The Journal of Systems and Software 85 (2012) 2812–2828

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

ADAM: Identifying defects in context-aware adaptation

Chang Xu [a,b,*], S.C. Cheung [c], Xiaoxing Ma [a,b], Chun Cao [a,b], Jian Lu [a,b]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
[b] Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China
[c] Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Kowloon, Hong Kong, China

ARTICLE INFO

Article history:
Received 30 April 2011
Received in revised form 11 April 2012
Accepted 27 April 2012
Available online 11 May 2012

Keywords:
Context-aware adaptation
Defect
Error detection
Failure

ABSTRACT

Context-aware applications, as a typical type of self-adaptive software systems, are receiving increasing attention. These applications continually adapt to environmental changes in an autonomic way. However, their adaptation may contain defects when the complexity of modeling all environmental changes is beyond a developer's ability. Such defects can cause failure to the adaptation and result in application crash or freezing. Relating these failures back to responsible defects is challenging. In this paper we propose a novel approach, called ADAM, to assist identifying defects in the context-aware adaptation. ADAM monitors runtime errors for an application, logs relevant error information, and relates them to responsible defects in this application. To make our ADAM approach feasible, we investigate the error types that are commonly exhibited by various failures reported in context-aware applications. ADAM detects these errors in order to identify responsible defects in context-aware applications. To detect these errors, ADAM formally models the adaptation semantics for context-aware applications, and integrates into them a set of assertion checkers with respect to these error types. We experimentally evaluated ADAM through three context-aware applications. The experiments reported promising results that ADAM can effectively detect errors, identify their responsible defects in applications, and give useful hints on how these defects can be fixed.

1. Introduction

In recent years, context-aware applications are receiving increasing attention. As a typical type of self-adaptive software systems, these applications inherit the nice ability of adapting their behavior to environmental changes in an autonomic way. However, they also suffer from the challenges of developing reliable applications to cope with various environments. Nowadays many successful context-aware applications are developed by following an autonomic control loop of "collect-analyze-decide-act" (Dobson et al., 2006) to perceive their environmental conditions and make smart adaptation. Still, developing reliable context-aware applications is non-trivial. It must address a number of research challenges arising from modeling dimensions, requirements, engineering, and assurances (Cheng et al., 2009). Various studies have been made to address these challenges and led to fruitful results, such as research methodologies and techniques on self-adaptive application modeling (Cheng et al., 2009; Andersson et al., 2009) and architecture-based self-management (Garlan et al., 2003; Kramer and Magee, 2007).

Context-aware applications are now becoming increasingly popular due to the continual advances in sensing technologies. These applications use *context* information about their environments and make self-adaptation. Typical examples include those emerging Android and iOS applications. They run on mobile phones with various built-in sensors, such as GPS, Bluetooth transceiver, and accelerometer. They customize the behavior based on the contexts collected from these sensors. Some popular applications like Locale (Locale, 2011) and Tasker (Tasker, 2011) can also adapt their working modes or profiles at runtime in response to their contexts. For example, Locale can automatically turn a phone into the vibration mode when its user is in a meeting.

While promising, *context-awareness* couples an application's behavior with its numerous *situations* (e.g., in a meeting or at home) introduced by its changeable environments. When the complexity of modeling these situations grows, a developer may fail to correctly specify how an application should adapt to these situations. The quality of such applications is thus difficult to guarantee (Lu et al., 2006, 2008; Mamei and Zamonelli, 2009).

We studied three commercial Android applications (Locale (Locale, 2011), Tasker (Tasker, 2011), and Setting Profiles (Setting Profiles, 2011)). Their bug-fixing histories show that 66.0%, 58.6%, and 77.5%, respectively, of their reported failures relate to defects in

* Corresponding author at: Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China. Tel.: +86 25 89680919; fax: +86 25 83593283.
E-mail addresses: changxu@nju.edu.cn (C. Xu), scc@cse.ust.hk (S.C. Cheung), xxm@nju.edu.cn (X. Ma), caochun@nju.edu.cn (C. Cao), lj@nju.edu.cn (J. Lu).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.04.078

their context-related application logics. These cover context handling, situation evaluation, and adaptation (a detailed study is given later in Section 2). In other words, context-related application logics have contributed most of these reported failures. This percentage is quite high.

For clarity, in this paper we interpret *defect* in an application as a place where the implemented application logic does not follow its developer's intention or requirement. For example, a defect can be a mistakenly specified condition "a ∨ b", which was originally supposed to be "a ∧ b". At runtime, a defect may be executed by the control flow of an application, making the application enter an *error*. The error may be further propagated to the application's output as an observed *failure*. In terms of Android applications, an observed failure is typically an application *crash* (abnormal exit and stop of running), or *freezing* (no response to any button or touch-screen input).

Tracking an observed failure back to its original defect is generally difficult. It may be more difficult for context-aware applications. First, setting up a useful test oracle for context-aware applications is challenging, because an observed failure is usually the consequence of a series of adaptations. There is no general way to know which particular adaptation in this series has gone wrong, leading to the final observed failure. A typical test oracle used in practice is to observe whether an application has crashed or become freezing. This is, however, not that useful, as it may be the result of a long series of adaptations. Second, when a failure occurs, collecting all relevant contexts is not easy. This is because the contexts are obtained from sensors outside an application, whose working states are difficult to record precisely and completely. Third, even if all relevant contexts could be collected, it is still not easy to repeat an observed failure. A slight variation in sensors or environments can easily cause an application to behave differently. Essentially, people cannot fully control a physical environment for debugging a context-aware application.

In this paper, we propose a novel approach to addressing these challenges. It allows us to detect an application's errors instead of failures at runtime, and track these errors back to their original defects. This approach can alleviate the test oracle problem from which the failure detection suffers, since errors have more criteria to judge than just using application crash or freezing. It can also be more useful since errors contain more information about defects than failures do. This is because errors are usually closer to defects physically, considering that failures have to manifest themselves as application crash or freezing after a series of adaptations. However, some new questions also arise: *Do failures in context-aware applications exhibit common error types? If yes, how to effectively detect them? Further, how to effectively identify defects based on these detected errors?* We shall study these research questions in this paper.

We restrict our scope in this paper to *model-based context-aware applications* (or MCAs for short). These applications represent one popular computing paradigm toward the construction of reliable context-aware software. This paradigm separates an application's logic into two concerns: (1) acquiring its contexts from physical environments and (2) conducting adaptation based on the situations detected from these contexts. Such a paradigm has been widely adopted by recent context-aware middleware infrastructures or frameworks (Capra et al., 2003; Kulkarni and Tripathi, 2010; Ranganathan and Campbell, 2003; Sama et al., 2010a; Xu and Cheung, 2005). Still, MCA defects have received inadequate attention. One line of existing work focuses on isolating noisy contexts from impacting an application's behavior. This is achieved by checking contexts against consistency constraints (Huang et al., 2009; Rao et al., 2006; Xu and Cheung, 2005; Xu et al., 2010) or probabilistic models (Bu et al., 2006; Khoussainova et al., 2006; Xu et al., 2008). These pieces of work do not directly identify defects in MCAs. Another line of existing work investigates defects in an MCA and proposes techniques to identify them statically or dynamically. These pieces of work mostly focus on *non-determinism errors* that may lead to multiple but conflicting adaptations (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Ranganathan and Campbell, 2003). Sama et al. (2010a,b) additionally investigated several other error types, but the aforementioned three research questions remain unanswered.

We in this paper make the following contributions:

(1) **An archival study of three commercial context-aware applications**. We examined a total of 285 real-world failures reported from 80 releases of three commercial context-aware applications. Based on these failures, we performed an error classification and studied the relationships between these failures and their associated errors.
(2) **An ADAM model for specifying an MCA's adaptation semantics (ADAM stands for Adaptation Modeling)**. This model enables the precise specification of adaptation semantics for context-aware applications, and allows for the effective error detection in the adaptation semantics at runtime.
(3) **An evaluation with three context-aware applications (MCAs)**. We evaluated ADAM's ability of detecting errors in context-aware adaptation and identifying defects from these errors through three MCAs.

The rest of this paper is organized as follows. Section 2 presents our archival study of three commercial context-aware Android applications and their reported failures. Based on them, we perform a classification of common error types exhibited by these failures. Section 3 introduces our ADAM model for specifying an MCA's adaptation semantics, and explains how the specification is executed at runtime to make context-aware adaptation. Within the ADAM model, Section 4 formally defines our earlier classified error types, and explains how they can be effectively detected and analyzed for identifying MCA defects. Section 5 evaluates our ADAM approach for its ability of identifying defects in context-aware applications and compares it to existing work. Section 6 presents and discusses the related work, and finally Section 7 concludes this paper.

2. Archival study

In this section, we investigate three context-aware Android applications, namely, Locale (Locale, 2011), Tasker (Tasker, 2011), and Setting Profiles (Setting Profiles, 2011). Locale is from top 10 winners of the first ADC (Android Developer Challenge competition), and Tasker is from top 30 winners of the second ADC. All the three applications are successful commercial products, which are receiving increasing downloads at the Android market. The three applications offer release notes or change logs at their web sites. These provide us information about their experienced failures and defects, which enable us to analyze their bug-fixing histories.

2.1. Application background

The three applications feature a common MCA paradigm: a reusable execution engine perceives environmental conditions and makes self-adaptation according to a set of user-defined *adaptation rules*. These rules are customized by users for different situations by adjusting rule conditions and their associated actions. When these rule conditions are satisfied, their associated actions are conducted as response. Adaptation rules thus play an important role in implementing an MCA's logic.

Failures of the three applications were reported during their development and deployment. From the first release v1.0 to the

Table 1
Application releases, failures, and defect partitioning (percentages sum up to 100% for each row).

Application	Total releases (#)	Total failures/defects (#)	Defect partitioning (%)		
			Sensing defects	Adaptation defects	Other defects
Locale	16	47	36.2%	29.8%	34.0%
Tasker	38	198	25.3%	33.3%	41.4%
Setting Profiles	26	40	32.5%	45.0%	22.5%

latest one v1.4.3, Locale has a total of 16 releases with 47 failures reported. Tasker, with more sophisticated functions, has experienced an even tougher development period. It has a total of 38 releases (v0.1–v0.38b) with 198 failures reported. Finally, Setting Profiles has a total of 26 releases with 40 failures reported. Table 1 lists these statistics data for the three applications.

From the descriptions in these applications' bug-fixing histories, we can manually figure out by what defects these reported failures have been triggered. For convenience, we assume that each reported failure corresponds to one defect. As such, the numbers of total defects experienced by the three applications are 47, 198, and 40, respectively. We shall discuss the impact of this treatment later in Section 2.5.

In this paper, we are interested in partitioning these defects into two types: *sensing defect* and *adaptation defect*. This is because they correspond to two aforementioned concerns in the MCA paradigm: (1) acquiring contexts from sensors and (2) using them for adaptation. Then, the first objective of our study is to investigate *how common these two defect types are in MCAs?*

2.2. Defect partitioning

To fulfill our first objective, we manually analyzed the 285 failures/defects associated with the three applications, and partitioned them into the following three categories:

(1) **Failures due to sensing defects**: An application fails (with a failure report) due to defects in its application logics concerning sensor data handling. This can be identified by manually analyzing the bug-fixing history associated with this failure report. For example, Tasker's (v.20b) Wi-Fi state "often never becomes inactive". This Wi-Fi state handling defect may cause the application attempts repeated radio controls (e.g., trying to turn off the Wi-Fi connection but always failing to do so). Then the phone hosting this application loses power quickly, causing the application to crash or exit abnormally.
(2) **Failures due to adaptation defects**: An application fails due to defects in its application logics concerning condition evaluation and adaptation. For example, Locale (v1.0) may "rapidly bounce between situations or Defaults when accuracy is low or when entering a building". This instable adaptation switches a phone's profiles quickly without reaching a stable state, causing the application unable to respond to any user input (i.e., application freezing).
(3) **Failures due to other defects**: An application fails due to defects other than the aforementioned reasons. They can be GUI, backup, or log defects.

