Sequential Event Pattern Based Context-aware Adaptation

Chushu Gao[1,3], Jun Wei[1]
[1]Technology Centre of Software Engineering
Institute of Software, Chinese Academy of Sciences
Beijing, China
{gaochushu,wj}@otcaix.iscas.ac.cn

Chang Xu[2,3]
[2]State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, China
changxu@cse.ust.hk

S.C. Cheung[3]
[3]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Kowloon, Hong Kong, China
scc@cse.ust.hk

ABSTRACT
Recent pervasive systems are designed to be context-aware so that they are able to adapt to continual changes of their environments. Rule-based adaptation, which is commonly adopted by these applications, introduces new challenges in software design and verification. Recent research results have identified some faulty or unwanted adaptations caused by factors such as asynchronous context updating, and missing or faulty context reading. In addition, adaptation rules based on simple event models and propositional logic are not expressive enough to address these factors and to satisfy users' expectation in the design. We tackle these challenges at design stage by introducing sequential event patterns in adaptation rules to eliminate faulty and unwanted adaptations with features provided in the event pattern query language. We illustrate our approach using the recent published examples of adaptive applications, and show that it is promising on designing more reliable context-aware adaptive applications.

Categories and Subject Descriptors
D.1.m [**Programming Techniques**]: Miscellaneous--rule-based programming; F.1.1[**Models of Computation**]: Automata

General Terms
Design, Reliability, Languages.

Keywords
Context-aware adaptation, pervasive computing, sequential event pattern

1. INTRODUCTION
Advanced built-in sensor technologies enable mobile and pervasive applications to adapt their configurations and behaviors to continual changing context values, such as variations of GPS readings, Bluetooth connections, and battery power. These applications are commonly referred to as context-aware adaptive applications (CAAAs) [11].

CAAAs are generally built on top of context-aware middleware (CAM) that facilitates collection and manipulation of contexts [2][3][4][13]. Although having different focuses on supporting context-awareness, most of context-aware middleware shares two well separated components in their conceptual model as depicted in Figure 1: (1) a context manager to collect, manipulate and feed context information required by the application, and (2) an adaptation manager to evaluate the adaptation rule based on the context provided by context manager and decide how the application should react to the context changing.

The developers of CAAAs define the adaptive behavior by specifying adaptation rules. Existing CAAAs and adaptation rules in recent publications suffer from some common forms of faults such as *non-determinism* (i.e., when a context value is updating, more than one rule can be triggered) and *instability* (i.e., the application's state depends on the length of time a context value holds) [11][9]. These faults, although detectable statically in design time, are not easy to eliminate.

Even carefully designed deterministic and stable set of adaptation rules may fail to satisfy the users' expectation due to the *context hazard* [11]. Context hazard is a type of adaptation faults caused by the asynchronous nature of context value updating and different refreshing rate of context resources. Context hazard may rise unwanted adaptations and lead CAAAs to unexpected states. For instance, when start driving, delayed Bluetooth connection may cause the *PhoneAdapter* application [11] failed to detect the hand-free system in car, and adapting the phone's profile to *Jogging* instead of *Driving* if GPS senses a user's speed of more than 5 miles per hour. Even the Bluetooth connection re-detected afterward, the application is not easy to recover to the correct state when GPS continually detects the speed of more than 5 miles per hour. There is no direct adaptation rule available to change phone's profile from *Jogging* to *Driving* since it is not a very reasonable scenario in real life. Intuitively, context hazards can be eliminated by introducing fixed delays to ensure all related context values to be updated before adaptation is taken. However, fixed delay is not always computable if context refresh rates are unknown. Moreover, delays may degrade applications' performance for those hazard-free adaptations.

It is observed that *instability* adaptation faults and *context hazard* in designing CAAAs and adaptation rules are significantly related to timing issues of context value updating. From the event-driven perspective, occurrence of them essentially depends on the order

The Second Asia-Pacific Symposium on Internetware(Internetware'10), November 3–4, 2010, Suzhou, China.

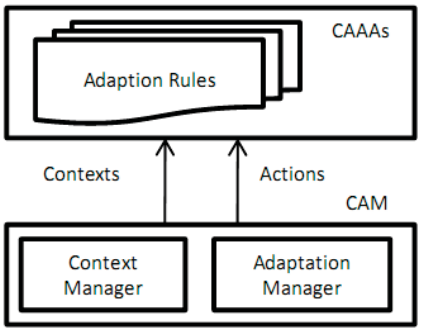

Figure1. Conceptual model of CAAAs.

of context updating events. Existing adaptation rule specification approach has its expressive limitation to describe this information.

In this paper, we present a novel approach to design adaptation rules with sequential event pattern query language to address this limitation. In addition, we illustrate that sequential event pattern query language can be used to design more reliable adaptation rules.

