A Generative Programming Approach to Developing Pervasive Computing Systems

Damien Cassou, Benjamin Bertran, Nicolas Loriant, Charles Consel

A Generative Programming Approach to Developing Pervasive Computing Systems

Damien Cassou Benjamin Bertran Nicolas Loriant Charles Consel
INRIA/LaBRI/ENSEIRB
{damien.cassou, benjamin.bertran, nicolas.loriant, charles.consel}@inria.fr

Abstract

Developing pervasive computing applications is a difficult task because it requires to deal with a wide range of issues: heterogeneous devices, entity distribution, entity coordination, low-level hardware knowledge... Besides requiring various areas of expertise, programming such applications involves writing a lot of administrative code to glue technologies together and to interface with both hardware and software components.

This paper proposes a generative programming approach to providing programming, execution and simulation support dedicated to the pervasive computing domain. This approach relies on a domain-specific language, named DiaSpec, dedicated to the description of pervasive computing systems. Our generative approach factors out features of distributed systems technologies, making DiaSpec-specified software systems portable.

The DiaSpec compiler is implemented and has been used to generate dedicated programming frameworks for a variety of pervasive computing applications, including detailed ones to manage the building of an engineering school.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures, Languages, Patterns; D.3.4 [*Programming Languages*]: Processors—Code generation, Retargetable compilers

General Terms Design, Languages

Keywords pervasive computing, generative programming, DSL

1. Introduction

Pervasive computing systems are being deployed in a rapidly increasing number of areas, including building automation, assisted living, and supply chain management. Regardless of their target area, pervasive computing systems have a typical architectural pattern. They aggregate data from a variety of distributed sources, whether sensing devices or software components, analyze a context to make decisions, and carry out decisions by invoking a range of actuators. Because pervasive computing systems are standing at the crossroads of several domains (*e.g.,* distributed systems, multimedia, and embedded systems), they raise a number of challenges in software development.

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.

Heterogeneity. Pervasive computing systems are made of off-the-shelf entities, that is, hardware and software building blocks. These entities run on specific platforms, feature various interaction models, and provide non-standard interfaces. This heterogeneity tends to percolate in the application code, preventing its portability and reusability, and cluttering it with low-level details.

Lack of structuring. Pervasive computing systems coordinate numerous, interrelated components. A lack of global structuring makes the development and evolution of such systems error-prone: component interactions may be invalid or missing.

Combination of technologies. Pervasive computing systems involve a variety of technological issues, including device intricacies, complex APIs of distributed systems technologies, undocumented interfaces, and middleware-specific features. Coping with this range of issues results in code bloated with special cases to glue technologies together.

Dynamicity. In a pervasive computing system, devices may either become available as they get deployed, or unavailable due to malfunction or network failure. Dealing with these issues explicitly in the implementation can quickly make the code cumbersome.

Testing. Pervasive computing systems are complicated to test. Doing so requires equipments to be acquired, tested, configured and deployed. Furthermore, some scenarios cannot be tested because of the nature of the situations involved (*e.g.,* fire and smoke). As a result, the programmer must resort to writing specific code to achieve ad hoc testing.

Software engineering approaches provide developers with general-purpose support, targeting a wide spectrum of areas, from business applications to scientific computing. For example, middlewares like CORBA [OMG 1995] offer a profusion of services, large APIs, and abstraction layers to hide low-level intricacies. Yet, they do not cover the specific range of issues involved in developing a pervasive computing system: heterogeneous entities, application structuring, combination of technologies, dynamicity, and testing. Leveraging existing software engineering approaches thus requires the programmer to write code to bridge the gap between general-purpose support and pervasive computing-specific needs.

Our approach

We propose a generative programming approach to providing support throughout the development of a pervasive computing system: programming, simulation and execution. This approach relies on a domain-specific language, named DiaSpec, dedicated to describing a pervasive computing system. Let us outline the development process underlying DiaSpec. This process revolves around key stages and roles depicted in Figure 1.

Taxonomy. Because of their heterogeneity, entities of pervasive computing environments need to be specified in a high-level manner to abstract over their variations. The number of existing entities also require to characterizing the ones that are relevant to a given area. To address these issues, we provide an area expert with a declarative language to define a hierarchy of building blocks (stage ①). Each class of building blocks is characterized in terms of the types of data that are gathered from the environment and the actions that are supported.

Architecture. A taxonomy definition is used as a basis to declare the architecture of pervasive computing applications. To do so, we introduce a declarative language inspired by architecture description languages (ADLs) [Medvidovic and Taylor 2000] but dedicated to an architectural pattern commonly used in the pervasive computing domain [Dey et al. 2001]. This architectural pattern consists of context components fueled by sensing building blocks from the taxonomy. These components process gathered data to make them amenable to the application needs. Context data are then passed to controller components that trigger actions in building blocks (stage ②).

Development. A DiaSpec description, consisting of a taxonomy definition and architecture declarations, is then passed to a compiler, called DiaGen. This compiler produces a Java programming framework, dedicated to the input DiaSpec description (stage ③).