Based on our failure/defect partitioning, Table 1 lists and compares the percentages of these three categories. We observe that:

(1) Sensing and adaptation defects are common.
(2) They account for over 66.0% of total failures in Locale and Setting Profiles.
(3) Although they account for 58.6% of total failures in Tasker, their occurrences (116) are actually more than the sum of their counterparts in Locale and Setting Profiles (62). It happens that Tasker's GUI is more sophisticated and error-prone, thus increasing the percentage of other defects.

As a summary, for the 80 releases of the three applications, sensing and adaptation defects account for a total of 58.6–77.5% of the 285 reported failures. It shows that these two defect types are common in context-aware applications. The high percentage confirms our earlier conjecture of extra challenges existing in developing context-aware applications. This strongly suggests the need for an effective approach to identifying such defects.

2.3. Error classification

Our follow-up question is how to identify these sensing and adaptation defects. To answer this question, we study the occurrences of undesirable properties (or errors) when such defects are executed and propagated to output as reported failures.

We distinguish errors from failures because the transient existence of an error may not necessarily lead to a failure. An example error is the loss of *predictability* of program behavior, i.e., an application's adaptation becomes random and unpredictable. Yet, the occurrence of an error indicates a potential defect since it is undesired.

Based on the bug-fixing descriptions of the three applications, we manually analyzed errors from their reported failures. We note that a reported failure may be associated with more than one error. That is, a failure may come with the occurrences of multiple undesirable properties. To better study these errors, we investigated existing work on testing and verifying context-aware or self-adaptive systems, and derived four common criteria for classifying these errors. Then, the second objective of our study is to investigate *whether the four criteria are adequate to cover these errors associated with MCA failures?*

We first introduce the four criteria for error classification. They concern the *predictability*, *stability*, *reachability and liveness*, and *consistency* of context-aware applications, respectively:

(1) **Predictability**. *Predictability* requires an application's adaptation behavior to be predictable upon any situation (Cheng et al., 2009), i.e., absence of *non-determinism*. For example, if an application contains multiple adaptation rules, the criterion would require that there is at most one rule that can be triggered at any situation. If all adaptation rules contain only atomic propositions and Boolean operators (Sama et al., 2010a) and keep static at runtime, static analysis can help find those rules whose triggering conditions overlap with each other. Such rules can cause non-determinism when triggered simultaneously due to overlapping conditions. In this case, they are considered as faulty (i.e., containing defects). However, if these rules use quantifications or are changeable at runtime, non-determinism would be difficult to judge statically and avoid at design time. Then some rules may be triggered simultaneously at runtime for adaptation, causing unpredictable application behavior. Many pieces of existing work consider this harmful (Ranganathan and Campbell, 2003; Sama et al., 2010a). Some work detects non-determinism (i.e., the loss of predictability) at runtime, and fixes

Table 2
Error classification for the three applications (percentages may sum up over 100% for each row).

Application	Failures due to sensing or adaptation defects (#)	Associated errors (%)				
		Pred. errors	Stab. errors	Reac. errors	Cons. errors	Others
Locale	31	41.9%	25.8%	29.0%	16.1%	35.5%
Tasker	116	21.6%	6.0%	7.8%	51.7%	23.3%
Setting Profiles	31	45.2%	32.3%	35.5%	25.8%	19.4%

or tolerates it as requested (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005).

(2) **Stability**. *Stability* requires an application to be stable after each adaptation (Cheng et al., 2009; Sama et al., 2010a). Otherwise, an application may continually adapt itself without reaching a stable state, forming a long adaptation trace or even running into an infinite loop. In this case, certain actions with respect to the states in this adaptation trace or loop may be lost. As a result, the application behaves as freezing (without any response to user input), and it may fail to reach certain states (as these states are skipped). The latter also relates to the *reachability* property to be discussed next. Both research work and development practice have considered such instability harmful (Locale, 2011; Setting Profiles, 2011; Tasker, 2011; Sama et al., 2010a).

(3) **Reachability and liveness**. Reachability and liveness are properties commonly studied in model checking for program's correctness. *Reachability* requires each state to be reachable from the initial state of an application. *Liveness* requires an application to be able to leave from any state and trigger any rule within a finite period of time (Sama et al., 2010a). The two properties necessitate the visit of each application state and triggering of each adaptation rule. When certain states or rules are never used, this is a strong indicator of potential defects in the application. For example, the aforementioned instability may cause certain states always skipped, leading to the loss of predesigned functionalities. This is usually undesired.

(4) **Consistency**. An application's perception of its environments (in terms of states) should be *consistent* with its actual environmental conditions (in terms of contexts). This requires that the application should always have its current state's *guard* conditions *hold* with respect to its available contexts. This is for realizing the consistency between an application's internal states and its external environments. Otherwise, the application's current state does not truly reflect its environmental conditions. Some work considers such inconsistency harmful (Kulkarni and Tripathi, 2010; Xu et al., 2010).

With these four criteria in mind, we analyzed the 178 failures caused by sensing or adaptation defects in the three applications. We classified these failures into five categories according to their exhibited errors with respect to the four criteria. These errors are due to the violation to the aforementioned four criteria. For ease of presentation, we name them *predictability error* (pred. error), *stability error* (stab. error), *reachability and liveness error* (reac. error), *consistency error* (cons. error), and *others* (none of the above). Table 2 lists this error classification for the three applications. We note that the percentages in each row may sum up above 100%, because a failure may exhibit, and therefore be associated with, more than one error type as mentioned earlier.

We give several examples for illustration:

(1) Locale v1.0 may encounter "concurrency issues that could cause the Wi-Fi setting to fail if Locale was simultaneously checking location" – *predictability error* (checking the Wi-Fi state may fail or not, depending on whether Locale is simultaneously checking its user's location).

(2) Locale v1.0 may also "rapidly bounce between situations or Defaults when accuracy is low or when (its user is) entering a building" – *stability error* (continual adaptations occur in a loop and never stop).

(3) Locale v1.1 may "add invisible conditions to Defaults, the end result being that Defaults might not always be active" – *reachability and liveness error* (the state associated with the Defaults profile becomes never active).

(4) Tasker v0.24b needs to clear profile data but may forget to do so, causing new contexts inconsistent with its actual state: "newly created contexts could (still) show as active" – *consistency error* (Tasker believes that an obsolete profile is still active, but actually it is not).

2.4. From errors to defects

From Table 2, the three applications reported a total of 178 failures caused by sensing or adaptation defects. Among these failures, we observe that 75.3% (134) of them are associated with at least one of the aforementioned four error types. Table 2 also gives the error distribution for the three applications. From the distribution, we observe that *the four error types have given a good coverage of the failures caused by sensing or adaptation defects in MCAs*. This strongly suggests the correlations between the four error types and sensing or adaptation defects. It motivates us to formulate an effective approach to detecting these errors in order to identify MCA defects.

We consider detecting these errors at runtime (i.e., dynamically). This is because some errors are hard to detect by static analysis. This happens when an MCA allows its application logic to evolve at runtime (e.g., changing its state, rules, or even contexts), or when its adaptation rules use quantifications (e.g., universal or existential quantifiers, which make the rule subsumption checking undecidable). For example, Locale allows rule changes with location changes, and its rules use quantifications to check whether any user's location falls in the scope of earlier registered locations. Such dynamics and expressive power requirements make it hard for static analysis to detect errors precisely.

In order to effectively detect errors and relate them back to original defects in MCAs, we identify the following three requirements:

(1) **An expressive model that allows precise specification for an MCA's adaptation semantics**. The model should be capable of precisely specifying an MCA's adaptation behavior at runtime. A common limitation in existing work (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Kulkarni and Tripathi, 2010; Ranganathan and Campbell, 2003; Sama et al., 2010a) about modeling MCAs is the lack of support for *runtime dynamics* (allowing application logic's evolution at runtime and context's dynamic update) and *high expressive power* (allowing quantifications). As discussed earlier, such support is necessary for precisely modeling an MCA's adaptation behavior.

(2) **Formal definitions of the four discussed error types, and their automated detection mechanism built into the model**. This would enable the detection of these errors to be done in a systematic way rather than manually to facilitate later defect analysis.

(3) **An ability of relating detected errors to their responsible defects in the model and of providing their occurring conditions (i.e., states, rules, and contexts under which these errors have occurred)**. Such information would be useful for analyzing these defects and giving constructive hints how they can be fixed.

In Section 3, we shall explain in detail how these requirements can be addressed using our ADAM approach.

2.5. Further explanations

Our archival study has simplified some analysis processes, and they need some explanations.

2.5.1. From failures to defects

In Table 1, we assume that each failure corresponds to one defect, and we partition these defects using the numbers of their corresponding failures. Although this assumption may not strictly hold in practice, the difference between the numbers of failures and defects will not be high. First, a failure being reported in a bug-fixing history implies that this failure has been really caused by some defect (since the failure/defect is fixed later). Second, within one application release, failures that are caused by the same defect are typically grouped together (unlikely occupying multiple bug-fixing entries). Third, across different application releases, newly reported failures are most likely caused by new defects (since earlier defects have been fixed before new releases). Therefore, the numbers of failures and their associated defects are most likely close to each other. Even if the two numbers are not strictly equal, it does not affect our conclusion significantly that sensing and adaptation defects account for a large proportion of all reported failures.

2.5.2. From failures to errors

Analyzing and classifying errors exhibited by failures is more challenging, and therefore we cannot 100% guarantee the correctness of our error classification. The classification may not be fully precise or complete since it is essentially a manual process. To be more precise, we allowed a failure to be associated with more than one error as long as it exhibits some undesired properties according to its bug-fixing descriptions. By doing so, we tried to avoid the difficulty of choosing a single error for each failure. Regarding the completeness, there may be more error types applicable for the three applications rather than only four as studied in this paper. However, we note that our focus is to study whether those failures due to sensing or adaptation defects can be well covered by a set of error types. Therefore, increasing error types does not change our conclusion that these failures have been already well covered by the four error types (75.3%). Adding more error types may help better identify MCA defects by detecting more error types, i.e., improving the effectiveness of our ADAM approach. We leave it to our future work.

3. The ADAM model

Our ADAM approach contains two parts. The first part is a model, called ADAM model, for specifying the adaptation behavior of a given MCA. The second part is a set of assertion checks built in the ADAM model, for automatically detecting errors in this MCA at runtime. In this section, we present the first part (the ADAM model), and the second part is presented in the next section.

3.1. Overview

Our ADAM model contains some unique features. It specifies the adaptation behavior of an MCA in terms of *states*, *adaptation rules* expressed in first-order logic, and *configurations* that describe this MCA's runtime attributes. Assuming discrete-time semantics, the ADAM model allows a precise modeling of an MCA's adaptation behavior, as compared to existing work that does not support *quantifications* or *timing requirements*, or that does not model how an MCA's logic *evolves with context changes* (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Sama et al., 2010a).

Given an MCA, its ADAM model consists of its *static aspect* $M_S := (S, R, s_0, R_0)$ and *dynamic aspect* $M_D := G_t := (C_t, s_t, R_t)$. We explain the two aspects below.

For the static aspect M_S, S is the set of all possible states in this MCA, and R is the set of all rules designed for this MCA. Among all states, state s_0 ($\in S$) is special, and defined as the *initial* state when the MCA starts. Here we assume that an MCA can be at one state at any time, and s_0 is its first state when it starts. Similarly, a rule subset R_0 ($\subseteq R$) is the *initial* set of *active* rules when the MCA starts. Here, "active" means "in use". Not all rules are in use at any time, and therefore an MCA uses only a subset of all rules at a time. We note that all elements (S, R, s_0, R_0) in the static aspect M_S can be decided at design time, and do not change at runtime (thus called *static*).