The remainder of the paper is organized as follows. Section 2 uses a motivating example to analyze the limitation of existing approach specifying adaptation rule. Section 3 introduces background on sequential event pattern query. Section 4 gives technical details in designing adaptation rule based on sequential event pattern queries and analyzes what and how instability faults and context hazard can be eliminated. Section 5 reviews the related work, and finally Section 6 concludes the paper with a summary of our contribution and plans for future work.

2. MOTIVATIONS

In this section, we use a simple but typical pervasive application, Self-controlling Mini-car, to illustrate our problem. The motivating example exposes to the kinds of adaptation faults found in other CAAAs with existing modeling approaches.

2.1 Self-Controlling Mini-Car Application

Self-Controlling Mini-Car is a simplified version of a recently implemented real-life mini-car application. The mini-car is equipped with eight ultrasonic sensors, two programmable wheels and powered with batteries. The raw sensor reading of various ultrasonic sensors and moving direction can be used to derive situations such as how many entities are in the car's detectable area and how far entities are away from the car. The precise refresh rate of such derived contexts is not easy to know since its computation time of may vary. For convenience, we directly use the derived context to model the adaptation behavior of the mini-car application.

The mini-car application aims to guide the mini-car exploring an unknown map area till it runs out of battery or receives power-off signal. In the meanwhile, the car should avoid colliding with various obstacles in the area. Self-controlling has two folds of meaning. On one side, the mini-car can control its movement using different predefined exploring algorithms. On the other side, the mini-car can switch between different running modes governed by adaptation rules. The latter is the subject of our study.

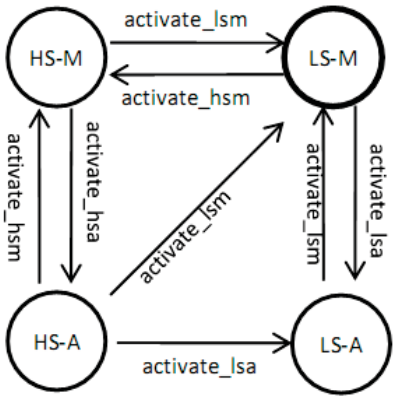

Figure2. Self-controlling mini-car application.

The mini-car supports two categories of adaptive behavior: (1) *speed adjustment* switches the mini-car's speed mode between *high-speed mode* (HS) and *low-speed mode* (LS); (2) *task adjustment* switches the car's task mode between *normal moving* (M) and *obstacle avoidance* (A). We use a similar finite-state machine approach, Adaptation Finite-State Machine (A-FSM) in [11] to describe the application. The combination states of the four modes and transitions between them are shown in Figure 2. Self-controlling mini-car application's four modes are designed to accomplish the exploring task with different moving strategies according to the sensed environment around the car:

(1) *Low-speed Moving (LS-M)*: employs a normal exploring strategy when surrounding obstacles are rare and with a safe distance away from the car, and uses a low speed;

(2) *High-speed Moving (HS-M)*: employs an aggressive exploring strategy when there is no obstacle detected, and uses a high speed;

(3) *High-speed Avoidance (HS-A)*: employs an optimistic obstacle avoiding strategy assuming obstacles are rare but in a closer distance than safe and switches the car from normal moving algorithm to obstacle avoiding algorithm while maintaining a high speed;

(4) *Low-speed Avoidance (LS-A)*: employs a conservative obstacle avoiding strategy assuming obstacles are heavily found around the car or in a critical dangerous distance to the car, and switches to obstacle avoiding algorithm while decreases the speed.

2.2 Problem in Existing Adaptation Rule Modeling

The adaptive behavior of a rule-based CAAA can be modeled by a finite-state machine as in [11]. In our mini-car application, we have experienced the existing A-FSM approach using a simplified model without considering priority of the rule.

2.2.1 Application Modeling

The *adaptation rules* are defined by a set $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{C} \times \mathcal{S} \times \mathcal{A}$. $\mathcal{S}$ is a set of possible states of a CAAA (e.g., High-speed Moving mode in the mini-car application). $\mathcal{A}$ is a set of actions to change the behavior of CAAA when transition to a new state *S* happened (e.g., "increase car speed" or "switch moving algorithm"). $\mathcal{C}$ is a set of situations definable over a set of context variable *C* (e.g., '*Power.level()<70*'). In A-FSM, *C* is a set of propositional context variables and $\mathcal{C}$ is a set of logical predicates defined over *C* using logic operators "and", "or", and "not".

