Int J Software Informatics, Vol.4, No.4, December 2010, pp. 419–436 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
©2010 by Institute of Software, Chinese Academy of Sciences. All rights reserved. Tel: +86-10-62661040

Sequential Event Pattern Based Design of Context-Aware Adaptive Application

Chushu Gao[1,3], Jun Wei[1], Chang Xu[2,3] , and S.C. Cheung[3]

[1] (TCSE, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2] (State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

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

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 the 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 in designing more reliable context-aware adaptive applications. We also introduce adaptive rule specification patterns to guide the design of adaptation rules.

Key words: context-aware adaptation; pervasive computing; sequential event pattern; property pattern

Gao CS, Wei J, Xu C, Cheung SC. Sequential event pattern based design of context-aware adaptive application. *Int J Software Informatics*, **2010, 4(4): 419–436. http://www.ijsi.org/1673-7288/4/i66.htm**

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)[20,21].

* This work is sponsored by the National Natural Science Foundation of China under Grant Nos. 60773028 and 60736015, 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, the National Key Technology R&D Program under the Grant No. 2009BADA9B02. The research was partially supported by the Research Council of Hong Kong (Project No. 612309).
Corresponding author: Chushu Gao, Email: gaochushu@otcaix.iscas.ac.cn
Received 2010-08-29; revised 2010-12-10; accepted 2010-12-25.

CAAAs are generally built on top of context-aware middleware (CAM) that facilitates collection and manipulation of contexts[2,3,4,6,13]. Implementations of context-aware middleware mostly share two well separated components in their conceptual models as depicted in Fig.1: (1) a *context manager* to collect, manipulate and feed context information required by context-aware applications, and (2) an *adaptation manager* to evaluate adaptation rules based on the contexts provided by context manager, and to decide how an application concerned should react to the context changes.

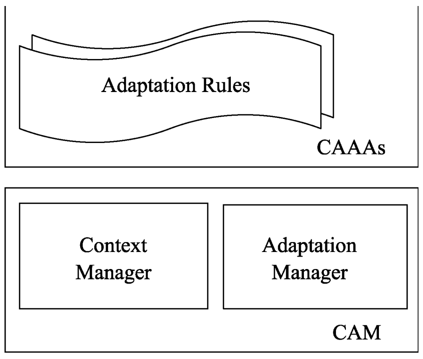


Figure 1. Conceptual architecture of CAAAs and CAM

The developers of CAAAs define adaptive behavior by specifying adaptation rules. Existing CAAAs and adaptation rules in recent publications commonly suffer from two types of faults: *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) [15,16,20,21]. These faults, although statically detectable at design time, are not easy to eliminate.

Adaptation rules are subject to *context hazard* [19,20], which 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 trigger unwanted adaptations and lead CAAAs to unexpected states. For instance, when drivers enter a car, the *PhoneAdapter* application[20,21] of their smart phones should be able to detect the new situation via a Bluetooth connection with the car's hands-free system, and hence put themselves to a *Driving* mode. However, if the Bluetooth connection suffers from delays, the phones may incorrectly put themselves in a *Jogging* mode when the drivers in the car start to move slowly at a speed of a few miles per hour. Even the Bluetooth connection is successfully established afterwards, it is not easy to restore the application to a correct state when GPS continually detects the speed of a few 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 hazard can be eliminated by introducing fixed delays to ensure all related context values to be updated before adaptation is performed. However, the determination of a delay's period is undecidable if context refresh rates are unknown. Moreover, frequent delays degrade applications' performance.

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, their occurrences essentially depend on the order of context updating events. Existing adaptation rule specification approaches have limitation in expressing this information. Various forms of formal logic have been proposed to express these properties over streamed events[5,18,19]. However, it is difficult to understand by practitioners when they specify system behavioral properties in these formal logics[7].

In this paper, we present a novel approach to designing adaptation rules in a sequential event pattern query language to address this limitation whilst maintaining the comprehensibility of the specification language by introducing adaptive rule specification patterns. In addition, we illustrate that sequential event pattern query language can be used to design reliable adaptation rules. We finally propose to design adaptive rules based on property specification patterns.