The dynamic aspect M_D is defined as the MCA's *configuration* G_t at time t. Configuration G_t is a runtime artifact that describes available contexts of this MCA at time t, its state at time t, and its active rules at time t. Formally, G_t is defined as (C_t, s_t, R_t), in which C_t is the set of available contexts, s_t ($\in S$) is the MCA's state, and R_t ($\subseteq R$) is the MCA's set of active rules at time t. We note that all elements (C_t, s_t, R_t) in a configuration G_t (i.e., the dynamic aspect) would be decided at runtime (thus called *dynamic*). Developers or users do not have to specify a configuration manually as it can be automatically calculated (to be explained later).

The static aspect is the part that may contain defects since it is designed by developers or users. These defects manifest themselves as observed failures through the dynamic aspect, which contains an MCA's runtime attributes. This is why we model both the static aspect and dynamic aspect for an MCA, while existing work focused on the static aspect only.

For convenience, when t is the current time, we also call s_t "current state" and R_t "current set of active rules". We assume that the time is 0 when an MCA starts running (discrete time semantics). Then, the initial configuration (when the MCA starts) is $G_0 := (C_0, s_0, R_0)$, where C_0 is typically an empty set (no available contexts at the beginning), and s_0 and R_0 come directly from M_S. With the progression of time t, G_t evolves accordingly. We discuss G_t's evolution later.

In the following, we present an illustrative application scenario, and then explain an MCA's *states*, *rules*, *configurations*, and *conditions* in turn using this application scenario.

3.2. Illustrative application scenario

We explain these concepts using a stock tracking application for illustration. The application is based on our past experiences when deploying this application for a paper manufacturing company, where it tracks stock item transportations in a warehouse.

A forklift continually transports stock items from the loading bay of this warehouse to its storage bay. Several sensing technologies have been deployed in the warehouse to offer different types of contexts to support task automation. These contexts are fed to an embedded system integrated with the forklift, and help the system decide where the forklift is located and what its next task is.

For example, radio frequency identification (RFID) readers are used to detect the locations of stock items (attached with RFID tags) under transportation. Pressure sensors are installed at the loading bay and storage bay, for detecting whether the forklift has arrived at the loading bay (where a next set of stock items can be loaded for

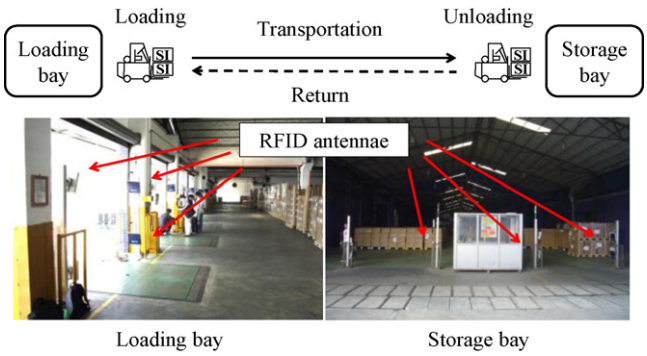

Fig. 1. A stock tracking application scenario (SI: stock item).

transportation) or the storage bay (where these transported stock items can be unloaded and checked).

Fig. 1 gives a conceptual illustration of this application scenario as well as its physical scenario photos. The application is used in the following for illustrating concepts, and will also be used as one of experimental subjects in later evaluation.

3.3. State

State represents an MCA's perception of its physical environment. It can be an explicit, unique name or an implicit representation in terms of values of important variables in this MCA. For convenience, we do not distinguish these two cases, instead giving a unique name (one string) for each state in the MCA.

At each time point, an MCA stays at one state. As time goes, the MCA's state may change accordingly, representing its changing perception of its environment. As such, an MCA can transit from one state s_1 to another s_2 freely at runtime. However, such state transition may not always be valid. When a state transition is triggered by a defect in an MCA's adaptation logic, the MCA's state may become inconsistent with its available contexts (recall our aforementioned consistency criterion). Therefore, one has to verify such state transition. This can be done by adding a guard condition *s.guard* to each state *s*. We require that before the transition s_1.*guard* should hold, and after the transition s_2.*guard* should hold. This requirement is also known as *one-point adaptation* semantics (Zhang and Cheng, 2006a,b) in self-adaptive systems.

By doing so, an MCA can make sure that its perception of its environment (state) is always consistent with its contexts, and that this consistency has not been violated by its adaptation. Besides, after a transition, the MCA's new state's guard condition s_2.*guard* should keep holding as long as the MCA still stays at this state. This is because at state s_2, the MCA may conduct other adaptations that do not necessarily change its state (e.g., only changing its active rules). In this case, we still need to guarantee the consistency for such adaptations (Kulkarni and Tripathi, 2010).

Take the stock tracking application for example. The application can be at one of the four states "Loading", "Transportation", "Unloading", and "Return", as shown in Fig. 1. When the application stays at the "Unloading" state, the forklift and its pallet (for transporting stock items) are supposed to be *both* at the storage bay. This assumption forms a guard condition for the "Unloading" state. If this condition is violated, then either the forklift or its pallet is not at the storage bay. This conflicts with the "Unloading" state, in which the forklift is supposed to be able to unload its stock items. It implies the inconsistency between this application's perception of its environments and its contexts. Therefore, this guard condition can help the application judge whether its current state is valid and whether its last adaptation leading to this state contains any defect.

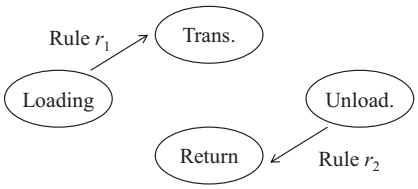

Fig. 2. Illustration for two state transitions ("Trans.": Transportation; "Unload.": Unloading).

3.4. Rule

Rule closely relates to an MCA's adaptation, and is also called *adaptation rule*. Rules form a structured paradigm for identifying interesting situations and making corresponding adaptations for an MCA. Each rule connects a situation to an adaptation. When the situation is detected, the adaptation is conducted as response. Formally, a rule $r := (guard, trigger, act, priority)$ consists of four attributes. We explain these attributes in the following.

3.4.1. Guard condition

A rule can be *active* (in use) or *inactive* (not in use) when an MCA runs. Although an MCA may have many rules designed in advance, not all of them are necessarily active *at any state* and *at any time*. An MCA's set of active rules tends to vary with the MCA's current state and/or time. When an MCA transits from one state to another, its active rules may change, reflecting the change of this MCA's allowed adaptations *across states*. Even if an MCA keeps staying at the same state, its active rules may still vary as time goes, reflecting the change of this MCA's allowed adaptations *inside one state*.

Due to such dynamics, each rule needs a guard condition to ensure its validness as being *active* for an MCA. Similar to a state's guard condition, a rule *r*'s guard condition *r.guard* is used to verify whether this rule *r* can become active (or called *enabled*) for an MCA's current state.

Take the stock tracking application for example. Suppose that the application uses a rule r_1 at the "Loading" state to make itself to transit to the "Transportation" state (i.e., r_1 is associated with an adaptation, which triggers the MCA to make a state transition), and uses another rule r_2 at the "Unloading" state to transit to the "Return" state, as shown in Fig. 2. Then the two rules r_1 and r_2 can be *validly* enabled for states "Loading" and "Unloading", respectively. Such associations (enabling a certain rule for a certain state) are reasonable.

Suppose that developers make a mistake in the application logic by enabling rule r_2 for the "Loading" state and rule r_1 for the "Unloading" state (i.e., the two associations are mistakenly switched). Then such adaptations do not make sense. This would result in a logical flaw or defect (i.e., trying to transit from the "Loading" state to the "Return" state and from the "Unloading" state to the "Transportation" state). To avoid such problems, rules r_1's and r_2's guard conditions r_1.*guard* and r_2.*guard* should be specified. These guard conditions must hold before rules r_1 and r_2 can be enabled for an MCA's current state, and must keep holding as long as rules r_1 and r_2 are still active for this state. For example, condition r_1.*guard* can be specified as "both the forklift and its pallet are at the loading bay", and condition r_2.*guard* can be specified as "both the forklift and its pallet are at the storage bay". If these conditions are not satisfied (not holding), rules r_1 and r_2 are not allowed to become active. Such guard conditions make sense, because rule r_1/r_2 can only be enabled for state "Loading"/"Unloading", where the forklift with its pallet must be at the loading/storage bay, respectively.

Therefore, a rule's guard condition can be used to check this rule's validity with respect to an MCA's current state, as well as

disclosing the *inconsistency* between this rule and the MCA's contexts (when the contexts cause the rule's guard condition violated).

3.4.2. Triggering condition and action list

Triggering condition and *action list* are two other attributes in a rule. Different from guard condition, which checks the validity of a rule as being active, *triggering condition* must be satisfied so that this rule can be *triggered*. Here, "triggered" means that the rule can be executed, i.e., all actions in the rule's *action list* can be conducted as an MCA's adaptation.

As such, a rule r's triggering condition $r.trigger$ specifies an interesting situation for an MCA. When $r.trigger$ is satisfied, we say that the situation is detected and rule r can be triggered. When rule r is triggered, rule r's associated action list $r.act$ is conducted. Conducting an action list is to execute all actions specified in this list. These actions can include the following types:

(1) **Update state: (update, s)**. The MCA transits to a new state s.
(2) **Update active rule: (enable, r_1) or (disable, r_2)**. The former enables a rule r_1 (becoming active) for the MCA, and the latter disables a rule r_2 (becoming inactive).
(3) **Update rule's priority: (change, r_3, p)**. Rule r_3's priority is replaced with a new value p (to be explained soon).
(4) **Update context: (add, c_1) or (remove, c_2)**. The former adds a new context c_1 to the MCA's set of available contexts, and the latter removes an existing context c_2 from it.
(5) Other *immaterial* actions that do not update states, rules, or contexts in the MCA's ADAM model.

We propose the above action types based on our experiences of investigating existing MCAs (e.g., three commercial context-aware applications in Section 2) and of developing research-oriented MCAs (e.g., three experimental subjects used in our later evaluation). We do not intend to propose a complete list, and therefore such action types are extensible. Many practical actions can belong to the last type (immaterial actions) as long as they do not update states, rules, or contexts in an MCA at runtime, even if they look different. The point here is that we *explicitly* model actions in a rule and shall handle their *impact* on an MCA's application logic and available contexts later, while existing work oversimplified or omitted this part.

3.4.3. Priority

The last attribute in a rule, *priority*, indicates the rule's selection priority. It is a natural number ($\in N$), representing the priority order for a rule to be selected from those rules whose triggering conditions have been satisfied simultaneously. When several active rules *all* have their triggering conditions satisfied with respect to a set of available contexts, all of them can be triggered. To avoid non-determinism, only one rule is selected for triggering. Priority is used for this purpose.

Although such a priority mechanism has been widely used for addressing the non-determinism issue (Ranganathan and Campbell, 2003; Sama et al., 2010a), there are various reasons limiting its usefulness. For example, assigning a *unique* priority value to every rule may not always be possible, especially when rules do not have any ordering in their semantics. In addition, if a rule's priority is subject to change at runtime, non-determinism still remains as an issue. For example, a smart streetlight application (Lu et al., 2006) may dynamically update the priority for a rule that increases the illumination when many visitors are nearby. With such dynamics, non-determinism cannot be fully avoided purely by a static priority mechanism. Therefore, predictability (one of our aforementioned criteria) may still be violated at runtime and cause MCA errors.

S1: $R_{tr} := \{\, r \mid r.trigger(C_{t2}) = \text{true} \wedge r \in R_{t1} \,\}$;

S2: $r_{tr} :=$ selectHighestPriority(R_{tr});

S3: $s_{t2} := s$, if there exists (update, s) $\in r_{tr}.act$; s_{t1}, else;

S4: $R_{t2} := R_{t1} \cup \{\, r \mid (\text{enable}, r) \in r_{tr}.act \,\} - \{\, r \mid (\text{disable}, r) \in r_{tr}.act \,\}$;

S5: $r.priority := p \mid (\text{change}, r, p) \in r_{tr}.act$;

S6: $C_{t2} := C_{t2} \cup \{\, c \mid (\text{add}, c) \in r_{tr}.act \,\} - \{\, c \mid (\text{remove}, c) \in r_{tr}.act \,\}$;

Fig. 3. Configuration evolution process (Steps S1–6).