Table 1. Adaptation rules of self-controlling mini-car

Rule Name	Current States	New States	Full predicate	Abbreviation
activate_hsm	LS-M, HS-A	HS-M	!Power.level()<70 and Sensor.count()=0	$!A_{power}$ and A_{sesnor}
activate_lsm	HS-M, HS-A, LS-A	LS-M	Power.level()<70 or (Sensor.count()<3 and !Sensor.dist()<50)	A_{power} or (B_{sensor} and ! D_{sensor})
activate_hsa	HS-M	HS-A	!Power.level () <70 and ((!Sensor.count()<3 and Sensor.dist() <50) or (Sensor.count()<3 and dist()<30))	$!A_{power}$ and (($!B_{sesnor}$ and D_{sesnor}) or (B_{sesnor} and C_{sesnor})
activate_lsa	LS-M, HS-A	LS-A	!Sensor.count()<3 and Sensor.dist()<30	$!B_{sensor}$ and C_{sesnor}

A rule r in $\mathcal{R}$ is a tuple of (s, c, s', a), where s and $s' \in \mathcal{S}$, $c \in \mathcal{C}$, and $a \in \mathcal{A}$. The semantic of this rule is to say if the situation c is satisfied and current state is s, the CAAA will adapt to a new state s' and take the action a once enter the new state s'.

Finally, the finite-state machine is defined as $\mathcal{M} = (\mathcal{S}, \delta, S_0, \mathcal{S}_f)$, where the transition relation is defined as $\delta = \{(s,r,s') | \exists r = (s,c,s',a) \in \mathcal{R}\}$.

In the mini-car application, two classes of contexts are used. The contexts reasoned from raw ultrasonic sensor readings include the number of detected entities and the distance to the nearest detected entities. The context sensed by battery sensor indicates the power level of the mini-car. Mini-car application adapts between four states governed by four different rules defined over five propositional context variables listed in Table 1. Although it is very simple, mini-car application suffers from the same sort of adaptation faults and context hazards as found in *PhoneAdapter*.

2.2.2 Instability Fault and Context Hazard

The A-FSM assumes that every entrance to some state and every value updating of context variable will trigger the CAAA to re-evaluate all the active rules in that state. In former case, even no new value is assigned to any of the context variables, the CAAA may trigger a rule r in the new state s' if the situation c is satisfied by current values of context variables, and consequently adapt the CAAA to state s''. This is a typical scenario of *adaptation race*. In latter case, when a context value updating event arrives, the CAAA evaluates the situation c in a rule r based on the snapshot of all context variables at the time point of event arrival. The snapshot at a point of time may not capture the situation correctly due to the *stale* context variables. The evaluation actually works on part of relevant variables because of the asynchronous nature of context value updating. It usually results in hustled adaptation. Unfortunately, adaptation rules based on propositional logic do not have the ability to describe the commutation order of relevant variables. These factors cause the CAAA triggering unwanted adaptations which is identified as *context hazards*.

The timing issue in context value updating is out of descriptive capability of the existing adaptation rule definition used in A-FSM. Let us illustrate the limitation of propositional logic based adaptation rules by examples. To ease of presentation, we use *bit string* to represent a snapshot of context variable assignments. For the *bit strings* used in the whole paper, the bit values are assumed to comply with the following order of context variable: A_{power}, A_{sensor}, B_{sensor}, C_{sensor}, D_{sensor}. For example, a bit string '11111' specifies the truth assignment of '*Power.level()<70*', '*Sensor.count()=0*', '*Sensor.count()<3*', '*Sensor.dist()<30*', and '*Sensor.dist()<50*' respectively.

2.2.2.1 Instability Faults

Table 2 shows a transition table of a context variable bit string that causes instability faults in mini-car application. The underscored bit values are used ones in adaptation rule in the current state S_c. As the true value holds, mini-car start with state *HS-A* will pass through states *LS-A, LS-M, LS-A, LS-M*, … The final state of this adaptation chain is random one between *LS-A* and *LS-M*. It is dangerous and annoying to have such unwanted adaptation. In existing adaptation rules specifications, such faults are hardly removable because the adaptation rules cannot easy introduce additional temporal constraints. The sensed context is discrete representation of continuous physical context values in real environment. It brings unexpected adaptation fluctuation when evaluating boundary conditions (e.g.,'*Power.level()<70'*) in adaptation rules. For instance, mini-car application may switch very frequently between state HS-M and LS-M when value of '*Sensor.count()*' varies around '3'. Propositional logic cannot be used to express assertions on a series unbounded number of events, say "The average value of '*Power.level(')* in last 10 seconds is greater than 70". Such aggregation functions can be used to eliminate fluctuation effects in CAAAs.

Table 2. Instability adaptation

S_c	A_{power}	A_{sensor}	B_{sensor}	C_{sensor}	D_{sensor}	S_n
HS-A	1	0	<u>0</u>	<u>1</u>	1	LS-A
LS-A	<u>1</u>	0	0	1	1	LS-M
LS-M	1	0	<u>0</u>	<u>1</u>	1	LS-A

2.2.2.2 Context Hazard

Table 3 shows a transition table that starts from state *HS-M* in three different cases when considering context variable bit string commute from '00000' to '00101'.

