A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing

Julien Mercadal, Quentin Enard, Charles Consel, Nicolas Loriant

A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing

Julien Mercadal*† Quentin Enard*§ Charles Consel*†‡ Nicolas Loriant*

*INRIA †LaBRI ‡ENSEIRB §Thales
Bordeaux, France
first-name.last-name@inria.fr

Abstract

The challenging nature of error handling constantly escalates as a growing number of environments consists of networked devices and software components. In these environments, errors cover a uniquely large spectrum of situations related to each layer ranging from hardware to distributed platforms, to software components. Handling errors becomes a daunting task for programmers, whose outcome is unpredictable. Scaling up error handling requires to raise the level of abstraction beyond the code level and the try-catch construct, approaching error handling at the software architecture level.

We propose a novel approach that relies on an Architecture Description Language (ADL), which is extended with error-handling declarations. To further raise the level of abstraction, our approach revolves around a domain-specific architectural pattern commonly used in pervasive computing. Error handling is decomposed into components dedicated to platform-wide, error-recovery strategies. At the application level, descriptions of functional components include declarations dedicated to error handling.

We have implemented a compiler for an ADL extended with error-handling declarations. It produces customized programming frameworks that drive and support the programming of error handling. Our approach has been validated with a variety of applications for building automation.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures

General Terms Design, Languages

Keywords Domain-Specific Languages, Architecture Description Languages, Exception, Pervasive Computing

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

1. Introduction

Pervasive computing systems coordinate a wide range of entities (devices and software components), communicate using a variety of network protocols, rely on intricate distributed systems technologies, and invoke a host of complex APIs. As a result, a pervasive computing system needs to address a uniquely large spectrum of errors pertaining to a variety of levels including physical infrastructure (*e.g.,* power loss), hardware (e.g., device malfunction), operating system (*e.g.,* resource exhaustion), network (*e.g.,* protocol timeout), middleware (*e.g.,* remote invocation), and API (*e.g.,* improper inputs). Detecting and recovering from these errors is critical to make a pervasive computing system reliable. To do so, a systematic and rigorous approach to handling errors is required.

Much progress has been achieved in middleware to abstract over underlying technologies. However, even the ones dedicated to the pervasive computing domain (*e.g.,* [29] and [28]), cannot abstract over errors: whether low level or high level, errors need to be propagated to trigger application-specific treatments. A systematic treatment of these errors comes at the price of polluting the entire application code with error-handling computations. Typically, a try-catch block is introduced when invoking an operation (or a group of them) that may fail. The resulting code is bloated and entangled, as demonstrated by Lippert and Lopes [20].

Even when errors are systematically addressed, the error-handling logic is often ad hoc and local, precluding system-wide reasoning. Because of a lack of high-level programming support, writing error-handling logic is a daunting task, neglected by most programmers. This situation results in the implementation of crude error-handling strategies, defeating the purpose of these errors (*e.g.,* log and ignore an error, or signal a different error).

An inherent feature of the try-catch block is that it addresses error-recovery concerns at two different levels: locally, by enabling execution to resume whenever possible, and globally, by taking action to preserve system-wide consistency. In the former case, the exception is not propagated and, if a value is missing, it may be replaced by a default

one. Hereafter, this kind of error handling is said to be *application level*. Alternatively, the programmer may choose a *system-level* handling of an error where the erroneous computation is aborted and the control transferred to a system-wide error-recovery handler. Such handler may replace a faulty sensor by a backup one, for example. The conflict between application-level and system-level handling of errors is exacerbated by the device-intensive nature of a pervasive computing system. Furthermore, real-size pervasive computing systems require to cope with a host of potential errors, making it critical to raise the level of abstraction beyond the try-catch block.

Scaling up error handling requires to raise the level of abstraction beyond the code level and the try-catch construct, approaching error handling at the software architecture level. This level permits a software system to be decomposed into components (*e.g.,* [31]), paving the way for the architecturing of error handling treatments. Addressing error handling at system-design time provides information to guide and support the implementation process, and to generate runtime support.

Our approach

To architecture error handling, we introduce an Architecture Description Language (ADL) that is extended with error-handling declarations. To further raise the level of abstraction, our approach revolves around a domain-specific architectural pattern, commonly used in pervasive computing. In doing so, we provide domain-specific declarative and programming support for error handling. In our approach, an error can be treated both at the application and system level, allowing it to be compensated locally and managed globally by platform-wide strategies.

Specifically, our extended ADL allows to decompose a pervasive computing system into functional components and error-handling components. Declaring a functional component includes specifying what errors are handled at the application level, and how they are handled. These declarations make the flow of errors explicit in an architecture description, enabling high-level reasoning. Error-extended architecture descriptions are used to generate customized programming frameworks. These frameworks guide and support the implementation of a component logic, as well as its application-level handling of errors. For example, a software architecture may require a component to fully handle device-related errors, shielding client components from these concerns.

To define platform-wide strategies for managing errors, we further extended our ADL, allowing system-level components to be declared. These components handle classes of errors that are relevant to a given strategy. Raising the level of abstraction at which strategies are described is a key enabler to enforce global constraints. For example, occupants of a building may need to be evacuated if a massive failure of fire detection devices occurs. To make this constraint explicit in the architecture, one can introduce a component that requires to receive all malfunction-related errors from these devices. Ensuring this constraint then amounts to verifying that the component implementation takes appropriate actions when a failure rate is reached. Because the underlying programming framework is generated from the architecture description, our system-level component is guaranteed to be sent errors of the required types.

To implement our approach, we have enriched an ADL dedicated to the pervasive computing domain with error-handling declarations. We have extended its framework generator to include support for error handling. We validated our approach by architecturing and implementing various building-automation applications.

Our contributions

Our contributions can be summarized as follows.

- *Architecturing error handling.* We propose a novel approach that raises the level of abstraction of error handling from programming to architecturing. Our approach allows reasoning, and programming is driven by this extended form of software descriptions.
- *An extended domain-specific ADL.* We have extended a domain-specific ADL with declarations dedicated error handling. These architecture-level declarations provide a separation between functional and error-handling concerns. Furthermore, error handling is made specific by decomposing it into application and system compensation strategies.
- *Programming support for error handling.* Architecture descriptions are processed by a compiler that generates dedicated programming frameworks in Java. We have extended this compiler to produce additional programming support for signaling, propagating and treating errors that originate as Java exceptions. This support makes the programming of error handling more rigorous and systematic.
- *Validation.* We have used our approach to develop a variety of dependable applications in areas including home/building automation and healthcare. Our largest case study is a system for managing a *13 500*-square meter building, amounting for more than *3 000* LOC.

Outline The rest of this paper is organized as follows. Section 2 identifies the key requirements to architecturing error handling. Section 3 presents our error handling model. Section 4 examines our implementation. In Section 5, we describe the evaluation of our approach. Related works are discussed in Section 6 and conclusions are given in Section 7.

2. Requirements to error handling

This section introduces DiaSpec, a domain-specific architecture description language dedicated to pervasive computing

systems [6], and identifies key requirements for handling errors in this domain.

DiaSpec is domain specific in that it is dedicated to describing pervasive computing systems and it relies on an architectural pattern commonly used in the pervasive computing domain [12]. The DiaSpec language is introduced using a working example: a fire management application. This application is part of a larger software system aimed to fully automate the management of an engineering school building [4].

Our building management system consists of a set of applications, each of which manages a range of situations. For example, our working example is dedicated to managing fire situations; it detects a fire by analyzing data produced by smoke and temperature sensors that populate the building. When a fire occurs, the fire management application is responsible for triggering sprinklers and alarms, and releasing fire doors.

