model it.

2.4.2 Assumption 2: not considering dynamic information in analysis

A-FSM is a static analysis approach. It does not consider the impact of variable dependency, physical constraints, and rule actions on its fault detection results. Therefore, the results can be *imprecise*. First, variable dependency and physical constraints enforce specific relationships among contexts, such that context variables cannot take arbitrary values. This is basically ignored by A-FSM. Second, rule actions introduce dynamic context changes at runtime. A-FSM is fully unaware of it.

Still take our stock tracking application as example. A-FSM can detect an instability fault at the "energy_saving" state. This fault manifests itself when value assignments to concerned context variables are: pallet(t_L) = true, flArv(r_L) = true, flGone(r_L) = true, pallet(t_S) = false, flArv(r_S) = true, flGone(r_S) = false, and proceed(c_S) = false. The application would make a state transition from "energy_saving" to "loading" (since flArv(r_L) = true), and then to "transporting" immediately (since flGone(r_L) = true), without taking any new context changes. During these two continual transitions, collecting paper box RFID readings may or may not be skipped, depending on rule execution speed and context update rate. However, although this fault is undesired, it would never occur in reality. This is because the associated value assignment is invalid due to the variable dependency between flArv(r_L) and flGone(r_L). They can never both take "true" at the same time.

It could be argued that such variable dependency should be studied in advance for refining later fault detection results. We note that there are three concerns. First, variable dependency can be a lot, and there does not exist an automated way to derive a complete and precise set of variable dependency for any given CAAA. Second, modeling variable dependency may itself require expressive power beyond that supported by a propositional logic based specification language. Third, physical constraints and rule actions add extra complexity by uncontrollable dynamic context changes, which cannot be modeled in advance.

For example, consider a physical constraint like "a forklift cannot jump from the loading bay to the storage bay suddenly". It requires that "pallet(T_{load}) = true" should not be followed *immediately* by a context change that demands "pallet(T_{stor}) = true". Exploring a complete set of such constraints needs extensive knowledge on physical laws about the real world and does not seem to be an easy task. Besides, the application can modify contexts at the "cross reading" or "missing reading" state, and this would update values of concerened context variables. Static analysis fails to incorporate such dynamic information into its fault detection, and has to suffer from the *imprecision* problem.

2.5 Challenges in improvement

Summing up our preceding analyses, the inadequate expressive power of a propositional logic based specification language and the nature of a static analysis based fault detection approach bring along two major limitations. They cause the fault detection precision inevitably impaired. Therefore, we first consider a migration from a propositional logic based specification language to a first-order logic based one. However, the migration is not that straightforward.

First, consider a rule that is specified by a first-order logic based specification language like rule **r8**, "$\exists b_L \in B_{load}$ ($\forall b_S \in B_{stor}[t]$ (not (matched(b_L, b_S))))", in the stock tracking application. Each context variable (e.g., b_L and b_S) in the rule can take an arbitrary value from its associated context set (not only "true" or "false"), and it can refer to a set of different values rather than only one value. Not only what contexts can become the values for such a context variable is unclear (should be decided at runtime), but also how many of them can be in this set is no longer knowable in advance (should be decided by an interval t). As such, exhaustically exploring all possible value assignments to context variables used in a rule becomes impossible. Then, the idea of detecting all potential adaptation faults by exhaustive exploration of a rule's space is no longer feasible to first-order logic specified rules.

Second, since an exhaustive exploration for the space formed by a first-order logic specified rule cannot be done statically, one may migrate from static analysis to dynamic analysis, i.e., detecting faults at runtime. This can also avoid the concern that there is no way to automatically derive a complete and precise set of variable dependency and physical constraints for CAAAs. However, detecting faults at runtime requires high efficiency because otherwise an application's responsiveness to context changes would be impaired, denying its original purpose of being context-aware.

Regarding the above two challenges and their analyses, we propose our novel adaptation model (AM) approach to detect CAAA adaptation faults at runtime. Our ideas are as follows:

1. **Completeness:** Deploy our first-order logic based specification language to model CAAA adaptation rules. The language supports both simple and complex rules. It also allows application-specific functions to be specified by developers or users.

2. **Precision:** Detect adaptation faults at runtime to avoid false warnings. Our AM approach detects CAAA faults dynamically and therefore must report real faults.

3. **Efficiency and scalability:** Detect adaptation faults in an incremental way. Our AM approach would reuse previous rule evaluation results and incrementally handle new context changes for efficiency and scalability.

3 AM Approach

In this section, we elaborate on our AM approach. We first introduce its underlying model for specifying how a CAAA runs with its state transitions. Based on this model, we then propose an algorithm to detect a CAAA's adaptation faults dynamically. Finally, we discuss how we reduce the impact of runtime fault detection on a CAAA's responsiveness to its received context changes.

3.1 Adaptation model

Our AM approach contains a built-in model for specifying CAAAs. This model, named *adaptation model*, is a finite-state machine, which contains a set of *states* S and a set of *adaptation rules* R. Each state is a unique string that distinguishes it from other states. Rules are further defined as: $R \subseteq S \times C \times S \times A \times N$. Here, C is a set of rule conditions specified in our aforementioned first-order logic based specification language. A is a set of actions that can be executed by a CAAA, and they may modify the values of concerned context variables. Finally, N is a set of natural numbers presenting rule priorities.

Thus, a rule r is represebted as a tuple of five attributes (s, c, s', a, n). If a CAAA is currently resident at state s and rule r's condition c is evaluated to "true", we say that rule r is *triggered*. All rules (like r) starting from state s (as the first element in r's representation tuple) are considered *active* for state s. If multiple active rules are triggered, then only one of them can be selected for execution. The choice depends on their priorities (n). When rule r is executed, the CAAA would conduct action a and transits to a new state s'. For our stock tracking application, the first-part adaptation rules **r1-5** (for normal workflow) have a normal priority (say 5), the second-part and third-part adapation rules **r6-12** (for exception handling and energy-awareness) have a high priority (say 10).