(1) Case 1: assumes the context variables B_{sensor} and D_{sensor} are commute simultaneously shown as underscored bit values, the evaluation result of predicates 'A_{power} or (B_{sensor} and $!D_{sensor}$)' in rule *activate_lsm* and '$!A_{power}$ and $!B_{sesnor}$ and D_{sesnor}' in rule

(a) Query 1

```
PATTERN SEQ(Shelf a, ~(Checkout b), Exit c)
WHERE   skip_till_next_match (a, b, c) {
        a.id = b.id
  and   b.id = d.id }
WITHIN 8 hours
```

(b) Query 2

```
PATTERN SEQ (GPS+ g1[], ~(BT b), GPS g2)
WHERE   skip_till_next_match (g1[], ~b, g2){
        g1[1].speed > 0
  and   g1[i].speed > avg(g1[..i-1].speed)
  and   g2.speed > 5 }
WITHIN 1 min
```

Figure3. Examples of event pattern queries.

activate_hsa remain *false*. The mini-car application should stay in the state *HS-M.*

(2) Case 2: assumes the context variables B_{sensor} and D_{sensor} are commuted in the list order due to the different refreshing rate. Rule *activate_lsm* is satisfied when B_{sensor} commutes (depicted as bold bit value) and drives the mini-car application to state *LS-M.* The succeeded commutation of D_{sensor} won't recover mini-car application from state *LS-M* to *HS-M.*

(3) Case 3: similar to Case 2, a reversed commutation order of context variable B_{sensor} and D_{sensor} will end up with state *HS-A* instead of *HS-M.*

Table 3. Context hazards

S_c	A_{power}	A_{sensor}	B_{sensor}	C_{sensor}	D_{sensor}	S_n
HS-M	0	0	0	0	0	HS-M
HS-M	0	0	1	0	1	HS-M
HS-M	0	0	**1**	0	0	LS-M
LS-M	0	0	1	0	**1**	LS-M
HS-M	0	0	0	0	**1**	HS-A
HS-A	0	0	**1**	0	1	HS-A

The commutation order of context variables plays a key role in choosing adaptation path in a CAAA. Again, this information cannot be specified by existing approach in A-FSM. It suggests that the adaptive action according to evaluation of single context updating event is not so desirable. Adaptation rule should check more events before it invokes actions and trigger transition between states.

As analyzed above, existing adaptation rule modeling techniques do not have sufficient expressive power to specify required adaptation behaviors:

(1) Lack of temporal constraints to describe time properties of context event updating;

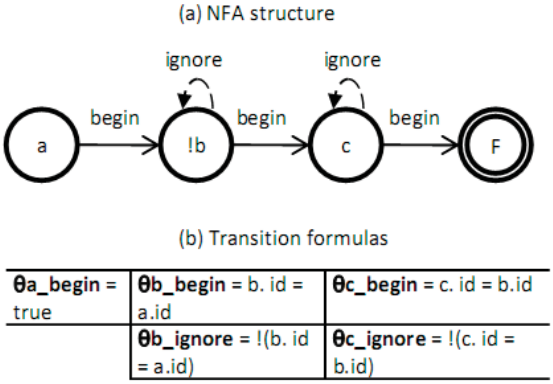

Figure4. The NFA^b automaton of Query 1.

(2) Cannot express commutation order of context variables;

(3) Cannot support user-defined domain-specific functions in smoothing fluctuated context values.

We therefore need to explore more expressive language in modeling adaptation rules. Sequential pattern query language naturally fits our purpose in tackling these issues. Our approach seeks to guide the CAAA developer designing more complex adaptation rules without the threats of instability faults and context hazards.

3. BACKGROUND

Sequential event pattern query over streams has attracted significant interests in areas such as RFID-based applications, and electronic health systems [6][12]. Recent studies have addressed the efficiency issues of pattern matching over streams [1][10]. Research results show that complex event processing can be dealt with in real-time and possibly with limited computation resources. To our best knowledge, applying sequential event pattern query in CAAAs has not been adequately studied. In this paper, we adopt pattern query language similar to SASE+ [1][5][14] in our adaptation rule modeling since it is shown to cover nearly all features in other pattern languages.

3.1 Event Model and Pattern Query

3.1.1 Event model

An ***event stream*** is a potentially infinite sequence of all types of events of interest. Each event represents an instantaneous occurrence of interest at a point of time. We denote these occurrences by ***event instances*** and use lower-case letter (e.g. *a, b, c*) to represent. An event instance contains a timestamp for its generated time, a name of its ***event type*** (e.g. GPS, BatteryPower), and a set of attribute values defined in the schema of each event type. Event stream is assumed to be composed of events in the order of their occurrence time.

