Architecturing Conflict Handling of Pervasive Computing Resources

Henner Jakob, Charles Consel, Nicolas Loriant

Architecturing Conflict Handling of Pervasive Computing Resources

Henner Jakob[1], Charles Consel[1], and Nicolas Loriant[2]

[1] INRIA Sud-Ouest, Bordeaux, France,
{henner.jakob,charles.consel}@inria.fr
[2] Imperial College, London, UK,
nloriant@doc.ic.ac.uk

Abstract. Pervasive computing environments are created to support human activities in different domains (*e.g.*, home automation and healthcare). To do so, applications orchestrate deployed services and devices. In a realistic setting, applications are bound to conflict in their usage of shared resources, *e.g.*, controlling doors for security and fire evacuation purposes. These conflicts can have critical effects on the physical world, putting people and assets at risk.
This paper presents a domain-specific approach to architecturing conflict handling of pervasive computing resources. This approach covers the software development lifecycle and consists of enriching the description of a pervasive computing system with declarations for resource handling. These declarations are used to automate conflict detection, manage the states of a pervasive computing system, and orchestrate resource accesses accordingly at runtime. In effect, our approach separates the application logic from resource conflict handling. Our approach has been implemented and validated on various building automation applications.

1 Introduction

The advances in telecommunication technologies and the proliferation of embedded networked devices are allowing the seamless integration of computing systems in our everyday lives. Nowadays, pervasive computing systems, as envisioned by Weiser [16], are being deployed in an increasing number of areas, including building automation and assisted living.

Typically, a pervasive computing environment consists of multiple applications that gather data from sensing devices, compute decisions from sensed data, and carry out these decisions by orchestrating actuating devices. For example, in building automation, motion and temperature sensors are used to automate lighting and regulate heating.

The rapid development of new devices (*i.e.*, resources), and development tools opened to third-parties, have paved the way to an increasing number of applications being deployed in pervasive computing environments. These applications access resources without any coordination between them because a pervasive computing platform needs to evolve as requirements change. In this situation, it is very common for a resource to be accessed by multiple applications, potentially leading to conflicts. For example, in a building management system, a security application that grants access inside the building can conflict with another application dealing with emergency situations like fires,

preventing the building to be evacuated. In fact, conflicts do not only occur across applications but also within an application. For example, different modules of an application may be developed independently of each other, creating a risk that conflicting orders to be issued to devices. Detecting, resolving and preventing intra- and inter-application conflicts is critical to make a pervasive computing system reliable. To do so, a systematic and rigorous approach to handling conflicts throughout the development lifecycle is required.

Detecting conflicts is a daunting task. Pervasive computing systems are complex and involve numerous applications that may conflict on one or multiple resources. Scaling up conflict handling for real-size pervasive computing systems requires to distinguish potential conflicts from safe resource sharing. This may depend on the type of a resource, for example, a conflict may occur on a resource providing mutually exclusive operations (*e.g.,* locking and unlocking a door). This may also depend on the applications being deployed in a pervasive computing environment (*e.g.,* two applications may access a device inconsistently), precluding application developers from anticipating potential conflicts. Without any support, detecting potential conflicts requires to examine the code of all the applications to identify each resource usage, and determine whether it may conflict.

After potential conflicts are pinpointed, it is necessary to resolve each of them. It requires intimate knowledge about the code of the corresponding applications to resolve the conflicts by making code changes. Because of the lack of high-level programming support, writing system-wide conflict-handling strategies is often overlooked. This situation results in polluting the logic of applications with ad hoc code, compromising the system maintainability.

The situation is exacerbated by the fact that pervasive computing environments are prone to changes: applications as well as resources emerge, evolve, and may disappear over time. These changes directly impact conflict management. This problem is well known in the telecommunications domain where it was observed that the number of potential conflicts grows exponentially as new applications are added to an existing system [10]. Manually handling conflict thus becomes impractical.

Our approach

Managing conflicts is often decomposed into three stages: detection, resolution and prevention [10]. In practice, these stages crosscut the development lifecycle of applications and pervasive computing systems.

We introduce an approach to conflict management that covers the lifecycle of a pervasive computing system. It consists of a design method for applications, supported by declarations and tools, separating conflict management tasks. This approach facilitates the work of architects, developers and administrators: requirements for conflict management are propagated throughout the development stages.

