Expressive Exceptions for Safe Pervasive Spaces

Eun-Sun Cho* and Sumi Helal**

Abstract—Uncertainty and dynamism surrounding pervasive systems require new and sophisticated approaches to defining, detecting, and handling complex exceptions. This is because the possible erroneous conditions in pervasive systems are more complicated than conditions found in traditional applications. We devised a novel exception description and detection mechanism based on "situation"- a novel extension of context, which allows programmers to devise their own handling routines targeting sophisticated exceptions. This paper introduces the syntax of a language support that empowers the expressiveness of exceptions and their handlers, and suggests an implementation algorithm with a straw man analysis of overhead

Keywords—Exceptions, Safety, Programming models for Pervasive Systems, Pervasive Computing, Contexts, Situations

1. INTRODUCTION

The pervasive computing paradigm, one of theimportant advancements in modern computing models, has drawnconsiderable attentionever since it was born in 2000.Although not all the technical challenges it presented have been overcome, there are prospects growing thatits context aware and invisible computing approach canfacilitate our dailylives in various ways. These days, active works on its early applications in such domains as smart spaces, ubiquitous healthcare, and ubiquitous learning systems result in actual deployments.

However, it does not mean that pervasive computing systems areentirelywelcomed in our daily lives.People may feel uneasy abouthidden interactions with computers,worrying about incorrect and dangerous reactions from *non-intrusive* and invisible computing.Devices and networks engaged in pervasive systems are more vulnerable to side effects from physical environments, which could possibly lead to more serious damages with less robustness than traditional desktop computers.

Previous solutions torobustness problems in pervasive systems have focused mainly on the individual device and network faults, which appear exigent to pervasive systems. They adopted *fault management technology*, which was originally developed for continuous reasonable-quality operations of a system in the presence of faults. A system wide monitoriterates a closed control loop for error detection and proper reaction 0.

However, this approach is far from sufficient, since pervasive systems, like other automatic control and dynamic systems, are not simple collections of devices and networks0.Thus rather

Manuscript received August 29, 2011; accepted March 5, 2012.
Corresponding Author: Sumi Helal
* Dept. of Computer Science and Engineering, Chungnam National University, Daejeon Korea (eschough@cnu.ac.kr)
** Dept. of Computer and Information Science and Engineering, University of Florida, Gainesville Florida USA (helal@cise.ufl.edu)

Copyright ⓒ 2012 KIPS (ISSN 1976-913X)

than being dedicated on individual device and network faults, errors in pervasive systems encompass a wide range of undesirable ways of executions. Following theclassificationof undesirable statuses in pervasive systems according to the seriousness of side effects provides some of the following examples:

(1) Inconvenient or foolish results: these kind of errors cause unsatisfactory pervasive services. For instance, in the case where a user's PDA is broken, the user suffers from inconvenience (he may want to use a substitute). As another example, it would be unnecessary for a robot cleaner to be stuck or hovering around and under a bed.
(2) Annoying results: they are a little more serious than inconvenient or foolish cases, but are still tolerable. For instance, an alarm clock might be ringing noisily while the user lays sick in bed0.
(3) Dangerous errors: this category causes most serious results. For instance, a malfunctioning door might open and close repeatedly in an odd way. In another example, an automatic temperature control might heat up the room while a window is open, resulting in overheating and possible appliance damage.

Note that such classifications do not have absolute criteria, but may vary based on the applications and users' perception. For instance, a flickering light could be dangerous for an elderly woman who just had eye surgery, while for others it could just be annoying.

Furthermore,with a system-wide fault tolerant support that usually operatesunder the ignorance of the semantics of application and domain knowledge,remedies for faultsare constrained tonaiveactions--usually haltingthe applications or switching to using backup devices.

Thus to make a pervasive system robust, we must consider application and domain knowledge.*Exception handling*is one of the candidates to achieve this goal. Itallowsprogrammer-described abnormal cases to be detected and remedied according to semantic-aware handler programs. With programming language based exception handling tools, programmers are supposed to expressexceptions and handlers based on application semantics.

However, relatively little focus has been placed on exception handling mechanismsfor pervasive computing. In addition, a traditional exception handling mechanismin a general-purpose language (C++/C#/Java)is highly likely to complicate the application codes in pervasive computing. Most of all, it is inefficient for knowing the exact sources of the abnormality due to the intricate delegation of exceptions that areoften along the asynchronous call chains over interwoven devices and networks.

This paper is concentrating on improving exception handling mechanisms in pervasive computing. Our goal is to provide application programmers in pervasive computing with an expressive and efficient way to describe exceptions, which allows for keeping programs as more manageable and less error-prone. We believe that the suggested mechanism helps programmers' knowledge and expertise about the application (including domain experts who usually support the programmers) to play a pivotal role in determining what is erroneous and what could go wrong.

The remainder of this paper is organized as follows. Section 2 presents related work and positions our approach. Section 3 introduces situations as a powerful and useful concept as utilized by our definition of exceptions. Section 4 presents a programming interface of exception handling that is based on situations. Section 5 presents two algorithms that implement the pro-

gramming interface and presents a simple analysis of their performance. Section 6 concludes the paper and reports on our ongoing and future work.

2. Related Works

Unfortunately, existing efforts do not provide adequate mechanisms to address undesirable executions (unsafe states) with respect to a specific application. They seem to suggest that application-specific safety is achieved in astraightforward manner by using traditional language based exception-handling features.

However, detecting and handling erroneous cases in pervasive systems are challenging, since unlike C++/C#/Java, their exceptions are not limited to memory status violations. For instance, in general purpose languages, an exception can be raised after checking the value of a variable as follows:

```
flag = file.read(80);
if (flag < 80) raise exception;
```

In an application of pervasive systems, an error in the current context can be raised in the same style:

```
flag = heater.get("temperature");
if (flag != NORMAL) raise exception;
```

However, this style is not adequate for other more complex and probable exceptions that could form over time from various devices in a pervasive system. One way to handle such exceptions is to resort to directly code forking operations that make new threads to monitor the system over a period of time However, such an approach is not appropriate because the monitoring code is not isolated from the main logic, which will complicate the original application and make it difficult to test and debug.

Some previous works in pervasive systems extend the runtime systems with separate subsystems to detect and handle the errors from surrounding contexts 00. K. Damasceno 0supports a device agent for each device, which monitors its corresponding device to recognize errors and raise an exception if there is any. However, the limitation of this approach is lack of support for the errors that have been determined collectively from multiple devices. The following code is for a heater agent, which raises an exception if the heater seems to be experiencing failure:

```
Heat.start(userpref.get("Temperature"));
UnableToHeat unaheat = new UnableToHeat();
Unaheat.getContext().setStringProperty("Thermostat", "noanswer");
…
EHMechanism.throw(unaheat);
```

To overcome such limitations,0 introduces a mechanism to handle a wider range of exceptions. Instead of the exceptions from unit devices, they consider the following four kinds of predefined exceptions: (1) service discovery/reconfiguration failures, (2) service-level binding fail-

ures, (3) service-level exceptions, and (4) context invalidation. The first two types of exceptions arise in service discovery and binding that are found in service-oriented architectures0. The third type of exceptions is related to service invocation, which can be handled in ways that are similar to the remote service invocation (RMI)'s exception handling mechanism 0.