3.1.2 Pattern Query

A ***pattern query*** specifies a correlated sequence of relevant events in order in the targeted event stream. The correlation and relevance of events are specified via constraints over the attribute values. Figure 3 shows examples of such query. Query 1 is to detect book theft in a RFID-based book store management

application [1]. The physical meaning of Query 2 is to detect a user with a GPS phone accelerates to the speed of 5 miles per hour before a car's hand-free device is detected by the phone's Bluetooth connection.

The **PATTERN** clause specifies a sequence of events of interest in a particular order using the **SEQ** construct. The sequence in Query 1 depicts an event of a shelf reading, succeeded by a non-occurred event of checkout reading, and followed by an event of an exit reading. The negation operator '~' denotes a non-occurred event, which is also referred to as *negative* event. Events without negation operator are referred to as *positive* event.

The **WHERE** clause encloses *predicates* on attributes of individual events or associated attributes across multiple events by the paired symbol '{' '}'. In Query 1, all the matched events in the pattern should have identical 'id' number.

The **WITHIN** clause confines the pattern query in a *time window* of predefined length of time, for example, '8 hours' in the above case. It is important to note that the semantic of **WITHIN** is a bit different when *negative* event presented at the end of sequential query. We will further explain it in Section 4 when we use *negative* events together with **WITHIN** clause in our adaptation rule design.

In Query 2, the event definition 'GPS+ g1[]' in sequence pattern is an expression of *Kleene closure*. Symbol '+' is a *Kleene plus* operator to capture unbounded number of events with particular properties which are specified with *predicates* in **WHERE** clause. 'g1[]' is declared as an array variable to store captured events.

Beside above constructs and features, pattern query language further provides *event selection strategy* to single out relevant events from a merged event stream consisting both relevant and irrelevant events for a given query. The strategy is declared as a function applying to the whole scope of **WHERE** clause. Unlike typical regular expression matching against strings, selected events in sequential pattern query are not always necessarily to be contiguous. For instance, in Query 2, only events of type GPS and BT are deemed as relevant. Other types of events are irrelevant and can be simply skipped when the event stream is evaluated against the pattern query. To support different usage scenarios, the pattern query language we adopt supports four types of strategies: *strict contiguity*, *partition contiguity*, *skip till next match*, and *skip till any match*. We will explain their semantic if used in this paper. Interested readers are suggested to refer to [1] for details on the semantics of these strategies.

3.1.3 Evaluation of Pattern Queries

The pattern queries of syntax in this section are translated to a formal evaluation model. The evaluation model uses a new type of nondeterministic finite automaton (NFA^b) where 'b' stands for a match buffer. Query 1 can be compiled to an automaton depicted in Figure 4. The sequential event patterns queries over input event stream is evaluated on this formal model. The transition formulas are evaluated when relevant events are selected and the automaton decides to enter correspondent state according to the semantic of the edge. For instance, *begin edge* means the automaton consume the current selected event and proceed to next state.

4. DESINGING ADAPTATION RULES WITH SEP

In this section, we introduce sequential event pattern (SEP) in designing adaptation rules for developing more reliable adaptation behavior of CAAA. With the simulated mini-car application, we compare our approach with existing rule specification to show the advantage of SEP in avoiding instability faults and context hazards.

4.1 Requirements of Adaptation Rule Design

Adaptation faults are application-specific according to the criteria of each CAAA. In our definitions, it is said to be *adaptation fault* when particular adaptation behavior is not as expected by users. The *instability* faults and *context hazards* are deemed as faults because they may lead CAAA to uncertain or unwanted states as illustrated in Section 2.

Let us first review the limitation of existing adaptation rule modeling approach summarized in Section 2 and argue why existing rule modeling language is not capable of specifying those unwanted situations.

4.1.1 Temporal Constraints

The timing issue is the main reason of *instability* faults and *context hazards* in existing adaptation rule design. The actual length of time a context variable holds its value is unknown. Existing adaptation rule cannot specify user's preference on how long the application should wait for a particular relevant context variable updates.

SEP can define a series event of particular characteristics in a specified time window. We can possibly confine the context updating events of interest in a fixed time window. It also makes possible to evaluate the application's situation and adaptation conditions based on historical events.

4.1.2 Order of Events

Context hazards exist due to asynchronous updating of context values. Since synchronizing the updates of context value is practically hard, the application is not able to maintain its current state as user may prefer when multiple context variables updated in a very short period of time. This is usually called *hold hazard* [11]. As we illustrated in Table 3, asynchronous updating context value may lead the mini-car application to two different states. Existing adaptation rule cannot specify the context value updating order in which the adaptation rules count on. Obviously, application's adaptive behavior may be not as expected by the user.