3.5. Configuration

We have introduced states and rules in an MCA. From them, our ADAM model's static aspect M_S can be decided for this MCA. As mentioned earlier, the ADAM model's dynamic aspect M_D is decided by the MCA's configuration at runtime. Let the MCA's configuration be $G_t := (C_t, s_t, R_t)$ at time t.

State s_t reflects the MCA's perception of its physical environment (i.e., state) at time t, and R_t represents the MCA's set of active rules (i.e., rules in use). Then, what kind of adaptations is allowed at time t is fully decided by configuration G_t. To see it, consider the set of available contexts C_t at time t. Some rules in R_t may be triggered with respect to available contexts C_t. By applying priorities, one active rule is selected from them for execution. The actions in this rule's action list are conducted as the adaptation. All relevant factors (e.g., C_t, R_t) have been included in the configuration G_t. Therefore, configuration G_t fully decides the MCA's behavior at time t.

A configuration can evolve with time. Suppose that at time t_1, an MCA's configuration is G_{t1}. Then at the next time t_2 ($=t_1 + 1$), context changes (from C_{t1} to C_{t2}) are perceived from physical environments. These context changes may trigger some adaptation, which in turn updates the MCA's current state, active rules, or other elements (e.g., a rule's priority). As a result, the MCA's configuration G_{t2} at t_2 may differ from G_{t1}. This process is called *configuration evolution*.

When an MCA's configuration evolves from one to another, its current state and active rules may change accordingly. This implies that its perception of its physical environment (i.e., current state) and allowed adaptations (i.e., set of active rules) may also change. Therefore, *configuration evolution is essentially an adaptation process*, in which an MCA's application logic evolves with time.

Fig. 3 lists six main steps (S1–6) in our configuration evolution process. We brief them below. For ease of presentation, the six steps are assumed to execute *sequentially* (parallel execution is not impossible, but it is beyond the scope of this paper):

(1) **Rule triggering (Steps S1–2)**. Suppose that from time t_1 to t_2, the MCA's set of available contexts changes from C_{t1} to C_{t2}. Steps S1–2 are to select a triggered rule due to this change. To do so, all active rules (R_{t1}) are checked on whether the triggering condition of any of them is satisfied under contexts C_{t2}. If yes, such rules are put into a set R_{tr}. From it, the one r_{tr} that carries the highest priority is selected as *triggered*.
(2) **State update (Step S3)**. Consider the triggered rule r_{tr}. If it contains a state update action in its action list $r_{tr}.act$, then Step S3 sets the MCA's state s_{t2} to this new state; otherwise s_{t2} remains at the original state s_{t1} without any change.
(3) **Rule update (Steps S4–5)**. Similarly, following action list $r_{tr}.act$, Step 4 proceeds to update the MCA's set of active rules, and Step 5 updates any rule's priority if requested.
(4) **Internal context update (Step S6)**. Although the set of available contexts has already changed from C_{t1} to C_{t2}, Step S6 may *further* update this set by inserting newly generated contexts and discarding removed contexts as requested in action list $r_{tr}.act$.

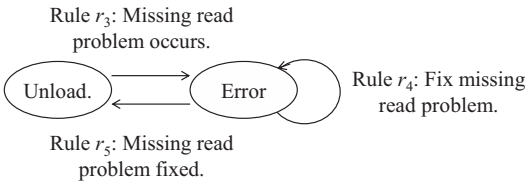

Fig. 4. An internal context update scenario ("Unload.": "Unloading").

$f :=$ let $v = c$ (f) | forall v in $X[t]$ (f) | exists v in $X[t]$ (f) |
(f) and (f) | (f) or (f) | (f) implies (f) |
not (f) | $predicate(v, \ldots, v)$

Fig. 5. Language syntax for specifying conditions.

The first five steps (S1–5) are straightforward. We explain a little bit more about Step S6. Take the stock tracking application for example. Suppose that the application stays at the "Unloading" state at some time. The application checks whether the set of stock items detected at the storage bay *matches* its counterpart at the loading bay. By this, the application ensures that no stock item gets lost during its transportation. If the check fails, it may be the case that a missing record problem occurs, i.e., some stock items detected at the loading bay can no longer be detected again at the storage bay. This may not necessarily be the physical loss of a stock item. It can be caused by a "missing read" error commonly found in real-life RFID deployments. That is, the stock item is still on the pallet, but the RFID reader at the storage bay just accidentally missed detecting it. Regarding this problem, the application transits from the "Unloading" state to the "Error" state, and invokes a rule r_4 for fixing this problem. The actions associated with rule r_4 would *regenerate* this missing read.

We note that by doing so, the stock item contexts are *internally updated* (i.e., Step S6), as compared to those *external updates* directly caused by context changes from physical environments. Fig. 4 illustrates such a scenario of internal context update supported by our ADAM model. We also note that existing work (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Sama et al., 2010) has omitted the modeling of such internal context updates in specifying an MCA's adaptation behavior, and therefore the modeling is not precise.

Finally, we note a special case in the configuration evolution process. From time t_1 to t_2, the MCA's set of available contexts may keep *unchanged*, i.e., $C_{t2} = C_{t1}$ (no new contexts perceived from physical environments). Or, even if $C_{t2} \neq C_{t1}$, there are no active rules whose triggering conditions are satisfied under C_{t2}, i.e., $R_{tr} = \phi$. For these two cases, there is no triggered rule, and therefore the configuration G_{t2} at time t_2 would *trivially follow* the configuration G_{t1} at time t_1 without any change in s_t and R_t (of course, the set of available contexts still changes from C_{t1} to C_{t2}).

3.6. Condition

Our ADAM model is almost ready for specifying the adaptation behavior of a given MCA. Its static aspect is built from the states and rules of this MCA. Its dynamic aspect is the MCA's configuration, which evolves at runtime (and therefore is automatically calculated). Still, we have not explained how to specify a state's guard condition, and a rule's guard condition and triggering condition. In the following, we present a first-order logic (FOL) based language for specifying these conditions.

Using this language, a condition is specified by a formula that is recursively constructed using the syntax shown in Fig. 5. This language is built on related work and has been extended for modeling MCA conditions:

(1) The "and", "or", "implies", and "not" formulae follow their traditional FOL interpretations, i.e., *conjunction*, *disjunction*, *implication*, and *negation* (Capra et al., 2003; Chomicki et al., 2003; Sama et al., 2010a).
(2) The "forall" and "exists" formulae use universal and existential quantifications to define a variable v that can take any value in its domain (Nentwich et al., 2002; Xu et al., 2010). In our ADAM model, such variable can take any context from a given set $X[t]$. Here, $X[t]$ can refer to any *finite* set of contexts. We use parameter t to restrict a time scope for the contexts considered in $X[t]$. A negative value means "in the past $-t$ time" and a positive value means "in the future t time". By this restriction, we make the contexts constrained by a timer and ensure the evaluation of "forall" and "exists" formulae *decidable*.
(3) The "let" formula defines a variable v that can take only one value (Xiong et al., 2009). In our ADAM model, such variable can be assigned with a context c, which can be a constant or a value subject to change at runtime.
(4) The terminal *predicate* can refer to any user-defined predicate in the application domain. By this, the language is extensible for domain-specific predicates.

We give two examples for illustrating how to specify MCA conditions.

Example 1. Triggering condition for rule r_3 in Fig. 4: *There exists at least one stock item that has been detected at the loading bay but cannot be detected again at the storage bay within its transportation time*. This condition can be specified as follows:

exists r_l in $LOAD[-t_1]$ (forall r_s in $STOR[t_2]$ ($unmatch(r_l, r_s)$)).

In this condition, $LOAD[-t_1]$ represents the set of RFID product codes collected at the loading bay in the past t_1 time. $STOR[t_2]$ represents the set of RFID product codes to be collected at the storage bay within the following t_2 transportation time. Predicate *unmatch* judges whether a pair of RFID product codes does not match each other (i.e., does not refer to the same RFID product code or stock item).

Example 2. Guard condition for rule r_1 in Fig. 2: *The forklift and its pallet should be both at the loading bay*. This condition can be specified as:

(let $p = pres_{load}$ ($down(p)$)) and (let $t = palt_{load}$ ($dect(t)$)).

In this condition, context $pres_{load}$ is the status of the pressure sensor installed at the loading bay, and function *down* judges whether the forklift stays right here (i.e., the pressure sensor is pressed down). Context $palt_{load}$ is the detection result of the RFID reader installed at the loading bay, and function *dect* judges whether the pallet is detected here (true means that the pallet is detected right at the loading bay).

3.7. Complete ADAM model

With the preparation of states, rules, configurations, and conditions, our ADAM model is ready for specifying the adaptation behavior for a given MCA. Fig. 6 illustrates the complete ADAM model for our stock tracking application. It includes the aforementioned features illustrated in Figs. 1, 2 and 4, as well as some new features such as supporting energy-awareness and extended error fixing mechanisms. We label states (ST1–9), rules (RL1–14), possible state transitions (by arrows) in Fig. 6. All rules' triggering

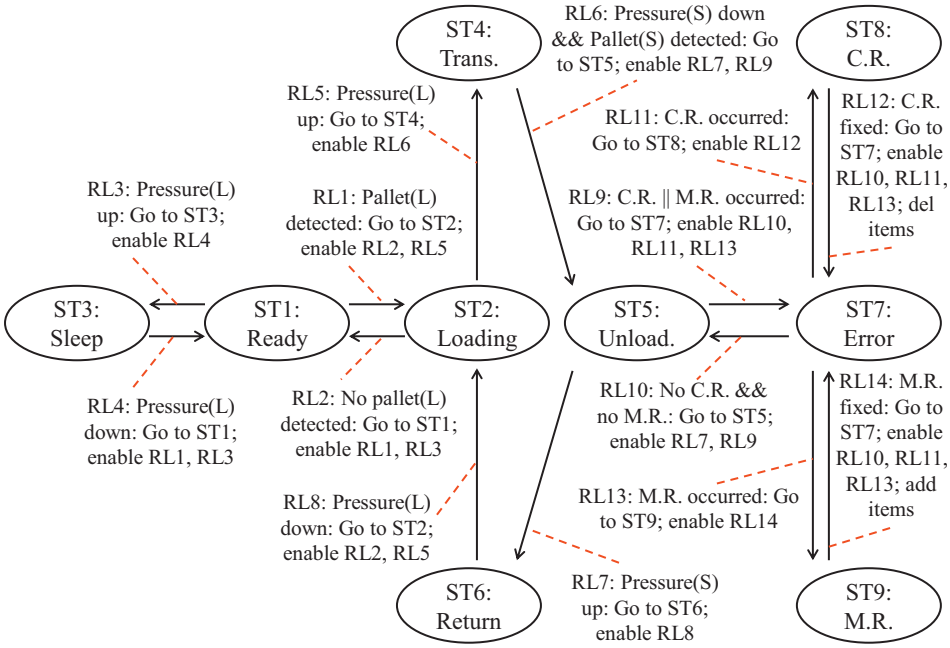

Fig. 6. The complete ADAM model for the stock tracking application ("Trans.": Transportation; "Unload.": Unloading; "C.R.": Cross read; "M.R.": Missing read; Pressure(L) represents the pressure sensor installed at the loading bay and pressure(S) represents the one at the storage bay; Pallet(L) represents a pallet being detected at the loading bay).

conditions and their associated actions are annotated in a simplified text manner, and guard conditions are omitted for simplicity. The application starts with state ST1 ("Ready") with two active rules (RL1 and RL3).