DiaGen generates high-level, DiaSpec-specific operations to support the development of devices and applications (stages ④ and ⑤). In doing so, the generated support abstracts away from underlying technologies. As well, it raises the level of abstraction by implementing a language to query a pervasive computing environment to discover entities. This language is specific to each DiaSpec description.

Simulation. To test pervasive computing applications prior to deploying a real environment, the DiaGen compiler generates simulation support. It produces support to implement the simulated building blocks from a set of predefined behaviors, whenever possible. This generated support is then linked to our pervasive computing simulator, named DiaSim [Bruneau et al. 2009]. DiaSim provides an editor to build a simulated environment and a 2D-renderer to guide and visualize the simulation (stage ⑥).

Deployment. Last, the system administrator deploys the pervasive computing system. To this end, a distributed systems technology needs to be selected. Currently, DiaGen offers a back-end for the following targets: Web Services [Consortium 2004], RMI [Downing 1998], CORBA [OMG 1995] and SIP [Rosenberg et al. 2002]. In doing so, we close the gap between these general-purpose software layers and the pervasive computing-specific needs (stage ⑦). This targeting does not necessitate any changes in the application code. Furthermore, the simulated environment can be mixed with the actual environment. This form of hybrid simulation enables applications to migrate incrementally to an actual environment, performing unit tests on actual components.

Our contributions

In this paper, we propose a domain-specific approach to supporting the development of pervasive computing systems. We generate a programming framework dedicated to a high-level description of a pervasive computing system. Our contributions can be summarized as follows.

- *DiaSpec.* We introduce a declarative language dedicated to specifying pervasive computing systems. DiaSpec includes (1) a language to define a taxonomy of the kinds of building blocks relevant to a target area and (2) an ADL-inspired language to describe application architectures. DiaSpec has been designed to permit the generation of support covering the development cycle of a pervasive computing system: programming, simulation and execution.
- *DiaGen.* DiaSpec descriptions are processed by a compiler, named DiaGen, that generates dedicated programming frameworks. A dedicated framework guides the development of the application code and raises the level of abstraction by providing the programmer with high-level operations for entity discovery and component interactions.
- *Retargetable compiler.* DiaGen leverages existing distributed systems technologies. It generates glue code to customize them with respect to the needs of pervasive computing, without manual intervention. Currently, DiaGen offers back-ends for the following targets: Web Services, RMI, SIP and CORBA.
- *Simulation generation.* DiaSpec declarations are also used to generate simulation support. It takes the form of predefined behaviors for the most common existing building blocks. It also generates skeletons to implement the simulation logic of a new kind of building block. Finally, code is produced to configure our pervasive computing simulator, DiaSim.

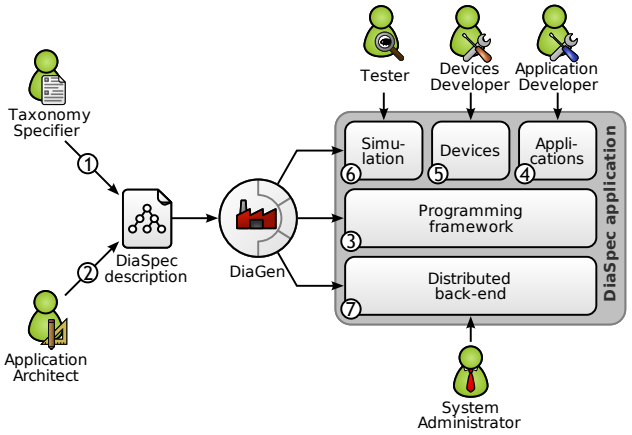

Figure 1. Development cycle

The rest of this paper is organized as follows. Section 2 introduces DiaSpec throughout an example. Section 3 examines the different elements generated in the dedicated programming framework. This section also describes how to develop on top of a DiaGen-generated programming framework. In Section 4, we present the generation of a layer specific to a distribution systems technology. Section 5 discusses the generation of a simulator configuration. In Section 6, we describe our implementation and the evaluation of our approach. Related works are discussed in Section 7 and conclusions are given in Section 8.

2. DiaSpec

This section introduces the DiaSpec language. First, we examine how to specify the taxonomy of a pervasive computing environment. Then, we present declarations to define the architecture of a pervasive computing application.

2.1 Working example

To illustrate our approach, an example of fire management is used throughout this paper. This example is part of a larger project aimed to automate an entire engineering school building. This project is briefly presented in Section 6.

To manage fire situations, we assume fire is detected by analyzing data from smoke and temperature sensors. When a fire occurs the corresponding application must trigger sprinklers and alarms, and unlock doors to allow occupants to evacuate the building.

2.2 Specifying a taxonomy

Pervasive computing applications interact with the environment through building blocks. Whether software or hardware, we view these building blocks as devices. A taxonomy is a collection of device declarations, each of which characterizes a set of possible device implementations sharing common functionalities. Device functionalities consist of data sources and actions. A data source specifies values sensed by a device. An action declares a set of operations supported by a device. Device declarations also include attributes, characterizing properties of devices instances. Device declarations are organized hierarchically allowing devices to inherit attributes, sources and actions from other devices.