Given a CAAA, we specify its adaptation model as: $AM = (S, R, s_0, S_f, s_c, V)$. Here, S and R are the set of states and set of adaptation rules, respectively, used by this application. A special state $s_0 \in S$ is this application's initial state. When the application starts, it is set to this initial state. Subset $S_f \subseteq S$ is this application's final states. When the application reaches any one of them, it stops running. All these belong to static information. On the other hand, s_c and V capture runtime information. Here, $s_c \in S$ is the application's current state, and V is the current value assignment for all context variables used by this application: $\wp\ (CTX\_VAR \times \wp\ (CTX\_VAL))$. 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

Now we explain the detection of non-determinism faults and instability faults. The detection is based on the adaptation model associated with a CAAA. As explained earlier, non-determinism fault violates the property that, for each state and each possible value assignment to context variables, there is at most one active rule that can be triggered. Instability fault violates the property that an application's current state' duration is independent of context update rate and rule execution speed. When an instability fault occurs, given context changes would produce a sequence of continul adaptations, such that which state the application would eventually stop at depends on how long concerned context variables would hold their values, which are unpredictable. This would cause the application to run in an unstable way (called *adaptation race*) or maybe worse in an infinite loop (called *adaptation cycle*)[15,16].

We give our fault detection algorithm in Fig. 3. It is composed of three parts. The first part is an overall framework, and the other two parts detect two adaptation

```
Algorithm (Part 1): detectFaults
Input: AM M, context change c
Output: faults F
M.evaluateRules(c)
F = {}
r = checkNondeterminism(M, F)
if r != null then
    state = M.getState()
    cycle = false
    R = checkRaceCycle(M, r, F, cycle)
    if size(R) > 1 then
        if cycle then
            F.add({"cycle", state, R})
        else
            F.add({"race", state, R})
        end if
    end if
end if
return F

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

Algorithm (Part 3): checkRaceCycle
Input: AM M, triggered rule r, faults F, boolean cycle
Output: race rules R, faults F, boolean cycle
R = {r}
while r != null && !cycle do
    changes[] = M.executeRule(r)
    M.evaluateRules(changes[])
    r = checkNondeterminism(M, F)
    if r != null then
        if r ∈ R then
            cycle = true
        end if
        R.add(r)
    end if
end while
return R
```

Figure 3. Fault detection algorithm

faults, respectively. When a new context change c is received, Part-1 Algorithm (`detectFaults`) is invoked. A CAAA's associated adaptation model M would evaluate all its active rules to see whether any rule is triggered due to this context change (Lines 1-3 of Part 1). Part-2 Algorithm (`checkNondeterminism`) is then invoked to detect 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, the control flow goes back to Part-1 Algorithm to continue to detect adaptation race or cycle faults (Lines 4-7 of Part 1); otherwise, the whole algorithm stops. The adaptation race or cycle detection forms a loop, in which Part-3 Algorithm (`checkRaceCycle`) executes the selected rule r and collects potential context changes (Line 3 of Part 3), then reevaluates active rules against these new changes and detects 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 rule can be triggered or the selected rules have already formed a cycle (Line 2 of Part 3). If the loop is not infinite, it indicates an adaptation race fault. It implies that no new context change is received, but the application keeps triggering itself and making continual adaptations. If the loop is infinite (cycle detected), it indicates an adaptation cycle fault. It implies that the same states and rule sequences would repeat themselves forever.

All these faults are harmful to CAAAs and can be detected by the fault detection algorithm. We note that since these faults are detected dynamically during the application runs, they are always real, i.e., no false warning. This is also confirmed by our later evaluation.

3.3 Incremental rule evaluation

Our fault detection algorithm works at runtime, and therefore its time complexity is a key issue. In the algorithm, when a context change is received, function `evaluateRules` would be invoked to check whether this change would cause any active rule to be triggered. The checking 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 this rule's action. These new changes would also need to be examined by function `evaluateRules` to see whether any new rule is triggered. As such, this function can be called multiple times even in one invocation of the fault detection algorithm.

We conjecture that this could be time-consuming. To study its complexity at runtime, we conducted initial experiments to investigate the percentage of time spent solely by this function against the total time. We found that this percentage could be up to 99%, which is very significant. This finding suggests that reducing the time complexity for the `evaluateRules` function is the key to improving the overall efficiency and scalability of our runtime fault detection algorithm.

3.3.1 Rule evaluation tree

We propose an incremental rule evaluation (IRE) technique to address this concern. IRE builds on our previous work on incremental consistency checking[20,23], and extends it to support efficient handling of three types of context change as well as enhanced reuse for both data structures and evaluation results.

IRE represents the evaluation for each adaptation rule by a tree structure (called *rule evaluation tree* or RET). The tree structure contains multiple layers, and each layer corresponds to a nested level of the associated rule. For example, consider rule **r6** "$\exists b_S \in B_{stor}$ $(\exists b_G \in B_{gate}[-t]$ (matched$(b_S, b_G)))$" in our stock tracking application. Suppose that at the time we evaluate this rule, B_{stor} contains its only value b_{d1}, and $B_{gate}[-t]$ contains two values $\{b_{g1}, b_{g2}\}$, which have been collected within the past t interval. Then the evaluation of rule **r6** can be represented by an RET, as illustrated in Fig. 4 (top-left diagram). This RET spans itself by listing all possible value assignments for concerned context variables. A post-order traversal (i.e., visiting each tree node and evaluating its truth value based on its associated value assignment) to the RET would return the truth value of the whole rule **r6**. Intermediate truth values (also called intermediate evaluation results) can be recorded in each node's data structure for later reuse.

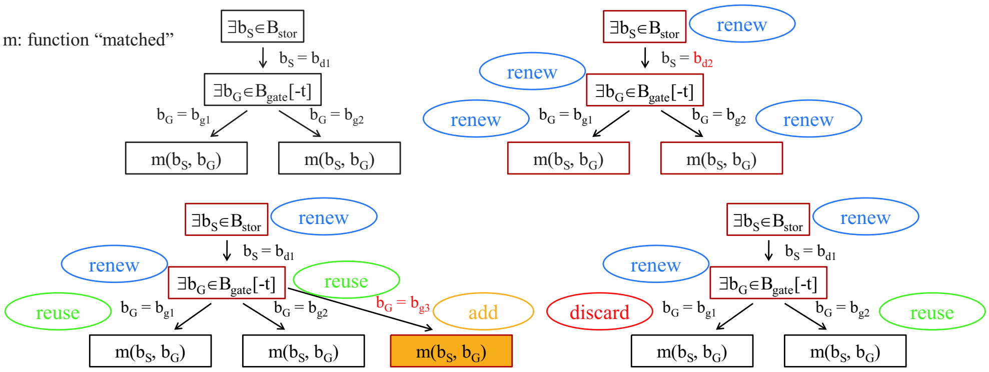

Figure 4. Rule r6s associated RET (top-left), tree change due to context update (top-right), tree change due to context addition (bottom-left), and tree change due to context deletion (bottom-right).