The remainder of the paper is organized as follows. Section 2 uses a motivating example to analyze the limitation of existing approaches for specifying adaptation rules. 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, and the design guidance on the basis of property specification pattern. Section 5 reviews the related work, and finally Section 6 concludes the paper with a summary of our contributions and future research plans.

2 Motivations

In this section, we use a simple but typical pervasive application, Autonomous Mini-car, to illustrate our problem. The motivating example is subject to common adaptation faults found in other CAAAs with existing modeling approaches.

2.1 Autonomous mini-car application

Autonomous 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 these entities are away from the car. The precise refresh rate of such derived contexts is not easy to know since its computation time 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 explore an unknown map area till the mini-car runs out of battery or receives a power-off signal. The car needs to avoid the obstacles blocking its way. 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)[20,21] to describe the application. The composite states of the four modes and transitions between them are shown in Fig.2. Autonomous 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)*: a slow-speed mode that employs a normal exploring strategy when the chances of hitting surrounding obstacles are rare;

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

(3) *High-speed Avoidance (HS-A)*: a high-speed mode that employs an optimistic obstacle avoiding strategy when nearby obstacles are detected but the chances of hitting them are low;

(4) *Low-speed Avoidance (LS-A)*: a slow-speed mode that employs a conservative obstacle avoiding strategy when close obstacles are detected.

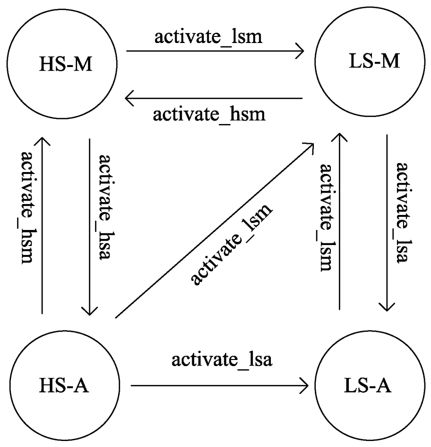


Figure 2. A-FSM modeling of the autonomous mini-car application

2.2 Problem in existing adaptation rule modeling

The adaptive behavior of a rule-based CAAA can be modeled by a finite-state machine[20,21]. We have conducted experiments by applying the existing A-FSM approach to our mini-car application 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".

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 it enters 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 derived 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. The mini-car application adapts among the four states governed by four different rules defined over five propositional context variables as listed in Table 1.

Table 1 Adaptation rules of autonomous 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$_{sensor}$
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$_{sensor}$ and D$_{sensor}$) or (B$_{sensor}$ and C$_{sensor}$))
activate_lsa	LS-M,HS-A	LS-A	!Sensor.count()<3and Sensor.dist()<30	!B$_{sensor}$ and C$_{sensor}$

2.2.2 Instability fault and context hazard

The approach of A-FSM assumes that every entrance to some state and every value updating of a context variable can trigger the CAAA to re-evaluate all the active rules in that state. In the former case, even no new value is assigned to any of the context variables, the CAAA may trigger a rule r in the new states' if the situation c is satisfied by current values of context variables, and consequently adapt the CAAA to states''. This is a typical scenario of *adaptation race*. In the 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}, and 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 the mini-car application. The underscored bit values are those involved in the rule evaluation at the current state S_c. According to the value assignment in the table, the state of the mini-car will oscillate between *LS-A* and *LS-M* repeatedly after it has reached the state *HS-A*. The final state of the car is therefore non-deterministically chosen from *LS-A* and *LS-M*. This illustrates a dangerous and undesirable adaptive behavior. In existing adaptation rules specifications, such faults are hard to fix because of the difficulties in adding new temporal constraints to the adaptation rules. The sensed context can be considered to be a 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

$\boldsymbol{S}_c$	$\boldsymbol{A}_{power}$	$\boldsymbol{A}_{sensor}$	$\boldsymbol{B}_{sensor}$	$\boldsymbol{C}_{sensor}$	$\boldsymbol{D}_{sensor}$	$\boldsymbol{S}_n$
HS-A	1	0	$\underline{0}$	$\underline{1}$	1	LS-A
LS-A	$\underline{1}$	0	0	1	1	LS-M
LS-M	1	0	$\underline{0}$	$\underline{1}$	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'.