This stock stocking application will also be used in our later evaluation. In the following, we discuss how to detect errors inside an ADAM model at runtime so as to identify defects in its corresponding MCA.

4. Detecting errors and identifying defects

Our goal is to identify defects in an MCA. Some defects are naive. They are directly observable from the MCA's associated ADAM model. For example, a rule's action list may contain conflicting actions that set this MCA to different states simultaneously, or enable and disable the same rule simultaneously, or change a rule's priority to different values simultaneously. As these defects are easy to identify, we do not elaborate on them. In the following, we discuss detecting the four error types mentioned in Section 2. Detecting these errors helps identify non-trivial defects in an MCA.

We design various *assertion checks* for detecting these error types. Fig. 7 illustrates four types of assertion checks for predictability errors, stability errors, reachability and liveness errors, and consistency errors, respectively. They are *predictability check* (or *pc* for short), *stability check* (or *sc*), *reachability and liveness check* (or *rc*), and *consistency check* (or *cc*). In Fig. 7, boxes S1–6 refer to the aforementioned six steps in the configuration evolution process. Our assertion checks are integrated into this process. In the following, we explain these checks in turn.

4.1. Predictability check

As mentioned earlier, predictability requires an MCA's adaptation behavior to be always predictable. In other words, at any time the number of active rules whose triggering conditions are satisfied

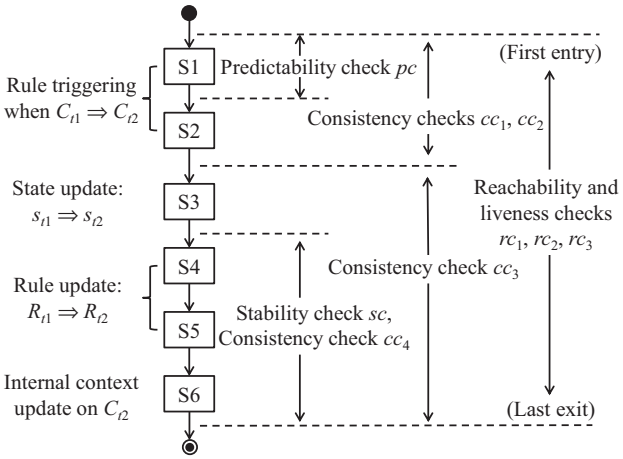

Fig. 7. Four types of assertion check: *pc*, *sc*, *rc*, and *cc*.

with respect to this MCA's contexts should be always no more than 1.

When an MCA starts, its set of available contexts is an empty set. As time goes, this set may contain new contexts captured from physical environments, and thus trigger active rules. Predictability check requires that whenever context changes occur (i.e., new contexts added into the set of available contexts or obsolete contexts removed from it), the number of active rules whole triggering conditions are satisfied can only be *at most* 1.

Let function *triggered*(R, C) return a subset of rules ($\subseteq R$) such that the triggering condition of each rule r in this subset is satisfied under the given set of contexts C. Let function $hp(R)$ return those rules that carry the highest priority among the rules in R. Then, a *predictability error* is defined as the failure of the following predictability check *pc*:

pc: assume (before S1): $|hp(triggered(R_{t1}, C_{t1}))| = 0$,
assert (between S1, S2): $|hp(triggered(R_{t1}, C_{t2}))| <= 1$.

The predictability check examines whether any context change (from C_{t1} to C_{t2}) would cause the occurrence of non-determinism

(i.e., more than one adaptation is possible at a certain situation). If the check fails, a predictability error is detected and logged for our later analysis.

4.2. Stability check

Stability requires an MCA to be *stable* after each adaptation. In other words, *after* each adaptation (i.e., a rule is selected to conduct its actions), there should be no other active rule whose triggering condition is *already* satisfied without further context changes. Otherwise, this new rule would *immediately* be triggered to make *continual* adaptation, causing an adaptation race or cycle as explained earlier.

Formally, a *stability error* is defined as the failure of the following stability check *sc*:

sc: assume (between S3, S4): $|hp(triggered(R_{t1}, C_{t2}))| = 1$,
assert (after S6): $|hp(triggered(R_{t2}, C_{t2}))| = 0$.

The stability check examines whether an adaptation would be followed *immediately* by another adaptation without taking any new context changes. Assertion "$|hp(triggered(R_{t2}, C_{t2}))| = 0$" requires that there should be no active rule that can be triggered after the previous adaptation (i.e., rule updates from R_{t1} to R_{t2} and internal context updates on C_{t2}).

4.3. Consistency check

Consistency requires an MCA's perception of its environments to be consistent with its actual environmental conditions. It concerns the check of guard conditions for an MCA's current state and all its active rules. Consistency check occurs both before and after each adaptation.

Formally, a *consistency error* is defined as the failure of any of the following four consistency checks: cc_1, cc_2, cc_3, and cc_4.

cc_1: assume (before S1): $s_{t1}.guard(C_{t1}) = \text{true}$,
assert (between S2, S3): $s_{t1}.guard(C_{t2}) = \text{true}$.

cc_2: assume (before S1: $\forall r \in R_{t1}$ $(r.guard(C_{t1}) = \text{true})$,
assert (between S2, S3): $\forall r \in R_{t1}$ $(r.guard(C_{t2}) = \text{true})$.

cc_3: assume (before S3): $s_{t1}.guard(C_{t2}) = \text{true}$,
assert (after S6): $s_{t2}.guard(C_{t2}) = \text{true}$.

cc_4: assume (before S4): $\forall r \in R_{t1}$ $(r.guard(C_{t2}) = \text{true})$,
assert (after S6): $\forall r \in R_{t2}$ $(r.guard(C_{t2}) = \text{true})$.

The above four consistency checks can be divided into two groups. Checks cc_1 and cc_2 examine whether all guard conditions (of an MCA's current state and of its active rules) keep holding after context changes from C_{t1} to C_{t2}. The two checks occur *before* each adaptation. Checks cc_3 and cc_4 examine whether all guard conditions (of the MCA's current state and of its active rules) keep holding after the state update from s_{t1} to s_{t2}, rule updates from R_{t1} to R_{t2}, and internal context updates on C_{t2}. The two checks occur *after* each adaptation.

4.4. Reachability and liveness check

Reachability and liveness concern the usefulness of designed states and rules in an MCA. We set up two fields *reached* and *live* for each state and one field *used* for each rule. All these fields are initialized to false when the MCA starts. The *reached* field of a state becomes true when this state has *once* been the current state. The *live* field of this state becomes true when the state has *once* been a non-current state after it has been a current state (i.e., leaving from this state). The *used* field of a rule becomes true when this rule has *ever* been triggered (i.e., its triggering condition is satisfied and its associated actions are conducted).

Formally, there are three checks for these three fields, respectively. A *reachability and liveness error* is defined as the failure of any one of the following three checks rc_1, rc_2, and rc_3.

Table 3
Logged data for each error type.

Error type	Assertion checks	Logged data
Predictability error	Predictability check *pc*	R_{t1}, C_{t1}, C_{t2}
Stability error	Stability check *sc*	R_{t1}, R_{t2}, C_{t2}
Consistency error	Consistency check cc_1	s_{t1}, C_{t1}, C_{t2}
	Consistency check cc_2	R_{t1}, C_{t1}, C_{t2}
	Consistency check cc_3	s_{t1}, s_{t2}, C_{t2}
	Consistency check cc_4	R_{t1}, R_{t2}, C_{t2}
Reachability and liveness error	Reachability and liveness check rc_1	$s.reached$ $(s \in S - \{s_0\})$
	Reachability and liveness check rc_2	$s.live$ $(s \in S)$
	Reachability and liveness check rc_3	$r.used$ $(r \in R)$

rc_1: assume (at beginning): $\forall s \in S - \{s_0\}$ $(s.reached = \text{false})$,
assert (at end): $\forall s \in S - \{s_0\}$ $(s.reached = \text{true})$.

rc_2: assume (at beginning): $\forall s \in S$ $(s.live = \text{false})$,
assert (at end): $\forall s \in S$ $(s.live = \text{true})$.

rc_3: assume (at beginning): $\forall r \in R$ $(r.used = \text{false})$,
assert (at end): $\forall r \in R$ $(r.used = \text{true})$.

The above three checks examine three aspects, respectively: (1) whether any state in an MCA has never been reached (with a false *reached* field), (2) whether any state has never been able to leave from after being a current state (with a false *live* field), (3) and whether any rule has never been used (with a false *used* field). We note that they differ from the other three types of assertion checks mentioned earlier, in that they occur before the *first entry* to the six Steps S1–6 and after their *last exit*. This is because these reachability and liveness errors are judged based on a series of observations to an MCA in its whole lifecycle.

4.5. Check logs and defect analysis

Given an ADAM model of an MCA, the above assertion checks detect at runtime the occurrences of the aforementioned four error types for this MCA. The detection of these errors result in corresponding error logs that contain relevant data for the analysis of their associated defects inside this MCA. Table 3 lists such logged data for each error type. They are exactly the data referred to in corresponding assertion checks. We in the following discuss two error types as example to explain how to analyze for associated defects from these data.

4.5.1. Predictability error

When a predictability error occurs, R_{t1}, C_{t1}, and C_{t2} are logged. This error log explains that under the context changes from C_{t1} to C_{t2}, the set of active rules R_{t1} exhibits a predictability error. The rules whose triggering conditions are simultaneously satisfied then can be directly derived from R_{t1} and C_{t2}. This error log can be combined with nearby consistency error logs to infer associated defects.

For example, if a predictability error occurs by itself (i.e., no nearby consistency errors then), it can be inferred that the rules involved in this error (i.e., those rules whose triggering conditions are satisfied simultaneously) have overlapping conditions. This can be a design defect, causing these triggering conditions specified imprecisely. In this case, developers should fix this defect by reexamining these conditions. On the other hand, if this predictability error occurs together with other consistency errors, then it implies that the MCA has already had its state inconsistent with its available contexts then. As a result, this predictability error occurs as a side effect of this inconsistency. It is likely that the MCA fails to consider certain context noises. Due to this defect, the MCA cannot precisely perceive its physical environment and have to make wrong adaptation (thus causing inconsistency). In this case, developers should examine the types of context noises that have been missed from consideration.

4.5.2. Consistency error

When a consistency error occurs due to the failure of consistency check cc_3, the states s_{t1}, s_{t2}, and the context C_{t2} are logged. This error log explains that under the adaptation from s_{t1} to s_{t2} (i.e., state update), the new state s_{t2} exhibits a consistency error. This error log can be combined with earlier consistency error logs related with state s_{t1} (if any) to infer associated defects.

For example, if a consistency error occurs by itself (i.e., no other consistency errors then), it can be inferred that the adaptation from s_{t1} to s_{t2} can be a faulty action. This defect causes inconsistency between the MCA's new state and its available contexts. In this case, developers should consider revising such state update action. On the other hand, if this consistency error occurs together with other consistency errors related with state s_{t1}, then it implies that this consistency error is probably a side effect of other consistency errors detected earlier at state s_{t1}. Since state s_{t1}'s guard condition has been already violated (leading to earlier consistency errors), this new consistency error trivially follows if the MCA is not able to recover by itself from inconsistency. In this case, developers should consider fixing defects associated with earlier consistency errors or enhance this MCA with some self-recovery mechanisms.

In the following, we experimentally evaluate our ADAM approach for its error detection and defect identification ability. The evaluation includes practical defect analysis examples for selected MCA subjects.

5. Evaluation

We implemented our ADAM approach in Java 6. We name it the ADAM toolkit. To use the toolkit, one first specifies an MCA's application logic by its corresponding ADAM model. The model is expressed as a set of XML documents that describe the MCA's states, rules, and conditions. At runtime, the toolkit loads this model into memory for real execution or simulation. The aforementioned assertion checks are integrated inside the toolkit. They report errors and generate error logs for defect analysis if any. For ease of presentation, we use terms "ADAM toolkit" and "ADAM" interchangeably in this section.