3.3.2 Types of context change

In order to incrementally evaluate adaptation rules, one needs to adjust these rules' associated evaluation trees incrementally with respect to received context changes. There are a total of three types of context change. They are *context update*, *context addition*, and *context deletion* changes:

1. **Context update change:** Modify the current value of a context variable. Such context variables can take only one value. One example is variable b_S in rule **r6**.

2. **Context addition change:** Add a new value for a context variable. Such context variables can take multiple values from a context set. One example is variable b_G in rule **r6**. This change actually affects this variable's associated context set.

3. **Context deletion change:** Remove a previous value for a context variable. Such context variables can also take multiple values from a context set. One example is variable b_G in rule **r6**. This change actually affects this variable's associated context set.

Context update change works for simple rules, which are specified by a propositional logic based specification language. Handling such context changes supports the case that the current value of a context variable is replaced by its latest value. Context addition and deletion changes work for complex rules, which are specified by a first-order logic based specification 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 timeliness requirements).

3.3.3 Handlig context changes incrementally

To incrementally handle context changes, one needs to reuse existing tree structures and previous evaluation results as many as possible. Our basic idea is outlined by the following four operations, which have different levels of reuse:

1. **Reuse.** Both existing tree structures and previous evaluation results are reused (as they are still useful).
2. **Renew.** Existing tree structures are reused, but previous evaluation results are renewed (as they need update).
3. **Discard.** Both existing tree structures and previous evaluation results are discarded (as they are no longer useful).
4. **Add.** Both tree structures and evaluation results are newly created (as they do not exist before).

We in the following explain how to incrementally handle context changes using these four operations. We explain our idea by illustrative examples.

Context update change. Suppose that a context update change modifies a context variable b_S's value from b_{d1} to b_{d2}, as illustrated in Fig. 4 (top-right diagram). We observe that all nodes in this RET are reusable but their evaluation results need update. Our IRE technique would first locate the node that contains the context variable b_S and then update its associated variable assignment from "$b_S = b_{d1}$" to "$b_S = b_{d2}$". Since all nodes below this node are affected by this update, their evaluation results need renewal. Finally, this node itself and all nodes above this node also renew their evaluation results in a bottom-up manner.

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 a context variable b_G, as illustrated in Fig. 4 (bottom-left diagram). Our IRE technique would first locate the node that contains the context variable b_G and then add one branch to it. The new branch corresponds to a new value assignment of $b_G = b_{g3}$ and a new node "matched(b_S, b_G)" (rightmost one) is created for this new assignment. IRE would then evaluate this node as it is completely new, but both the tree structures and previous evaluation results of the other two "matched(b_S, b_G)" nodes (two left ones) can be fully reused. Finally, for the two top nodes "$\exists b_G \in B_{gate}[-t]$" and "$\exists b_S \in B_{stor}$", their tree structures are reusable but IRE would renew their evaluation results as their child nodes 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 a context variable b_G, as illustrated in Fig. 4 (bottom-right diagram). Our IRE technique would first locate the node that contains the context variable b_G and then remove the branch corresponding to the value assignment $b_G = b_{g1}$ from it. All nodes on this branch are discarded, but the tree structures and previous evaluation results on the "$\exists b_G \in B_{gate}[-t]$" node's other branches are still reusable. Finally, IRE would renew the evaluation results for the two top nodes "$\exists b_G \in B_{gate}[-t]$" and "$\exists b_S \in B_{stor}$" because their child nodes have updated evaluation results, but their tree structures keep unchanged and thus still reusable.

3.3.4 Time complexity analysis

Our IRE technique is very efficient. To see it, we analyze its time complexity. Let the number of nodes in an RET be n and one node visit be the unit time (1). Without IRE, one has to create the whole RET, evaluate all nodes on this RET. No matter what type of context change it is, the RET creation and evaluation is always complete. Therefore, the time complexity is $\mathrm{O}(n)$. On the other hand, with IRE, the time complexity is greatly reduced due to significant reuse of existing tree structures and previous evaluation results. It is $\mathrm{O}(1) \sim \mathrm{O}(n)$ for handling a context update or addition change, and $\mathrm{O}(1)$ for handling a context deletion change.

We note that even if our IRE technique reaches its worst-case complexity $\mathrm{O}(n)$, its actual spent time would still be much less than the time required by a non-incremental technique (i.e., without IRE). This is because IRE handles affected nodes only, whereas a non-incremental technique has to recreate and reevaluate all nodes. We shall illustrate the significant performance difference between using and not using IRE, as well as the great difference in the number of nodes created for rule evaluation in our following evaluation.

4 Evaluation

In this section, we conduct both simulated and real-world experiments to evaluate our AM approach and compare it to existing A-FSM approach in detecting adaptation faults for CAAAs. We study two research questions:

1. **Effectiveness:** *Compared to A-FSM, how effective can our AM approach be in detecting adaptation faults for CAAAs?*
2. **Efficiency:** *How efficiently does our AM approach support runtime fault detection for CAAAs?*

We select two applications as our experimental subjects. One is the stock tracking application discussed throughout the paper. We use it with simulated configurations and parameters, which were adapted from its field test data in our pilot study. The other experimental subject is a self-controlling robot-car application. It is a real-world system build on real hardware platform and tested with realistic physical data.

4.1 Comparison metrics

To answer our two research questions, we need to first select comparison metrics in our experiments. We compare the effectiveness of our AM approach with that of the A-FSM approach as follows.

Given a CAAA that has been specified by our AM model, we first project it on simple rules. The resulting model is named AM_{simple} model. We then apply the A-FSM approach to this model to detect adaptation faults, since A-FSM can handle simple rules only. The set of thus detected faults is named $\text{Faults}_{simple,A-FSM}$. Next we apply our AM approach to the AM model directly to detect adaptation faults, since AM can handle both simple and complex rules. The set of thus detected faults is named $\text{Faults}_{all,AM}$. For comparison purposes, we further divide $\text{Faults}_{all,AM}$ into two disjointed subsets: $\text{Faults}_{simple,AM}$ and $\text{Faults}_{other,AM}$ (i.e., $\text{Faults}_{all,AM} = \text{Faults}_{simple,AM} \cup \text{Faults}_{other,AM}$). Here, $\text{Faults}_{simple,AM}$ contains those faults that are related to simple rules only, and $\text{Faults}_{other,AM}$ is defined as: $\text{Faults}_{all,AM} - \text{Faults}_{simple,AM}$.