Table 3 Context hazards

$\mathbf{S}_c$	$\boldsymbol{A}_{power}$	$\boldsymbol{A}_{sensor}$	$\boldsymbol{B}_{sensor}$	$\boldsymbol{C}_{sensor}$	$\boldsymbol{D}_{sensor}$	$\mathbf{S}_n$	
Initial state	HS-M	0	0	0	0	0	HS-M
Case 1	HS-M	0	0	$\underline{1}$	0	$\underline{1}$	HS-M
Case 2	HS-M	0	0	$\underline{\mathbf{1}}$	0	0	LS-M
	LS-M	0	0	1	0	$\underline{\mathbf{1}}$	LS-M
Case 3	HS-M	0	0	0	0	$\underline{\mathbf{1}}$	HS-A
	HS-A	0	0	$\underline{\mathbf{1}}$	0	1	HS-A

(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_{sensor} and D_{sensor}' in rule *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*.

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;

(2) Cannot express commutation orders of context variables;

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

We therefore need to explore a 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 (RFID stands for Radio Frequency Identification), and electronic health systems[11,25]. Recent studies have addressed the efficiency issues of pattern matching over streams[1,17]. 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 a pattern query language similar to SASE+[1,10,26] 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 '$\sim$' 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

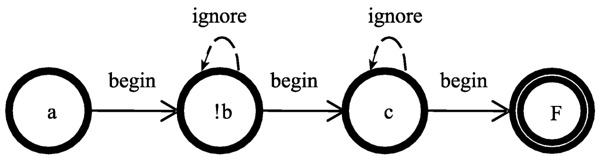
The pattern queries of syntax in this section are translated to a formal evaluation model. The evaluation model uses a new type of nondeterministic finite automaton (NFA^b) where 'b' stands for a match buffer. Query 1 can be compiled to an automaton depicted in Fig.4. The sequential event patterns queries 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.

```
(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 }
```

Figure 3. Examples of event pattern queries

(a) NFA structure



(b) Transition formulas

θa_begin= true	θb_begin= b.id = a.id	θ c_begin= c.id = b.id
	θb_ignore= !(b.id = a.id)	θ c_ignore= !(c.id = b.id)

Figure 4. The NFA^b automaton of query 1

4 Designing 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[20,21]. 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 not be 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.

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 $\mathcal{C}$ 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 Fig.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.