We propose to declare a pervasive computing system and its applications using a domain-specific architecture description language (ADL), named DiaSpec [5], developed in our research group. This ADL serves two purposes: (1) it allows domain experts to describe the available resources in the pervasive computing environment, and (2) it is used by software architects to design applications with respect to the declared

resources. We extended DiaSpec with conflict-handling declarations that allow domain experts to characterize resources from a conflict-management viewpoint. This information, in combination with the architecture descriptions, allows to automatically pinpoint places where conflicts can occur.

To resolve the detected conflicts, we propose to raise the level of abstraction beyond the code level, by providing declarative support for conflict resolution. Within an application, the developer uses declarations to specify states for a pervasive computing system and order them with respect to their critical nature (*e.g.,* fire is more critical than intrusion). These states are enabled and disabled depending on runtime conditions over the pervasive computing system (*e.g.,* fire detection). State changes are used to update access rights to conflict-sensitive resources (*e.g.,* in case of fire, the fire module takes precedence over the intrusion module). Our approach is incremental in that states and priorities can be added as a pervasive computing system is enriched with new applications. Its declarative nature allows to prevent conflict-handling logic from polluting the application logic.

Conflict-extended architecture descriptions are used to generate customized programming frameworks. These frameworks guide and support the implementation of the conflict-handling logic. Generating the underlying framework from the architecture description guarantees that the architecture implementation can only access the required resources. Additionally, runtime support ensures that access to resources are granted in conformance with conflict-handling declarations.

Our contributions can be summarized as follows.

- *Extended development cycle* – We have identified the requirements at different development stages to detect, resolve, and prevent conflicts. We have seamlessly integrated conflict-management activities into a software development lifecycle.
- *Conflict-handling declarations* – We have extended a domain-specific ADL to declare conflict resolution at an architectural level. A declarative approach is introduced to define the states of a pervasive computing system and their critical nature. Such declarations form the basis to define the conflict-handling logic of a pervasive computing system.
- *Programming support* – Conflict-handling declarations are used to augment the generated programming framework with code dedicate to conflict handling. This code (1) guides the implementation of the conflict handling logic within and across applications, and (2) generates code that manages resource accesses to prevent runtime conflicts.

The rest of this paper is organized as follows. Section 2 identifies the key requirements to manage resource conflicts. Section 3 presents how to integrate conflict management into the development cycle. Section 4 outlines our implementation. Section 5 evaluates our approach. Related works are discussed in Section 6, and concluding remarks are given in Section 7.

2 Background and requirements

In this section, we first present a domain-specific architecture description language, named DiaSpec [5]. The underlying development process is illustrated with a working

example of building management. Second, we examine the requirements for managing resource conflicts when developing pervasive computing applications.

2.1 Background

The DiaSpec language enforces an architectural pattern, named sense-compute-control, commonly used in the pervasive computing domain [6]. This pattern distinguishes three types of components, as depicted in Figure 1: (1) *resources*, which provide sensing and actuating capabilities on a pervasive computing environment[3], (2) *contexts*, which aggregate and process sensed data, and (3) *controllers*, which receive information from contexts and invoke actuators. This architectural pattern goes beyond the pervasive computing domain and enables high-level programming support and a range of verifications [3, 4, 7].