In the experiments, we study the following three metrics:

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

2. **Potential false positives**, which are defined as: $\text{Faults}_{simple,A-FSM} - \text{Faults}_{simple,AM}$. This part gives the faults related to simple rules and they are potentially false positives. This is because these faults are detected by the A-FSM approach, 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**, which are defined as: $\text{Faults}_{simple,AM} - \text{Faults}_{simple,A-FSM} \cup \text{Faults}_{other,AM}$. This part gives the faults missed by the A-FSM approach, but reported by our AM approach. They are real faults because they have been observed in our runtime detection. They have been missed by A-FSM due to various reasons (e.g., some rule actions have modified contexts at runtime but this is not consided by A-FSM, or some faults relate to complex rules and cannot be handled by A-FSM).

We would measure and compare the above three metrics in evaluating the effectiveness for our AM approach and existing A-FSM approach. Regarding the efficiency comparison, we would measure the time spent in fault detection for both approaches.

4.2 Stock tracking application

To run the stock tracking application, we set up simulated warehouse scenarios based on configurations and parameters from our engineers according to practical environments in our field study. Each forklift undertakes multiple transportation tasks, and each transportation task costs 20-60 seconds. During transportation, a forklift carries 50 paper boxes, each of which is attached with an RFID tag for unique tracking. The cross reading rate and missing reading rate are both set to 5%. Besides, a forklift may be randomly delayed in its return trip back to the loading bay by other uses due to resource sharing. The probability of such events is set to 10%, and the delay thus incurred is random and can be up to 40 seconds.

The experiments with the stock tracking application were conducted on a PC with Intel® Pentium® 4 CPU @3.2GHz and 2GB RAM. The operating system is Windows XP Professional SP3. We implemented both the A-FSM and AM approaches in Java (Oracle/Sun JRE 7). They shared the same data structures for collecting and manipulating contexts. We integrated both approaches in Cabot context middleware[19,24] as plug-in services. The stock tracking application ran with contexts fed by Cabot.

4.2.1 Effectiveness

We now compare the existing A-FSM and our AM approaches using discussed metrics. The AM model for the stock tracking application contains a total of 12 adaptation rules (Fig. 2), which include both 6 simple rules and 6 complex rules. Then the corresponding AM_{simple} model applicable to A-FSM contains all 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 belonging to instability faults) on the AM_{simple} model. On the other hand, AM detected faults on the AM model directly. It detected one non-determinism fault and two adaptation race faults. Seemingly, A-FSM has detected much more faults, but we are more interested in their detection qualities. We further investigated all these detected faults, and obtained the following findings according to our comparison metrics.

Real faults. There is only one adaptation race fault detected by both approaches. This fault shows that one adaptation race occurred at the "energy_saving" state when the "start_loading" rule was executed and immediately followed by the triggering of another "start_transporting" rule. This happened when the application resided at the "energy_saving" state, at which its controlled forklift and pallet were taken away by other uses, and then the pallet came back earlier than the forklift did. As a result, the application's state transited from "energy_saving" to "loading" and then immediately to "transporting", as both adaptation rules' conditions were satisified. However, the activity of reading RFID tags of all paper boxes being transported could be skipped at the "loading" state.

Potential false positives. A-FSM also detected other 163 adaptation race faults and 12 adaptation cycle faults. Unfortunately, they are all potential false positives. After our further analysis, all of them were confirmed as false positives due to violation with variable dependency or physical constraints. First, 132 adaptation race and 12 cycle faults violate variable dependency among four context variables flArv(r_L), flGone(r_L), flArv(r_S) and flGone(r_S). For example, flArv(r_L) and flGone(r_L) cannot take "true" at the same time. This is because the pressure sensor installed at the loading bay must report "down" or "up", implying that the forklift arrives at or leaves the loading bay, respectively. Two cases cannot be together. Second, all the remaining 31 adaptation race faults violate four physical constraints. They are: "forklift and pallet cannot be separated at two places at the same time", "forklift and pallet 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". As such, although A-FSM detected much more faults, most of them are unreal (175/176 = 99.4%).

False negatives. There are two faults (one non-determinism fault and one adaptation race fault) detected by AM but missed by A-FSM. This is interesting as although A-FSM detected much more faults, it still missed real faults. This non-determinism fault occurred at the "loading" state, at which both rules "save_energy" and "start_transporting" were triggered. This happened when the forklift stayed at the loading bay and its pallet had been taken away for some time, and then the forklift was also taken away and the application would now face a non-deterministic situation. This is because both rules "save_energy" and "start_transporting" had their conditions satisified and thus triggered. The adaptation race fault occurred at the "unloading_1" state, at which the application started to transit to the "unloading_2" state. At the new state, if some missing readings were present, the application would immediately transit to the "missing_reading" state (the adaptation rule's condition already satisified). Then the preparation for missing reading checking could be skipped at the "unloading_2" state. These two faults are real, but missed by A-FSM, causing false negatives to its detection results.

We summarize these comparison results in Table 1, from which we observe that: (1) AM is precise; (2) A-FSM contains both false positives and false negatives; (3) AM reports more effectively developer-cared faults than A-FSM (three real faults vs. one real fault; no false fault vs. 177 false faults).

Table 1 Effectiveness comparison between two fault detection approaches

Approach	Non-determinism faults		Adaptation race faults		Adaptation cycle faults	
	Detected	**Analysis**	**Detected**	**Analysis**	**Detected**	**Analysis**
A-FSM	0	1 missing	164	1 real, 163 unreal, 1 missing	12	12 unreal
AM	1	1 real	2	2 real	0	0

4.2.2 Efficiency

Our AM approach detects adaptation faults in CAAAs at runtime, and therefore its efficiency is a key issue. We deploy our IRE technique for this purpose. To see how well it works, we study AM's efficiency. We name the AM version with our IRE technique IC (i.e., incremental checking) and the one without our IRE technique NI (i.e., non-incremental checking).

Handling time. We first measure and compare the total time used in handling contexts (fault detection included). We simulated forklift transportation 50 times. We observe that NI took 209,261 ms to handle all contexts, whereas IC took only 43,413 ms (20.7%). This shows that IC works much more efficiently.