(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
```

Figure 5. Pattern query in adaptation rules

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*[20,21].

4.2.2 Rule to avoid hold hazard

The *hold hazard* is commonly found in our mini-car application and *PhoneAdapter* application[20,21]. 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.

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$_2$c$_3$}	{c$_2$c$_3$d$_2$}	~~{$c_2c_3d_2d_3$}~~	~~{c$_2$c$_3$d$_2$ d$_3$'}~~	...
					{c$_3$}	{c$_3$d$_2$}	~~{$c_2d_2d_3$}~~	~~{$c_3d_2d_3'$}~~	
5	$\boldsymbol{B}_{sensor}$	0	0	<u>**1**</u>	1	1	1	1	...
6	$\boldsymbol{D}_{sensor}$	0	0	0	0	0	1	1	...
7	**state A-FSM**	hsm	hsm	**lsm**	lsm	lsm	lsm	-	...
8	**state SEP**	hsm	hsm	hsm	hsm	hsm	hsm	**lsm**	...

When c_2 arrives, the pattern query will add it to the result set. In A-FSM, Bsensor 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 *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 the commutation order of two context variables can be specified in SEP based adaptation rules. It is the same to specify a commutation order for 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, in particular, 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.

On the other side, even with the language features provided by sequential event pattern query, the property expressed in a particular event pattern is not explicit enough for developer to use and understand. The specification using sequential event pattern is still discretional and without disciplined guidance. Temporal properties can be specified using property patterns[7]. Patterns are recurring solutions for particular problems. Adaptation rules essentially express how adaptive application should behave when certain property is hold. We follow the definition of property specification pattern and introduce the adaptive rule specification pattern to direct the design of sequential event pattern query used in adaptation rules.

4.3 Adaptive rule specification pattern

4.3.1 Property specification pattern

Dywer et al.[7] propose property specification pattern to bridge the gap between practitioners and automatic model checking tools and methods, and ease the property specification via a given pattern system. Property specification pattern can express property such as "An occurrence of event P must be followed by an occurrence of event Q", "Event R must not happen between events P and$Q$". The pattern template can be translated to various state-based or event-based formalisms.

According to the property expressed, the patterns fall into two main catalogues, namely *Occurrence* patterns and *Order* patterns, and are organized into hierarchy (see Fig.6.). The detailed meaning of the patterns is as below:

Absence A given event P does not occur within a scope;

Existence A given event P must occur within a scope;

Bounded existence A given event P must occur at least/exactly/at most k times within a scope;

Universality A given event P occurs throughout a scope;

Precedence A given event P must always be preceded by a given event Qwithin a scope;

Precedence chain A chain of events must always be preceded by another chain of events within a scope;

Response A given event P must always be followed by a given event Q within a scope;

Response chain A chain of events must always be followed by another chain of event within a scope.

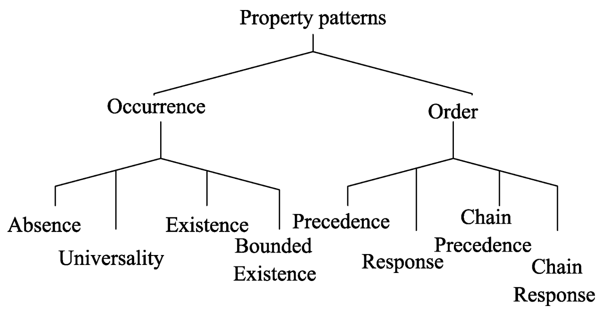


Figure 6. Pattern hierarchy

The patterns are constrained with scope. The scope is defined according to the occurrence of other events. E.g., "*After Q*" means the pattern is valid after an event Q occurred. These atomic patterns can be composed and extended to construct new patterns for specific purpose.

4.3.2 Implementing property pattern with SEP

Let us revisit the requirements of adaptation rule design and adaptation rules aforementioned. The temporal constraints and order of events in adaptation rule designing can be specified using extended property specification patterns.

Temporal constraints are used to specify the freshness of context events. Basic property pattern does not allow specifying such timed properties. We use a variation of property specification pattern proposed by Gruhn et al.[9]. The extended patterns allow specifying property like "An occurrence of event P must be followed by an occurrence of event Q within k time units". We denote this sort of property pattern as *Timed* pattern. All the basic patterns can be extended to time bounded ones. After a carefully review of all our SEP query based adaptation rules in mini-car and other applications in recent publications, a set of selected basic patterns are extended to construct our preliminary adaptive rule specification patterns and the correspondent SEP templates are given in Table 5.