In DiaSpec, a pervasive computing system is described in two stages: (1) a taxonomy of entities is defined for a target area (*e.g.,* building automation), and (2) an architecture is described for each application of the pervasive computing system (*e.g.,* fire management), given a taxonomy definition. DiaSpec provides a language layer for each stage.

2.1 Defining an application area

DiaSpec offers a language layer that abstracts over heterogeneous entities, whether hardware or software. It is a taxonomy language dedicated to describing classes of entities that are relevant to a given application area. An entity declaration models sensing capabilities that produce data, and actuating capabilities that provide actions. Specifically, a declaration includes a data source for each one of its sensing capabilities. An actuating capability corresponds to a set of method declarations. Additionally, attributes are included in an entity declaration to characterize properties about instances (*e.g.,* their location). Entity declarations are organized hierarchically, allowing entity classes to inherit attributes, sources, and actions.

Let us now illustrate the taxonomy language with the fire management area. An extract of this taxonomy is shown in Figure 1. Entity classes are introduced by the **device** keyword. For simplicity, note that the same keyword is used to introduce both software and hardware entities. At the root of our taxonomy is the `Device` node (lines 1 to 3). It introduces the `location` attribute. Attributes are used as area-specific values to discover entities in a pervasive computing environment.

The sensing capabilities of an entity class are declared by the **source** keyword. For example, the `TemperatureSensor` entity class defines the `temperature` data source of type `Temperature` (lines 9 to 12). Interestingly, the source declaration abstracts over how the data are supplied, that is, whether the data are pushed into the application, or pulled from the entity by the application.

```
1  device Device {
2    attribute location as Location;
3  }
4  device FireSensor extends Device {}
5  device FireActuator extends Device {}
6  device SmokeDetector extends FireSensor {
7    source smoke as Smoke;
8  }
9  device TemperatureSensor extends FireSensor {
10   attribute accuracy as Accuracy;
11   source temperature as Temperature;
12 }
13 device Sprinkler extends FireActuator {
14   action OnOff;
15 }
16 device FireDoor extends FireActuator {
17   action Release;
18 }
19 device Alarm extends Device {
20   action Activation;
21 }

23 action OnOff {
24   on();
25   off();
26 }
27 action Activation {
28   activate(type as AlarmType);
29   deactivate();
30 }
31 action Release {
32   release();
33 }
34
35 enumeration LockedStatus {LOCKED, UNLOCKED}
36 enumeration AlarmType {FIRE, INTRUSION}
37 enumeration TemperatureUnit {CELSIUS, FAHRENHEIT}
38 enumeration Accuracy {LOW, NORMAL, HIGH}

40 structure Temperature {
41   value as Integer;
42   unit as TemperatureUnit;
43 }
44 structure Smoke {
45   isDetected as Boolean;
46 }
```

Figure 1. Extract of the fire management taxonomy

Actuating capabilities are declared by the **action** keyword. As an example, consider the `Sprinkler` declaration (lines 13 to 15). This entity class defines the `OnOff` action interface to be invoked by an application to activate sprinklers. An action interface consists of the signatures of methods supported by an entity class (lines 23 to 33).

Requirements. The entities of a pervasive computing system are off-the-shelf software components and devices that may fail, raising a variety of errors [7]. For example, entities may become unavailable due to malfunction (*e.g.,* power losses) or network failures. They may also operate incorrectly due to bugs (*e.g.,* faulty sensors [19]). Entities may directly raise an error when they are capable of diagnosing a malfunction. Alternatively, the runtime environment

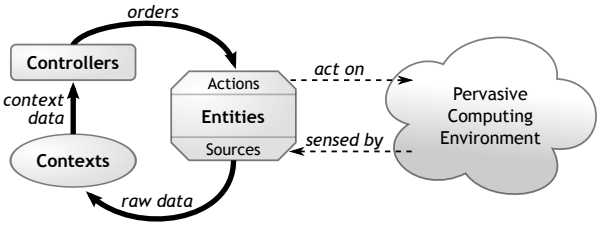

Figure 2. DiaSpec architectural style

may raise an error if it cannot operate an entity because of platform-related problems.

In the spirit of an IDL (*e.g.,* WSDL [8]), the declaration of an entity class should expose the errors it may raise to ensure that its clients provide appropriate treatments. Yet, we need to go beyond this approach because such error declarations do not take into account the usage context of its clients. For example, an error raised by a temperature sensor is declared once regardless of whether the sensor is used for informational purposes or is part of a safety-critical application. A more accurate approach should allow an entity client to express requirements on entity errors to take into account the usage context.

2.2 Architecturing an application

To architecture applications, the DiaSpec language provides an ADL layer. It is based on an architectural pattern depicted in Figure 2. This pattern decomposes an application into context and controller components. Context components are fueled by sensing entities, declared by the taxonomy. These components interpret, filter and aggregate these data to make them amenable to the application needs. Controller components receive application-level data from context components and determine the actions to be triggered on entities.

Let us illustrate the ADL layer of DiaSpec with the architecture description of our fire management example. The overall functional architecture of this application is graphically represented in Figure 3. The corresponding DiaSpec architecture declarations are presented in Figure 4. At the bottom of this architecture are the smoke detectors and the temperature sensors, declared in the taxonomy. These sensors respectively detect the presence of smoke and calculate the temperature of a room in the building. These data are respectively sent to the `SmokeDetected` and `AverageTemperature` components, declared using the **context** keyword. In particular, consider the `SmokeDetected` context (lines 1 to 4). This component is responsible for determining whether a given room of the building is filled with smoke. To do so, it aggregates and processes sources of information from smoke detectors deployed in the building. These sources are declared using the **source** keyword that takes an identifier and a class of entities (line 3). As a result of this declaration, the `SmokeDetected` context takes a `smoke` data source as input, produced by

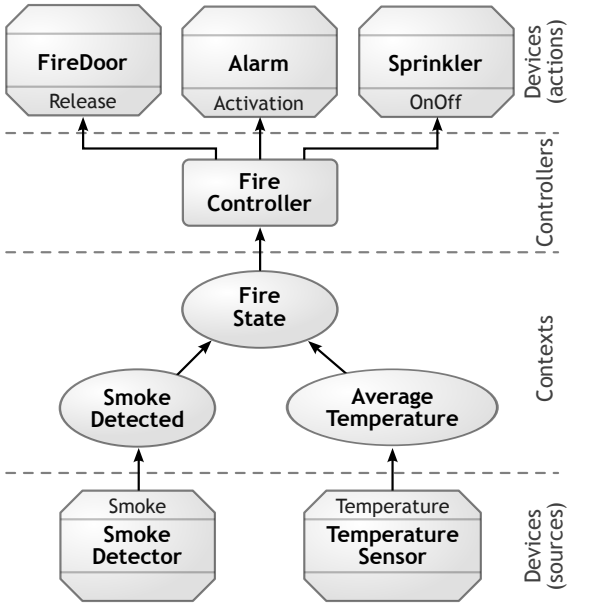

Figure 3. A data flow view of the fire management architecture extract

```
1  context SmokeDetected as Boolean
2    indexed by location  as  Location  {
3    source smoke from SmokeDetector;
4  }
5  context AverageTemperature as  Temperature
6    indexed by location  as  Location  {
7    source temperature  from TemperatureSensor;
8  }
9  context FireState   as  Boolean
10   indexed by location  as  Location  {
11   context SmokeDetected;
12   context AverageTemperature;
13 }

15 controller  FireController   {
16   context FireState;
17   action Release  on FireDoor;
18   action OnOff on Sprinkler;
19   action Activation   on Alarm;
20 }
```

Figure 4. Extract of the fire management architecture

the `SmokeDetector` entities. The `SmokeDetected` component signals the presence or the absence of smoke in a room by producing a value of type `Boolean`. To facilitate the use of this information by other components, each output value is attached an index, namely the location of the room where the smoke is detected. This is done by declaring the `SmokeDetected` component as being *indexed*, using the **indexed by** keyword (line 2). The `FireState` component combines the output of the `SmokeDetected` component with the average temperature of the target room, provided by the `AverageTemperature` context. This infor-

mation determines whether a fire is occurring in the target room; it is passed to the `FireController` component, declared using the **controller** keyword (lines 15 to 20). This component is responsible for extinguishing the fire and performing other emergency tasks. To do so, it operates sprinklers, alarms, and fire doors; using the **action** keyword, it declares invoking the `OnOff` action on the `Sprinkler` entity class, the `Activation` action on the `Alarm` entity class, and the `Release` action on the `FireDoor` entity class (lines 17 to 19).