Response time. The above handling time only compares the total time cost. We then measure and compare the application's response time, as illustrated in Fig. 5. *Response time* is the interval between an application receives a context and it conducts required actions. 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. This shows that IC works much more efficiently also for its every required response.

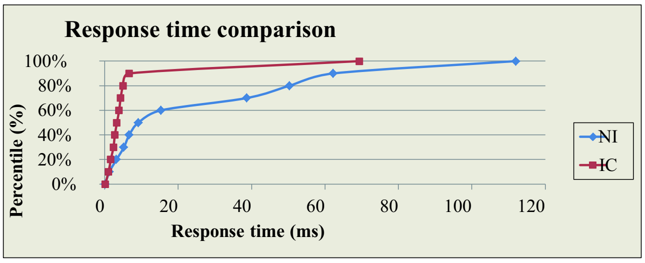

Figure 5. Response time comparison between two rule evaluation techniques

Created nodes. We owe AM's efficiency to its IRE technique (i.e., the IC version). To further understand how IRE contributes to AM's efficiency, we measure and compare how many nodes are created in NI and IC, respectively. As illustrated in Table 2, NI created a total of 43,993,249 nodes for tree structures in our experiments. On the other hand, IC created only 903,829 nodes (2.0%), which are much less. IC realized this by reusing 42,524,948 nodes (96.7%) and renewing 564,472 nodes (1.3%). That is, the reuse of existing tree structures ("renew" case) and previous evaluation results ("reuse" case) helped AM work much more efficiently at runtime.

Table 2 Created nodes comparison between two rule evaluation techniques

Technique	Reused(♯)	Renewed(♯)	Added(♯)
NI	0	0	43,993,249
IC	42,524,948	564,472	903,829
Percentage	96.7%	1.3%	2.0%

Scalability. Finally, we compare NI's and IC's scalability and study its impact on rule triggering when an application's complexity increases. We set up a scale factor F (from 1 to 10) to control the number of transportations running in parallel. With the increase of the scale factor, the number of contexts to be handled in the unit time would increase and the interval between contexts would decrease accordingly. This would thus increase the rule evaluation workload for both NI and IC.

We observe from Fig. 6 that NI's occupied time increases quickly with the scalability factor F's growth. Here, *occupied time* measures the percentage of context handling time against the total time allowed. IC's occupied time grows clearly much slower. Besides, we observe that starting from $F = 5$, NI's occupied time becomes so close to 100%. At the same time, its rule triggering number decreases quickly (down to below 60% as compared to IC's value at $F = 10$). The continuous decreasing of this number indicates that the application's adaptation rules cannot be triggered as expected. This would seriously affect the application's responsiveness (i.e., no response when context changes occur). On the other hand, IC's rule triggering number keeps quite stable (almost no change). This shows that our IRE technique really contributes to AM's efficiency as well as its scalability for

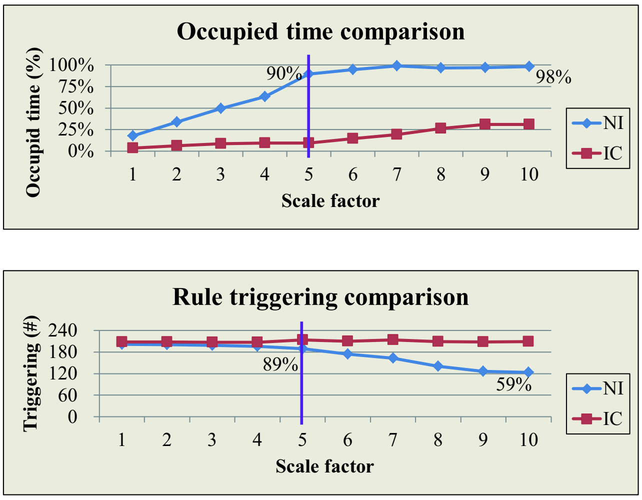

Figure 6. Scalability comparison between two rule evaluation techniques

complex applications.

4.3 *Self-Adaptive robot-car application*

Our second experimental subject is a self-adaptive robot-car application. It is a realistic system with real hardware, as shown in Fig. 7. The car is developed on a 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). These sensors can detect obstacles at each orientation. The car's two rear wheels are programmable for different walking speeds, and its two front wheels are equipped with speed sensors, which can measure how far the car has walked.

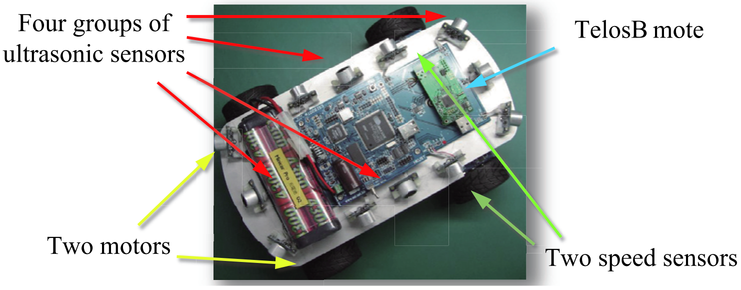

Figure 7. Robot-car system

The self-adaptive robot-car application runs on the car. It collects contexts (ultrasonic sensor and speed sensor data) periodically (every 350 ms). It then analyzes these contexts, evaluates adaptation rules, and decides how to guide the car to walk. The application aims to guide the car to explore an unknown area. It supports two context-aware features: (1) automatic speed adjustment (high-speed mode HS and low-speed mode LS); (2) automatic obstacle avoidance (normal walking mode W and obstacle avoidance mode A). The two features can combine together with different modes, as illustrated by the state transition diagram in Fig. 8.

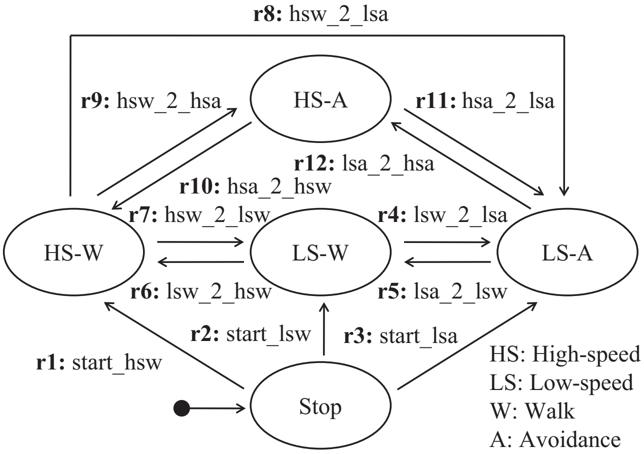

Figure 8. Self-adaptive robot-car application (its state transition diagram with 5 states and 12 rules)

The application starts with its initial "Stop" state, from which it can transit to one of three different states, depending on its contexts then. For example, it transits to the "HS-W" state if no obstacle is detected nearby, meaning that the car can walk with a high speed. Otherwise, it may transit to the "LS-A" or "LS-W" state with a low speed, depending on whether any detected nearby obstacle is already very close or not. The application's state would keep being adjusted when the car's surrounding obstacle conditions change. It works for the goal of exploring the area efficiently and at the same time keeping safety (i.e., without bumping into any obstacle).