ADAM can work with external context middleware such as Cabot (Xu and Cheung, 2005) to receive contexts for MCA execution. These contexts can be from physical sensors or simulation algorithms. This enables ADAM to be tested with various setups.

In the following, we evaluate ADAM's error detection and defect identification ability through three MCAs and compare it to existing work.

5.1. Evaluation subjects

Our work was motivated by three commercial Android applications, namely, Locale (Locale, 2011), Tasker (Tasker, 2011), and Setting Profiles (Setting Profiles, 2011). However, since we do not have access to their detailed adaptation logics, we had to evaluate our ADAM toolkit using three other MCAs. They were developed by research projects of our two universities.

The three MCAs are Avatar simulator, Stock tracking application, and Obstacle-avoiding car. Their application logics are similar to those of common MCAs by adapting their behavior based on user-defined adaptation rules. The adaptation logics may also evolve with time, allowing state, rule, and context changes at runtime.

(1) **Avatar simulator**. The Avatar simulator is an MCA whose application logics are derived from the scenarios of a well-known movie Avatar (http://www.avatarmovie.com/). It allows a virtual user to enable new skills when necessary knowledge is learned, and perform such skills based on environmental changes.

(2) **Stock tracking application**. The Stock tracking application is the illustrative one we have used throughout the paper for explaining ADAM concepts. Its application logics have been illustrated in Fig. 6. We also use this application in our evaluation for a complete story.

(3) **Obstacle-avoiding car**. The Obstacle-avoiding car is a real system running on an embedded motor-car. The system is built on a Cirrus Logic EDB9302 board with an ARM920T CPU, and the car is equipped with various sensors for perceiving environmental contexts. For example, ultrasonic sensors are used to measure the distances between the car and surrounding obstacles so that the car can avoid these obstacles during its area exploration. Speed sensors are used to estimate the car's moving speed and suggest its obstacle-avoiding strategy based on surrounding obstacle conditions.

The three MCAs each include 5–9 states, 13–14 rules, 18–23 guard conditions, and 13–14 triggering conditions. These data are seemingly not large, but they actually constitute a very large dynamic space. First, each rule can be independently enabled or disabled for each state. Second, an MCA's available contexts can be arbitrary at runtime, being unpredictably affected by internal and external context changes. Therefore, identifying potential defects inside such a dynamic space is non-trivial.

In our following experiments, the Avatar simulator and Stock tracking application were exercised with simulated contexts for better control. The Obstacle-avoiding car was exercised with real contexts from its installed sensors. This treatment considers bother simulation and real execution. Besides, we examined all 61 guard conditions and 40 triggering conditions used by the three MCAs. We found that they have covered all eight formula types used in our condition language in the ADAM model (see Section 3.6). It shows that the selection of the three MCA subjects does not specially bias a particular subset of all formula types.

5.2. Experimental design

We designed three setups for evaluating our ADAM toolkit.

5.2.1. Setup 1

The three MCAs came along with a small set of test cases (less than 10 each). These test cases were originally designed for testing major functions of the MCAs. This follows the current practice of engineering Android and iOS applications: many of such applications have very few test cases. With these limited resources, we first investigated whether these test cases are useful for identifying defects in an MCA. Under this setup, we can only observe failures caused by executing these test cases (i.e., no error detection). To do so, we ran each MCA with its test cases using our ADAM with its error detection ability *disabled*.

5.2.2. Setup 2

For comparison, we also reran these MCAs with their same test cases using our ADAM with its error detection ability *enabled*. Under this setup, we collected all error logs and studied whether these logs help in identifying defects in these MCAs.

5.2.3. Setup 3

We note that all available test cases have been designed based on *expected contexts*. This is for driving an MCA to make an intended adaptation so as to test its pre-specified functions. To be complete, we also investigated the effects of these test cases with *mutated contexts*. This is for simulating the effects of noisy contexts from physical environments. To be fair, we first fixed all defects identified in early experiments (under Setups 1 and 2), and then reran

Table 4
Experimental results under Setup 1 (error detection disabled).

MCA	Observations
Avatar simulator	2 test cases caused application freezing and other test cases had no observable output.
Stock tracking application	All test cases had no observable output.
Obstacle-avoiding car	All test cases had no observable output.

these MCAs with their test cases based on mutated contexts (mutation rate: 20%). This rate is decided according to existing empirical estimation of error rate for common context types like location and RFID contexts (Rao et al., 2006; Xu et al., 2010). Mutation was realized by adding a new random context, randomly deleting an existing context, or replacing an existing context with a random one in its domain. The three cases were evenly split in our experiments. However, we treated the Obstacle-avoiding car differently, because it uses raw contexts captured directly from its physical environments. Mutation is not necessary for this MCA since its contexts already contain natural noises. Under this setup, we investigated whether the three MCAs are vulnerable to unexpected contexts, and whether our ADAM still helps in identifying new defects in these MCAs.

5.3. Experimental results and analyses

Our experiments were conducted on a machine with an Intel® Core™ i5 CPU @3.20 GHz and 3.5 GB RAM. The machine is installed with Microsoft Windows 7 Professional (SP1) and Oracle/Sun JRE 6 (update 25). We give experimental results in Table 4, Table 5, and Table 6 for the three setups, respectively, and explain them below.

5.3.1. Setup 1

Under Setup 1, ADAM ran with its error detection ability disabled. It acted as a common context sensing and adaptation engine as in many existing MCAs. Under this setup, we observed that only for the Avatar simulator, there are two test cases that caused application freezing (i.e., no response to any user input). For other test cases or other MCAs, there is no observable output at all, as shown in Table 4.

Based on this limited information, one can infer that the Avatar simulator may contain some defects, which caused such application freezing. However, there are no extra clues about these defects. One seems to have no effective way to quickly locate such defects, not to mention fixing them.

Besides, one can hardly infer whether the Stock tracking application and Obstacle-avoiding car contain any defect, since these two MCAs have no observable output at all. This is a typical example of common challenges in engineering practical MCAs.

5.3.2. Setup 2

Under Setup 2, ADAM ran with its error detection ability enabled. Then ADAM detected a total of 18 errors for the three MCAs with their same test cases. Based on these detected errors' associated logs, we identified a total of 14 defects for these MCAs, as shown in Table 5.

For example, the Avatar simulator's two application freezing cases now can be analyzed based on their associated error logs. They were caused by two adaptation cycles (i.e., stability errors). When such adaptation cycles occurred, the Avatar simulator transited from one state to another and back infinitely. As a result, the Avatar simulator could not process any new contexts, behaving as no response to any user input. According to the errors' associated logs, two related rules were identified from the Avatar simulator's ADAM model. These two rules' triggering conditions were both satisfied under certain contexts, such that the two rules were triggered in cycle. That is, one rule was triggered to enable another rule, which was immediately triggered to enable the first one, and then it repeated this process. These two faulty rules and their triggering contexts (under which the rules were triggered) were already included in the error logs. Therefore, one can easily identify their responsible defects: developers have forgotten to remove a key context after the first adaptation. This context would cause the Avatar simulator to adapt back to the original state, and then the first adaptation would be triggered again, causing an adaptation cycle. Based on this information, one can easily fix this defect.

Sometimes several errors may relate to one responsible defect only. This implies that these errors are correlated and have been caused by one defect. For example, three reachability and liveness errors in the Avatar simulator all relate to a single defect: a typo in one rule's triggering condition caused a keyword "calm" written as "clam". As all occurrences of "calm" are mistakenly written as "clam" in the Avatar simulator, this defect cannot be identified by static analysis. However, at runtime this defect manifested itself by making the Avatar simulator enter a state that cannot be left, causing its leaving rule never used and target state never reached. The error logs provide the information about these two states as well as the adaptation rule connecting them. Based on this information, one can easily identify this defect and fix it.

We note that very occasionally a few reported errors may be *false positives*. However, this does not mean that these errors are wrong. Instead, they are correct, but these errors do not relate to any MCA defect. For example, ADAM reported 2 *benign* stability errors for the Stock tracking application (see the number in brackets in Table 5). They are considered as benign, because they were caused by developers' intended adaptations. These adaptations quickly went across several states only for efficiency purposes. Due to their similarity to stability errors, which ignore user inputs, they were detected as well. However, the occurrences of such false positives are infrequent (2 against all 18 errors in Setup 2 and even less in Setup 3 as shown later), and they are easily distinguishable from others.

5.3.3. Setup 3

Under Setup 2, we identified a total of 14 defects. We fixed all these 14 defects, and then proceeded to Setup 3. We reran the three MCAs with aforementioned mutated contexts (the Obstacle-avoiding car still took raw contexts as explained earlier). We detected a total of 75 new errors (4 are benign). Based on their associated error logs, we identified a total of 58 new defects, as shown in Table 6. These large numbers show the vulnerability of the three MCAs when exposed to unexpected contexts. This validates concrete challenges existing in engineering MCAs for real-world scenarios.

We analyzed these errors and responsible defects. We found that most defects are because these MCAs either fail to consider situations caused by unexpected contexts, or even if such situations have been considered (by designing guard conditions), these MCAs just

Table 5
Experimental results under Setup 2 (error detection enabled).

MCA	Pred. errors	Stab. errors	Reac. errors	Cons. errors	Defects
Avatar simulator	0	2	3	1	4
Stock tracking application	1	3 (2)	0	3	5
Obstacle-avoiding car	4	1	0	0	5

Table 6
Experimental results under Setup 3 (mutated contexts).

MCA	Pred. errors	Stab. errors	Reac. errors	Cons. errors	Defects
Avatar simulator	0	5	0	17	19
Stock tracking application	1	12 (4)	0	15	19
Obstacle-avoiding car	1	7	0	17	20

Table 7
Partitioned errors and defect-fixing suggestions.

MCA	Avoidable errors	Errors to be addressed and their associated defect-fixing suggestions
Avatar simulator	3 stab. errors, 17 cons. errors	2 stab. errors: specify more precise guard conditions for 2 rules.
Stock tracking application	5 stab. errors, 15 cons. errors	1 pred. error: set a proper priority for a rule; 3 stab. errors: specify more precise guard/triggering conditions for 3 rules.
Obstacle-avoiding car	1 pred. error, 17 cons. errors	7 stab. errors: specify more precise triggering conditions for 3 rules.

leave the situations there without addressing them. Although it is possible to enhance each MCA with an individual handling mechanism for addressing these situations, this approach can be very time-consuming and is most likely not reusable for other MCAs.

To address this problem, we investigated all these detected errors. We found that consistency errors are typically coupled with other types of errors. For example, among 5 stability errors in the Avatar simulator, 3 of them (60.0%) have been caused by 7 consistency errors (totally 17 consistency errors). Similarly, 5 of 8 stability errors (62.5%, 4 benign ones already removed) have been caused by 8 consistency errors (totally 15 consistency errors) in the Stock tracking application. This strongly suggests that if one can fix these consistency errors at runtime (e.g., by resolving context inconsistency (Xu and Cheung, 2005; Xu et al., 2011)), many other errors can have been avoided. The workload of fixing their associated MCA defects can be therefore greatly reduced.

Based on this idea, Table 7 partitions all 71 detected errors (4 benign ones removed) into two groups: *errors that can be avoided* and *errors that have to be addressed*. We observe that 58 errors (81.7%) can be avoided if context inconsistency can be resolved at runtime (i.e., guaranteeing no consistency error). The other 13 errors (18.3%) have to be addressed, and they are associated with 9 MCA defects. In other words, one needs to focus on 9 defects instead of the aforementioned 58 defects. This is a great workload reduction (15.5% only).

Fixing these 9 defects requires either making relevant guard/triggering conditions more precise or adjusting relevant rules' priorities. As ADAM's generated error logs have already provided information about these conditions, rules, and the contexts under which such errors have occurred, fixing these defects is not difficult. Still, automatically fixing runtime errors (e.g., fixing consistency errors by inconsistency resolution to avoid other types of errors) and even defects is an interesting research issue, and we keep it as our future work.