Requirements. As shown in our fire management example, context and controller components are dependent on entities, whether or not directly. Entities are the main point of origin of errors. Furthermore, they may need to be propagated throughout the application data flow, impacting a chain of components. Rather than delegating the handling of errors to the implementation, a high-level approach should leverage component declarations. Enriched declarations could enable default treatments to be generated automatically and the development of specific treatments to be enforced during component implementation. Such declarative approach would raise the level of abstraction of error handling, allowing reasoning at the architecture level, generating error-handling code in default cases, and providing guidance to the programmer for other situations.

Considering our fire management system, there are a number of situations that could be expressed in the declarations of context and controller components. For example, the building designer may not provide an alternative to the failure of `FireDoor` entities. Consequently, the pervasive-computing system architect could take this situation into account by declaring a `FireController` component that ignores errors from `FireDoor` entities. For another example, the `SmokeDetector` entities should be considered as critical to the safety of the building occupants. Consequently, the software architect should be able to require the developer of the `SmokedDetected` component to implement code to compensate for errors at the application level. Alternatively, the software architect could also provide a declaration that precludes any error handling within a given component, providing a complete spectrum to the architect.

In addition to application-specific processing, there is a need to define error-handling strategies that are consistent over an entire pervasive computing platform. Doing so requires global handlers that would centralize the treatment of a given range of errors and provide a system-wide treatment that complements the application-level processing. For example, an unreachable entity not only requires an application-level treatment to compensate for the missing value, but it also necessitates system-level measures that could be factorized.

2.3 Implementing a pervasive computing system

The DiaSpec compiler generates a Java programming framework with respect to a taxonomy definition and an architecture description. This dedicated framework contains an abstract class for each DiaSpec declaration (entity, context and controller) that includes generated methods to support the implementation (*e.g.,* operations for entity discovery and entity invocation). The generated abstract classes also include abstract method declarations to allow the developer to program the application logic (*e.g.,* triggering entity actions). Implementing a DiaSpec-declared object, whether entity or component, is done by sub-classing the corresponding generated abstract class. This enables to precisely guide the programmer in the development process.

Figure 5 presents an excerpt of the implementation of the `SmokeDetected` context declaration. This implementation is done by extending the corresponding generated abstract class. Because this context is declared as taking a `Smoke` input source from smoke detectors, the generated framework provides the required methods to discover, select, and interact with instances of this entity class. Specifically, all available smoke detectors are discovered by invoking the `allSmokeDetectors` method (line 4); events of type `Smoke` are published by calling the `setSmokeDetected` method (line 11) for a given location; the `Smoke` source of this context component is accessed in the pull mode via `getSmoke` (line 10), and the push mode via `smokeChanged` (line 7).

```
1  public class MySmokeDetected extends SmokeDetected
2  {
3   public MySmokeDetected() {
4     allSmokeDetectors().subscribeSmoke(this);
5   }
6   @Override
7   public void smokeChanged(SmokeDetector smokeDetector, Smoke
          smokeDetected) {
8     ...
9     try {
10      if (sd.getSmoke().isDetected)    {
11        setSmokeDetected(location,  new Smoke(true));
12      }
13    } catch (DiaGenCommunicationException e) {...}
14    ...
15  }
16 }
```

Figure 5. An implementation of the `SmokeDetected` context component

Requirements. Because entities are extensively invoked by an application, their potential errors propagate the need to guard against these errors throughout most application components. A key problem with existing approaches is that, when implementing the handling of an error, the programmer is left wondering whether it should be ignored or compensated for; whether it requires system-wide actions or can be treated locally; whether it is fatal to the system or can be

recovered from. The nature of the error is informally specified and often requires thorough code examination. As a consequence, developers may ignore errors or provide inappropriate treatments.

Let us illustrate these issues by considering again the `SmokeDetected` context implementation (Figure 5). The call to a smoke detector (line 10) is guarded against an error. However, the try-catch construct offers a very crude mechanism, considering the many issues to address. For example, depending on the error type, a system-wide treatment may be required given the safety-critical nature of the sensor. For another example, depending on whether the error is found to be transient, a default value could compensate for the missing temperature value.

How errors are handled should not be optionally and informally communicated to the programmer. Instead, programming should be driven by architecture design decisions. The architecture description should be a repository of these architecture decisions, providing a system-wide view of error handling, down to its constituent parts.

These issues are specific to the constituent parts of the architecture of a software system and should be specified at that level. The implementation should then be closely driven by the design choices made in the architecture.

```
1  device TemperatureSensor extends FireSensor {
2    attribute accuracy as Accuracy;
3    source temperature as Temperature raises MeasureException;
4  }
5  device Sprinkler extends FireActuator {
6    action OnOff;
7  }
8  action OnOff {
9    on() raises WaterPressureException;
10   off();
11 }
```

Figure 6. Fire management taxonomy with exception declarations

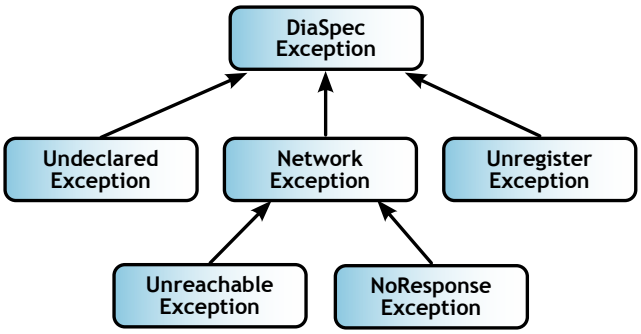

Figure 7. Hierarchy of built-in exceptions

3. Error-Handling Model

We now present our error-handling model that addresses the requirements discussed earlier. This model is introduced in the context of DiaSpec and is illustrated by the fire management example. We first describe how errors are characterized using the taxonomy definition of a pervasive computing environment. Then, we explain how to extend an ADL such as DiaSpec to architecture error-handling at both the application and system level. Finally, we examine the programming support generated by our architecture-driven approach to facilitate the implementation of the declared components.

3.1 Characterizing errors

Typically, errors occur during interactions with entities. We have extended our DiaSpec taxonomy language with declarations for specifying that a data source or an action method may throw exceptions.

Figure 6 revisits the taxonomy for fire management, decorating entity declarations with exception information. Exceptions raised by a data source are introduced by the **raises** keyword. In our example, an exception of type `MeasureException` is raised by temperature sensors when an error occurs during the computation of the temperature measure (line 3). Exceptions raised by an action method are also introduced by the **raises** keyword. In our example, an exception of type `WaterPressureException` is raised by sprinklers when an error occurs during their opening (line 9).