On the other hand, the last type of exceptions requires new mechanisms to handle them, since they address unique errors in the pervasive computing environment. As mentioned earlier, programmers' insight and knowledge are needed in order to detect and handle such errors. This is followed in 0, which provides their programmers with a new feature named "ContextGuard," which enables describing the conditions that must be maintained during the execution. Breaking the condition will cause a predefined exception named ContextInvalidationException. The following code shows a context guard that ensures that whenever a user changes the status of a room in a hospital, the user should remain with a doctor; otherwise an exception will be raised.

```
ContextGuard {
   When Event RoomStatusChangeEvent
   GuardCondition
      CurrentWard.isPresent(thisUser)
      CurrentWard.isPresent(members(Doctor))
}
```

However, in dealing with this kind of exception, the mechanism in0still has limitations. One of the noticeable restrictions is the expressiveness of the ContextGuard structure, which is based on First Order Logic (FOL) 0. Although programmers using FOL can describe snapshots of some contexts, they cannot use the same to express real and complicated scenarios and time based exceptions. Unlike traditional exception definitions that are normally defined as a single context, programmers in pervasive systems should be capable of capturing complex contexts that evolve and take shape over time. Repeated patterns that themselves are the erroneous execution (such as flapping doors and flickering lights) cannot be described in FOL.

```
ContextGuard {
        When Event LightChangeEvent
        GuardCondition
        // hard to described in FOL
        // "PreviousChangingTime is not 10sec before.
        //     This happens repeatedly"
}
```

Additionally, exceptions related to actuation (or the effect of actuation), such as "turn on the air-conditioner" cannot be immediately detected after the actuation instruction. However, they can be possibly detected several minutes afterwards that again cannot be captured without intertwining threads of control of the program, which is a complicated procedure.

This paper focuses on addressing this limitation and suggests an extensional notion of contexts called "situations." A situation is a temporal sequence of contexts, on which we base our safety approach. It provides a "natural" interface for programmers to express real and complicated scenarios and time based exceptions and handlers.

As a simple example, the light flickering problem can be described as a situation in which turnOn and turnOff events happen repeatedly every 10sec. or less from each other, as shown in the code segment below:

```
on exception(
        repeat([turnOn (l); turnOff(l)], 10sec])
) {… /* handler code comes here */ }
```

The main idea of situation-based exception handling was presented at the IEEE/IPSJ International Symposium on Applications and the Internet0 as a short paper. Without an actual page limitation, this paper gives detailed descriptions ofsyntacticstructures of exception handling and detection algorithms.

3. SITUATIONS: A USEFUL DERIVATIVE OF CONTEXTS

Situations are powerful abstractions that are utilized by our approach as follows:

(1) Situations enable programmers to define scenario-style exceptions, which are powerful means to capture complex semantics and temporal causalities among devices, services, and contexts.
(2) Situations facilitate the asynchronous detection of exceptions with respect to observation for a specific time period, which greatly simplifies programming.

Let us begin with brief examples. First, when a door repeatedly opens and closes with less than a 10sec. time-gaps, a situation for this is described as follows:

```
repeat([doorOpen(d); doorClose(d)], 10sec])
```

For the case when a user enters a room more than twice without a single exit, meaning that the output sensor might be broken or malfunctioning, the situation is described as follows:

```
[userEntered(u);!userExit(u); userEntered(u)]
```

Programmers can also describe a malfunctioning robot cleaner that is repeatedly getting into bed (less than) every 15 sec., no farther than 300mm from the bed, for more than two minutes, as:

```
[repeat(robot.("xpos")-bed.get("xpos") < 300mm &&
        robot.("ypos")–bed.get("ypos")<300mm), 15sec)] for 2min
```

The following code is the situation where too many (more than 20) users entered a room during a period (ten minutes). "e.type" is for userEntered, which is a predefined event type, and it is assumed to have "nomofevents" property for the cardinality of the same typed events:

```
[e = userEntered(u); e.type.get("numofevents") > 20] for 10 min
```

An example of asynchronous exception detection is a room temperature that should be checked three minutes from now:

```
[later_on [room.get("temperature") > 110F] after 3 min]
```

Before describing our approach, we clarify related concepts for "situations." A "context" is viewed as a mapping from context items to their values. "Context items" are attributes describing pertinent components of a pervasive space such as "humidity "and "temperature."

As suggested in0, a "fault" is defined as a context in which a device is not working correctly from the point of view of the system. Unlike0, we use an "exception" or "error" to refer to an unacceptable context (or a sequence of contexts) from the point of view of the application. Exceptions (or errors) may be caused by faults. We use "exceptions" and "errors" interchangeably in this paper, although "exceptions" is often used in the context of programming languages, whereas "errors" is used in a broader sense.

Context is a vector of values for context items, where a context item is a unit that has a value in a pervasive space. A history of contexts, along with selected events is called a situation. An exception occurs when a situation representing an undesirable condition that is disallowed by the application is detected.

3.1 Situation Patterns

Programmers responsible for describing exception-handler pairs must first specify the exceptions. To lessen the programmers' burden and to avoid tedious and erroneous repetition of similar exception descriptions, languages such as C++ and Java allow for constructing exceptions as objects. The subtyping between the exception objects denotes inclusion of the exceptional cases that those objects represent, which enables the hierarchical construction of exception handlers and greatly simplifies pairing between exceptions and handlers.

However, simple objects and their hierarchy are not sufficient for specifying context and situation based exceptions in pervasive systems. For instance, combinations of elementary exceptions (e.g., as temporal sequences) are likely to be helpful for describing exceptions. Those descriptions may also embody important system events unrelated to any contexts.

To reduce the burden of the description of complex exceptions, we introduce *situation patterns* – a regular expression based compositional description of exceptions in pervasive systems. The semantics of a situation pattern is an (infinite) set of situations that matches the pattern.

Detecting an exception is a process of matching the corresponding situation pattern to a sliding window over the system's event queue; if it matches, the exception is detected. Note that we do not look over the entire event queue from start to end as this could entail an enormous overhead. Instead, guided by the exception definition, we only look into some recent portions of the queue, which we call the *queue window*. If an exception is related to the most recent 10 minutes, the size of the window will follow and will also be 10 minutes.

Like building blocks, a situation pattern is constructed from selected base exceptions (context exceptions). Predefined event types can be automatically considered as situation patterns, which are unbreakable units.

The syntax of an exception definition starts by defining a related variable. We call this variable a *situation variable*, which has formal or actual parameters. These parameters will be uni-

fied with actual values or other parameters. The definition of an exception is embodied by a situation description that is assigned to the situation variable as shown below[1]:

```
exception _definition :=
        define exceptionsituation_var [ ( parameter* )]?:= situation_description;
```

A situation (pattern)[2] description denotes a pattern of temporal sequence of events and other situations. It is composed of the events that are necessary for the situation to be satisfied, including time operators as well as conditions that the subcomponents are subject to, in addition to other situations. In the situation description syntax, there is one simple and three composite situation descriptions as follows:

```
situation_description:= unit_situation
                      | filtered_situation
                      | aggregated_situation
                      | situation_description fornum
                      | later_on situation_description
```

"for num" represents the window size, which optionally specifies the sliding window size of the event stream to match the situation description. If it is omitted, a default window size will be applied.