SEP inherently describes the order of events according to their occurrence time. It is possible to precisely designate the particular order of a series of context value updating events. Thus, the commutation order is predefined in adaptation rules and the adaptation path could be expected in design.

4.1.3 User-Defined Function

A special class of *instability* faults is called *fluctuation race* that results from fluctuation of sensed context value when evaluating boundary conditions in adaptation rules. The fluctuation may arise from real fluctuation in physical world, such as speed variation of the car. Unreliable context provision may cause context value fluctuating as well, such as temporary disconnection, reconnection of Bluetooth devices.

Table 4. Comparison of adaptive behavior of A-FSM and SEP

1		Event	c_1	d_1	c_2	c_3	d_2	d_3	d_3'	…
2		Value	4	60	2	2	55	45	45	…
3		Time	0	1	**2**	4	5	**6**	**8**	…
4		Result	{}	{}	$\{c_2\}$	$\{c_2c_3\}$ $\{c_3\}$	$\{c_2c_3d_2\}$ $\{c_3d_2\}$	~~$\{c_2c_3d_2d_3\}$ $\{c_3d_2d_3\}$~~	$\{c_2c_3d_2\ d_3'\}$ ~~$\{c_3d_2d_3'\}$~~	…
5		B_{sensor}	0	0	**1**	1	1	1	1	…
6		D_{sensor}	0	0	0	0	0	1	1	…
7	s	A-FSM	hsm	hsm	**lsm**	lsm	lsm	lsm	-	…
8	s	SEP	hsm	hsm	hsm	hsm	hsm	hsm	**lsm**	…

Selection strategy, kleene plus operation, together with user-defined aggregation function over historical context values can be used to filter out noisy events and prevent some of *fluctuation race*.

4.2 Adaptation Rule with SEP

We focus on two types of faults: *fluctuation race* fault and *hold hazard*. As analyzed before, the former results from context value fluctuation. The latter is caused by the unknown commutation order.

4.2.1 Rule to avoid fluctuation race

As we discussed above, a major cause of *fluctuation race* is the frequent variation of context values. We use three language features in combination to alleviate the fluctuation effects.

In A-FSM, the evaluation of boundary conditions is conducted in every context value updating event's arrival and consequently only checks the current value of all the relevant events. Missing or delayed context updates, every fluctuation will trigger a hustle adaptation if an adaptation rule is satisfied by this fluctuation. Unlike A-FSM, we model situation C in adaptation rules with pattern queries, which can decide a situation by checking a series of historical events rather than only current one.

Query 3 in Figure 5(a) specifies such an event pattern for rule *activate_hsm*. We use a kleene plus component '*p[]*' and a predicate on event *p* with aggregation function to capture the context updating event whose power level is greater than 70. The pattern query checks if a context updating events to notify the number of detected obstacles equals to zero right after some context updating events notify the high level power status. If any event sequence is found matched in 30 seconds, the situation will be deemed as satisfied and the adaptation rule is triggered.

This pattern query has a good property to tolerant temporary low level power reading. When a context updating event notifies a level less than 70, the pattern query matching will not fail immediately. It will skip this event because of the semantic of *skip till next match* allows the pattern matching waiting to see if any other events saying power level is high exists. In general, a pattern query will output more than one successful matching result. In this query, any matching of described sequence will satisfy the adaptation rule.

It is always possible to use these three language features to smooth and tolerate fluctuated context value updating. We can also apply similar pattern query to prevent *fluctuation race* found in *PhoneAdapter* [11].

4.2.2 Rule to avoid hold hazard

The *hold hazard* is commonly found in our mini-car application and *PhoneAdapter* application [11]. It is hard to synchronize the context updating with different refreshing rate in practice. We design a pattern query with particular acceptable commutation order, *negative* commutation event and a fixed time window to specify the adaptive behavior more precisely than in A-FSM.

Query 4 captures a sequence start with a context updating event that notifies less than 3 obstacles detected. And the detected obstacles are always less than 3 before the nearest distance to obstacles larger than 50 is detected. And the most important part of this query is to guarantee that no obstacles are detected in the distance less than 50 after the distance larger than 50 is detected within 5 seconds.

This kind of pattern query has two advantages over adaptation rules in A-FSM:

(1) Explicitly specified acceptable commutation order. The event pattern in SEQ constructs requires the adaptation rule accept commutation of *Count* context first and then *Dist* context. If we project these two contexts to A-FSM model, they correspond to context variable B_{sensor} and D_{sensor}. That means only commutation order (B_{sensor} D_{sensor}) is accepted. And if this commutation happens, we could exactly know the next state after HS-M is LS-M.

(2) Negative event and WITHIN clause can be used postpone evaluation decision if we want the mini-car application to stay in state HS-M after context variable commuted.