Other exceptions that are not directly related to sensing and actuating capabilities of entities, can be raised by the runtime environment during an access to a data source or a method call. These exceptions enable to signal errors coming from the pervasive computing platform. For example, when an entity is unavailable due to a network failure, an exception of type `NetworkException` is raised. We call such exceptions *built-in*; they form a hierarchy shown in Figure 7.

Our two-level approach to error handling results in signaling an error at both the application level and the system level. At the application level, the error handling logic is only concerned about compensating for the error, not repairing the malfunction. At the system level, error handling is aimed to analyzing the cause of an error and determining repair policies that preserve global consistency. Conceptually, both application and system-level error handling logic are executed simultaneously, when both levels are impacted by an error. When an error is detected proactively (*e.g.,* an expired regisration lease for an entity), it can be handled by a system-level component, prior to any application invocation, allowing a recovery that is transparent to the application.

3.2 Architecturing error handling

By addressing errors at the software-architecture level, our approach abstracts over the wide spectrum of errors occurring in a pervasive computing system, and provides information to guide and support the implementation of error-handling code. We leverage the DiaSpec architecture language presented in Section 2 to architecture errors at both

application and system level. Note that our approach can apply to other ADLs that are based on our architectural pattern.

3.2.1 At the application level

Architecturing errors at the application level consists of specifying how an exception impacts the control flow of the architecture components. More precisely, the architect determines at system-design time (1) whether the developer of a given component must define the logic to handle an exception, continuing execution transparently; (2) whether handling this exception is optional, amounting to a Java *unchecked* exception; (3) whether the exception should be automatically propagated to the calling component; or (4) whether no exception can occur. To express these policies we have introduced four new keywords in DiaSpec, respectively: **mandatory catch**, **optional catch**, **skipped catch**, and **no catch**. These extensions allow the architect to require an error to be handled differently depending on its criticality in the pervasive computing system.

```
1  context SmokeDetected as Boolean
2    indexed by location as Location {
3    source smoke from SmokeDetector [skipped catch];
4  }
5  context AverageTemperature as Temperature
6    indexed by location as Location {
7    source temperature from TemperatureSensor [mandatory catch];
8  }
9  context FireState as Boolean
10   indexed by location as Location {
11   context SmokeDetected [mandatory catch];
12   context AverageTemperature [no catch];
13 }

15 controller FireController {
16   context FireState [no catch];
17   action Release on FireDoor [skipped catch];
18   action OnOff on Sprinkler [optional catch];
19   action Activation on Alarm [mandatory catch];
20 }
```

Figure 8. The fire management architecture with exception declarations

Let us illustrate these extensions with our fire management example. To do so, a revised version of the architecture description with exception declarations is displayed in Figure 8. In this description, the architect considers the `FireState` component as being central to determining a fire situation in the building. Consequently, error handling of smoke detectors is not delegated to the `SmokeDetected` component. This design decision is expressed in line 3, where the `SmokeDetected` component is declared as not being allowed to process errors from smoke detectors. This task is assigned to the `FireState` component, where the treatment of smoke-detector errors is declared as mandatory (line 11).

In contrast, the `AverageTemperature` component is assumed to serve various purposes, including heating control. Its role is secondary in the fire management application: it is only used to corroborate the information delivered by the `SmokeDetected` component. As such, the `AverageTemperature` component is declared as one that compensates for errors from temperature sensors (line 7), always providing a value, even at the expense of accuracy. Because errors from temperature sensors are fully treated by the `AverageTemperature` component, the `FireState` component does not have to consider the reliability of this context component, as is declared in line 12.

Our exception declarations also play a key role in controller components to interact with actuators. They express design decisions regarding what the implementation of a component should do in case the invocation of an actuator fails. If a failure cannot be compensated for at the application level, the exception can be ignored (line 17). If alternative strategies to a failure can be chosen at the application level, the mandatory declaration is used to enforce an implementation (line 19). If no specific treatment is defined at design time, the optional declaration is used (line 18).

Note that the **no catch** declaration can be inferred: when a component is required to treat errors (line 7), its clients can automatically assigned a **no catch** declaration (line 12). Furthermore, a syntactic verification is performed to check the consistency of the exception declarations.

3.2.2 At the system level

Architecturing errors at the system level takes the form of DiaSpec context and controller components that process DiaSpec events signaling errors. These DiaSpec events are called *exceptional events*[1]; they are of the same type of the original Java exceptions, providing all the information about the causes of the error. System-level context components differ from ordinary DiaSpec components in that they take as input exceptional events from entities. They process these events and refine them into application-specific values. Controller components receive these refined values and execute a repair strategy by invoking actuators. In doing so, platform-wide repair strategies are raised to an architectural level: they are decomposed into context and controller components. This approach enables system-wide reasoning about errors in a pervasive computing system, enabling stringent implementation constraints to be enforced.

To illustrate our approach, consider the system level of the error-handling architecture for our working example displayed in Figure 10; the data flow view of this architecture is showed in Figure 9. Conceptually, a system-level context component is dedicated to handling errors that are related from an application viewpoint. For example, we declare a context component dedicated to handling failures of fire detection devices and another component addressing

[1] This distinction is purely conceptual. Exceptional events are declared as such but are actually implemented as DiaSpec events (see Section 4 for details).

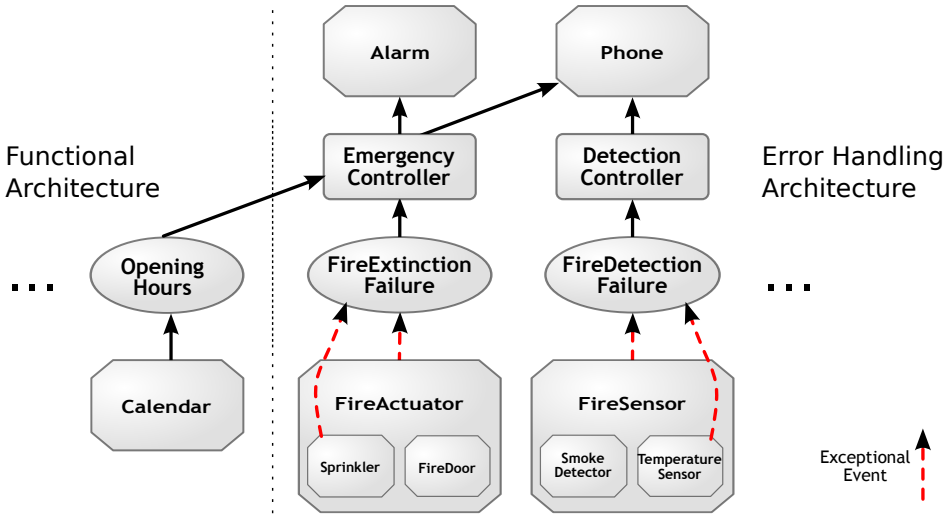

Figure 9. Extract of the flow of exception handling in the fire management application

```
1  context FireExtinctionFailure  as Boolean
2    indexed by location  as Location {
3      exception WaterPressureException  from Sprinkler;
4      exception BatteryException  from Sprinkler;
5      exception PowerException from FireActuator;
6      exception NetworkException from FireActuator;
7      exception UnregisterException  from FireActuator;
8  }
9  context FireDetectionFailure  as Boolean
10   indexed by location  as Location {
11     exception MeasureException from TemperatureSensor;
12     exception PowerException from FireSensor;
13     exception NetworkException from FireSensor;
14     exception UnregisterException  from FireSensor;
15 }

17 controller EmergencyController {
18   context FireExtinctionFailure;
19   context OpeningHours;
20   action Activation  from Alarm;
21   action PhoneSomeone from Phone;
22 }
23 controller DetectionController  {
24   context FireDetectionFailure;
25   action PhoneSomeone from Phone;
26 }
```

Figure 10. Extract of the error-handling architecture at the system level for the fire management example

failures of fire extinction devices (`FireDetectionFailure` and `FireExtinctionFailure`, respectively). These specific components analyze the input errors and publish a status information, whenever measures need to be taken. For example, occupants of a building may need to be evacuated when there is a massive failure rate of fire extinction devices. To do so, the implementation of the `FireExtinction-Failure` component should, among other tasks, keep track of faulty fire-extinction devices. When a failure threshold is reached, it should publish a failure alert. As declared in line 18 of Figure 10, the `EmergencyController` component is a client of this information and may trigger an evacuation alarm (line 20), if a failure alert occurs during office hours (line 19).