To design a context-aware adaptation rule, one can follow the guidance of an adaptive rule specification pattern, find correspondent SEP template, and substitute parameters in SEP template to construct a concrete SEP query for adaptation rule specification. In some cases, adaptation rules cannot found a suitable basic specification pattern. A variation pattern that recursively applies different basic patterns can be used to specify these adaptation rules. E.g., the adaptation rule specification of Query 4 can be instantiated by using *Timed Response Chain* pattern, where the first chain of events can be instantiated by *Timed Bounded Existence* pattern and substitute the parameters with *Count* events; and the second chain of events can be instantiated by *Timed Bounded Existence* pattern and substitute the parameters with *Dist* events plus a *Timed Absence* pattern instance with another *Dist* event.

Table 5 Adaptive rule specification pattern and SEP template

Adaptive rule specification pattern	**Pattern description**	**SEP template**
Timed Absence	An event P must not occur for k time units from t_0	SEQ ($\sim P$) WHERE $\{P.time > t_0\}$ WITHIN k
Timed Existence	An event P must occur within k time units from t_0	SEQ (P) WHERE $\{P.time > t_0\}$ WITHIN k
Timed Bounded Existence	A given event P must occur at least n times within k time units from t_0	SEQ($+P$[n]) WHERE $\{P[i].time > t_0\}$ WITHIN k
Timed Response	A given event P must always be followed by a given event Q within k time units from the time P occurred	SEQ(P,Q) WHERE {} WITHIN k
Timed Response Chain	A chain of events must always be followed by another chain of event within k time units	SEQ($P_1, P_2, \ldots, P_i, Q_1, Q_2, \ldots, Q_j$){} WITHIN k

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[6] and context-aware middleware to support the development and execution of context-aware applications. For instance, CARISMA[3,4], EgoSpaces[13], Cabot[29,30,31], and RCSM[32].

Context-aware applications are significantly impacted by the quality of context. The noisy and corrupted context stream is not uncommon. Jeffery et al.[12] and Xu et al.[29,31] 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.[20,21] 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.[3,4] 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.[15,16] 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.[17] observe the out-of-order events in event stream based applications and propose aggressive and conservative strategies to tackle such problem with different assumptions on the presence rate of out-of-order events. This work can complement ours in the presence of out-of-order context updating events. One close work is[24] which uses declarative event pattern as specifications and Aspect Oriented Programming techniques to implement protocols. The main focus of this work is to improve the comprehensibility, maintainability, and traceability.

Property specification pattern is widely used for finite-state verification[9,14], design-time compliance management[8], and runtime monitoring[22,23,27,28]. Our work focuses on the use of property specification pattern to help specifying context-aware adaptive rule.

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 further have mapped the sequential event pattern query onto property specification property to help better understand of properties expressed in event pattern queries.

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 study the automatic translation from adaptive rule specification pattern to SEP queries and explore other 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.

References

[1] Agrawal J, Diao Y, Gyllstrom D, et al. Efficient Pattern Matching over Event Streams. SIGMOD. 2008. 147–159.

[2] Bellavista P, Corradi A, Montanari R, et al. Context-Aware Middleware for Resource Management in the Wireless Internet. IEEE Trans. on Software Engineering, December, 2003, 29(12):1086-1099. DOI=http://dx.doi.org/10.1109/TSE.2003.1265523

[3] Capra L, Emmerich M, Mascolo C. A Micro-economic Approach to Conflict Resolution in Mobile Computing. SIGSOFT FSE. Charleston, South Carolina, USA. 2002. 31–40.

[4] Capra L, Emmerich M, Mascolo C. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Trans. on Software Engineering, October 2003, 29(10):929–945. DOI= http://dx.doi.org/10.1109/TSE.2003.1237173