Let us now describe device declarations in more detail by examining the taxonomy for fire management given in Figure 2. Device classes are introduced by the `device` keyword. At the root of our taxonomy is the `Device` node (line 1). It introduces the `location` attribute. The `SmokeDetector` device extends the root node and defines the `Smoke` data source, using the `source` keyword (lines 3 to 5). Another example of device declaration is `Door` (lines 15 to 18). Besides a data source giving the door state, `Door` defines the `Locking` action interface to operate a door lock. An action interface defines the signatures of methods supported by a device (lines 20 to 22). These methods are triggered by applications. As explained in Section 3, the hierarchy of device classes and the attribute declarations are used to generate support for device discovery.

```
1  device Device(Location location){}
2
3  device SmokeDetector extends Device {
4    source Smoke;
5  }
6  device TemperatureSensor(Accuracy accuracy) extends Device {
7    source Temperature;
8  }
9  device Sprinkler extends Device {
10   action OnOff;
11 }
12 device Alarm extends Device {
13   action Activation;
14 }
15 device Door extends Device {
16   action Locking;
17   source LockedStatus;
18 }
19
20 action OnOff {on(); off(); toggle();}
21 action Activation {activate(AlarmType alarmType); deactivate();}
22 action Locking {lock(); unlock();}
23
24 enum LockedStatus {LOCKED, UNLOCKED}
25 enum AlarmType {FIRE, INTRUSION}

27 struct Temperature {
28   int value;
29   enum {CELSIUS, FAHRENHEIT} unit;
30 }
31 struct Smoke {
32   boolean isDetected;
33 }
```

Figure 2. Extract of the ENSEIRB taxonomy in the DiaSpec syntax

2.3 Declaring an application architecture

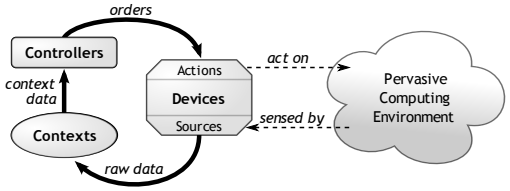

Figure 3. DiaSpec architectural style

DiaSpec offers constructs to declare the architecture of an application following an architectural pattern commonly used in the pervasive computing domain [Dey et al. 2001]. As depicted in Figure 3, a pervasive computing system senses the environment via devices, introduced in a taxonomy definition. The resulting data are refined by context components to match the application needs. Context data are then passed to controller components to make decisions by triggering device actions, declared in the taxonomy. A data flow view of the fire management example is presented in Figure 4. The bottom part shows the data sources corresponding to smoke detectors and temperature sensors. These data sources are aggregated by the `FireState` context to determine whether there is a fire. This information is passed to the `Fire` controller, which triggers device actions to put out the fire and perform other emergency tasks (top part of the figure).

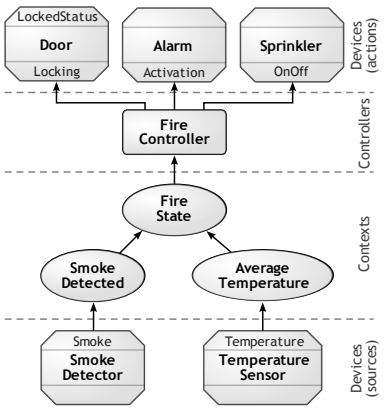

Figure 4. A data flow view of the fire management application

The DiaSpec declarations of the fire management application are presented in Figure 5. The declaration of a context component (the `context` keyword) consists of a name, a list of context/source names as input, and a type as output. For example, the `SmokeDetected` context takes a `Smoke` source as input, produced by the `SmokeDetector` device class (line 1). The restriction to this device class is expressed by the `from` keyword. `SmokeDetected` produces a value of type `Smoke`. In addition to functionalities, context components are declared as producing data relative to some situation, expressed as an index. For example, the `SmokeDetected` component produces a boolean value that is relative to the `location` index, indicating where the smoke is detected. When multiple indices are associated with a given context value, it is relative to all the declared indices. When a context component does not declare an index, its output context data are not relative to any situation.

For another example, consider the `FireState` context component that produces a value of type boolean indicating whether a

fire is occurring. This value is computed from two input contexts, namely `SmokeDetected` and `AverageTemperature`, and is relative to the `location` index.

Note that the declaration of the source of a context does not make explicit how the values are supplied, *i.e.,* push or pull. Indeed, the generated framework provides the programmer with both modes.

The declaration of a controller component (the `controller` keyword) consists of a name, input context names and action clauses. For example, the `FireController` controller component takes the `FireState` context as input and trigger actions defined by device interfaces, namely `Locking`, `OnOff`, and `Activation` (lines 11 to 16). Interfaces may be narrowed down to a device class with the `on` keyword, just like a source of context may be restricted to a device class with the `from` keyword.