In the following three sections, we will introduce the meaning and usage of each kind of situation.

3.2 Unit and Composite Situations

The description of a unit situation is the simplest definition of a situation description.

```
unit_situation :=  any
                 | event_function
                 | situation_var
                 | situation_var := event_function
```

"any" is the simplest definition of a situation, which matches any exception patterns.

"situation_var" can be an event function, which is a predefined type of event, like userEntered(), followed by parameters:

```
event_function := event_type [ ( parameter* ) ]
```

We assume that event types abort(), operationStart(),and operationEnd() are predefined. The last two types of events enable access to the execution status of an operation (action) through the properties of an event. Parameters can be actual values or formal variables, but for simplicity,

---

1 Reserved words are in typewriter font, and reserved symbols are underlined,while other symbols denotegrammatical-construction.

2 We will omit the word "pattern" in the name of this grammar symbol to avoid lengthy symbol names.

we consider only null parameters in this paper.

A variable name of other situations can be a unit situation. This kind of a situation itself may look redundant, but unit situations are usually combined with other features to make a composite situation description.

The assignment form of a situation variable and an event function can be also a unit situation. Similar to the case of two consecutive situation descriptions; a temporary situation variable is first described with an event function and a new unit situation is then derived from the situation variable. The following example shows that there is a unit situation description using the built-in event type userEntered():

```
e := userEntered()
```

A unit situation description is used for a set of situations, each of which is made of at most a single event. A composite situation description is used for situations composed of multiple events. A composite situation is not appended to the queue, but is conceptually inserted into the position immediately following the last matched event.

The first category of composite situations is for filtered situation descriptions, which have conditions attached to them:

```
filtered_situation := [ (situation_description)? : condition ]
```

The condition is a first order predicate empowered with some programming features like forall/foreach and optional quantifiers, which introduce bounded variables. The predicate itself is based on event data; of which the exact definition is dependent on the specific language used. In the later part of this paper, we assume Java syntax:

```
condition := (forall var | foreach var | some var)? predicates using event data
```

For example, a simple filtered situation with conditions can be described as follows:

```
define exception UserEntered(User u) :=
            [e:= userEntered():u.id == e.uid;];
define exception guestEntered(User u):=
            [userEntered(u): u.id != owner];
```

This has userEntered(A)as a matched situation where A is a user ID.

The second category of filtered situations is an aggregated situation description for the sophisticated temporal assets of situations. It is theoretically based on the combination operators of temporal event logic0. Typical approaches to combine two events are ANDing (two situation descriptions match at the same time), ORing (one of the two situations matches), and Negation (no matching situation exists). In addition, descriptions of sequences of situation patterns as well as repeating situation patterns are useful. These basic temporal notions constitute these first five definitions in the aggregated_situation, as described below. For instance, “;” simply means the order of the two situation occurrences.

```
aggregated_situation := situation_description && situation_description
                      | situation_description || situation_description
                      | not situation_description
                      | situation_description ; situation_description
                      | ( situation_description )*
                      | situation_with_time_constraint
```

While the above five expressions do not assume time constraint or intervening event-related constraints, they do require a sliding queue window, the size of which is defined in the situation description using the "for num" phrase. If this phrase is omitted, the system will use a default window size. For instance, "s_1&&s_2" means both s_1 and s_2 match within the current window of the queue at the point of checking against this exception.

For the description of time constraints, time can qualify a single situation description or connect two situation descriptions together. Time constraints are made of time-operators together with constants and "time_variables.""#T" is a special variable denoting the duration of time (in milliseconds) (e.g., "#T < 100"). "%T" is a special variable denoting a specific time (e.g., "%T == 12:00:00/06/03/2010") We assume a global clock for simplicity.

```
situation_with_time_constraint := (situation_description)? time_description
                                   situation_description
                                  |repeat ( situation_description, time_description )
                                              (morethan num times)?

time_description := time_duration time_operator num
                  | time_stamp time_operator time_data
time_duration :=   #T
time_stamp :=      %T
time_operator :=   ≤ | ≥ | == | nearnum with
time_data :=       num :num :num /num / num / num
```

3.3 Advanced Situation Patterns

In addition to allowing for the basic description of situations, our proposed method allows the asynchronous exception handling code to be attached right after the corresponding invocation. This approach provides a simpler programming interface to programmers, and achieves more "separation of concern" 0than existing asynchronous exception handling support0.

For instance, let us assume a context update through the service invocation of aircond.set(on) , where the variable aircond is bound to an air-conditioner.

```
define exception too_hot:=
                    [: $. get("place").get("temperature") > 110F];
aircond.set(on)
          on exception ([later_on [#T == 2min: too_hot]]) {…}
```

Failure of aircond.set(on)is detected two minutes after the invocation by checking if the situation too_hot is matched.

Since "#T==" or "#T>" phrases are mandatory in the "later_on" situation description, we pro-

vide a simple syntactic sugar that is denoted by “after,” as shown below:

```
aircond.set(on) on exception ([later_on too_hot after 2 min]) {…}
```

A second feature is *parameterization*, which encourages the reuse of commonly used situation descriptions. The following situation definition refers to a specific pattern of situations happens frequently. To make it general enough for reuse, it parameterizes the window size (tt), the time gap between the two situations (t), and the situation (e).

```
define exception frequent (e, t, tt) :=[e (#T < t ; e)*] for tt;
```

By substituting the input parameters with real values, a situation description can yield more concrete situation descriptions. For example, frequent_entering is used for situations where people enter the smart place frequently:

```
define exception frequent_entering :=
                        [frequently(user_entered(u), 60, 300];
```

In the next example, the situation description e is assigned to the same user ID. It can be realized by attaching an imperative programming style condition with the keyword “forall.”

```
define exception frequent_same_user (e, t, tt) :=
      [e (#T < t ; e)*:
         forall i e[i].get(“user”) = e[i+1].get(“user”)];
```

Note that the input parameter e is used twice in the description, and both occurrences of e match the same situation description. However, they can represent different instantiations, e[1]and e[2], if necessary. In addition, it embeds the repetition of e in “(#T < t ; e)*,” so that e[2]can be instantiated again by e[2][1], e[2][2], e[2][3]...e[2][n], in case a series of n situations matching e are detected. In the following example, the same user u enters the roomfrequently for some reason:

```
define exception frequent_entering_of_same_user(u) :=
[frequent_same_user (user_entered(u), 60, 300]
```

4. DEFINING HANDLERS OVER SITUATIONS

One challenge to the broader adoption of exception handling among programmers is the independence of exception handler writing from the main control flow of the application. Thus we propose that programmers define exception handlers while they define exceptions, entailing that exceptions and their handlers are separate from the main application. This strategy may simplify the development of safer and more robust applications 0.