[5] Clarke EM, Emerson EA, et al. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. ACM Trans. on Programming Languages and Systems, April, 1986, 2: 244–263.

[6] Dey AK, Salber D, Abowd GD. A Conceptual Framework and a Tookit for Supporting the Rapid Prototyping of Context-Aware Applications. Human-Computer Interaction (HCI) Journal, 2001, 16 (2–4): 97–166.

[7] Dwyer MB, Avrunin GS, Corbett JC. Patterns in Property Specifications for Finite-State Verification. ICSE. ACM. Los Angeles, CA, USA. 1999. 411–420.

[8] Elgammal A, Tretken O, van den Heuvel WJ, et al. Root-Cause Analysis of Design-Time Compliance Violations on the Basis of Property Patterns. ICSOC. San Francisco, CA, USA. 2010: 17–31.

[9] Gruhn V, Laue R. Patterns for Timed Property Specifications. Electr. Notes Theor. Comput. Sci. (ENTCS). 2006, 153(2): 117–133.

[10] Gyllstrom D, Agrawal J, Diao Y, Immerman N. On Supporting Kleene Closure over Event Streams. ICDE. Poster.

[11] Harada L, Hotta Y. Order Checking in a CPOE Using Event Analyzer. CIKM. 2005. 549–555.

[12] Jeffery SR, Garofalakis M, Franklin MJ. Adaptive Cleaning for RFID Data Streams. Proc. of VLDB, 2006. 163–174.

[13] Julien C, Roman GC. EgoSpaces: Facilitating Rapid Development of Context-aware Mobile Applications. IEEE Transactions on Software Engineering, 2006, 32(5):281–298. DOI=http://doi.ieeecomputersociety.org/10.1109/TSE.2006.47

[14] Konrad S, Cheng BHC. Real-time Specification Patterns. ICSE. ACM. St. Louis, Missouri, USA. 2005. 372–381.

[15] Kulkarni D, Tripathi A. Application-Level Recovery Mechanisms for Context-Aware Pervasive Computing. SRDS. Napoli, Italy. 2008. 13–22.

[16] Kulkarni D, Tripathi A. A Framework for Programming Robust Context-Aware Applications. IEEE Transactions on Software Engineering, 2010, 99(RapidPosts): 184–197. DOI=http://doi.ieeecomputersociety.org/10.1109/TSE.2010.11

[17] Liu M, Golovnya D, Rundensteiner EA, et al. Sequential Pattern Query Processing over Out-of-Order Event Streams. ICDE, 2009. 784–795.

[18] Manna Z, Pnueli A. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag New York, Inc.. 1992.

[19] Ramakrishna YS, Melliar-Smith PM, Moser LE, et al. Interval Logics and Their Decision Procedures: Part I + II. Theoretical Computer Science. 1996, 170(1-2): 146–147.

[20] Sama M, Rosenblum DS, Wang Z, et al. Model-based Fault Detection in Context-aware Adaptive Applications. Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM. Atlanta, Georgia. November 09-14, 2008). 261–271. DOI= http://doi.acm.org/10.1145/1453101.1453136

[21] Sama M, Rosenblum DS, Elbaum S, et al. Context-Aware Adaptive Applications: Fault Patterns and Their Automated Identification. IEEE Trans. Software Eng. 2010, 36(5): 644–661.

[22] Simmonds J, Chechik M, Nejati S, et al. Property Patterns for Runtime Monitoring of Web Service Conversations. RV, Budapest, Hungary. 2008. 137–157.

[23] Simmonds J, Gan Y, Chechik M, et al. Runtime Monitoring of Web Service Conversations. IEEE T. Services Computing (TSC), 2009, 2(3): 223–244.

[24] Walker RJ, Viggers K. Implementing Protocols via Declarative Event Patterns. SIGSOFT FSE. ACM. Newport Beach, CA, USA. 2004. 159–169.

[25] Wang F, Liu P. Temporal Management of RFID data. VLDB, 2005. 1128–1139.

[26] Wu E, Diao Y, Rizvi S. High-performance Complex Event Processing over Streams. Proc. of the 2006 ACM SIGMOD International Conference on Management of Data, ACM. Chicago, IL,

USA. June 27–29, 2006. 407–418.
DOI=http://doi.acm.org/10.1145/1142473.1142520

[27] Wu G, Wei J, Huang T. Flexible Pattern Monitoring for WS-BPEL through Stateful Aspect Extension. ICWS. Beijing, China. IEEE Computer Society, 2008. 577–584.

[28] Wu G, Wei J, Ye C, et al. Detecting Data Inconsistency Failure of Composite Web Services through Parametric Stateful Aspect. ICWS. IEEE Computer Society. Miami, Florida, USA. July 5–10, 2010. 68–75.

[29] Xu C, Cheung SC. Inconsistency Detection and Resolution for Context-aware Middleware Support. ESEC/SIGSOFT FSE 2005. 336–345.

[30] Xu C, Cheung SC, Chan W. 2006. Incremental Consistency Checking for Pervasive Context. ICSE. 2006. 292–301.

[31] Xu C, Cheung SC, Chan W, et al. Partial Constraint Checking for context Consistency in Pervasive Computing. ACM Trans. Softw. Eng. Methodol., Jan. 2010, 19(3): 1–61.
DOI=http://doi.acm.org/10.1145/1656250.1656253

[32] Yau SS, Wang Y, Karim F. An Adaptive Middleware for Applications in Ubiquitous Computing Environments. Real-Time Systems, 2004, 26(1): 233–238.