Similarly, the `FireDetectionFailure` component determines whether every room of the building has functioning devices for detecting a fire (*e.g.,* smoke detectors) by handling the exceptional events related to this device class.

Let us examine further the declarations of system-level components in Figure 10. Using the **exception** keyword, the `FireExtinctionFailure` component (lines 1 to 8) defines as input exceptional event types from various entities: `WaterPressureException` and `BatteryException` from `Sprinkler`, and `PowerException`, `NetworkException` and `UnregisterException` from `FireActuator`. Whereas some declarations directly draw errors from an entity class (*e.g.,* `Sprinkler`), others leverage the hierarchy of the taxonomy of entities, allowing handlers to be defined at the appropriate level of granularity (*e.g.,* `FireActuator`).

A system-level component, whether context or controller, processes both exceptional events and ordinary values, produced by context components, whether or not functional. This is illustrated by the `EmergencyController` component whose logic depends on the `OpenHours` component (line 19).

3.3 Implementing error handling

Given a taxonomy definition and architecture declarations, a domain-specific programming framework is generated, providing support for the implementation of components (*i.e.,* entities, contexts and controllers) of a pervasive computing system. We have developed an extended version of the DiaSpec compiler to address error handling. This extended compiler generates frameworks with programming support

```
1  public class TemperatureSensorProxy {
2    ...
3    // only generated if annotated mandatory or optional
4    public Temperature getTemperature (TemperatureContinutation tc) {
5      ... // code generated by the DiaSpec compiler
6    }
7    public interface TemperatureContinuation {
8      public Temperature onError ();
9    }
10 }
```

Figure 11. Extract of the `TemperatureSensorProxy` proxy class generated for the `AverageTemperature` component

to ease and guide the development of error handling code at both the application and system level.

3.3.1 At the application level

At the application level, error handling essentially amounts to compensate for either a missing value from a data source or a failed invocation of an entity action; This is done without considering the nature of the error. Error handling is triggered by a unique exception called *the application-level exception*. This exception only retains the control flow effect of the exception mechanism, allowing the error to be propagated up the call stack.

To handle the application-level exception, we propose a continuation-style approach to compensate for a missing value. In this approach, if the component being implemented is declared as handling errors, the developer must pass a typed continuation code as an additional input to the operation. If error handling is declared as being disallowed, there is no continuation parameter. This continuation-passing style approach requires the developer to provide compensation code on a per-operation basis. This constrasts with the try-catch construct that can cover a group of operations. Because the programming framework is generated with respect to an architecture description, the entity interfaces provided to the developer are compliant to their error-handling declarations.

Let us examine the `TemperatureSensorProxy` proxy class generated for the `AverageTemperature` component[2]. An extract of this proxy class is displayed in Figure 11. This class provides a method named `getTemperature` (line 4) to request the `temperature` source provided by the temperature sensors. Because this source was declared as **mandatory catch** in the `AverageTemperature` component, the generated proxy requires the programmer to compensate for a missing value in case of failure. To do so, a continuation parameter is introduced in the `getTemperature` method; it corresponds to recovery code to be executed in case of failure. Since the Java programming language does not support continuations per se, they are mimicked by generated interfaces (lines 7 to 9) declaring an `onError` method, whose return type is the same as the overloaded method (*i.e.,* Temperature). The following code presents examples of continuations for the `getTemperature` method in case the invocation of the sensor fails. The first continuation provides a default value. The second continuation retries to query the sensor once; if it fails again, a default value is returned.

```
1  // default value
2  Temperature temperature = t.getTemperature (
3    new TemperatureContinuation () {
4      public Temperature onError () {
5        return new Temperature(21, TemperatureUnit.CELCIUS);
6      }
7    }
8  );

10 // retry + default value
11 Temperature temperature = t.getTemperature (
12   new TemperatureContinuation () {
13     public Temperature onError () {
14       return t.getTemperature(
15         new TemperatureContinuation() {
16           public Temperature onError() {
17             return new Temperature(21, TemperatureUnit.CELCIUS);
18           }
19         }
20       );
21     }
22   }
23 );
```

If the client component had declared the `Temperature` source as skipping errors (*i.e.,* **skipped catch**), the generated proxy would only include a method without a continuation parameter, disallowing the programmer to handle errors. In this case, when an error occurred, the application-level exception would automatically be propagated to the calling components until one caught it. Because pervasive computing systems typically rely on a reactive-execution model, the application-level exceptions ignored by the top-level calling component, can be soundly intercepted by our error-handling model, without aborting the application. In doing so, the developer is not forced to write error-handling code, nor does he need to modify every method signature where an exception is not handled to declare its propagation, as is done in Java.

Alternatively, the client could have declared this source as optionally treating errors (*i.e.,* **optional catch**). In this case, both methods, with and without the continuation parameter, would have been generated in the proxy, delegating to the developer whether to handle errors.

For another example of continuation consider the sprinkler actuator, invoked by the `FireController` component. This interaction is declared as **optional catch**. As a result, the programmer may either invoke the `OnOff` method without a continuation, letting the exception propagate in case of an error, or consider that the `FireController` component should continue taking actions to extinguish the fire, ignoring the error. The latter option is achieved by the continuation shown below.

[2] A proxy class is generated for each component interaction declared in the DiaSpec architecture.

```
// do nothing
sprinkler.on (
  new OnContinuation () {
    public void onError () {
      // do nothing
    }
  }
);
```

3.3.2 At the system level

System-level error handling takes the form of contexts that refine exceptional events, originating from Java, into application-specific values. These values are used by controllers to maintain the platform consistency by invoking entity actuators. These context and controller components are implemented by extending the corresponding abstract class generated by the DiaSpec compiler. This is illustrated by the implementation of the `FireExtinctionFailure` context, shown in Figure 12. It is aimed to detect locations where there is no functioning devices to extinguish a fire. Its implementation is done by extending the generated abstract class, named `FireExtinctionFailure`. Because this context is declared as catching `PowerException` and `WaterPressureException` from fire actuators and sprinklers respectively (Figure 10, line 5 and line 3), the generated framework provides support to discover fire actuators and sprinklers, and to monitor the exceptions they raise. For example, the generated `allFireActuators` method discover all fire actuators and is used to subscribe to `PowerException` from fire actuators (Figure 12, line 4). Similarly, the generated `allSprinklers` method is used to subscribe to `WaterPressureException` from sprinklers (line 5). The `MyFireExtinctionFailureImpl` class implements the abstract methods (*e.g.,* `onNewWaterPressureException`) that are called upon exceptions (lines 8 to 15).

4. Implementation

The DiaSpec compiler, named DiaGen, is implemented using the ANTLR parser generator [25]. A DiaSpec-specified software system is agnostic to the target distributed-systems technology. To do so, DiaGen leverages existing distributed-systems technologies by generating glue code to customize them with respect to the needs of pervasive computing. Currently, DiaGen offers back-ends for the following targets: Web Services [9], RMI [13], SIP [30] and CORBA [24].