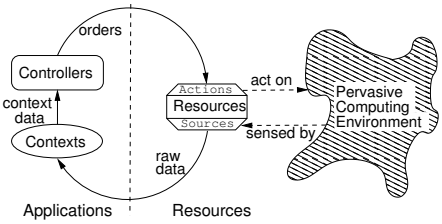

Fig. 1. DiaSpec architectural pattern

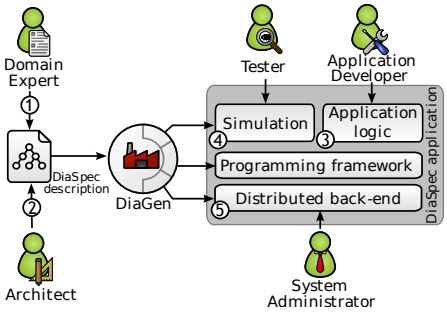

Fig. 2. DiaSpec development cycle

Figure 2 shows how a DiaSpec description drives a five-stage development process. (1) A domain expert declares a taxonomy of resources that can be found in the pervasive computing environment. (2) An architect describes the interactions between resources, contexts and controllers. Given a taxonomy and an architecture description, a compiler, named DiaGen, generates a customized programming framework in Java. (3) The generated framework is used by the developer to implement the application. (4) The application code can be tested as is, prior to deployment, using a simulator for a pervasive computing environment, named DiaSim [1]. (5) A system administrator can deploy the application in a real pervasive computing environment. The suite of tools supporting our development process is called DiaSuite[4].

We now focus on the first three steps of our development process with an application that treats different types of emergencies in a building.

[3] Resources are devices (*e.g.,* a motion detector) or software components (*e.g.,* an address book).
[4] DiaSuite is freely available `http://diasuite.inria.fr` and open source.

```
1  device LocDevice {
2    attribute location as Location;
3  }
4  device SmokeSensor extends LocDevice {
5    source smoke as Float;
6  }
7  device TempSensor extends LocDevice {
8    source temperature as Float;
9  }
10 device Door extends LocDevice {
11   source status as LockedStatus;
12   action LockUnlock;
13 }
14 device Alarm extends LocDevice {
15   action OnOff;
16 }
17 device Sprinkler extends LocDevice {
18   action OnOff;
19 }
20 device Logger { action Log; }
21
22 action LockUnlock {
23   lock();
24   unlock();
25 }
26 action Log {
27   logEvent(event as String);
28 }
29 action OnOff { on(); off(); }
```

Fig. 3. Extract of the emergency management taxonomy

```
1  context AvgTemp as Float
2      indexed by location as Location {
3    source temperature from TempSensor;
4  }
5
6  context SmokeDetected as Boolean
7      indexed by location as Location {
8    source smoke from SmokeSensor;
9  }
10
11 context Fire as Boolean
12     indexed by location as Location {
13   context AvgTemp;
14   context SmokeDetected;
15 }
16
17 context DoorStatus as Boolean
18     indexed by location as Location {
19   source status from Door;
20 }
21
22 controller FireCtrl {
23   context Fire;
24   context DoorStatus;
25   action LockUnlock on Door;
26   action OnOff on Alarm, Sprinkler;
27   action Log on Logger;
28 }
```

Fig. 4. Extract of the architectural description of the fire module

Describing the environment First, the domain expert declares the available resources of a pervasive computing environment, as is done using an interface description language (*e.g.,* WSDL) to declare external resources. In DiaSpec, this process is supported by a language layer 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.

Figure 3 shows an excerpt of the taxonomy for the emergency application. Specifically, to detect a fire, the application uses temperature and smoke sensors deployed in the building. Upon fire detection, the doors are unlocked to ensure the safe evacuation of all the building occupants. Additionally the alarms of the building are turned on, as well as the sprinklers nearby the fire. All actions are logged for later analyses.

The domain expert introduces the resource classes with the **device** keyword. In Figure 3, a root device with a location attribute is declared (see lines 1 to 3). Attributes mainly serve as filters for resource discovery in the pervasive computing environment. The **source** and **action** keywords define the capabilities of a resource. For example, line 5 declares that the smoke sensor produces a float value, indicating the current smoke intensity. The `Door` device provides the `LockUnlock` action (line 12), which is further detailed in lines 22 to 25.