5.4. Additional comparisons and measurements

We owe ADAM's error detection and defect identification ability to its two distinct characteristics, as compared with existing work (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Ranganathan and Campbell, 2003; Sama et al., 2010a). First, ADAM has a set of built-in assertion checks for detecting four types of common errors, which have been recognized by our archival study to be able to well cover most sensing and adaptation MCA defects. Second, ADAM has a more expressive model for specifying the adaptation behavior of an MCA and the impact of its internal context updates, which have been oversimplified by existing work. The usefulness of the first characteristic (i.e., our assertion checks) has been illustrated by three groups of experiments in Section 5.3 (more discovered errors and defects). We now explain a bit more about the usefulness of the second characteristic (i.e., our ADAM model) in helping discover more errors and defects in MCAs.

As introduced earlier, our ADAM model is more expressive than existing work (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Ranganathan and Campbell, 2003; Sama et al., 2010a). It can adequately specify the adaptation behavior of an MCA by supporting its first-order logic based condition specification and modeling its timing constraints and internal context updates. We note that such features are not fully supported by existing work, but they are required for precisely specifying an MCA's adaptation behavior. For comparison purposes, we created a model *M'* to represent existing work (which does not support our aforementioned features). We replaced our ADAM model with this model *M*' to study whether this treatment would affect our error detection and defect identification results.

We still used the earlier three MCA subjects, and specified their adaptation behavior using this new model M'. We note that by doing so, their application logics can only be partially specified, and therefore only part of their application logics can be checked for errors at runtime. We used this model M' (less expressive than our ADAM model), but enhanced it with our error detection ability. We then conducted experiments again under Setup 2 and Setup 3 (Setup 1 was omitted as it did not use ADAM's error detection ability). We collected experimental results and compared them in Table 8.

We observe that only 6 errors (33.3% of all 18 errors) were detected for the three MCAs under Setup 2, and 39 errors (52.0% of all 75 errors) were detected under Setup 3. It shows that the error detection ability was greatly reduced when our ADAM model was not used. In this case, our assertion checks still generated some error logs for defect analysis (although less). Based on them, we identified 6 defects (42.9% of all 14 defects) under Setup 2, and 32 defects (55.2% of all 58 defects) under Setup 3. We can observe that the usefulness of the generated error logs in helping identify MCA defects was also affected when our ADAM model was not used. This comparison validates the importance of our ADAM model's expressive power to assisting the error detection and defect identification ability of our ADAM approach.

Besides, we also measured ADAM's runtime overhead for its error detection. We measured the ratio between the execution time used by ADAM with its error detection enabled and that with this ability disabled. The ratio provides the magnitude of time incurred by ADAM's error detection. We found that the measured ratio falls into the range of 107.5–110.7%. It suggests quite small overhead (7.5–10.7%) incurred by ADAM for better detecting errors and identifying defects in MCAs.

Regarding the space overhead, it is mainly caused by the error logs generated for defect analysis. However, since these error logs are only generated when errors are detected, their sizes are very small. In our experiments, the caused space overhead is almost

Table 8
Error detection and defect identification comparisons.

MCA	Setup 2		Setup 3	
	Detected errors	Identified defects	Detected errors	Identified defects
All three MCAs	6 (33.3%)	6 (42.9%)	39 (52.0%)	32 (55.2%)

negligible (55–184 bytes for each MCA run). This additionally justifies the practical usefulness of our ADAM in detecting errors and identifying responsible defects for MCAs.

6. Related work

In this section, we discuss the related work in recent years. It covers self-adaptive software system, application situation modeling, error detection, property checking, and software testing and debugging.

6.1. Self-adaptive software system

Developing reliable self-adaptive software systems has been always challenging. Today, most successful self-adaptive software systems are developed by following the well-known "collect-analyze-decide-act" autonomic control loop (Dobson et al., 2006). This gives practically useful guidelines for developing such complex software systems in a proved way. Still, a number of research challenges exist, covering four aspects, namely, modeling dimensions, requirements, engineering, and assurances (Andersson et al., 2009; Cheng et al., 2009). These four aspects are closely connected. For example, a goal designed at the modeling phase may not be adequately assured at runtime, thus leading to violation to its requirement. This can cause solid challenges to engineering such software systems. Some pieces of existing work focus on the reliability support for self-adaptive software systems at an architecture level. For example, Kramer and Magee (2007) proposed a three-layered architecture model for realizing the self-management in self-adaptive software systems. Garlan et al. (2003) used a notion of *style* to enable architecture-based self-repairing for self-adaptive software systems. Our focus in this paper is mainly at a modeling and implementation level, complementing these pieces of existing work.

6.2. Application situation modeling

As a typical type of self-adaptive software systems, developing reliable context-aware applications depends much on a precise modeling of the adaptive behavior of these applications. Situation has been well recognized as a key modeling concept for such applications. A situation refers to a special condition interesting to an application and deserving this application's response at runtime. Detecting a pre-specified situation would trigger an application to adapt itself at runtime in order to deliver better services.

Situations are typically specified by *pattern-based* or *logic-based* languages. For example, Adi and Etzion (2004) and Agrawal et al. (2008) used event patterns to specify situations. Julien and Roman (2006), Mamei and Zamonelli (2009), and Murphy et al. (2006) specified situation patterns based on context tuples. Context-aware applications may also be interested in certain relationships among situations. To describe such relationships, propositional logics have been commonly used (Capra et al., 2003; Sama et al., 2010a). Sometimes, first-order or temporal logics may also be used if more complex situation relationships or scenarios need to be described (Mok et al., 2004; Ranganathan and Campbell, 2003; Zhang and Cheng, 2006a,b).

These pieces of work mainly focus on how to specify interesting situations and detect them effectively at runtime. Such situations are modeled as rules in an application, and typically assumed to be static at runtime. Some work (Ranganathan and Campbell, 2003; Sama et al., 2010a) also considered the possibility of evolving an application's logic (e.g., rules) at runtime. This is achieved by allowing the set of active rules to be changeable across different states. This idea echoes our ADAM model in the paper, but we further relax the relationship between a state and the set of rules that can be enabled for this state (i.e., purely dynamic at runtime). Besides, we also model how an application's contexts evolve (i.e., internally updatable) due to its triggered adaptations. These features allow one to be able to precisely model the adaptive behavior of an application. As shown in Section 5, this helps detect more errors and identify hidden defects in applications.

6.3. Error detection

Context-aware applications are prone to errors at runtime. The existing work on detecting errors for context-aware applications can be roughly divided into two groups: *context-level* work and *application-level* work.

The first group (context-level work) mainly focuses on isolating noisy contexts from impacting applications that may be vulnerable to such noises. For example, Rao et al. (2006) proposed detecting and removing noises in RFID contexts by matching these contexts against anomaly patterns. This work has focused on a special type of context, i.e., RFID context. There is also a large body of work focused on detecting noises in general types of contexts. Such noises are modeled as *context inconsistency*, behaving as violation to predefined consistency constraints for certain application domains. These constraints can include property constraints (Huang et al., 2009), heuristic rules (Xu et al., 2008), inconsistency triggers (Xu and Cheung, 2005), and integrity constraints (Khoussainova et al., 2006). When any constraint is violated, we say that context inconsistency is detected. This implies that noisy contexts do exist and they should be prevented from being accessed by context-aware applications. Otherwise, these applications may behave abnormally at runtime. This group of context-level work has tried to avoid application errors by detecting and isolating noisy contexts. However, it does not guarantee 100% detection of all noisy contexts, as consistency constraints can specify only necessary conditions for the correctness of application contexts.

The second group (application-level work) instead focuses directly on detecting errors for context-aware applications. Most such work focuses on detecting non-determinism errors (Capra et al., 2003; Chomicki et al., 2003; Insuk et al., 2005; Ranganathan and Campbell, 2003), which are similar to predictability errors studied in this paper. Very few pieces of work studied consistency errors (Kulkarni and Tripathi, 2010) or stability errors (Sama et al., 2010a). As a comparison, our ADAM approach aims to detect a wide range of errors, including all these studied error types. In addition, we consider how these errors manifest under complex scenarios that should be specified with the modeling support of quantifications, timing constraints, and internal context updates. Our evaluation justifies the necessity of such an expressive ADAM model for detecting a wide range of errors. Finally, our ADAM

approach also helps analyze such errors' responsible defects to assist defect-fixing tasks.

6.4. Property checking

Context-aware applications belong to the category of self-adaptive software in some sense. The latter often concerns properties in the adaptation and their runtime verification. This is similar to our ADAM approach that sets up assertion checks for runtime evaluation in order to detect application errors. Gorbovitski et al. (2008) proposed a framework for systematically specifying properties and checking their violations at runtime. The work focuses on techniques for incremental evaluation of such properties. This shares a similar goal with our previous work on incremental context consistency checking (Xu et al., 2006, 2010). Both pieces of work have targeted at making runtime property checking more efficient, but not relating property violations to application errors or defects.

Conditions often need to be expressed in property specifications. Our ADAM approach adopts a first-order logic based language for specifying them. The language uses a temporal construct to restrict the scope of contexts under consideration, and allows user-defined predicates to express domain-specific functions. Still, there is existing work that supports more expressive languages. For example, Basin et al. (2008) proposed a dynamic monitor for verifying at runtime those properties specified by a metric first-order temporal logic. The work provides a flexible modeling support for "past and future" temporal operators, but its modeling language is highly theoretical and does not seem to be directly applicable to practical MCAs. Besides, Grunske (2008) presented a set of specification patterns for defining probabilistic temporal properties for practical applications. Such patterns are useful for describing probabilistic relationships among contexts in complex application scenarios. However, considering that our ADAM approach is already adequate for modeling MCAs discussed in this paper, we leave as future work: (1) investigating whether it is necessary to further improve our ADAM model's expressive power and (2) investigating whether such expressiveness improvement would bring unexpected impact to our error detection and defect identification objectives.

The adaptation semantics based on which our assertion checks are conducted is one-point adaptation. Here we have followed what has been proposed by existing work ((Zhang and Cheng, 2006a,b). Although straightforward, the one-point adaptation enables us to focus on a clearly formulated adaptation process such that different properties can be defined at different phases of an adaptation process. Such properties can then be checked using our ADAM approach's assertion checks to identify MCA defects. Zhang and Cheng (2006a,b) also proposed other two types of adaptation semantics, which are more complex and have been discussed only in the scope of static analysis. We plan to consider in future how to extend our ADAM model to these two other types of adaptation semantics, and study how to detect errors and identify defects under such new adaptation semantics.

6.5. Software testing and debugging

Our work has restricted its scope to model-based context-aware applications (i.e., MCAs) in this paper. Such applications formulate their adaptation logics as rules, and adaptation errors may be triggered at runtime if these rules contain defects. Sama et al. (2010b) focused on the same type of applications by analyzing common MCA defects from an architecture's point of view. They tried to assign such defects to different layers in a context-aware computing architecture. Their discussed defects are statically identifiable (Sama et al., 2010a), while this paper focuses on more challenging defects that may manifest into errors only at runtime.

There are some pieces of work on testing context-aware applications, but they mainly focus on code-based programs, different from our scope in this paper. For example, Lu et al. (2006, 2008) and Wang et al. (2007) studied techniques for increasing the testing coverage for context-aware applications. They based their work on certain interactions unique to such applications, i.e., those interactions between application logics and changeable contexts from environments. Testing application logics has been extensively studied by traditional testing techniques, while changeable contexts are usually beyond the control of an application and therefore have been largely omitted. Although our work focuses on different types of context-aware applications, our insights are similar. Our ADAM model explicitly models the impact of adaptation on the evolution of an application's logics as well as internal context changes brought about by such adaptation.