Our error-handling model extends the DiaSpec taxonomy and architecture languages, the DiaSpec framework generator, and the DiaSpec runtime. Most of these extensions are mapped directly into existing DiaSpec concepts. For example, system-level context components rely on the publish-subscribe paradigm, leveraging the implementation of DiaSpec sources and contexts. This seamless integration strategy allows to transparently reuse existing DiaSpec tools, such as the existing back-ends and DiaSim – a simulator for pervasive computing systems [4].

```
1  public class MyFireExtinctionFailureImpl extends
       FireExtinctionFailure
2  {
3    public MyFireExtinctionFailureImpl() {
4      allFireActuators().subscribePowerException();
5      allSprinklers().subscribeWaterPressureException();
6      ...
7    }
8    @Override
9    public void onNewPowerException(FireActuator fireActuator,
           PowerException power) {
10     setFireExtinctionFailure(fireActuator.getLocation(),
             noMoreCorrectFireActuator(fireActuator.getLocation()));
11   }
12   @Override
13   public void onNewWaterPressureException(Sprinkler sprinkler,
           WaterPressureException waterPressure) {
14     setFireExtinctionFailure(sprinkler.getLocation(),
             noMoreCorrectFireActuator(sprinkler.getLocation()));
15   }
16   ...
17 }
```

Figure 12. Extract of the `FireExtinctionFailure` context implementation class

We now briefly describe the implementation of our error handling model. We examine how an error is transformed into the application-level exception and an exceptional event. Then, we discuss how they are propagated through components.

4.1 Handling errors

Java exceptions, regardless of their origin, are intercepted by an intermediate layer, between the entity client and the entity implementation. This layer is part of the DiaSpec generated framework. It provides a uniform treatment of exceptions, converting them into the application-level exception and an exceptional event.

An error may either be user-defined or built-in. User-defined errors correspond to taxonomy-declared exceptions; they are raised by entity implementations, using the Java *throw* statement. If not taxonomy-declared, an exception is considered built-in. For example, when a networked entity becomes unreachable, an exception is raised by the client proxy. In addition, DiaGen generates a number of error detection mechanisms (*e.g.,* watchdog and heartbeat). These mechanisms allow system-level components to handle errors proactively (*i.e.,* prior to any application invocation), improving the reliability of the pervasive computing platform.

The layer between a client proxy and the implementation of a `TemperatureSensor` entity is displayed in Figure 13. The `MeasureException` Java exception is mapped into both an exceptional event (line 4) and the application-level exception (line 5). Similarly, the built-in Java exception of type `Exception` is published (line 7) and propagated as the application-level exception (line 8).

As can be noticed, the transport of exceptional events at the system level leverages DiaSpec events: each exceptional event is published as a DiaSpec source (lines 4 and 7). Source notification follows the publish-subscribe paradigm. This paradigm is mapped by DiaGen into the target distributed-systems technology [6] (*i.e.,* Web Services, RMI, SIP, and CORBA).

```
1 try {
2   return temperatureSensor.getTemperature();
3 } catch (MeasureException e) { // user−defined
4   publishMeasureException(e);
5   throw new ApplicationLevelException();
6 } catch (Exception e) { // built−in
7   publishUndeclaredException(new UndeclaredException(e));
8   throw new ApplicationLevelException();
9 }
```

Figure 13. Mapping errors into the DiaSpec model

4.2 Propagating errors

Once converted into an exceptional event, an error is refined by context components. These components interpret, aggregate and transform information to produce application-specific values, combining exceptional events with other data sources. This refinement process propagates information throught context components until values can be used by controller components to take actions.

At the application level, the application-level exception is either propagated or handled in a continuation. It is propagated up the call stack, when the corresponding exception was declared as `skipped`. In case of an `optional` or `mandatory` declaration, the DiaSpec compiler generates a method that catches the application-level exception and executes the continuation as illustrated in the following.

```
1 public Temperature getTemperature(TemperatureContinuation tc) {
2   try {
3     return this.getTemperature();
4   } catch (ApplicationLevelException e) {
5     return tc.onError();
6   }
7 }
```

5. Assessment

We applied our error handling model on three existing pervasive computing applications developed in our research group. These applications are part of a larger project aimed to automate a 13,500-square meters building, hosting an engineering school [4]. This project combines six different applications involving 21 entity classes, 20 components and over 400 entity instances, amounting for more than 3,000 LOC written in Java.. The fire manager is one of these applications. We now briefly describe the two others, namely the newscast manager and the intrusion manager. We identify errors pertaining to these applications, and we give benefits provided by our error-handling approach. Then, we discuss how errors would be handled without our approach, directly using Java exceptions.

5.1 Newscast Manager

This application manages information displayed on screens deployed in the engineering school. Information such as news from RSS feeds or daily class schedules are selected according to surrounding student profiles, including their dominating department affiliation. Badge readers are located near screens to detect proximity of students and their identities. Finally, a profile database is queried to obtain student profiles from their identity.

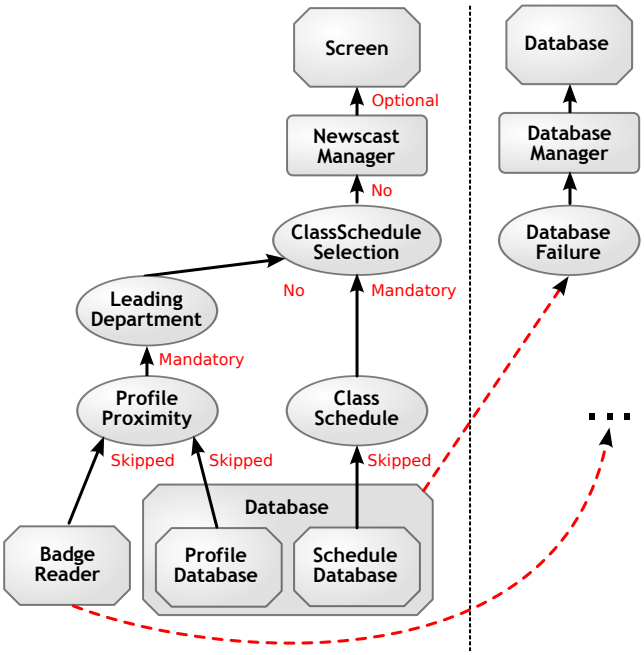

Figure 14. Extract of the newscast DiaSpec architecture

Figure 14 (left part) presents the part of the application architecture that displays class schedules on screens. The `ProfileProximity` component uses entities of type `BadgeReader` and `ProfileDataBase` to provide the profiles of nearby students to the `LeadingDepartment` component. In turn, this component defines a process that combines the profiles of the surrounding students, delivering to the `ClassScheduleSelection` component the leading profile. This information is used to request the class schedule of the corresponding class from the `ClassSchedule` component, and is provided to the `NewscastManager` component. The class schedule is obtained from the `ScheduleDatabase` entity.

Error handling strategies. Badge readers can stop working due to hardware or network failures. Yet, accurately monitoring people in the proximity of a screen is not required in the context of the newscast manager: missing the detection of a badge is unlikely to change the leading profile in an area. Thus, the `ProfileProximity` context

can delegate error handling to LeadingDepartment context, which can compensate for the application-level error by willingly ignoring it. This is expressed in the architecture by annotating with the **skipped catch** keyword the edges between ProfileProximity and its children, namely, BadgeReader and ProfileDatabase. Additionally, we annotate with **mandatory catch** the edge between LeadingDepartment and ProfileProximity. Similarly, the ClassScheduleSelection context can compensate failures of the ScheduleDatabase entity with a default value.