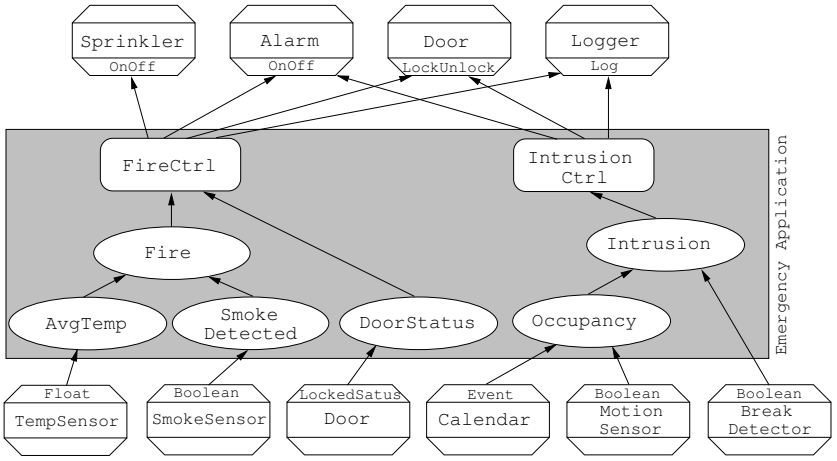

Fig. 5. Architecture of the emergency application

Describing the architecture To support application design, the DiaSpec language offers an ADL layer, based on the architectural pattern depicted in Figure 1, and comprises resource, context and controller components.

To illustrate the ADL layer, let us examine the emergency application. Figure 4 presents an excerpt of the corresponding DiaSpec declarations, describing the fire module. Figure 5 shows a graphical view of the emergency application, including the intrusion module. The arrows indicate the flow of information. The resources at the bottom of the diagram provide information to context components; the resources at the top provide the controller components with actions on the environment.

The temperature sensors of a room send their values to the `AvgTemp` component. Figure 4, lines 1 through 4, introduces this component using the **context** keyword. It includes a **source** declaration, defining the input of this component. The **as** keyword, line 1, is followed by the type of the output value (`Float`). The value is indexed by a location: the room where the average temperature is measured. Another context component, `SmokeDetected`, gathers information from smoke detectors. Both contexts, the average temperature and the smoke information, are used by the `Fire` component to determine whether there is a fire in the building and its location. Eventually, if there is fire, the `FireCtrl` component is invoked. It is declared by the **controller** keyword (line 22). This component declares two input sources using the **context** keyword and referring to `Fire` and `DoorStatus` (lines 23 to 24). The **action** keyword defines the actuator operations that can be invoked by a controller component. In our example, the `FireCtrl` component can lock/unlock doors, turn on/off alarms and sprinklers, and log events (lines 25 to 27).

Implementing an application The customized programming framework produced by DiaGen consists of an abstract class for each DiaSpec component (resource, context, and controller). The abstract class includes methods that implement the programming

support (*e.g.,* resource discovery and component communication mechanisms). The application logic to be provided by the developer is declared as abstract methods. Implementing the application logic is done by subclassing a generated abstract class.

2.2 Requirements

Let us now define our notion of resource conflict and examine the issues to be resolved within the DiaSpec development approach.

Intra-application resource conflicts Sensors and actuators need to be distinguished when it comes to resource conflicts. Indeed, sensors can sustain many consumers, requesting values either directly (*e.g.,* remote procedure call) or via some runtime support (*e.g.,* notification server). The situation would be comparable for actuators, if only they did not have side effects on the environment. This is illustrated in Figure 5, where the `FireCtrl` and `IntrusionCtrl` controllers share resources. These controllers can, for example, have conflicting effects on the door resource, depending on whether the current state of the pervasive computing environment requires anti-intrusion or firefighting measures.

What this example illustrates is that resolving resource conflicts relies on some notion of state that determines which consumer should acquire the resource. A pervasive computing environment can be in different states depending on a variety of conditions. Expressing these conditions is a key to providing a practical approach to conflict resolution. To separate this concern from the application logic, the approach should target the architecture level. In the door example, we would need to introduce states, enabled by conditions over relevant sensed data (*e.g.,* smoke intensity, motion detection). Based on the enabled states, the attempts of the controllers to acquire the doors would be prioritized.

Note that some actuators can be insensitive to conflicts. An example is the `log` action (lines 26 to 28): it can record data in any order, assuming each invocation has the necessary contextual information (*e.g.,* a time stamp).

Inter-application resource conflicts The emergency application is only a part of the building management system. The system administrator also deploys a security application to manage access in the building. Figure 6 shows a graphical representation of two applications: emergency and security. Both applications operate the same type of resources, in this case door and logger.

As can be noted, resource conflicts occur at different levels and require to be managed globally. Even though, conflicting usage of resources can be resolved with respect to a given state, there needs to be a global, system-wide approach to combining unitary strategies in a transparent and predictable way.

3 Conflict management

This section presents our approach to conflict management. It addresses the requirements discussed previously, and illustrates the approach with the building management system.