Once the application architecture is declared, the programmer implements the components. This development is supported by a dedicated framework discussed in the following section.

```
1  context SmokeDetected[Location location]:  Smoke {
2    source Smoke from SmokeDetector;
3  }
4  context AverageTemperature[Location location]:  Temperature {
5    source Temperature from TemperatureSensor;
6  }
7  context FireState[Location  location]:  boolean {
8    context SmokeDetected, AverageTemperature;
9  }

11 controller  FireController  {
12   context FireState;
13   action Locking on Door;
14   action OnOff on Sprinkler;
15   action Activation  on Alarm;
16 }
```

Figure 5. Extract of the ENSEIRB fire management application

3. Dedicated programming framework

DiaGen generates a programming framework with respect to a set of declarations for device classes, context components and controller components. DiaGen produces an *abstract class* for each entity declaration (device, context and controller), providing methods to support the development (discovery and interactions). It also generates abstract method declarations to allow the developer to program the application logic (*e.g.,* triggering device actions). Implementing a DiaSpec-declared entity is done by subclassing the corresponding generated abstract class. In doing so, the developer is required to implement each abstract method. To facilitate this process, most Java IDEs are capable of generating class templates based on super abstract classes. The developer writes the application logic in subclasses and not in the generated abstract classes.

Generating abstract classes instead of incomplete source code better separates programming support and developer code. As a result, generating a new programming framework, after changes in a DiaSpec description, does not override the code of the developer. However, these changes might modify the generated support, requiring the developer to revise his already-written code.

Let us now describe the dedicated support generated for devices, contexts and controllers.

3.1 Device implementation

The compilation of a device declaration produces a dedicated skeleton in the form of an abstract class depicted in Figure 6. Let us examine what is generated for each part of a device declaration: actions, sources and attributes.

```
1  public abstract  class  Door {
2    protected void setLocation(Location  location)  {...}
3    protected void setLockedStatus(LockedStatus  lockedStatus)  {...}
4    public abstract  void lock();
5    public abstract  void unlock();
6    ...
7  }
```

Figure 6. The abstract class `Door` generated by DiaGen from the declaration of the Door device (Figure 2, lines 15 to 18)

Actions An action corresponds to a set of operations supported by a device. It takes the form of an interface included by the abstract class generated for a device declaration. Each operation is to be implemented by the device developer. This implementation is aimed to bridge the gap between the declared interface and an actual device instance.

The following code fragment wraps a `Door` device that is controlled using X10, a protocol commonly used in home automation [X10].

```
1  public class  Door_X10 extends Door {

3    ... // defines  x10Ctrl  and x10Addr

5    public Door_X10(Location location)  {
6      setLocation(location);
7    }

9    @Override
10   public void lock()  {
11     x10Ctrl.addCommand(new Command(x10Addr, Command.OFF));
12     setLockedStatus(LOCKED);
13   }

15   @Override
16   public void unlock()  {
17     x10Ctrl.addCommand(new Command(x10Addr, Command.ON));
18     setLockedStatus(UNLOCKED);
19   }
20 }
```

Figure 7. A developer-supplied implementation of a Door device using the X10 protocol in Java

According to the declaration of the Door device (Figure 2, lines 15 to 18), the generated `Door` class (Figure 6) declares abstract methods `lock` and `unlock` (lines 4 and 5). The `Door_X10` implementation of these two methods (Figure 7, lines 9 to 19) relies on the X10 library.

Sources A device declares sources that make its state and sensed data available to context components. The generated support provides the developers with dedicated methods to update source data. In the above example, the generated `setLockedStatus` method (Figure 6, line 3) sets the door state and is called by the developer (Figure 7, lines 12 and 18). This method is generated thanks to the LockedStatus declaration (Figure 2, line 17).

Attributes Devices are characterized by attributes. These attributes can be assigned values at runtime. Attributes are managed by generated getters and setters. For example, the `Door` device inherits the `location` attribute from the `Device` node (Figure 2, line 1), which triggers the generation of a `setLocation` method (Figure 6, line 2). In Figure 7, the `setLocation` method is used (line 6) to set the door location to a value passed as a constructor parameter.

3.2 Developing application logic

For each context and controller declaration, the DiaSpec compiler generates a skeleton, implemented as an abstract class. The devel-

opment of the application logic thus consists of implementing the generated abstract classes.

3.2.1 Implementation of context components

From a context declaration, DiaGen generates support to develop the context processing logic. This support allows distributed devices to be selected through service discovery. Proxies are generated to interact with selected devices.

The code fragment in Figure 8 presents the implementation of the `SmokeDetected` context declaration. This is done by extending the corresponding generated abstract class named `SmokeDetected` (not shown here, but similar to the one in Figure 6) with a new class named `MySmokeDetector`. Because this context is declared as taking a `Smoke` input source from smoke detectors (Figure 5, line 2), the generated framework provides support to select and interact with instances of this device class. For example, the `allSmokeDetectors` method is generated in the abstract class `SmokeDetector` to discover all available smoke detectors and is used in Figure 8 line 4. The `subscribeSmoke` method is invoked to subscribe to the `Smoke` input source.