When all error flow paths to a component have been assigned a mandatory treatment, the remaining edges can be annotated with the **no catch** keyword. At this point, the architecture description, enriched with error-handling declarations, can be used to generate a new programming framework. The Java compiler then points the programmer to errors that are due to missing continuations for calls to entities requiring a *mandatory* compensation.

Badge readers and databases errors are now compensated at the application level. Yet, it is necessary to deal with faulty entities to prevent further problems. In our approach, this can be done without polluting the code of the newscast application. For example, consider repairing database corruptions using database built-in repair features (*e.g.,* rollback). This strategy is addressed by system-level components: the DatabaseFailure context abstracts over database errors and instructs the DatabaseManager controller to perform repair operations on database entities.

These system-level components do not impact the application level implementation. Generating the framework from a software architecture, extended with error-handling declarations, introduces only new abstract classes that must be subclassed to implement the system-level error recovery logic.

5.2 Intrusion Manager

This application is responsible for securing the engineering school. It defines what areas needs to be secured and detects intrusions in the secured ones using motion detectors deployed in the engineering school. When an intrusion is detected, alarms are triggered and the supervisor of the engineering school is alerted by displaying a warning message with a picture taken by a camera covering the intrusion area on his supervision screen.

An extract of the functional architecture of this application is graphically represented in Figure 15 (left part). When the entities of type MotionDetector detect a motion in an area of the engineering school, the IntrusionDetected component requests the SecuredArea component to determine whether the motion is occurring in an area identified as secured by the SecurityService entity. If so, the information is passed to the IntrusionManager component; it acts on the Alarm, Camera, and Screen entities to report the intrusion to the supervisor.

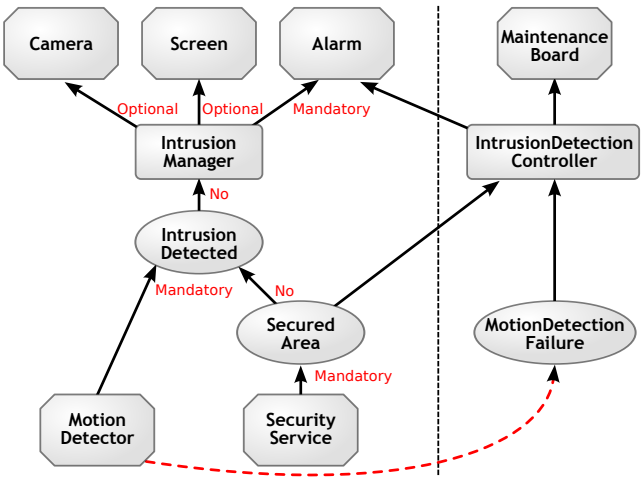

Figure 15. Extract of the intrusion manager DiaSpec architecture

Error handling strategies. Similarly to the FireState component in the fire management application, the architect considers the IntrusionDetected component as being central to determining an intrusion situation in the engineering school. Consequently, it must compensate for the application-level errors of the MotionDetector entities. This is expressed by annotating with the **mandatory catch** keyword the edge between MotionDetector and IntrusionDetected. Moreover, the SecuredArea context can compensate failures of the SecurityService entity with a default value (*i.e.,* the area is secured), implementing a conservative security strategy.

Because motion detectors are considered as critical to the safety of the engineering school, dealing with their failure is essential. To that end, a repair strategy must be introduced and addressed by system-level components. The MotionDetectionFailure component keeps track of faulty motion detectors. When a failure threshold is reached, it instructs the IntrusionDetectionController component to trigger alarms and to display an error report on the maintenance board.

5.3 Revisiting our use cases with Java exceptions

Let us now discuss the issues one would encounter in adding error handling in our use cases, solely using Java exceptions.

A system-level recovery strategy handles various errors from multiple entities. Doing this in Java requires developers to duplicate code for each occurrence of the corresponding exceptions. One may wonder whether letting these exceptions propagate higher up the call stack would enable factorizing the system-level error recovery strategy. In fact, such propagation prevents the application logic from compensating for an error and resuming normal execution. For example, implementing the database repair strategy in the newscast manager inside the ClassScheduleSelection component would prevent from compensating for the Profile-

`Database` exceptions in the `LeadingDepartment` component. Since, continuous execution is important in pervasive computing systems, developers should systematically perform system-level recovery strategies at the closest location from the exception source and rethrow exceptions to application-level strategies. Our model automates this repetitive and error-prone task by generating the glue code that systematically signals exceptions at the system level.

In the intrusion management application, the data provided by the motion detectors may only be pushed into the application, and never pulled. Consequently, there is no try-catch block used in the implementation of the `IntrusionDetected` component to deal with faulty motion detectors. This situation may lead developers to neglect critical errors for the pervasive computing system. Moreover, dealing with these errors requires developers to write ad hoc code for detecting and signaling potential failures of motion detectors. By providing failure detection mechanisms (*e.g.,* heartbeat) that raise proactively built-in exceptions (*e.g.,* `UnregisterException`), our approach ensures that device failures are systematically signaled at the system level.

Another issue stems from the fact that Java provides no guidance as to where and how exceptions should be handled. A common mistake is to mask an exception by catching it early or with a wrong type (*e.g.,* a parent class), without rethrowing it. This may forbid an appropriate exception handler up the call stack to compensate for the exception. For example, in the newscast manager application, Java does not prevent developers to handle `ScheduleDatabase` exceptions in the `ClassSchedule` component. It can make it impossible for some exceptions to be treated appropriately. In contrast, our approach lets the software architect declare at design time where exceptions should be propagated and where they should be handled.

6. Related Work

We now review existing approaches for handling errors in software systems, whether or not pervasive computing. To the best of our knowledge, there does not exist tool-based approaches to architecturing, declaring and supporting error handling, decomposing treatment into an application and system level.

Architecture. ADLs specify the structure and the behavior of a software system, whether or not distributed, to ensure some properties both at compile time and run time. Although most of them target specific aspects of a software system [3, 21, 22], few ADLs focus on the specification of error flow in architecture descriptions. Filho *et al.* [15] propose a conceptual framework, named Aereal, to describe and analyze exceptions that flow between architectural components in a given architectural style. In contrast, our approach goes beyond verification in that a programming framework is generated from an architecture description. Such programming framework implements the exception declarations, ensuring that the resulting software system is compliant with them. Our generative approach is made possible by the domain-specific nature of DiaSpec.

Exception patterns. The Portland Pattern Repository [1] proposes a set of exception pattern descriptions in the context of general-purpose programming languages. This repository is organized into categories: defining exception types, raising exceptions, handling exceptions, *etc*. These pattern descriptions are work in progress that can be freely extended and discussed. Our approach combines, adapts and automates two of these patterns to the needs of error handling in pervasive computing systems.

The application-level exception, proposed in our approach, can be seen as a *BottomPropagation* pattern in which an exception carries no information. It is also similar to the *Maybe* Monad[3] in Haskell. As for exceptional events at the system level, they are typically implemented with an *ExceptionReporter* pattern. This approach enables to decouple the treatment of errors from the control flow, allowing for example exceptions to be treated by multiple handlers. This exception pattern adapts the publish-subscribe [14] model for error reporting.

Not only does our approach revolve around both the *BottomPropagation* and *ExceptionReporter* exception patterns, but it automates their implementation in the generated programming framework.