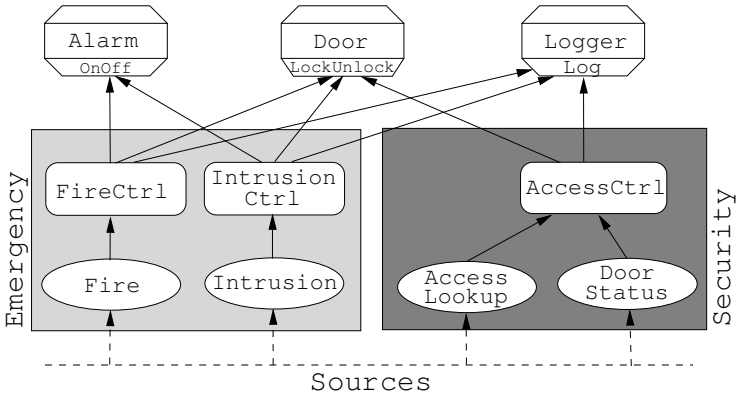

Fig. 6. Potential resource conflicts between multiple controller components

3.1 Detecting potential conflicts

Our approach to conflict management revolves around the DiaSpec description of an application. Such a description exposes the interactions with actuators, allowing resource conflicts to be detected within an application, for the application developer, and between applications, for the system administrator.

Let us examine how the intra-application conflicts between the fire and the intrusion modules are solved (Figure 6). The process is the same for inter-application conflicts.

In DiaSpec, conflicts may occur when a resource is used by more than one controller component. Information about the resource usage can be extracted from the DiaSpec description of an application. This information needs to be refined to account for actions that are insensitive to resource conflicts (*e.g.,* the `log` action).

Categorizing actions in the taxonomy We extended the taxonomy language of DiaSpec with effect declarations for resource actions. An effect declaration applies to an action (*i.e.,* an interface and its associated operations), which is part of a device declaration. In practice, we have identified three main effects that need to be expressed. First, a device includes an action with operations that are mutually exclusive in their effects. For example, a door is either locked or unlocked. Such an action is declared with the **exclusive** keyword. Second, a device combines operations that interfere with each other. For example, a multimedia device could include two actions: an audio player and a video player; if both players run simultaneously, they interfere with each other. The list of interfering actions of a device is declared with the **interfering** keyword[5]. Lastly, when an action is conflict insensitive, it is declared without effect keywords.

In our example, the domain expert has to enrich the declaration of the `Door`, `Alarm` and `Sprinkler` devices with the exclusive keyword, as is shown in Figure 7. The `Log` action is left unchanged because it is conflict insensitive.

[5] Interfering actions do not occur in our building management example.

```
device Door extends LocDevice {
  source status as LockedStatus;
  exclusive action LockUnlock;
}
device Alarm extends LocDevice {
  exclusive action OnOff;
}
device Sprinkler extends LocDevice {
  exclusive action OnOff;
}
```

Fig. 7. The taxonomy contains three devices with exclusive actions

Analyzing the architecture description Given the taxonomy declarations enriched with conflict-handling information, the application developer and, later in the process the system administrator, investigate potential resource conflicts. A resource usage raises a potential conflict when two or more controllers may access it. These controllers may be defined within an application or across applications. In our approach, potential resource conflicts are automatically detected from a DiaSpec description. Conflict resolution is expressed with declarations, leaving the application logic unchanged.

3.2 Declaring conflict resolution

To resolve conflicts, we partition resource users with respect to a set of *states* in which a pervasive computing environment can be. These states are totally ordered with respect to their assigned priority level; they are associated with resource users (*i.e.,* controller components). For example, our building can be in either of the following states, listed in order of increasing priority: normal, security, or emergency. In doing so, applications and controllers, within an application, can be assigned different states, resolving their access to conflicting resources.