For instance, the exception “too_hot” can be handled with pseudo code commands such as

"chill the system" and "notify the administrator," as follows (where the concrete routine is assumed to be in an abbreviated Java like language). In the following example, temperature over 110F would run a cooler and notify the administrator.

```
define exception too_hot :=
            [: $. get("place").get("temperature")> 110F];
too_hot.handler[1] = {chill_the_room; notify_the administrator}
```

However,real-world situations are more complicated because programmers cannot have knowledge about exception handling ahead of the actual exception checking (occurring) point. For instance, the same fire exception may be handled by halting the system in some cases, while in other cases it would be handled by cooling.

Thus we introduce a list of exception handler concepts, providing multiple handler options that the programmer can choose from while developing the main logic. In the following example, there are two options of handlers, numbered [1]and [2]for when the temperature is too high:

```
define exception too_hot :=
                 [: $.get("place").get("temperature") > 110F];
too_hot.handler[1] = { chill_the_room; notify_the _administrator}
too_hot.handler[2] = {halt}
```

In addition to basic handler definition capabilities, important issues to consider related to handlers are: (1) the range of exceptions; (2) frequency of exception monitoring; and (3) how to compose handlers. The remaining subsections cover these issues and introduce our solutions.

If an exception is raised within a specific block (like "try{}" in Java), the appropriate exception handler associated with the block processes the exception. In this paper, we assume that programs in pervasive systems are basically a set of ECA (Event-Condition-Action) rules, and the unit of code fragment that is bound to exception-handler pairs is a simplified case of an ECA rule. Thus the exception too_hot can be used as follows, where the *situation_variable*[1] is the first handler selected at this point in the program.

```
on event (…) condition (… ) action {
     …
} on exception(too_hot[1])
```

Our proposed mechanism allows global exception-handler pairs to be bound to the entire program. Such exceptions are usually related to real world safety. In the following code fragment, the previous example of abnormal temperature detection is re-used, with the only difference being that the "on exception" phase is stand-alone rather than being bound to particular ECA rules:

```
define exception too_hot :=
      [: $. get("place").get("temperature") > 110F];
too_hot.handler[1] = {
      chill_the_room; notify_the _administrator
}on exception (too_hot[1]) ;
```

5. IMPLEMENTATION ALGORITHMS

Since exceptions in pervasive systems need sophisticated temporal pattern description and matching, using Complex Event Processing (CEP)0 techniques to handle nested events will be useful. However, traditional CEP requires the frequent processing of events and continuous queries on the event stream, which degrades performance.

Recently, data stream techniques have been proposed for handling event matching over continuous queries with enhanced performance 0, 0. However, as in SQL, their techniques focus mainly on approximate and statistical data aggregations, and do not address the matching of multiple sequenced queries or nested patterns 0.

To support the situation concept efficiently, our system is required to handle both nested events like CEP as well as continuous queries found in stream database systems. In addition, our implementation algorithm should not incur any unnecessary overhead, and should exploit any available source of saving in performance overhead in terms of the frequency of event processing and situation assessment.

In the following section, we introduce a basic algorithm for the processing of events and the evaluation of situations. We assume that runtime events are placed in a queue of events as they occur via basic push, pull, or combined push/pull mechanisms as suggested in 0. Following the basic algorithm, we present an improved algorithm (the Boxing Algorithm) that exploits additional situation semantics.

5.1 Basic Algorithm

Since situation description is similar in structure than a regular expression, our algorithm uses FSA (Finite State Automata) as a state transition mechanism. We have a set of transition rules for each structure of situation descriptions. The simplest transition rule might be "e_1&& $e_2 \overset{e1}{\Rightarrow} e_2$," which means if e_1 is detected then the rule "e_1&& e_2"is partially matched and the system is waiting for e_2.

However a simple FSA is not sufficient for detecting a situation. One of the important reasons is that continuous evaluation needs a new FSA for a situation description every time the event stream is evaluated, which does not seem to be practical. In addition nested structured situations cannot be straightforwardly detected by FSA.

Moreover, we have to hold more than one state in the FSA for the condition evaluation of the filtered situations, otherwise backtracking would be largely involved. For instance, in the situation description [e1&& e2: e1.get("name")== e2.get("name")], events in the order of e2(*m*)-e1(*m*) will match the situation, when m and n denote the names of the events and e1 and e2 are types of events. But if events enter the system in the order of e1(*m*)-e1(*n*)-e2(*n*), we have to ignore e1(m), backtrack and start a new matching process for the sequence of e1(*n*)and e2(*n*). In addition, e1(*m*)-e1(*n*)-e2(*m*) should also make us hold e1(*m*) to meet e2(*m*), and ignore e1(*n*).

To avoid backtracking, which is very expensive in terms of performance, we devised eFSA – an extended notion of FSA, which is a merged form of multiple FSAs for all possible states for each situation. First, we extend the FSA system and transition so that we use a *collection* of possible states of a traditional FSA as a state in eFSA (state of states). It extends $\overset{e}{\Rightarrow}$ transition to represent the transition from/to the collections of states. To avoid confusion, we call a single state in traditional FSA an "item," while we use a "state" for the collection of the items. Thus, a state holds all possible items matched with currently arriving events. For instance, for the situa-

tion descriptions [e1&& e2]and [e1; e3] once e1 enters the system, the next state should include [e2]and[e3],which represents the remaining events to be matched.

One of the benefits of eFSA is reducing redundant items that could arise from the use of multiple FSAs. In addition, it is much more efficient to handle the dynamic addition/deletion of items as will be shown next. Our adoption and use of eFSA is mainly influenced by the famous LR parsing algorithm, which is widely used in compiler construction 0. Like an LR parser designed for context free grammar supporting nested rules, our eFSA is devised to handle nested structures of situations (described in context free grammars), rather than limited regular expressions, which are equivalent to FSA's.

The basic algorithm is based on maintaining the set of situation items for an application A (I_A), which is changed each time the system processes the event queue. More formally, the set of situation items I_U is defined as follows, where I_A is a subset of I_U:

Definition 1*A situation item* $i \in I_U = \{\langle s_1, s_2, b, max \rangle \mid s_1 \in \text{Situation}_U,\ s_2 \in \text{Situation}_U \cup$ "**accept**,"$b \in \text{Var} \rightarrow E_{ID},\ max \in \text{Timestamp}\}$, where Situation_U is the set of all possible situation descriptions, and Var is the set of variable names used for the condition evaluation of a situation. E_{ID} denotes the set of events arriving at runtime, whereas *max* is the current time plus application specified window size.

Usually s_2 is a sub-component of s_1, denoting the remaining portion of a situation by eliminating the currently matched prefix from s_1.

For instance, "⟨e1&& e2, e2, {⟨e1,#6⟩}, 1200⟩" represents a situation description of "e1 && e2" with a window size of 1,200 msec, when the current time of*t* is 0. This item describes the situation where after event #6 is matched with the situation (the type of event #6 matching that of e1), then the system should anticipate and await for e_2 to arrive in order to detect the situation in this example. This situation item will expire 1,200 msec after its insertion to I_A.An application specified window size is given by the "fornum" structure in the situation description.Its concept is the same as the window size in CEP or stream database systems, and is used for the detection of a sequence of events. Note that, without the time limit of the duration of a time period, the system has to hold the entire event stream for good in order to detect combinations of multiple events like "e1&& e2."This is why anapplication-widedefault window size is usuallyprovided for unspecified situations.