The application contains a toal of 5 states and 12 adaptation rules, as shown in Fig. 8. All these rules are complex ones. They check all reported ultrasonic sensor data within a recent interval to decide runtime obstacle conditions for each orientation. For example, the "lsw_2_hsw" rule would transit the application from the "LS-W" state to "HS-W" state (low-speed walking to high-speed walking), when all ultrasonic sensors report "clear" conditions recently: $\forall o_A \in OC_{all}[-t]$ (clear(o_A)). Here, "clear" means no obstacle detected at any orientation. Another rule "lsa_2_hsa" transits the application from the "LS-A" state to "HS-A" state (low-speed avoidance to high-speed avoidance), if no ultrasonic sensor reports a "dangerous" condition and the obstacle avoidance rountine is not so frequently called recently: (not ($\exists o_A \in OC_{all}[-t]$ (danger(o_A)))) and (not ($\exists h \in HS[-t]$ (frequent(h)))). Here, "dangerous" means very close obstacle detected at at least one orientation. Besides, if the car gets stuck (judged from speed sensor data), the application would clear all contexts, reset its state to "Stop", and then restart itself.

For this application, the A-FSM approach does not apply due to the presence of complex rules (all rules are complex) and runtime context changes (A-FSM assumes no runtime context change). Therefore, we only evaluate the effectiveness and efficiency of our AM approach in detecting adaptation faults for this application.

We ran the car in a 5.0m × 3.5m typical office environment, which contains several desks, chairs, and cabinets, as illustrated in Fig. 9. Its left part is considered as high-

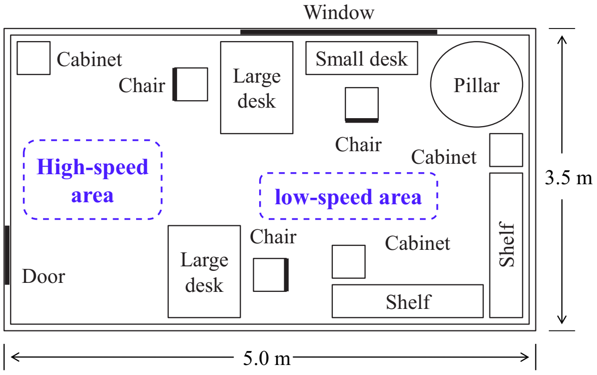

Figure 9. Office environment for field test

speed area due to its sparse layout, while its right part is considered as low-speed area as it is much denser with furniture. We ran the car ten times in this environment, each taking about 2.5 minutes due to its battery limitation. We recorded all experimental data and analyzed them.

4.3.1 Effectiveness

Our AM approach detected a total of 6 distinct non-determinism (named D1-6) and 8 distinct adaptation race faults (named R1-8) with various occurrences. Table 3 lists all these fault occurrences for each car run as well as their sums for each run and for each fault type. The sums of detected fault occurrences for each run have a range between 65 and 93 (with an average of 75), which are roughly close to each other. However, the sums of fault occurrences for each fault type have a significantly different distribution. For non-determinism faults, the distribution range is 3.1-36.3%, while for adaptation race/cycle faults, it is 0.3-44.7%. A graphical representation of these fault distributions is given in Fig. 10.

Table 3 Fault occurrences in the self-adaptive robot-car application

Run	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**Total**
1	4	9	2	0	0	0	1	22	2	5	2	0	18	0	**65**
2	4	13	6	1	2	0	4	16	2	0	9	1	23	2	**83**
3	15	11	1	2	0	0	8	15	5	10	0	0	13	0	**80**
4	2	0	3	0	0	6	2	28	7	0	5	0	39	1	**93**
5	6	10	3	2	1	0	5	18	0	3	4	0	27	0	**79**
6	1	5	1	1	2	0	0	21	0	0	6	1	28	1	**68**
7	1	0	8	0	0	1	1	18	1	0	7	0	30	1	**68**
8	1	5	5	0	0	0	1	23	0	0	7	0	31	0	**73**
9	3	3	1	2	0	0	3	20	0	2	3	0	31	0	**68**
10	7	2	3	1	0	4	5	19	3	0	4	0	23	0	**71**
All	**44**	**58**	**33**	**9**	**5**	**11**	**31**	**200**	**20**	**20**	**47**	**2**	**263**	**5**	**748**

This suggests that some faults are really hard to detect. The hardest non-determinism fault D5 (only 5 occurrences in 10 runs) happened at the “HS-W” state, where three adaptation rules “hsw_2_lsw”, “hsw_2_lsa”, and “hsw_2_hsa” were all triggered at the same time. The hardest adaptation race fault R6 (only 2

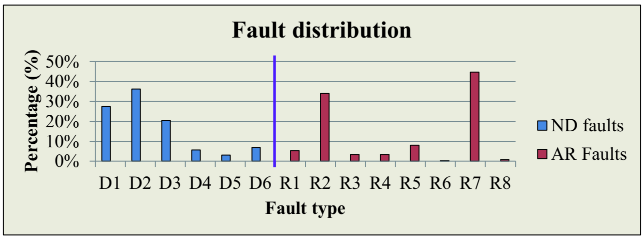

Figure 10. Fault distribution in the self-adaptive robot-car application

occurrences in 10 runs) happened also at the "HS-W" state, where two rules "hsw_2_hsa" and "hsa_2_lsa" were triggered in sequence.