To complete our approach, we need to enable and disable states depending on evolving conditions of the pervasive computing environment. This is done by introducing *state component*, leveraging the DiaSpec notion of context component. Recall that such a component receives information about the pervasive computing environment (*e.g.,* smoke, fire, ...). A state component uses this information to determine whether the conditions for a given state hold, producing a boolean value.

```
system state SecuritySt
    priority 5 to Security {
  source date from Calendar;
}
system state EmergencySt
    priority 10 to Emergency {
  application state FireASt;
  application state IntrusionASt;
}
```

Fig. 8. System state-component declarations (inter-application conflicts)

```
application state FireASt
    priority 15 to FireCtrl {
  source temperature from TempSensor;
  source smoke from SmokeSensor;
}
application state IntrusionASt
    priority 10 to IntrusionCtrl {
  context Intrusion;
}
```

Fig. 9. Application state-component declarations (intra-application conflicts)

Let us illustrate our approach with inter- and intra-application conflict resolution. Consider Figure 8 where two state components are defined (lines 1 to 9): `SecuritySt` and `EmergencySt`. These components are declared with the **system** keyword to indicate that they apply system-wide, allowing the system administrator to resolve inter-application conflicts. With the **priority** keyword, they are assigned priority values of 5 and 10, respectively, indicating that `SecuritySt` is less critical than `EmergencySt`. Following the **to** keyword is the applications to which the declared state applies. The conditions under which a state holds are parameterized by information sources, as is declared for the `SecuritySt` state with the `Calendar` source. As well, the conditions may be parameterized by other states, as is defined by the `EmergencySt` state with `FireASt` and `IntrusionASt`. In fact, these two states are used to resolve intra-application conflicts, promoting state-component reuse – the states are defined in Figure 9 (lines 1 to 9).

Application state components are declared with the **application** keyword by the application developer and apply to controller components declared within an application. For example, the `FireASt` state applies to the `FireCtrl` and `IntrusionASt` to `IntrusionCtrl`. Both controllers, and associated states, are local to the `Emergency` application. This local nature also applies to the priority defined by application states. That is, these priorities resolve conflicts within an application. In our example, these declarations prioritize `FireCtrl` over `IntrusionCtrl`. In doing so, intra-application conflicts for resources, such as doors, can get resolved.

3.3 Implementing conflict resolution

Declarations of conflict handling are enforced by additional code produced by DiaGen, shielding the application developer and system administrator from low-level implementation details.

Let us illustrate the implementation of the conflict-handling logic by considering the declaration of the `FireASt` state in Figure 9. This state component relies on two information sources, temperature and smoke, to determine whether the building is on fire.

Figure 10 shows an implementation of this state component. In lines 7 and 8, the component subscribes to all the required sensors. To keep track of the building situation, the component stores temperature and smoke values from all the locations within the building. Specifically, the component implementation updates the value (temperature or smoke) for each location (lines 12 to 21). After refreshing the value, it checks whether the condition for a fire holds by calling the `checkFire` method (line 17). This method determines whether or not a fire is occurring by publishing a boolean value (line 35), which in turn will enable or disable the corresponding state of the pervasive computing system (*i.e.,* `FireASt`).

4 Implementation

To achieve our conflict management approach, we have extended DiaSpec, DiaGen, and the DiaSpec runtime. The extended DiaSpec runtime is illustrated by our building

```
1  public class FireASt extends AbstractFireASt {
2
3    private Map<Location, Map<String, Value>> status;
4
5    public void initialize() {
6      status = new HashMap<Location, Map<String, Value>>();
7      allTempSensors().subscribeTemperature();
8      allSmokeDetectors().subscribeSmoke();
9    }
10
11   @Override
12   public void onNewTemperature
13         (Location loc, Temperature temperature) {
14     Map<String, Value> values = getValues(loc);
15     values.put("temperature", temperature.value());
16     status.put(loc, values);
17     checkFire();
18   }
19
20   @Override
21   public void onNewSmoke(Location loc, Smoke smoke) {...}
22
23   private Map<String, Value> getValues(Location loc) {...}
24
25   private void checkFire(){
26     boolean fireDetected = false;
27     for(Location loc : status.keySet()){
28       Map<String, Value> values = status.get(loc);
29       if(values.get("temperature").equals(Temperature.HIGH)
30         && values.get("smoke").equals(Smoke.HIGH)){
31           fireDetected = true;
32           break;
33       }
34     }
35     setFireASt(fireDetected);
36   }
37 }
```

Fig. 10. An implementation of the `FireASt` state component

example in Figure 11. We introduce a `ConflictCtrl` component that subscribes to state components to gather information about states that get enabled/disabled at runtime. It combines this information with state priorities to compute access rights, and update the `enforcer` components, associated with each resource class (*e.g.,* `Door`). The `enforcer` components intercept resource accesses and decide whether or not to grant them. Specifically, an `enforcer` component intercepts a method call and creates a request of the form `(controller, action, resource)`. Such a request is matched against an access control list (ACL) attached to each resource class; this ACL comprises rules of the following form.

```
(controller, action, resource, [true|false])
```

When the request matches an ACL entry, the access to the resource is granted depending on the boolean value of the corresponding rule.