A description of the algorithm in terms of situation items over time is depicted in Fig. 1. At each time point *t*=0, *t*=1, *t*=2,…, a long gray rectangle is assigned, where the ovals representthe situation items held in the corresponding time point. For simplicity, we do not include variable binding details. The first situation of an item, which is not varying as time goes by, is also omitted in the figure. The explicit keyword **max**is used for the*max* of the situation items.

Fig. 1 assumes that a programmer wants to detect two situation patterns e1&&e2and e1;e3. Let us call these situations*target situations*. At the time $t = 0$, I_A is initialized by two initial items ⟨e1&&e2, **max**=3⟩and ⟨e1;e3, **max**=2⟩. Note that theseitems are repeatedly inserted into the gray boxat every following time point, as denoted by brackets in the figure, which forces continuous matching incoming events with the target situations.

As the time point is shifted to the next point, each item is transformedalong the different arrows, according to its cases. (1) The bold arrows and double arrows are for the case in which some prefix of the situations in the item is matched,(2) the dotted arrows represent the case where no prefix is matched, but the item should be held because it might be matched after-

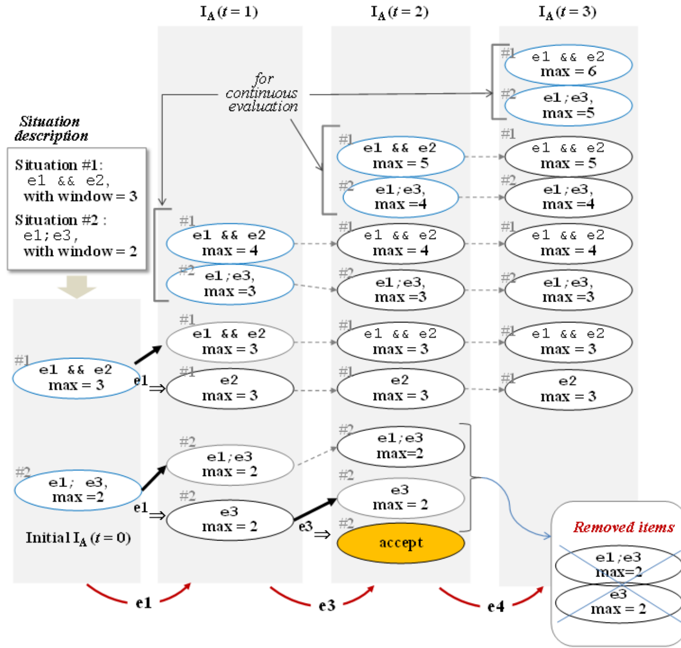

Fig. 1. Transition of I_A over timein a basic algorithm to match the sequence of events against multiple situation patterns

ward,and (3) the curved arrows denote that the item becomes removable because the lifetime of the item is expired.

In the case (1), a single item spawns two new items at the next time point. For instance, an item with a situation "e1 && e2"matched with the current event e1 moves along the bold arrow, resulting in a shorter situation ("e2") by eliminating the matched prefix. It also spawns the same item as in the previous time point (denoted by a double arrow)in case the matched prefix does not satisfy the condition part of a situation description. This arrow is necessary for back tracking cases, as in example of e1(m)-e1(n)-e2(n). (Note that our matching process concerns only types ofevents, but not the evaluation of condition part of a situation pattern with parameters. It has room for improvement, which is included in our future works.)

The maximum frequency possible for processing events (which should lead to the most responsive detection) cannot exceed the inverse of the minimum amount of time required for the processing of one event. We assume the maximum frequency in the illustration in Fig. 1. We also assume that I_A is initialized by two situation descriptions (⟨⟨e1&&e2, **max**=3⟩and ⟨e1;e3, **max**=2⟩), and the initial time t is assumed to be 0. And we assume that one event occurs at each transition to the next time point and in the bottom of the figure event e1-e3-e4 occurs sequentially, as is shown with curved arrows.

At time t = 1 and as an event whose type is the same as e1enters the system, a transition occurs and I_A is updated as follows:

- New items are inserted as the sub-portion of the situation is matched (⟨e2, **max**=3⟩ and ⟨e3, **max**=3⟩, shown in thin solid lined ovals in Fig. 1.) This is obtained by a predefined

$\overset{e}{\Rightarrow}$ transition, whose details are omitted for brevity.

- Items with initial values are re-inserted as shown in blue lined ovals (⟨e1 && e2, **max**=4⟩and ⟨e1;e3, **max**=3⟩ with a big arrow and an oval in Fig. 1) for continuous evaluation. Such re-insertions are made every time the event stream is evaluated. With every future re-insertion, the maximum time limit is adjusted accordingly (at t = 1, **max** values are changed to 4 and 3, instead of 3 and 2 at t = 0).
- Older items remain in I_A, ⟨e1 && e2, **max**=3⟩and ⟨e1;e3, **max**=2⟩ as shown in bold solid lined ovals). They remain after the transition just as they were, even though they matched the current event. Thus every step of transition involves holding onto matched items to avoid backtracking.

At time t=2, right after an event with the same type as e3 enters the event queue, I_A is updated as follows:

- The situation "e1; e3" is matched, thus ⟨e3, **max**=2⟩ in the previous I_A (that is, $I_A(t{=}1)$) makes a transition to "accept," which means that one situation description is matched by type and therefore an evaluation of condition will begin (solid yellow oval).
- The dotted ovals (⟨e1 && e2, **max**=3⟩, ⟨e2, **max**=3⟩ and ⟨e1;e3, **max**=2⟩) are the situation items remaining from $I_A(t{=}1)$ because they are not related at all with the event e3 being processed.
- As in $I_A(t{=}1)$, $I_A(t{=}2)$ also has newly created items with initial situation descriptions (in a blue lined circle)

```
IA = φ
for each time (according to maximum possible frequency)
        // get rid of expired items
        IA = IA – {⟨s, s', b, max⟩| ⟨s, s', b, max⟩∈ IA
                  and    max < current_time}
        // insert initial items with max as current_time+w_user
          IA = IA + {⟨s, s, ⊥, current_time + w_user⟩ |
                         s ∈ SituationA, w_user is user defined window size}

        if (no unprocessed event remains) break;
        or else for each event e
                  // execute the transition to get a new IA
                  IA = { i' | i e⇒ i', i ∈ IA }
                  for all   ⟨s, "accept", b, max⟩∈ IA
                         c = condition_of (s)  // c is condition part of s
                         tv = evaluate (c, b)  // evaluate c with binding b
                         if (tv is true) raise s
                         IA = IA - ⟨s, "accept", b, max⟩
                  end of for
        end of else
end of for
```

Fig. 2. Basic algorithm for I_A evaluation

- As in $I_A(t=1)$, previous items still remain in I_A (⟨e3, **max**=2⟩ in thick solid lined oval), even though the event type is matched as in $I_A(t=1)$. This redundancy is intended as we mentioned to avoid backtracking.