A context may access data from devices using two interaction modes: pulling (line 23) or pushing (line 4 for subscription and line 8 for notification). The implementation of a context component must provide values to its consumers, whether other contexts or controllers. To do so, methods are generated as illustrated by lines 11 and 14 where the context component code pushes a new boolean value to indicate the presence of smoke at a specific location. These interaction modes are further described in Section 3.4.

```
1  public class MySmokeDetected extends SmokeDetected {

3    public MySmokeDetected() {
4      allSmokeDetectors().subscribeSmoke(this);
5    }

7    @Override
8    public void smokeChanged(SmokeDetector smokeDetector, Smoke
         smokeDetected) {
9      Location location = smokeDetector.getLocation();
10     if (smokeDetected.isDetected)
11       setSmokeDetected(location, new Smoke(true));
12     else
13       // check whether there is still smoke at this location
14       setSmokeDetected(location, isSmokeDetected(location));
15   }

17   // Tests whether smoke is detected for a particular location
18   private Smoke isSmokeDetected(Location location) {
19     SmokeDetectorComposite detectors =
20                 discover(smokeDetectorWhere().location(location));
21     for (SmokeDetector sd : detectors) {
22       try {
23         if (sd.getSmoke().isDetected)
24           return new Smoke(true);
25       } catch (DiaGenCommunicationException e) {...}
26     }
27     return new Smoke(false);
28   }
29 }
```

Figure 8. A developer-supplied implementation of the SmokeDetected context in Java

3.2.2 Implementation of controller components

DiaGen generates support to create a controller component from its DiaSpec declaration. Controllers use the context information to take decisions that are carried out by calling actions on devices. To do so, support is generated to allow a controller to access context information in both push and pull modes. This is illustrated by the `FireController` generated abstract class (not shown here, but similar to the one in Figure 6). In this abstract class, and because the `FireController` depends on the `FireState` context (Figure 5, line 12), methods have been generated to provide both modes of interaction to access `FireState` context information. In Figure 9, the implementation of the `FireController` controller component is done by extending the corresponding generated abstract class named `FireController` with a new class named `MyFireController`. The implementation starts by invoking the `subscribeFireState` method, generated in the abstract superclass `FireController` to be notified when the `FireState` information changes (Figure 9, line 4). Correspondingly, the method `fireStateChanged`, declared abstract in the generated abstract class `FireController`, is overridden to receive these notifications (Figure 9, lines 7 to 28). Arguments passed to the notification callbacks are always the new context value, followed by the indices characterizing this value.

The controller declaration makes explicit what devices are controlled through what operations. This information is used to generate an abstract class that supports mechanisms to discover target devices. Device discovery is illustrated in line 13 of Figure 9 where methods `discover` and `doorsWhere` are called to select all the doors of the building. The implementation of these methods is generated in the abstract superclass `FireController`.

Methods `unlock` (line 14), `on` (line 18) and `activate` (line 25) are generated in the `Door`, `Sprinkler` and `Alarm` proxies, respectively. A proxy is a generated class representing a local view of a remote device. In doing so, a device invocation abstracts over distribution details.

After having presented the overall programming support given by a generated framework, let us focus on a key mechanism to cope with dynamicity, namely, device discovery.

```
1  public class MyFireController extends FireController {

3    public MyFireController() {
4      subscribeFireState();
5    }

7    @Override
8    public void fireStateChanged(boolean fireState, Location location)
9    {
10     Location building = location.getBuilding();
11     if (fireState) {
12       // Unlock all doors of the building in fire
13       DoorComposite d = discover(doorsWhere().location(building));
14       d.unlock();
15
16       // Switch on the sprinklers where the fire is located
17       SprinklerComposite s = discover(sprinklersWhere().location(
             location));
18       s.on();
19
20       // Activate the alarms
21       AlarmComposite alarms = discover(alarmsWhere().location(
             building));

23       // Equivalent to alarms.activate(AlarmType.FIRE):
24       for (Alarm a: alarms)
25         try { a.activate(AlarmType.FIRE); }
26         catch (DiaGenCommunicationException e) {...}
27     } else {...}
28   }
29 }
```

Figure 9. A developer-supplied implementation of the FireController in Java

3.3 Device discovery

Our dedicated programming frameworks provide support to discover devices based on a taxonomy definition. Device discovery returns a collection of proxies for the selected devices. Importantly, this collection is encapsulated in a composite object, an instance of the composite design pattern [Gamma et al. 1995]. An example of such collection, `DoorComposite`, is returned in line 13 of Figure 9. In this design pattern, the developer can process all elements of the collection either explicitly by using a loop or implicitly by invoking a method of the composite that implements an iteration. Line 14 in Figure 9 is an example of an implicit iteration. Lines 24 through 26 illustrate an explicit iteration.

Composites provide a way of factorizing exception handling. The generative approach has the potential of further leveraging this situation by parameterizing the generative process with respect to policies for exception handling and fault tolerance. We are currently investigating this direction.

To help developers express queries to discover devices, DiaGen generates a Java-embedded, type-safe DSL, inspired by the work of Kabanov et al. [Kabanov and Raudjärv 2008]. This technique makes it possible to keep using a standard Java compiler, instead of augmenting the Java grammar, as is done in Silver [Van Wyk et al. 2007] and ArchJava [Aldrich et al. 2002].

An example of the use of our query language is given in the `MyFireController` class shown in Figure 9. The `fireStateChange` method selects doors to operate. The call to `doorsWhere` (line 13) restricts the selection to doors in the building where the fire has been detected. On line 17, the `discover` method call returns sprinklers specifically located where the fire is occurring.