In Table 4, we simulate the adaptation rule evaluation process for A-FSM and our approach. The first three rows represent an event stream of two types of events (number of detected obstacles c_i and distance to the nearest obstacles d_i), the value of each instance, and the timestamp of each event instance. The fourth row records the pattern matching result after each event instance arrives. The fifth and sixth rows are evaluation results of two context variable used in A-FSM. The seventh and eighth rows are mini-car application's state using A-FSM and SEP respectively.

When c_2 arrives, the pattern query will add it to the result set. In A-FSM, B_{sensor} is evaluated to be *true* and commutes its bit value

from '0' to '1'. This commutation will cause the rule *activate_lsm* to be satisfied and adapt the mini-car application from *HS_M* to

(a) Query 3

```
PATTERN SEQ(Power+ p[], Count c)
WHERE   skip_till_next_match (p[], c) {
        p[1].level > 70
  and   p[i].level > min(p[..i].level)
  and   c = 0}
WITHIN 30 seconds
```

(b) Query 4

```
PATTERN SEQ(Count+ c[], Dist+ d1[], ~Dist d2)
WHERE   skip_till_next_match (c[], d1[],~d2){
        c[i].value < 3
  and   d1[i].value > 50
  and   d2.value < 50 }
WITHIN 5 seconds
```

Figure5. Pattern query in adaptation rules.

LS-M. A *hold hazard* occurred if the user wants to maintain the application in state *HS-M* when a close distance of obstacles detected in a very short time. Our approach is free of this type of *hold hazard*. The pattern query matching process will look forward to see if commutation of *Dist* context (an obstacle is detected to be near than 50) happens in recent 5 seconds. If it happens, as d_3 arrives (in 4 seconds after c_2 is selected and in 2 seconds after c_3 is selected), the evaluation of pattern query fails and the adaptation rule will not be triggered. Mini-car application thus stays at the *HS-M* state. Considering another case, if d_3' arrives in the 6 seconds after c_2 is selected, the pattern query is evaluated as successful, although another spontaneous matching is fail. The mini-car application will adapt to *LS-M* accordingly. The semantic of **WITHIN** in the presence of negative event will exactly guarantee this.

In this example, we show how commutation order of two context variables can be specified in SEP based adaptation rules. It is the same to specify commutation order of multiple context variables. We use similar pattern queries in both mini-car and *PhoneAdapter* applications. It can prevent all *hold hazard* of the kind.

4.2.3 Discussion

Currently, our approach is only applicable to these two particular adaptation faults which are very common ones both found in mini-car application and other CAAA such as *PhoneAdapter*. We found that it is not easy to verify other properties of adaptation rules when sequential event pattern has been introduced. Particularly, determinism, state liveness, rule liveness. Because the existing static analysis technique is not applicable since the space of possible event stream is infinite. We need to balance the advantages and disadvantages when using sequential event pattern to specify the adaptation rules.

5. RELATED WORK

In this section, we review the related work in context-aware pervasive computing and event pattern processing.

Context-aware applications situate in the central of mobile and pervasive computing. Several groups of researchers have proposed context-aware framework Context Toolkit[4] and context-aware middleware to support the development and execution of context-aware applications. For instance, *CARISMA*[3], *EgoSpaces*[8], *Cabot*[14], and RCSM[15].

Context-aware applications are significantly impacted by the quality of context. The noisy and corrupted context stream is not uncommon. Jeffery et al. [7] and Xu et al. [14] propose techniques to clean up low quality context data stream. These work focus on different level aiming to improve context provisions.

Adaptation rules plays important role in context-aware applications. Adaptation rules may suffer from non-determinism, instability faults, and possibly conflict to each other. Sama et al. [11] analyze and detect the adaptation faults on an A-FSM model statically, and use simple predicates to construct adaptation rules. However, it is not clear how to correct them in design time. Although we adopt the finite-state machine to modeling CAAA as well, our work differs from theirs in exploring guidance on more reliable adaptation rules design.

Other researchers also propose many runtime mechanisms to cope with context invalidation and adaptation conflicts. Capra et al. [3] argue that conflicts of adaptation policies cannot be resolved statically and use a microeconomic mechanism to mitigate policy conflicts in the runtime. Kulkarni et al. [9] integrate exception handling model in context-aware application design and provide a programming framework to support runtime recovery when various failure condition arising. It is our next step to investigate the integration of sequential pattern matching and failure recovery mechanisms.

Context-aware adaptation using sequential event pattern queries has not been studied adequately. Liu et al. [10] observe the out-of-order events in event stream based applications and propose aggressive and conservative strategies to tackle such problem with different assumption on the presence rate of out-of-order events. This work can complement ours in the presence of out-of-order context updating events.