$I_A(t=3)$ includes more dotted ovals since e4 is not related with any situation descriptions. Also, ⟨e1; e3, **max**=2⟩and ⟨e3, **max**=2⟩ are eliminated after transitioning on e4 at t=3, since their valid period ended at t=2, according to the application specific window size. The basic algorithm demonstrated in Fig. 1 is shown in Fig. 2.

I_A is initialized with situation descriptions and its application specific window sizes. At each timeiterating the outermost loop on the event stream, expired items are eliminated from I_A, and old situation items are reinserted with modified *max* values to I_A. Also new items are created and added as transitions occur (e.g., "$\overset{e1}{\Rightarrow}$ transition"). If an "**accept**" item is created, the system evaluates the corresponding condition part of the situation description and raises an exception if the evaluation result comes out to be *true*.w_{user} denotes user defined window size. To simplify the analysis, we assume that only one user window exists.

5.2 Boxing Algorithm

This section presents the Boxing algorithm, which is an improvement over the basic algorithm presented in the previous section, in the following two respects:

(1) Time constraints in the situation descriptions can be further exploited to reduce the size of I_A, by ignoring items that cannot occur at certain times according to these constraints. For instance, events such as waking up in the morning or eating a meal cannot be followed by the same events. In fact, and at a much lower scale, some types of sensors (e.g., RFID) need some minimal time to "forget" or "consume" a sensed event before detecting another (refer to the UserEntered event example in Section II). More explicitly, situation descriptions could provide such constraints conveniently (e.g., "later on ~ after"), which nicely allows for ignoring the corresponding situation items for a calculable amount of time.

(2) The frequency of update of I_A (outer loop in Fig. 2) can be significantly reduced byexploiting additionalexplicit information about situation items. Specifically, if certain types of events known prior towill not occur for some period of time, we can utilize this information to slow down the update frequency without loss of promptness. We refer to the resulting slower frequency (its inverse to be more accurate) as the "evaluation window,"which is different from the application-defined windoww_{user} (related to max). The former is unified over an application or a system, while the application defined window is situation-specific.

The improved algorithm is presented in Fig. 3. Situation items are different from the basic algorithm, with an additional field named *min*, which means the earliest time for evaluating the item. If no explicit time parameters like "later on~ after" or the minimum time from the sensor property exists, *min* is usually set to the current time. We call the pair of this *min* value and the *max* introduced in the basic algorithm a *time span* of a situation item, which focuses a situation evaluation within a box or an envelope (hence, we refer to our improved algorithm as the *Boxing algorithm*.)

In the algorithm, we denote the evaluation window size as w. Each window has its own temporal I_A, called I_W, which exists only for that window. From I_A, I_W is selected by eliminating those items of which time spans are not intersected with the current window. Whenever one window processing is completed, the new I_A is evaluated with a new min-value, based on *fre-*

```
I_A = φ
for each window with size w
        // remove expired items
        I_A = I_A – {⟨s, s', ⟨min, max⟩, b⟩ |
⟨s, s', ⟨min, max⟩, b⟩∈ I_A    and    max < current_time}
// insert the initial items with max      as current_time+w_user
        I_A = I_A + {⟨s, s, ⟨current_time, current_time+w_user⟩ , ⊥⟩|s ∈ SituationA}
        I_W = {⟨s, s', ⟨min, max⟩, b⟩| ⟨s, s', ⟨min, max⟩, b ⟩∈ I_A
                                        and min  ≤ current_time + w}
I_A = I_A - I_W

        if (no unprocessed event remains) break;
        or else for each event e
                //execute the transition to get a new I_W
                I_W = {i' | i =e=> i', i ∈ I_W}
                for each ⟨s, "accept", ⟨min, max⟩, b⟩∈I_W
                        c = condition_of (s)    //c is condition part of s
t_v = evaluate (c, b)  //evaluate c with binding b
if (t_v is true)    raise s
I_W = I_W - ⟨s, "accept", ⟨min, max⟩, b⟩
      end of for each

I_A = I_A- I_W
IA = {⟨s, s',⟨MIN(last (e)+1/frequency (e), min), max⟩, b⟩|
   ⟨s, s', ⟨min, max⟩, b⟩∈ I_A& type(e) ∈lookahead (s')}
      end of for each
    end of if
end of for each
```

Fig. 3. Boxing algorithm for I_A evaluation

quency(*e*) and *last*(*e*). *frequency*(*e*) is the frequency of occurrences of events of the same type as *e*, and $t \in last(e)$ is the time of last occurrence of event of type *e*. *lookahead*(*s*) means the first expected event(s) needed to match a certain situation description, which are calculated based on $\overset{e1}{\Rightarrow}$ transition rules prior to execution.

5.3 Performance Analysis

We now evaluate and compare our algorithms based on the following straw man performance model and notations:

n_s : # of situation descriptions of an application
n_e : average # of events in an evaluation window per unit time
m_s : average # of items that are accepted per a unit of time (whether or not the condition is true)
r_s: ratio of the increment of number of items after a $\overset{e}{\Rightarrow}$ transition
r_B: ratio of the # of the remaining items after the boxing (based on the expected next occur-

rence time) over the # of the original items
r_w: ratio of selection from I_A to form I_W

The situation detection time for a time quantum Δt is proportionally tied to the number of situation items and the number of events arriving within Δt. The overhead is consumed in searching the I_A and I_W spaces to find matching items. Thus, where c_0 is the cost to process each item:

$$T_{overhead} = \text{\# of items} \times c_0 \times n_e \times \Delta t. \quad (1)$$

In the Basic algorithm, if we denote $I_A(j)$ to be the number of items in the jth evaluation on the event stream, then the number of items for I_A can be estimated as follows:

$$I_A(0) = 0,\ I_A(1) = n_s$$
$$I_A(j) = (I_A(j\text{-}1) - expired_j + n_s)(1+r_s)^{ne\times(j-1)} - m_s,\ \text{if } j>1 \quad (2)$$

To calculate $I_A(j)$, we first subtract $expired_j$, which is the number of expired items from $I_A(j\text{-}1)$ and add n_s. That is, the number of inserted items that are initialized with the situation descriptions. This number increases by the rate of r_s after $\stackrel{e}{\Rightarrow}$ transition and powered by ($n_e\times$ (j-1)), which is the number of events that have arrived until that time. Then, the number of the matched items (ms) will be eliminated from I_A whether the condition part with the binding b is satisfied or not. If a situation item i was inserted into I_A at w_{user} before a certain time point, i, then those items created by the $\stackrel{e}{\Rightarrow}$ transition from i until that point in time will be eliminated. Since new items initialized with situation descriptions are inserted into I_A every time point, $expired_j$ can be further formulated as follows:

$$expired_j = 0\ ,\ \text{if } j < w_{user}$$
$$= n_s\times(1+r_s)^{ne\times wuser},\ \text{otherwise} \quad (3)$$