Globally, the conflict management process is performed in two stages. Statically, potential conflicts are detected based on the taxonomy and architecture declarations. For each detected conflict on a resource, the DiaSpec description is searched to identify state components dedicated to resolving it. This information is used to parameterize the `ConflictCtrl` component. Dynamically, this component will update the ACL of the `enforcer` component of all the resources impacted by a state change. The updated ACL is calculated based on the enabled/disabled states and their priorities.

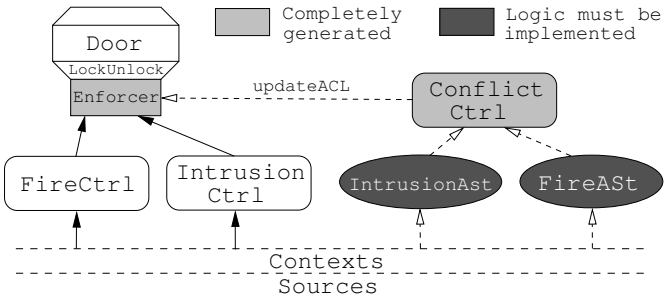

Fig. 11. Extended DiaSpec runtime system

5 Evaluation

To assess the usability of our approach, we applied it to the building management system. This case study was particularly interesting because it had been specified in DiaSpec and implemented, prior to the development of our approach. As a result, it could serve as a reference implementation, and a basis to be extended with our conflict-handling approach. We focus on the comprehensibility and reusability of conflict managing code, and the ability to detect conflicts. To test the correct behaviour of both implementations, original and extended, we used our pervasive computing simulator, DiaSim [1].

The original building management system was developed by members of our group who have expert knowledge in DiaSpec. They acted as architect, developer, administrator, and used their expertise to solve the foreseeable conflicts. The lack of proper support made them resort to ad hoc strategies to resolve resource conflicts. For example, to prevent three different controllers to conflict in accessing doors, they had to introduce a dedicated action to the door resource for each kind of controller in the taxonomy. This action would essentially mimic our conflict resolution strategy, taking a state as a parameter and determining whether to grant access to the door. In contrast to our approach, this ad hoc technique requires to structure the taxonomy with respect to conflict handling concerns and to pollute the application code with conflict handling logic.

With our approach, adding a new application to an existing system requires to declare and implement an additional system state component, if a new state is needed. In this case, the new system state component is independent from other components, besides the new priority level to be introduced.

6 Related Work

Conflicts are a major problem in a variety of domains. For example in telecommunications, Keck and Kühn show that feature interaction is an exponential problem that appears when new services are added to an existing system [10]. This problem can be directly mapped to pervasive computing, their services and features are our applications

and their actions on resources. Calder and Miller [2] use the Spin model checker to analyze telecommunication systems. To do so, a system (services and features) is modeled in Promela using temporal properties. Our approach circumvents the feature interaction problem by relying on existing system specifications and conflict-handling declarations provided by the domain expert and the application developer.

There exist different strategies to resolve conflicts in pervasive computing environments. The idea of proactively changing access control on resources is also used by Gupta *et al.* [8]. They present a criticality-aware access control approach that is only studied as a conceptual model. In contrast, we cover conflict management throughout the development lifecycle: from design to programming, to runtime.

Haya *et al.* assign a priority to every operation [9]. The priority is calculated by a central component using information about the current state, the caller and the type of operation. In comparison, our approach incurs little overhead for resource invocations because the *enforcer* component is coupled with the resource, preventing any central component from becoming a bottleneck.