6. CONCLUSION

In this paper, we have carefully analyzed the timing issue of context value updating, and the reason of instability faults and context hazards in existing approach to specify adaptation rule. We have proposed to introduce sequential event pattern in adaptation rule specification to prevent certain type of instability faults and context hazards.

We have evaluated our approach with a simulated CAAA. Although adaptation rules with sequential event pattern are more reliable than existing one in terms of the potential to eliminate instability faults and context hazards, it is still preliminary and way far from a comprehensive approach to design dependable adaptation rules in CAAA. We plan to explore the methodological guidance and design principles in designing sequential pattern query based adaptation rule in our future work.

Adaptation rules with sequential event pattern is much complex than with simple propositional logic. Whilst the language's expressive power brings many benefits in specifying adaptation behavior, it is also make harder to verify properties such as determinism, state liveness, and rule liveness. We also plan to further investigate new verification techniques in the presence of sequential event pattern queries.

7. ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China under Grant No. 60773028, the National Grand Fundamental Research 973 Program of China under 2008CBA20704, the National Science and Technology Major Project of China under the Grant No. 2009ZX01043-003-001, and the National Key Technology R&D Program under the Grant No. 2009BADA9B02.

8. REFERENCES

[1] Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N. 2008. Efficient Pattern Matching over Event Streams. In *SIGMOD*, 147-159.

[2] Bellavista, P., Corradi, A., Montanari, R., and Stefanelli, C.2003. Context-Aware Middleware for Resource Management in the Wireless Internet. In *IEEE Transactions on Software Engineering*, v.29 n.12, pp. 1086-1099, December, 2003. DOI=http://dx.doi.org/10.1109/TSE.2003.1265523

[3] Capra, L., Emmerich, M., Mascolo, C. 2003. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications, *IEEE Transactions on Software Engineering*, v.29 n.10, p.929-945, October 2003 DOI= http://dx.doi.org/10.1109/TSE.2003.1237173

[4] Dey, A. K., Salber, D., Abowd, G. D. A Conceptual Framework and a Tookit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction (HCI) Journal*, Vol 16 (2--4), 2001, pp. 97--166.

[5] Gyllstrom, D., Agrawal, J., Diao, Y., Immerman, N. 2008. On supporting kleene closure over event streams. In *ICDE*, 2008. Poster.

[6] Harada, L., Hotta, Y. 2005. Order checking in a CPOE using event analyzer. In *CIKM*, 549-555, 2005.

[7] Jeffery, S.R., Garofalakis, M., Franklin, M.J. 2006. Adaptive cleaning for RFID data streams. In *Proceedings of VLDB* 2006, pp.163-174. 2006.

[8] Julien, C., Roman, G. C. 2003. EgoSpaces: facilitating rapid development of context-aware mobile applications. In *IEEE Transactions on Software Engineering,* vol32,no.5, pp.281-298. 2006. DOI= http://doi.ieeecomputersociety.org/10.1109/TSE.2006.47

[9] Kulkarni, D., Tripathi, A. 2010. A Framework for Programming Robust Context-Aware Applications, *IEEE Transactions on Software Engineering*, vol. 99, no. RapidPosts, pp. 184-197, 2010. DOI= http://doi.ieeecomputersociety.org/10.1109/TSE.2010.11

[10] Liu, M., Golovnya, D., Rundensteiner, E. A., Claypool, K. 2009. Sequential Pattern Query Processing over Out-of-Order Event Streams. In ICDE, 784-795, 2009.

[11] Sama, M., Rosenblum, D. S., Wang, Z., and Elbaum, S. 2008. Model-based fault detection in context-aware adaptive applications. In *Proceedings of the 16th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (Atlanta, Georgia, November 09 - 14, 2008). SIGSOFT '08/FSE-16. ACM, New York, NY, 261-271. DOI= http://doi.acm.org/10.1145/1453101.1453136

[12] Wang, F., Liu, P. 2005. Temporal Management of RFID data. In *VLDB*, 1128-1139, 2005.

[13] Wu, E., Diao, Y., and Rizvi, S. 2006. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international Conference on Management of Data* (Chicago, IL, USA, June 27 - 29, 2006). SIGMOD '06. ACM, New York, NY, 407-418. DOI= http://doi.acm.org/10.1145/1142473.1142520

[14] Xu, C., Cheung, S. C., Chan, W., and Ye, C. 2010. Partial constraint checking for context consistency in pervasive computing. *ACM Trans. Softw. Eng. Methodol.* 19, 3 (Jan. 2010), 1-61. DOI= http://doi.acm.org/10.1145/1656250.1656253

[15] Yau, S.S., Wang, Y., Karim, F. 2004. An Adaptive Middleware for Applications in Ubiquitous Computing Environments. In *Real-Time Systems*, vol.26, no.1, pp. 233-238, 2004.