More complex queries can also be expressed, as illustrated below.

```
1 Location room1, room2;
2 ...
3 discover(doorsWhere().location(or(eq(room1),eq(room2))));
```

This query selects doors that are either located in room 1 or 2.

A method suffixed by `Where` is available for each device that can be discovered. These methods return a dedicated filter object on which it is possible to add specific filters over attributes associated wth the device class. For example, the `FireController` abstract class defines a `doorsWhere` method that returns a `DoorFilter`. This filter can be further parametrized by adding an attribute filter for the location attribute defined by the `Door` in the taxonomy. This is done by calling the `location()` method defined in the generated `DoorFilter` class. The parameter to this method is either a `Location` value or a logical expression. If a `Location` value is passed then the discovered devices must have this location in order to be selected. If a logical expression is chosen, the attributes of the selected devices hold with respect to the logical expression. A logical expression is made of relational and logical operators. New methods can be defined to further enhance the expressiveness of the query language.

Our approach allows developers to specify filters for more than one attribute, as is shown in the following example.

```
1 Accuracy minAccuracy;
2 Location room1;
3 ...
4 discover(temperatureSensorsWhere().location(room1).accuracy(gt(
      minAccuracy)));
```

This query will select all temperature sensors that are both in room 1 and provide a specified minimum accuracy. Our current implementation does not allow logical expressions across attributes. For example, it is not possible for a query to specify that a device must have a particular value for an attribute *or* another value for another attribute. We are working on this limitation.

This embedded DSL is both expressive and concise. It plays a key role in enabling the developer to handle the dynamicity of a pervasive computing environment without making the code cumbersome.

3.4 Interaction modes

As mentioned earlier, an application interacts with a device either to carry out an action or access context data. A generated programming framework supports the former case with the command interaction mode. The latter case is supported by both a push and pull mode.

Command. A command is a one-to-one asynchronous interaction mode, similar to a remote procedure call. The developer can pass arguments to a command according to signatures included in the DiaSpec taxonomy. Because a command is limited to operate a device, it does not return a value. However, errors can be expressed as exceptions. An example of command invocation is given in line 14 of Figure 9.

Pull. A context can fetch data from devices and other contexts. As well, a controller can fetch data from contexts. To achieve these interactions, the pull mode provides a one-to-one synchronous interaction mode with a return value. Accessing data from a device then consists of invoking the appropriate methods of the device proxy returned by the device discovery mechanism. This is exemplified by line 23 in Figure 8. Accessing data from a context is achieved by calling a method with the required context indices.

Push. This mode corresponds to the asynchronous publish/subscribe paradigm. When a device or a context needs to push some data (*e.g.,* whenever it changes), it calls a `set` method implemented in its abstract class. This is illustrated by the `SmokeDetected` context that publishes the `Smoke` event. To do so, its abstract class implements a `setSmokeDetected` method that takes a `Smoke` value as argument, together with an index value characterizing the event. This index value indicates the location where the smoke is detected. An event value is received by all entities that have subscribed to the event type. A subscription method is generated in an abstract class for each input source or context declared in a component.

The management of subscribers and the propagation of events are supported by the generated programming frameworks, easing the development process.

4. Targeting distributed technologies

A generated programming framework abstracts away from the underlying distributed systems technologies. Besides raising the level of abstraction, this strategy makes the application code portable across distributed systems technologies without any change. DiaGen currently offers four back-ends targeting Web Services, CORBA, SIP and RMI. Each of these technologies provides specific features and mechanisms with various benefits for the development of pervasive computing systems. For example, RMI is well-suited for testing because it requires a light infrastructure. SIP is needed when a pervasive computing system relies on a telephony infrastructure.

Each of these technologies is widely deployed, providing a profusion of existing components. For example, Web Services technology offers services in a wide range of areas (*e.g.,* weather forecasting and electronic commerce). In our approach, these existing components can be re-used as building blocks to develop DiaSpec applications. They are introduced as devices in a taxonomy. From a device declaration, DiaGen generates the configuration and code necessary to integrate device instances in DiaSpec applications.

The rest of this section describes how DiaGen compiles DiaSpec concepts into Web Services and CORBA. This compilation relies on generative programming tools existing in both distributed systems technologies. These tools allow a generative programming approach to be used throughout the compilation process. In doing so, boilerplate tasks are avoided, making the development of DiaSpec applications less error-prone.

4.1 Web services

The generation of a dedicated Web Services back-end is performed in three stages as illustrated by Figure 10.