For the Boxing algorithm, the number of items can be estimated as follows: let us denote $I_A(k)$ and $I_W(k)$ to be the cardinalities of I_A and I_W in the kth evaluation on the event stream. Note that k is different from j in the Basic algorithm since the time for the k window = $w\times$ j^{th} unit time with window size w. $I_W(k)$ is learned by the following formula ($expired_k$ is the same as in the Basic algorithm.)

$$I_A(0) = 0,\ I_W(1) = n_s\times r_w,\ I_A(1) = n_s\ '\ (1+r_s) - m_s$$
$$I_W(k) = ((I_A(k\text{-}1) - expired_k + n_s)\times r_B)(1+r_s)^{ne\times w\times(k-1)} - m_s$$
$$I_A(k) = (I_A(k\text{-}1) - expired_k + n_s)\times(1 - r_w) + I_W(k)\ ,\ \text{for } k>1 \quad (4)$$

At the beginning, $I_W(1)$ is inferred directly from the situation description, and no expired items exist. After that, at each window, to obtain $I_W(k)$ we subtract $expired_k$ from the previous $I_A(k\text{-}1)$ and add ns to it. Then we select only ther_B portion of the result due to boxing. This number will be increased by rate r_s and powered by the number of events. $I_A(k)$ is then adjusted reflecting the changes brought about by the new $I_w(k)$.

Fig. 4(a) shows the expected time overhead in each algorithm according to the time elapsed. The overhead exponentially increases with time, but the Boxing algorithm eliminates more

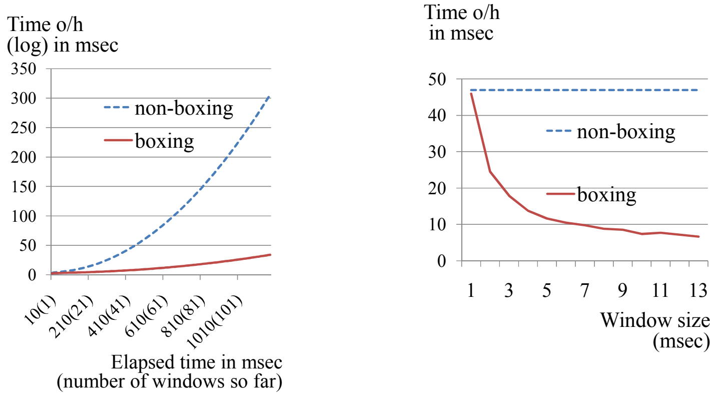

Fig. 4. (a) Time overhead according to elapsed times and (b) Time overhead according as (evaluation) the window size is growing0

items, which slows down the overhead curve. We assume that 1event comes every 1 msec, 1 out of 100 incoming events are matched to the current window and to 10situation descriptions. The processing time per an event is assumed to be10.0 msec 0, and 80% of the items are meaningful for the current window.

Fig.4(b) shows the expected time overhead and data loss of each algorithm as the window size is increased. It also shows the data loss when we use the average time of the arrival of events (such as the minimum of time spans of items), which is easier to estimate. We can see that the time overhead is decreasing as the window size grows. Thus,a bigger window size is better as long as it does not impair responsiveness. This is achieved by taking the smallest among the expected inter event times of all events as a window size.

Fig.5(a) shows how time overhead reacts to the inter-arrival time between events. It shows the Boxing algorithm to be more robust in face of eventful systems than the Basic algorithm.If some degree of irresponsiveness is tolerable or acceptable, the average arrival time of events can be

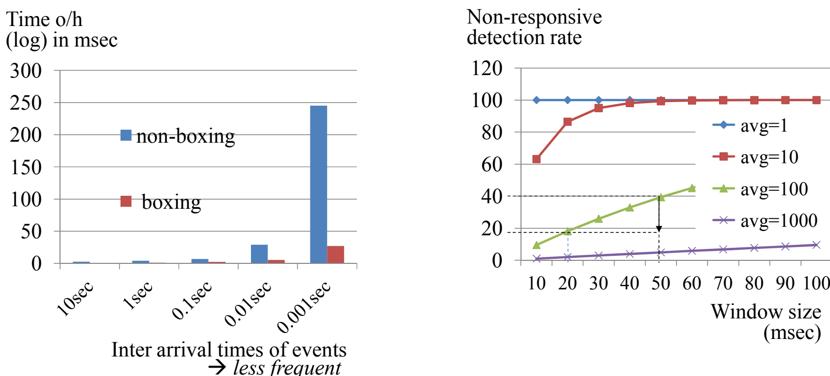

Fig. 5. (a) Time overhead according to inter event times, and (b) Non-responsive detection rates according to window sizes 0

used as a direct hint for choosing the window size. In this case, detection might be either missed or delayed because next events may arrive earlier than the average.When we assume that the arrival of events follows an exponential distribution, the probability that the actual waiting time is less than the window size is $1\text{-}e\text{-}^{(1/avg)\times w}$ where *avg* is the average inter event time. Fig.5(b) depicts the variation of this non-responsive detection rate according to the inter event time, showing the result of using the average of inter-arrival time of events as a window size instead of the minimum inter arrival time. Note that, if the average arrival time is 50 msec, more than 60% of the events are missed or delayed for detection. However, such a loss of time accuracy can be reduced if we take less than the average as the window size. If we take 20 msec as a window size rather than 50 msec, the rate of loss of time accuracy drops to around 30% when the average arrival time is 50 msec (as indicated by the arrow in the graph).

5.4 Architecture

During the development phase, programmers who utilize our methodare supposed to describe the exceptionsas situation patterns. To describe their related handlers within a given applicationa named exception is defined as a situation pattern, and handlersare defined and bound to the named exception as mentioned earlier.

During application provisioning (activation), as shown in the lower left part of Fig. 6, a Situation Rewriter pre-processor translates exception definitions into normal forms and then eFSAs. It then initializes the state transition tools used by the situation manager to execute the eFSAs at the run time.

As the system begins at run time, contexts are monitored and are buffered into an event queue by the Application Event Queue Manager, where situation patterns are searched for and identified based on the State Transition Engine for the registered eFSA. If the matching "catches" a situation pattern, the corresponding handler will be selected and executed by the handling engine.

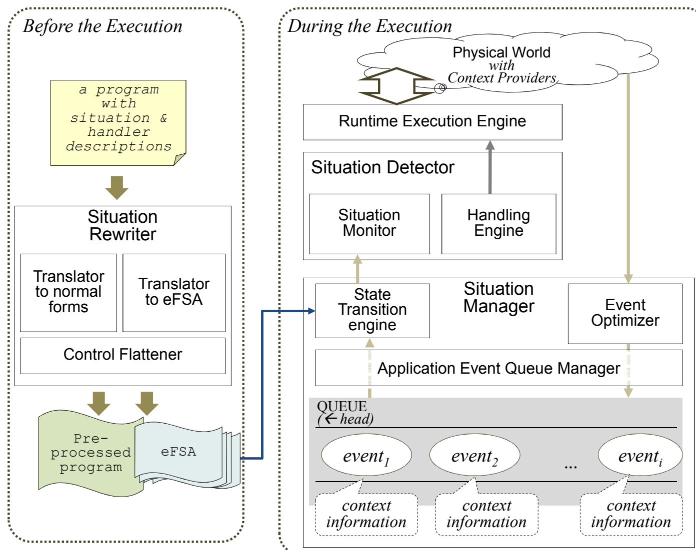

Fig. 6. Flow of the situation handling system

The matching process can be optimized during the eFSA generation phase before execution, or can be optimized based on the runtime status.