The work closest to ours is that of Retkowitz and Kulle [11]. They use the notion of dependency management for handling resource conflicts. It is exemplified in the context of smart homes where it allows fine-grained configuration of a conflict-aware middleware. It is designed so a user can interact with the system and set priorities for different applications. In comparison, our approach is not limited to the home automation domain and addresses conflict handling throughout the development lifecycle.

Tuttlies *et al.* have a different approach to resolving conflicts. They propose to describe the side effects of an application on the physical environment [15]. Additionally, each application states, what it considers a conflict. As a result, they can detect conflicts between interfering applications. Devising and applying a suitable strategy is left to the application developer. In contrast, we aim for a system-wide conflict management to allow system-wide reasoning.

7 Conclusion

In this paper we have presented a domain-specific approach to architecturing conflict handling of resources. This approach covers the development lifecycle of a pervasive computing system. Our approach includes a detection of potential conflicts, their resolution, and their prevention at runtime. We extended an ADL to add information that is required for these three stages of conflict management. This information is used to generate code that guides and supports the implementation of conflict management.

We have introduced new tasks dedicated to conflict management in the development process of a pervasive computing system. In the resulting process, application and conflict handling code are cleanly separated. Furthermore, our approach to conflict management is incremental and modular, preserving the independence between applications. This facilitates reuse of applications, and makes the conflict management easier to understand and verify.

To prevent conflicts, our implementation enforces an ACL for each pervasive computing resource. These ACLs are proactively updated based on the current system state.

Currently, our approach treats conflicts for classes of resources. This strategy applies to situations where applications act on all instances of a class (*e.g.*, the emergency application unlocks all doors). We plan to extend this approach by introducing a perimeter (*e.g.*, a room, a floor) in the conflict management declaration of an application.

We also plan to expand our model to include the access rights for users. Access control is a major problem in pervasive computing environments, since it must handle physical and virtual objects at the same time [12, 14]. A related research direction is to integrate user preferences into our model to resolve certain types of conflicts, as proposed by Shin *et al.* [13].

References

1. J. Bruneau, W. Jouve, and C. Consel. DiaSim, a parameterized simulator for pervasive computing applications. In *International Conference on Mobile and Ubiquitous Systems*, 2009.
2. M. Calder and A. Miller. Feature interaction detection by pairwise analysis of LTL properties - a case study. *Formal Methods in System Design*, 28(3):213–261, 2006.
3. D. Cassou, E. Balland, C. Consel, and J. Lawall. Architecture-driven programming for sense/compute/control applications. In *International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2010.
4. D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *International Conference on Software Engineering*, 2011.
5. D. Cassou, B. Bertran, N. Loriant, and C. Consel. A generative programming approach to developing pervasive computing systems. In *International Conference on Generative Programming and Component Engineering*, 2009.
6. A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *H.-C. I.*, 16(2):97–166, 2001.
7. S. Gatti, E. Balland, and C. Consel. A step-wise approach for integrating QoS throughout software development. In *European Conference on Fundamental Approaches to Software Engineering*, 2011.
8. S. K. S. Gupta, T. Mukherjee, and K. Venkatasubramanian. Criticality aware access control model for pervasive applications. In *International Conference on Pervasive Computing and Communications*, 2006.
9. P. A. Haya, G. Montoro, A. Esquivel, M. García-Herranz, and X. Alamán. A mechanism for solving conflicts in ambient intelligent environments. *J. UCS*, 12(3):284–296, 2006.
10. D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24:779–796, 1998.
11. D. Retkowitz and S. Kulle. Dependency management in smart homes. In *International Conference on Distributed Applications and Interoperable Systems*, 2009.
12. G. Sampemane. *Access Control For Active Spaces*. PhD thesis, University of Illinois, 2005.
13. C. Shin, A. K. Dey, and W. Woo. Mixed-initiative conflict resolution for context-aware applications. In *International Conference on Ubiquitous Computing*, pages 262–271, 2008.
14. W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong. Access control in collaborative systems. *ACM Computing Surveys*, 37:29–41, 2005.
15. V. Tuttlies, G. Schiele, and C. Becker. Comity - conflict avoidance in pervasive computing environments. In *International Workshop on Pervasive Systems*, 2007.
16. M. Weiser. The computer for the twenty-first century. In *Scientific American*, volume 265, pages 94–104, 1991.