Our ADAM approach also supports defect analysis based on generated error logs. This relates to existing work on path-based fault correlation (Le and Soffa, 2010). The work tries to track the matching of error signatures back to the sites in programs for these error occurrences. The concept of error signature is similar to that of error criterion or type in our work. However, since our error logs include the contexts under which certain errors have occurred, we are able to pinpoint faulty rules and suggest possible defect-fixing options, as shown in our evaluation. In addition, we also investigated common error types through an archival study. They were originally not clear for context-aware applications.

7. Conclusion

In this paper, we conducted an archival study on model-based context-aware applications. We investigated the development history of three commercial context-aware applications, and confirmed the solid challenges existing in developing these practical applications. We reported our findings of four common error types that have covered most defects collected in our study. Based on these results, we presented our ADAM approach to addressing the error detection and defect identification issue for context-aware applications. Our evaluation confirmed the effectiveness and practical usefulness of our ADAM approach through simulation-based and real-life experiments.

Our ADAM approach follows an intuitive computing paradigm, which has been widely adopted in many context-aware middleware infrastructures. This paradigm takes an event-driven style to accept context changes from environments, and adapts an application based on user-defined rules. Users are supposed to be able to specify such rules for their applications. This practice is similar to what they have done in order to use the aforementioned three Android applications (Locale, Tasker, and Setting Profiles). Therefore, there is no extra overhead for users. Our ADAM toolkit can run these user-specified rules, automatically report errors, and help analyze for potential defects in the rule design. Users are also allowed to use states to group different sets of rules, and make their mapping relationships totally dynamic at runtime. Although such treatment is optional, this can increase the flexibility in designing an application's logics. Of course, such flexibility may cause an application prone to defects. However, our ADAM toolkit exactly fits in this need for detecting errors and helping identify these defects.

There might be two issues about using our ADAM toolkit in practice. The first is about guard conditions. Guard conditions can be obtained from application specifications that express the expectations on what environments this application is supposed to run in. These conditions can also be derived from context consistency constraints. Such constraints have been studied in the existing work on inconsistency detection and resolution for context-aware computing (Huang et al., 2009; Xu and Cheung,

2005; Xu et al., 2010). The second issue is about test cases. Our ADAM toolkit uses test inputs only, and therefore a test oracle is essentially unnecessary. As shown in our evaluation, the ADAM toolkit used mutation techniques to generate random contexts as test inputs. Therefore, our ADAM toolkit does not rely much on test cases. It can detect runtime errors and generate associated error logs based on randomly mutated test inputs.

Currently, our evaluation used only research projects as experimental subjects. We plan to extend the use of our ADAM approach to other large-scale applications under realistic environments to further validate its practical usefulness. Meanwhile, a user study on how to use our ADAM toolkit to support developing reliable context-aware applications would be necessary. On the other hand, some related research issues may be interesting to our communities on self-adaptive software systems. For example, we are currently investigating how to automatically fix detected application errors by repairing context inconsistency at runtime. This works for self-healing or self-repairing systems at a model and implementation level, and can also complement the existing work on self-managed dependable systems working at an architecture level (Garlan et al., 2003; Kramer and Magee, 2007).

Acknowledgements

This research was partially funded by National Science Foundation (grants 61100038, 61021062) and 863 program (2011AA010103) of China, and by Research Grants Council (grant 612210) of Hong Kong. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

References

Locale, 2011. http://www.twofortyfouram.com/.
Setting Profiles, 2011. http://www.probeez.com/.
Tasker, 2011. http://tasker.dinglisch.net/.
Adi, A., Etzion, O., 2004. Amit: the situation manager. VLDB Journal 13 (2), 177–203.
Agrawal, J., Dial, Y., Gyllstrom, D., Immerman, N., 2008. Efficient pattern matching over event streams. In: Proceedings of the Joint ACM SIGMOD/PODS Conference, Vancouver, Canada, June 2008, pp. 147–160.
Andersson, J., Lemos, R.D., Malek, S., Weyns, D., 2009. Modeling dimensions of self-adaptive software systems. Software Engineering for Self-Adaptive Systems, LNCS 5525, 27–47.
Basin, D., Klaedtke, F., Muller, S., Pfitzmann, B., 2008. Runtime monitoring of metric first-order temporal properties. In: Proceedings of the IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, December 2008, pp. 49–60.
Bu, Y., Gu, T., Tao, X., Li, J., Chen, S., Lu, J., 2006. Managing quality of context in pervasive computing. In: Proceedings of the 6th International Conference on Quality Software, Beijing, China, October 2006, pp. 193–200.
Capra, L., Emmerich, W., Mascolo, C., 2003. CARISMA: context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering 29 (October (10)), 929–945.
Cheng, B.H.C., Lemos, R.D., Giese, H., Inverardi, P., Magee, J., 2009. Software engineering for self-adaptive systems: a research roadmap. Software Engineering for Self-Adaptive Systems, LNCS 5525, 1–26.
Chomicki, J., Lobo, J., Naqvi, S., 2003. Conflict resolution using logic programming. IEEE Transactions on Knowledge and Data Engineering 15 (February (1)), 244–249.
Dobson, S., Denazis, S., Fernandez, A., Gaiti, D., Gelenbe, E., Massacci, F., 2006. A survey of autonomic communications. ACM Transactions Autonomous Adaptive Systems 1 (December (2)), 223–259.
Garlan, D., Cheng, S.W., Schmerl, B., 2003. Increasing system dependability through architecture-based self-repair. Architecting dependable systems. LNCS 2677, 61–89.
Gorbovitski, M., Rothamel, T., Liu, Y.A., Stoller, S.D., 2008. Efficient runtime invariant checking: a framework and case study. In: Proceedings of the 6th International Workshop on Dynamic Analysis, Seattle, Washington, USA, July 2008, pp. 43–49.
Grunske, L., 2008. Specification patterns for probabilistic quality properties. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 2008, pp. 31–40.
Huang, Y., Ma, X., Cao, J., Tao, X., Lu, J., 2009. Concurrent event detection for asynchronous consistency checking of pervasive context. In: Proceedings of the 7th Annual IEEE International Conference on Pervasive Computing and Communications, Galveston, TX, USA, March 2009, pp. 131–139.
Insuk, P., Lee, D., Andhyun, S.J., 2005. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In: Proceedings of the 29th Annual International Computer Software and Applications Conference, Edinburgh, UK, July 2005, pp. 359–364.
Julien, C., Roman, G.C., 2006. EgoSpaces: facilitating rapid development of context-aware mobile applications. IEEE Transactions on Software Engineering 32 (May (5)), 281–298.
Khoussainova, N., Balazinska, M., Suciu, D., 2006. Towards correcting input data errors probabilistically using integrity constraints. In: Proceedings of the 5th International ACM Workshop on Data Engineering for Wireless and Mobile Access, Chicago, IL, USA, June 2006, Article 2.
Kramer, J., Magee, J., 2007. Self-managed systems: an architectural challenge. In: Proceedings of Future of Software Engineering, International Conference on Software Engineering, Minneapolis, MN, USA, May 2007, pp. 259–268.
Kulkarni, D., Tripathi, A., 2010. A framework for programming robust context-aware applications. IEEE Transactions on Software Engineering 36 (March/April (2)), 184–197.
Le, W., Soffa, M.L., 2010. Path-based fault correlations. In: Proceedings of the 18th International ACM SIGSOFT Symposium of Foundations of Software Engineering, Santa Fe, NM, USA, November 2010, pp. 307–316.
Lu, H., Chan, W., Tse, T., 2006. Testing context-aware middleware-centric programs: a data flow approach and a RFID-based experimentation. In: Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Portland, OR, USA, November 2006, pp. 242–252.
Lu, H., Chan, W., Tse, T., 2008. Testing pervasive software in the presence of context inconsistency resolution services. In: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 2008, pp. 61–70.
Mamei, M., Zamonelli, F., 2009. Programming pervasive and mobile computing applications: the TOTA approach. ACM Transactions on Software Engineering and Methodology 18 (July (4)), Article 15.
Mok, A.K., Konana, P., Liu, G., Lee, C.G., Woo, H., 2004. Specifying timing constraints and composite events: an application in the design of electronic brokerages. IEEE Transactions on Software Engineering 30 (December (12)), 841–858.
Murphy, A.L., Picco, G.P., Roman, G.C., 2006. LIME: a coordination model and middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering and Methodology 15 (July (3)), 279–328.
Nentwich, C., Capra, L., Emmerrich, W., Finkelstein, A., 2002. Xlinkit: a consistency checking and smart link generation service. ACM Transactions on Internet Technology 2 (May (2)), 151–185.
Ranganathan, A., Campbell, R.H., 2003. An infrastructure for context-awareness based on first order logic. Personal and Ubiquitous Computing 7, 353–364.
Rao, J., Doraiswamy, S., Thakkar, H., Colby, L.S., 2006. A deferred cleansing method for RFID data analytics. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 2006, pp. 175–186.
Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z., 2010a. Context-aware adaptive applications: fault patterns and their automated identification. IEEE Transactions on Software Engineering 36 (September/October (5)), 644–661.
Sama, M., Rosenblum, D.S., Wang, Z., Elbaum, S., 2010b. Multi-layer faults in the architectures of mobile, context-aware adaptive applications. The Journal of Systems and Software 83, 906–914.
Wang, Z., Elbaum, S., Rosenblum, D.S., 2007. Automated generation of context-aware tests. In: Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, May 2007, pp. 406–415.
Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H., 2009. Supporting automatic model inconsistency fixing. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering, Amsterdam, The Netherlands, August 2009, pp. 315–324.
Xu, C., Cheung, S.C., 2005. Inconsistency detection and resolution for context-aware middleware support. In: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, September 2005, pp. 336–345.
Xu, C., Cheung, S.C., Chan, W.K., 2006. Incremental consistency checking for pervasive context. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, May 2006, pp. 292–301.
Xu, C., Cheung, S.C., Chan, W.K., Ye, C., 2008. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In: Proceedings of the 28th International Conference on Distributed Computing Systems, Beijing, China, June 2008, pp. 713–721.
Xu, C., Cheung, S.C., Chan, W.K., Ye, C., 2010. Partial constraint checking for context consistency in pervasive computing. ACM Transactions on Software Engineering and Methodology 19 (January (3)), 1–61, Article 9.
Xu, C., Ma, X., Cao, C., Lu, J., 2011. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In: Proceedings of the 8th ICST International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2011), LNICST 104, Copenhagen, Denmark, Dec 2011, pp. 285–297.
Zhang, J., Cheng, B.H.C., 2006a. Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, May 2006, pp. 371–380.
Zhang, J., Cheng, B.H.C., 2006b. Using temporal logic to specify adaptive program semantics. The Journal of Systems and Software 79, 1361–1369.

Chang Xu is an associate professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology of Nanjing University. He received his Ph.D. degree in computer science and engineering from

the Hong Kong University of Science and Technology. His research interests include software engineering, software testing and analysis, and pervasive computing.

S.C. Cheung is a professor with the Department of Computer Science and Engineering of the Hong Kong University of Science and Technology. He received his Ph.D. degree in computing from the Imperial College London. His research interests include various software engineering issues related to program analysis, testing and debugging, services computing, cyber-physical systems, internet of things, and mining software repository. He was an editorial board member of the IEEE Transactions on Software Engineering (TSE, 2006–2009). He is currently serving on the editorial board of the Information and Software Technology (IST), Journal of Computer Science and Technology (JCST), and the International Journal of RF Technologies: Research and Applications. He is an extended member of the executive committee of the ACM SIGSOFT.

Xiaoxing Ma is a professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology of Nanjing University. He received his Ph.D. degree in computer science from Nanjing University. His research interests include middleware systems, software architecture, and self-adaptive software systems.

Chun Cao is an assistant professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology of Nanjing University. He received his Ph.D. degree in computer science from Nanjing University. His research interests include middleware systems, software architecture, and self-adaptive software systems.

Jian Lu is a professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology of Nanjing University. He received his Ph.D. degree in computer science from Nanjing University. He has a wide range of research interests in software engineering, including formal methods, software methodology, software agents, and automated software engineering.