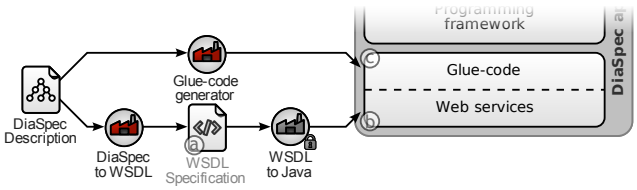

Figure 10. Distribution generation

In stage ⓐ, given a DiaSpec taxonomy, DiaGen generates interface descriptions written in the Web Service Description Language (WSDL). A translation is needed to adapt the high-level communication abstractions of our approach into Web Services concepts. For example, DiaSpec actions become *operations*. Figure 11 shows the WSDL translation of the `Alarm` device defined in Figure 2.

```
<message name="void" />
<message name="ActivateInput">
  <part name="alarmType" type="xsd:AlarmType"/>
</message>
<portType name="Activation">
  <operation name="activate">
    <input message="ns0:ActivateInput" />
    <output message="ns0:void" />
  </operation>
  <operation name="deactivate">
    <input message="ns0:void" />
    <output message="ns0:void" />
  </operation>
</portType>
```

Figure 11. Extract of the `Alarm` WSDL definition

In stage ⓑ, from these WSDL definitions, stubs and skeletons are generated using the Axis libraries[1]. Finally, in stage ⓒ, DiaGen creates the glue-code to plug the generated Web Services stubs and skeletons with DiaSpec dedicated programming framework.

4.2 CORBA

The back-end generation process for Web Services and CORBA are quite similar. The DiaSpec taxonomy is used to generate descriptions written in the Interface Definition Language (IDL). DiaSpec concepts are translated to the IDL constructs. Devices, contexts and controllers become interfaces. Actions are mapped into methods. DiaGen generates the glue-code to fill the gap between CORBA stubs and skeletons and the dedicated programming framework.

Note that DiaGen-generated WSDL and IDL descriptions can be used independently of our approach to develop Web Services and CORBA components. The compliance of these native components is checked by their compiler, enabling their integration in a DiaGen-generated software system.

[1] `http://ws.apache.org/axis/java/index.html`

5. Generation of the simulation part

Existing simulation tools for pervasive computing applications are usually either general-purpose, or ad hoc. In both cases, application testers have to write a lot of code to model the pervasive computing environment and to dynamically visualize its state. Introducing simulated entities requires the development of tedious and boilerplate code that becomes a major part of the final software system.

In this section, we show how the DiaSpec description is used to generate a simulation support.

5.1 Simulated environment

A DiaSpec taxonomy gives information about the physical environment. Device declarations define interactions with the environment. These interaction points are used to produce a simulation support.

In a concrete environment, performing actions on a device can impact the environment data by changing values. For example, a call to methods `activate` and `deactivate` of `Alarm` devices modifies the noise level around their locations. The application tester chooses and parameterizes the suitable behavior to translate an action into the simulated environment state.

A simulated environment is populated with devices. From a DiaSpec taxonomy, the tester selects devices and places them on a 2D representation of the physical space.

5.2 Simulated entities

The test of an application through simulation involves simulated devices. These simulated devices have to link their sources with the simulated environment. Also, their actions must fuel the simulation engine with updated environment data. Simulated devices are built above the same programming framework as the one used to implement contexts and controllers. In doing so, contexts and controllers are able to interact with simulated devices transparently. From a device declaration, DiaSim editor provides the tester with several predefined behaviors (periodical, threshold, *etc*) for each source. DiaGen generates an implementation of a device bridging the gap between the programming framework and the simulated environment.

5.3 Hybrid simulation

Given a device declaration, both real and simulated implementations share the same generated programming framework. Thanks to the framework abstractions, contexts and controllers interact uniformly with devices regardless of their nature. The application tester can include real devices into a simulated environment. The incremental integration of real devices facilitates the development and testing process.

6. Evaluation

In this section, we present experimentations we conducted using our approach to illustrate its practical benefits. First, we introduce a pervasive computing project developed by our research group to automate an engineering school building. Second, we present an assessment of a practical course we gave to introduce pervasive computing to engineering students using DiaSpec. Finally, we briefly outline the implementation of our tool chain.

6.1 Engineering school building

The DiaSpec description presented in Figures 2 and 5 is an extract from a larger project[2], whose goal is to manage a 13,500-square meters building, hosting an engineering school and research groups. The project features many applications including fire man-