Middleware and programming languages. Many software layers dedicated to the pervasive computing domain have been proposed to provide programming support for error handling [7, 26, 27]. For example, the one.world project [27] proposes a *check-pointing* mechanism that allows developers to capture the execution state of a component, and later to restore it to gracefully resume the execution of the component after a failure, such as power loss. It also enhances the robustness of pervasive computing systems by providing transaction-level persistence. Our error handling model is complemented to this approach. In fact, we plan to enrich DiaSpec further by integrating fault-tolerance declarations, generating built-in error recovery strategies in programming frameworks.

Dedecker *et al.* [10] have proposed a domain-specific language, called AmbientTalk, dedicated to developing applications in the context of a mobile network. In particular, they introduce a distributed exception-handling mechanism [23] to deal with mobile hardware characteristics [10], such as connection volatility. This mechanism consists of a set of language constructs that enables to handle exceptions at different levels of granularity in the application code: message, block and collaboration. A key difference with our approach is that AmbientTalk still relies on a form of try-catch block that does not allow a separation between application and system-level error handling.

[3] http://www.haskell.org/all_about_monads/html/maybemonad.html

Various communication paradigms have been used to develop mobile computing applications, whether data-based (*e.g.,* tuple spaces [16]), or control-based (*e.g.,* the actor model [2]). These paradigms enable to decouple communications between networked entities from their *spatial* and *temporal* dimension. This strategy improves failure resilience through error isolation and prevents a range of errors from being propagated. However, these approaches are restricted to network-related errors. In contrast, our approach tackles a range of errors, from faulty devices to platform errors. Also, it leverages an existing general-purpose programming language, namely Java, raising the level of abstraction of error handling by introducing extensions to an ADL.

Demsky and Dash [11] have proposed a task-based language, called Bristlecone, for developing robust systems. Their approach places a premium on continued execution and can tolerate some degradation from a specific designed behavior. The Bristlecone runtime uses task specifications to determine what task to execute. Because tasks in Bristlecone have a transactional semantics, when a task fails due to software, hardware or user errors, the Bristlecone runtime aborts the task's transaction. To avoid re-triggering the same underlying fault, the Bristlecone runtime records the combination of tasks that caused the failure, and executes other tasks. In that respect, Bristlecone relies on a unique strategy to handle errors at the granularity of a task. In contrast, our approach can recover from a faulty entity without aborting the execution of a component. The recovery logic is defined at both application and system level, enabling the implementation of a range of recovery strategies.

Aspect-oriented programming. The tangling between the functional and exceptional code in software systems is one of the consequences of the lack of programming support to handle exceptions. Aspect-Oriented Programming (AOP) is a programming technique for modularizing concerns that cross-cut the programs [17], such as exception handling. Lipper and Lopes studied the reengineering of exception handling code using the AOP language, AspectJ [18]. Their study demonstrates that AOP reduces code tangling related to exception handling, and greatly reduces the corresponding portion of code.

Our approach also modularizes and factorises the system-level error-handling logic. It keeps application-specific error-handling logic in the application code, allowing context-specific treatments. Also, the AOP approach is specific to a target program, making it difficult to apply weaving actions to a range of programs.

Cacho *et al.* [5] have proposed an aspect-oriented approach, called EJFlow, that extends AspectJ and enables developers to modularize system-level exception-handling code. To do so, mechanisms are introduced to associate exception handlers with exceptional flows in Java code. Unlike our generative approach, AOP does not provide developers with programming support and guidance to implement exception-handling logic.

7. Conclusion and Future Work

In this paper, we have presented a domain-specific approach to architecturing and supporting error handling. We have extended an ADL with error handling declarations and showed that these declarations facilitate the separation between error handling and the application logic. We have demonstrated that our approach makes the programming of error handling more rigorous and systematic.

Our approach was successfully applied to the development of realistic pervasive computing applications for managing a building. We have illustrated our approach with three applications from this large project and discussed its benefits.

This work is being actively expanded in various directions. One of them consists of using fault tolerance strategies at the architectural level to generate more support for error handling. Another direction is to widen the scope of our approach to other non-functional concerns like security and performance.

Availability

The tools from this paper are available at `http://diasuite.inria.fr`

Acknowledgments

We thank Damien Cassou and Benjamin Bertran for developing DiaGen. We thank all the members of our research group for insightful discussions and careful reading of earlier versions of this paper. We thank Rémi Douence, Julia Lawall, Anne-Françoise Le Meur, and Jacques Noyé for their helpful comments. We also thank the anonymous referees for their valuable feedback. Julien Mercadal is supported by a fellowship from the French ministry of research.

References

[1] Portland pattern repository, 1995. URL `http://c2.com/cgi/wiki?ExceptionPatterns`.

[2] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Tansactions on Software Engineering and Methodology*, 6(3):213–249, 1997. ISSN 1049-331X. doi: 10.1145/258077.258078.

[4] J. Bruneau, W. Jouve, and C. Consel. DiaSim, a parameterized simulator for pervasive computing applications. In *Proceedings of the 6th IEEE International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous '09)*, Toronto, CAN, jul 2009. ICST/IEEE.

[5] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. EJFlow: Taming exceptional control flows in aspect-oriented program-

ming. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD '08)*, pages 72–83, New York, NY, USA, 2008. ACM Press. ISBN 978-1-60558-044-9. doi: 10.1145/1353482.1353492.

[6] D. Cassou, B. Bertran, N. Loriant, and C. Consel. A generative programming approach to developing pervasive computing systems. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE '09)*, pages 137–146, Denver, CO, USA, 2009. ACM Press. doi: 10.1145/1621607.1621629.

[7] S. Chetan, A. Ranganathan, and R. Campbell. Towards Fault Tolerant Pervasive Computing. *IEEE Technology and Society Magazine*, 24(1):38–44, 2005.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001.

[9] W. W. W. Consortium. Web services architecture, 2004. URL `http://www.w3.org/TR/ws-arch/`.

[10] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35726-2.

[11] B. Demsky and A. Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP '08)*, pages 490–515, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8.

[12] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, 2001. ISSN 0737-0024.

[13] T. B. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998. ISBN 0764580434.

[14] P. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computer Survey*, 35(2):114–131, 2003. ISSN 0360-0300. doi: 10.1145/857076.857078.

[15] F. C. Filho, P. H. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79(10):1397–1418, 2006. doi: 10.1016/j.jss.2006.02.060.

[16] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985. ISSN 0164-0925. doi: 10.1145/2363.2433.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242, 1997.

[18] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003. ISBN 1930110936.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. ISSN 0164-0925. doi: 10.1145/357172.357176.

[20] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 418–427. ACM Press, 2000.

[21] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995. doi: 10.1109/32.464548.

[22] J. Magee and J. Kramer. Dynamic structure in software architectures. *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '96)*, 21(6):3–14, 1996. ISSN 0163-5948. doi: 10.1145/250707.239104.

[23] S. Mostinckx, J. Dedecker, E. G. Boix, T. V. Cutsem, and W. D. Meuter. Ambient-oriented exception handling. In C. Dony, J. L. Knudsen, A. B. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2006. ISBN 3-540-37443-4.

[24] OMG. The common Object Request Broker: Architecture and specification. Technical Report 2.0, Object Management Group, 1995.

[25] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[26] S. R. Ponnekanti, B. Johanson, E. Kiciman, and A. Fox. Portability, extensibility and robustness in iROS. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom '03)*, page 11, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1893-1.

[27] G. R. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004. ISSN 1536-1268. doi: 10.1109/MPRV.2004.1321024.

[28] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom '05)*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2299-8. doi: 10.1109/PERCOM.2005.26.

[29] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002. ISSN 1536-1268. doi: 10.1109/MPRV.2002.1158281.

[30] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. Technical report, RFC 3261, Aug. 2002. URL `http://www.ietf.org/rfc/rfc3261.txt`.

[31] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN 0470167742, 9780470167748.