After our further analysis, we found that both faults happened only when the car's two front ultrasonic sensors reported inconsistent obstacle conditions. For example, one sensor reported that "the car is approaching some obstacle but there is still some distance", and the other sensor reported that "the obstacle is already very close and the situation is dangerous"). We understand that we may not be able to easily control real-world contexts, 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[15,16]) may not be so useful in exposing such faults. This is because we measured that all states and rules had been already covered 100% for this application. One of our ongoing projects is to use improved static analysis with refined and ranked fault reports to more effectively detect such hard faults[9]. We are going this line. Still, we note that our AM approach can already detect so many real faults, which the existing A-FSM approach is unable to detect. This further confirmed AM's usefulness in detecting adaptation faults for practical CAAAs.

4.3.2 Efficiency

Finally, we evaluate our AM approach's efficiency in detecting adaptation faults for the self-adaptive robot-car application. We list the context handling time and occupied time for all ten car runs in Table 4.

Table 4 Handling time and occupied time in the self-adaptive robot-car application

Run	**Handling time (ms)**	**Occupied time (%)**
1	3,963	2.8%
2	3,741	2.6%
3	3,694	2.6%
4	5,175	3.7%
5	3,465	2.4%
6	4,254	3.0%
7	4,569	3.2%
8	4,190	2.9%
9	5,117	3.6%
10	5,086	3.6%
Average	**4,325**	**3.0%**

We observe that our AM approach's overhead is very small. For this application, its time for handling contexts (fault detection included) took 4,325 ms on average. This is with respect to a total car running time of about 2.5 mintues. Therefore, the average occupied time (i.e., handling time against total time allowed) is no more than 3.0%. When we further investigated the response time (i.e., interval between a car receives a context change and it takes a response action), it is about 1.0ms on average (80% less than 1.1ms and 90% less than 2.3ms). It shows that our AM's runtime fault detection only incurred negligible overhead in addition to normal application execution. This is desirable to practical context-ware applications as they usually need to be sensitive to context changes.

5 Related Work

Context-aware computing is attractive and receiving increasing attention from software developers and researchers. When applications become context-aware, they can actively perceive environmental changes and make seamless adaptation for delivering smart services to users. To support such applications, various middleware infrastructures[1,8,13,19,24,25] and application frameworks[3,5] have been proposed to assist application development and deployment. At the same time, many interesting context-aware applications have been developed and deployed for practical uses, such as highway collision avoidance system[8], Roaming Jigsaw[13], ConChat[14], Call Forwarding[18], and so on.

To ease application development, software developers usually assume that: (1) contexts can be *correctly* received from physical environments, and (2) adaptation rules can be *correctly* applied once their triggering conditions are satisfied at runtime. However, recent studies have disclosed that contexts from physical environments are often noisy, inaccurate, and easily obsolete. This fact often causes context-aware applications to run in an unexpected way, if they have not considered adequately how to alleviate the impact of such unreliable contexts.

Regarding this, Deshpande et al.[2] proposed to use threshold smoothing to filter out unreasonable contexts that have clearly exceeded allowable ranges. Our previous work[20,23] presented a constraint checking approach to identify correlated contexts whose inconsistency is difficult to judge by threshold solely. Regarding detected context problems or inconsistencies, Jeffery et al.[7] proposed a probabilistic approach to fix missing RFID contexts. It is effective but only applies to RFID contexts. Our previous work[21] complemented it in a heuristic way by enforcing general contexts to satisfy consistency constraints. Our previous work[22,27] also tried to resolve context inconsistencies in a way that minimizes potential side effect on an application's behavior. This is understandable as when contexts change by repairing actions, an application's behavior would also change accordingly. We have to validate such follow-up changes. Besides, applications may have conflicting resolution policies between each other, and repairing contexts for one application may be harmful to another application. Our previous work[28] also tried to isolate such across-application side effect by maintaining individual context views for each of these applications. This makes these applications run as if they are stand-alone.

On the other hand, while researchers are making great efforts in improving the quality of contexts collected from physical environments, applications themselves may still suffer from unanticipated faults or errors at runtime, when non-determinism and adaptation races or cycles arise. Insuk et al.[6] proposed to tolerate non-determinism as long as such simultaneously triggered adaptations do not conflict with each other in their semantics. This requires a precise modeling of such semantics, which costs manual efforts. Sama et al.[15,16] set up a stricter criterion by disallowing any occurrence of non-determinism, adaptation race, or adaptation cycle (i.e., the A-FSM approach). This is achieved by statically exploring all potential search space in an application. This can be done by a great deal of simplification in application modeling, and it also impairs the approach's effectiveness. For example, lacking dynamic information makes this approach probably report numerous false positives, which can easily overwhelm developers. This concern has been also echoed by Capra et al.[1], who believed that removing adaptation conflicts should be conducted dynamically. This is exactly what we have made efforts for in this paper.

Finally, researchers from software testing and debugging communities focused on detecting faults for code-based context-aware applications. They proposed new testing coverage criteria for covering those data flows especially affected by context changes[10,11], as well as new test case generation techniques for manipulating contexts into context-aware program points in an application[17]. We in this paper mainly focus on model-based context-aware applications, and extend over its earlier symposium version[26]. As compared to the existing A-FSM approach, we propose a new AM approach that owns increased expressive power for application modeling and increased precision for eliminating false warnings in fault detection. We also address the detection algorithm's runtime efficiency to make it useful for practical applications. This complements our previous work on disclosing the correlations between errors and failures in CAAAs. This also complements our ongoing effort of detecting adaptation faults statically with reduced false warnings and ranked fault reports[9]. This is done by deriving an application model that contains deterministic constraints to remove false warnings, and an environment model that contains likely constraints to rank remaining fault reports.

6 Conclusion

In this paper, we reviewed existing approaches for detecting adaptation faults in context-aware adaptive applications, and identified two aspects for improvement. We proposed our novel AM approach, and explained how it can be used to model complex rules in these applications and improve fault detection precision. Our AM approach works dynamically, and it supports such runtime fault detection via an efficient rule evaluation technique.

We evaluated our AM approach and compared it to a representative A-FSM approach using two context-aware applications. Both applications realize functionali- ties according to their perceived environmental changes in terms of states and adaptation rules. We selected these two applications in our evaluation for two reasons: (1) The stock tracking application was derived from a practical RFID application scenario with real parameters from on-site engineers; (2) The self-adaptive robot-car application runs on a real hardware platform, which collects realistic and noisy contexts from physical environments. By these two applications, we try to make our experiments more realistic, and measure AM's effectiveness and efficiency and compare it to A-FSM in a practical setting.