Preliminary partial implementation employsKnopflerfish 1.3.5 0 and ANTLRv3 0 on top of the ATLAS platform 0.

6. Conclusions

It is important to describe an exception with a powerful language model, because the possible erroneous conditions in pervasive systems are more complicated than conditions found in traditional applications. This paper proposes a novel exception handling mechanism in pervasive systems, which allows programmers to devise their own handling routines targeting sophisticated exceptions. Our approach utilizes situations, which is a novel extension of content. This empowers the expressiveness of exceptions and their handlers. We presented the syntax of a language support along with implementation algorithms. We also provided a straw man analysis of the performance of the algorithms in terms of overhead.We believe our work is an important starting step to enabling exception handler writing for pervasive systems programmers.Our future work will focus on enhancing the algorithms with optimization techniques in stream database systems.

References

[1] S. Bruning, S. Weissleder and M.Malek, "*A Fault Taxonomy for Service-Oriented Architecture*," *Proceedings of the10th IEEE High Assurance Systems Engineering Symposium*, Dallas, Texas, USA, November, 2007, pp.367-368.

[2] Safety Research & Strategies Inc., "*Toyota Sudden Acceleration Time Line*", http://www.safetyresearch.net/toyota-sudden-unintended-acceleration/toyota-sudden-acceleration-timeline/

[3] H.-I.Yang and Sumi Helal,"*Safety Enhancing Mechanisms for Pervasive Computing Systems in Intelligent Environments*,"*Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, Hong Kong, March, 2008, pp.525-530.

[4] K.Damasceno,N.Cachoand A. Garcia, A. Romanovsky and C. Lucena," *Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case,"Proceedings of Software Engineering for Large-scale Multi-Agent Systems*, Shanghai, China, May 2006,pp.37-44.

[5] D. Kulkarni,and A.Tripathi,"*A Framework for Programming Robust Context-Aware Application*," *IEEE Transactions on Software Engineering*, Vol.36, No.2, 2010,pp.184-197.

[6] Oracle, "*Java RMI Remote Exception*,"http://java.sun.com/j2se/1.4.2/docs/api/java/rmi/RemoteException.html

[7] A. Ranganathan and R. H. Campbell, "*An infrastructure for context-awareness based on first order logic*", *Personal and Ubiquitous Computing*, Vol.7, No.6, 2000, pp.353-364.

[8] E.-S. Cho, S. Helal, "*A Situation-based Exception Detection Mechanism for Safety in Pervasive Systems*", *Proceedings of 11th IEEE/IPSJ International Symposium on Applications and the Internet*, Munich, Germany, July 2011.

[9] B. Randell,"*Dependable pervasive systems*,"*Proceedings of 23rd IEEE International Symposium on Reliable Distributed Systems*, Florianpolis, Brazil, October, 2004, pp.2-2.

[10] I. Cervesato, M. Franceschet and A. Montanari, "A *Guided Tour Through Some Extensions Of The Event Calculus*", *Computational Intelligence*, Vol.16 No.2, 2000, pp.307-347.

[11] M. Lippert,C.V. Lopes,"*A study on exception detection and handling using aspect-oriented programming*,"*Proceedings ofInternational Conference on Software Engineering*, Limerick, Ireland, June 2000, pp.418-427.

[12] "*AsyncCallback Delegate-.NET Framework Class Library,*"*MSDN*, http://msdn.microsoft.com/en-

us/library/system.asynccallback(v=VS.71).aspx
[13] P. R. Pietzuch, B. Shand and J. Bacon, "*A framework for event composition in distributed systems,*"*Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Rio de Janeiro, Brazil, June 2003, pp.62-82.
[14] Ja. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "*Efficient Pattern Matching over Event Streams*,"*Proceedings of ACM SIGMOD conference*, Vancouver, BC, Canada,June 2008, pp.147-160.
[15] S.White, A.Alves,D.Rorke,"*WebLogic event server: a lightweight, modular application server for event processing,"Proceedings ofSecond international conference on Distributed Event-based Systems*,Rome Italy, July 2008, pp.193-200.
[16] M. Liu, M. Ray, E. A. Rundensteiner and D. J. Dougherty, "*Processing Nested Complex Sequence Pattern Queriesover Event Streams*,"*Proceedings of the 7th Workshop on Data Management for Sensor Networks*, Singapore,September 2010, pp.14-19.
[17] C. Chen, Y. Xu, K. Li and S. Helal, "*Reactive Programming Optimizations in Pervasive Computing*,"*Proceedings of 10th IEEE/IPSJ International Symposium on Applications and the Internet*, Seoul, Korea, July 2010, pp.96-104.
[18] A. Aho, R. Sethi,M. S. Lam and J. Ulman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., Prentice Hall, 2006, pp.241-246.
[19] N. H. Cohen and K. T. Kalleberg, "*EventScript: an event-processing language based on regular expressions with actions*", *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, Tucson, USA, June 2008, pp.111-120.
[20] R. Bose, J. King, H. El-zabadani, S. Pickles, and A. Helal, "*Building Plug-and-Play Smart Homes Using the Atlas Platform*,"*Proceedings of the 4th International Conference on Smart Homes and Health Telematic* (ICOST), Belfast, the Northern Islands, June 2006, pp.265-272.
[21] "Knopflerfish-Open Source OSGi,"*The Knopflerfish Project*,http://www.knopflerfish.org/
[22] "ANTLR Parser Generator v3,"*ANTLR Project*,http://www.antlr.org

Eun-Sun Cho
She received her BS, MS, and PhD degrees in Computer Science and Statistics from Seoul National University in 1991, 1993, and 1998, respectively. She is an Assistant Professor in the Department of Computer Science & Engineering at Chungnam National University in Daejeon, Korea. Her research interests include the area of programming languages and program analysis related issues, especially for pervasivecomputing environments.

Sumi Helal
He earned his B.E. and M.E. degrees in Computer Science and Engineering from Alexandria University in Egypt in 1982 and 1985, respectively, and received his Ph.D. in Computer Sciences from Purdue University in 1991. He held academic and industrial research positions at MCC, Purdue University, and the University of Texas in Arlington, and is nowa full Professor in the Computer and Information Science and Engineering Department (CISE) at the University of Florida.He is the co-founder and an editorial board member of the IEEE Pervasive Computing magazine, and Editor of the magazine's column on Standards, Tools, and Emerging Technologies. He has been on the editorial board of IEEE Transactions on Mobile Computing, and currently serves as the Networking Area Chair of the IEEE Computer magazine. His research interests span the areas of pervasive computing, mobile computing and networking, and Internet computing.