We note that detected non-determinism faults can be addressed by setting up a rule priority mechanism or refining the existing one. However, addressing detected adaptation race and cycle faults needs more effort, maybe requiring manual inspection. If the quality of fault reports is so low such that most reported faults are false warnings, then a great deal of inspection effort would be wasted. Our AM approach exactly fits here by helping report real faults only. In addition, due to its dynamic analysis nature, AM also reports the occurrence for each distinct fault type. This provides additional information on which faults have occurred more frequently and thus are more urgent for addressing. We note that this information is collected at runtime and thus reflects real fault conditions. It is helpful for debugging practice.

We are now studying ways of detecting hard adaptation faults that occur rarely at runtime. We are also interested in adapting our approach to code-based context-aware applications. We are going this line.

References

[1] Capra L, Emmerich W, Mascolo C. CARISMA: context-aware reflective middleware system for mobile applications. IEEE Trans. on Software Engineering, Oct 2003, 29(10): 929–945.

[2] Deshpande A, Guestrin C, Madden S. Using probabilistic models for data management in acquisitional environments. Proc. of the 2nd Biennial Conference on Innovative Data Systems Research. Asilomar, CA, USA. Jan 2005. 317–328.

[3] Dey AK, Abowd GD, Salber D. A context-based infrastructure for smart environments. Proc. of the 1st International Workshop on Managing Interactions in Smart Environments. Dublin, Ireland. Dec 1999. 114–128.

[4] Garfinkel S, Rosenberg B. RFID: Applications, Security, and Privacy. Addison-Wesley. Jul 2005.

[5] Henricksen K, Indulska J. A software engineering framework for context-aware pervasive computing. Proc. of the 2nd IEEE Conference on Pervasive Computing and Communications. Orlando, Florida, USA. Mar 2004. 77–86.

[6] Insuk P, Lee D, Hyun SJ. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. Proc. of the 29th Annual International Computer Software and Applications Conference. Edinburgh, UK. Jul 2005. 359–364.

[7] Jeffery SR, Garofalakis M, Frankin MJ. Adaptive cleaning for RFID data streams. Proc. of the 32nd International Conference on Very Large Data Bases. Seoul, Korea. Sep 2006. 163–174.

[8] Julien C, Roman GC. EgoSpaces: facilitating rapid development of context-aware mobile applications. IEEE Trans. on Software Engineering, May 2006, 32(5): 281–298.

[9] Liu Y, Xu C, Cheung SC. AFChecker: effective model checking for context-aware adaptive applications. The Journal of Systems and Software, Mar 2013, 86(3): 854–867.

[10] Lu H, Chan WK, Tse TH. Testing context-aware middleware-centric programs: a data flow approach and a RFID-based experimentation. Proc. of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Portland, Oregon, USA. Nov 2006. 242–252.

[11] Lu H, Chan WK, Tse TH. Testing pervasive software in the presence of context inconsistency resolution services. Proc. of the 30th International Conference on Software Engineering. Leipzig, Germany. May 2008. 61–70.

[12] Lv J, Ma X, Huang Y, Cao C, Xu F. Internetware: a shift of software paradigm. Proc. of the 1st Asia-Pacific Symposium on Internetware. 2009.

[13] Murphy AL, Picco GP, Roman GC. LIME: a coordination model and middleware supporting mobility of hosts and agents. ACM Trans. on Software Engineering and Methodology, Jul 2006, 15(3): 279–328.

[14] Ranganathan A, Campbell RH, Ravi A, Mahajan A. ConChat: a context-aware chat program.

IEEE Pervasive Computing, Jul-Sep 2002, 1(3): 51–57.

[15] Sama M, Elbaum S, Raimondi F, Rosenblum DS, Wang Z. Context-aware adaptive applications: fault patterns and their automated identification. IEEE Trans. on Software Engineering, Sep/Oct 2010, 36(5): 644–661.

[16] Sama M, Rosenblum DS, Wang Z, Elbaum S. Model-based fault detection in context-aware adaptive applications. Proc. of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. Atlanta, GA, USA. Nov 2008. 261–271.

[17] Wang Z, Elbaum S, Rosenblum DS. Automated generation of context-aware tests. Proc. of the 29th International Conference on Software Engineering. Minneapolis, MN, USA. May 2007. 406–415.

[18] Want R, Hopper A, Falcao V, Gibbons J. The active badge location system. ACM Trans. on Information Systems, Jan 1992, 10(1): 91–102.

[19] Xu C, Cheung SC. Inconsistency detection and resolution for context-aware middleware support. Proc. of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Lisbon, Portugal. Sep 2005. 336–345.

[20] Xu C, Cheung SC, Chan WK. Incremental consistency checking for pervasive context. Proc. of the 28th International Conference on Software Engineering. Shanghai, China. May 2006. 292–301.

[21] Xu C, Cheung SC, Chan WK, Ye C. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. Proc. of the 28th International Conference on Distributed Computing Systems. Beijing, China. Jun 2008. 713–721.

[22] Xu C, Cheung SC, Chan WK, Ye C. On Impact-oriented automatic resolution of pervasive context inconsistency. Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Dubrovnik, Croatia. Sep 2007. 569–572.

[23] Xu C, Cheung SC, Chan WK, Ye C. Partial constraint checking for context consistency in pervasive computing. ACM Trans. on Software Engineering and Methodology, Jan 2010, 9(3): Article 9, 1–61.

[24] Xu C, Cheung SC, Lo C, Leung KC, Wei J. Cabot: on the ontology for the middleware support of context-aware pervasive applications. Proc. of the NPC Workshop on Building Intelligent Sensor Networks. Wuhan, China. Oct 2004. 568–575.

[25] Xu C, Cheung SC, Ma X, Cao C, Lv J. ADAM: identifying defects in context-aware adaptation. The Journal of Systems and Software, Dec 2012, 85(12): 2812–2828.

[26] Xu C, Cheung SC, Ma X, Cao C, Lv J. Dynamic fault detection in context-aware adaptation. Proc. of the 4th Asia-Pacific Symposium on Internetware. Qingdao, Shandong, China. Oct 2012, Article 23, 1–10.

[27] Xu C, Ma X, Cao C, Lv J. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. Proc. of the 8th ICST International Conference on Mobile and Ubiquitous Systems, LNICST 104. Copenhagen, Denmark. Dec 2011. 285–297.

[28] Yang H, Xu C, Ma X, Zhang L, Cao C, Lv J. ConsView: towards application-specific consistent context views. Proc. of the 36th Annual International Computer Software and Applications Conference. Izmir, Turkey. Jul 2012. 632–637.