[2] `https://diasim.bordeaux.inria.fr/Examples/Enseirb/`

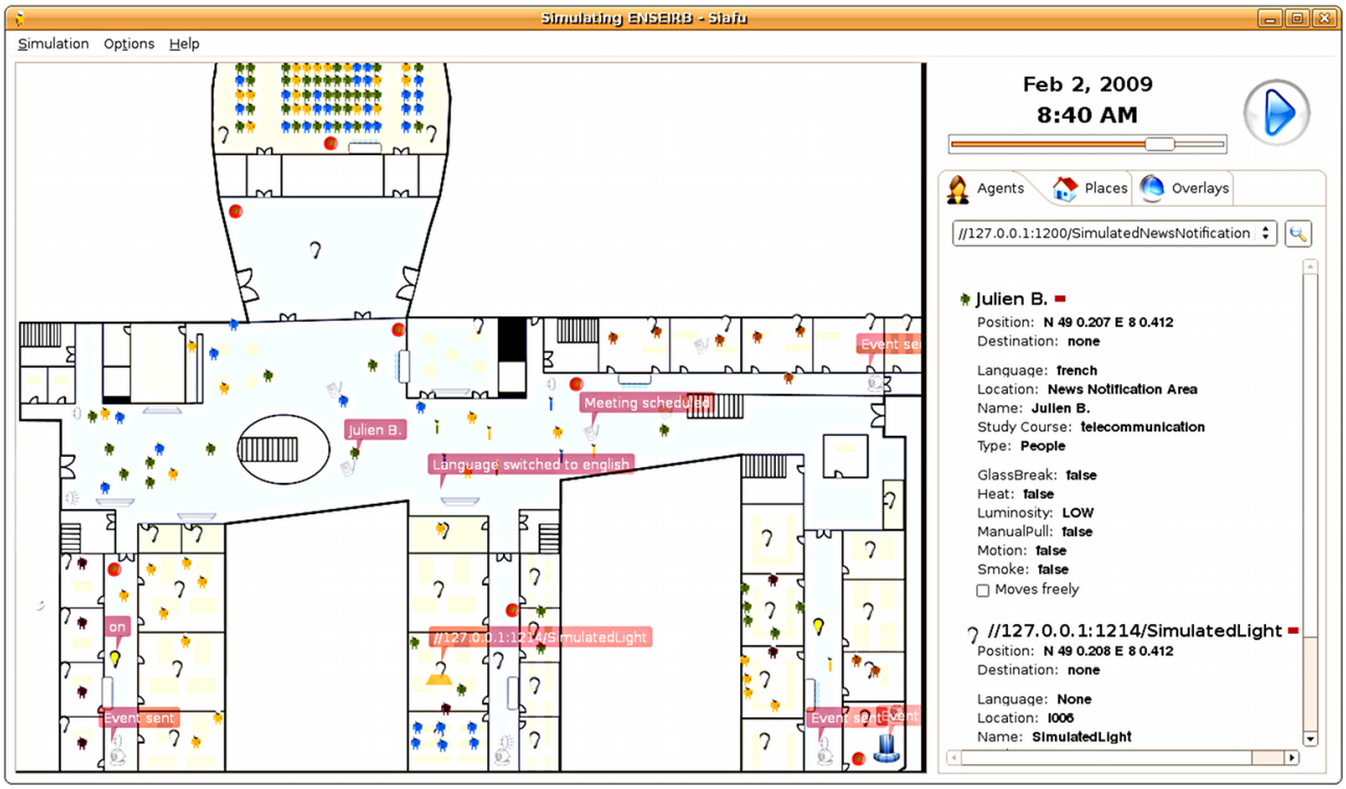

Figure 12. DiaSim simulation renderer for the ENSEIRB project on the automation of the engineering school. The left frame displays a 2D view of the first floor of the ENSEIRB engineering school building, the simulated devices deployed (*e.g.,* light and temperature sensors, display screens) and simulated occupants. This view also displays pop-up notifications for information sensed from the simulated environment. The right frame displays detailed information on both an occupant (*e.g.,* name, position) selected in the left frame and its surroundings (*e.g.,* light, temperature). The simulation renderer is interactive; it allows users to alter time, temperature and luminosity, to move occupants around...

agement, light and air conditioning management, and access control.

This project involves 21 device classes, 13 contexts, 7 controllers, 47 type declarations and 17 action definitions. The DiaSpec taxonomy represents 200 LOC, the architecture 130 LOC, the generated framework 7,000 LOC and manually written Java code 3,000 LOC.

The engineering school building is simulated using DiaSim[3]. Figure 12 displays part of the simulation rendering of the project. Table 1 lists the number of simulated devices involved in the fire management scenario that have been deployed. At the moment, more than 250 device instances and 300 occupants are simulated (*e.g.,* staff, researchers, students, visitors) with various behavioral patterns. We are planning on deploying part of the project in-situ to study the benefits of the DiaSpec approach for the incremental deployment of pervasive computing applications.

6.2 Teaching

We used the DiaSpec language and related tools in a course on pervasive computing system programming. The class consisted of twenty undergraduate computer science students with no prior experience. Their assignment was to develop a meeting manager application whose role was to notify participants of approaching meetings. The notification medium and the delays before starting

Devices	deployed
Smoke detectors	40
Temperature sensors	40
Sprinklers	20
Doors	16
Alarms	20

Table 1. Numbers of devices simulated in DiaSim involved in the fire situation management scenario of the ENSEIRB demonstration.

the meeting depended on the participant locations and surroundings (*i.e.,* email, voice mail, IM, or SMS).

All student groups designed a working DiaSpec architecture for the meeting manager and most completed the assignment with an implementation. This experience demonstrated that students become fluent in using DiaSpec in a short time.

6.3 Implementation of DiaSpec and DiaSim

DiaSpec[4] is a project actively developed and maintained by the INRIA Phoenix research group[5]. Its compiler, DiaGen, consists of a tool chain including a parser, a verifier and code generators. The implementation of DiaGen is based on the JastAdd compiler com-

[3] The simulation of the engineering school building has been presented at the demonstration track of PERCOM'09.

[4] `https://diaspec.bordeaux.inria.fr/`

[5] `http://phoenix.labri.fr/`

piler [Ekman and Hedin 2007]. The modifications to the JastAdd parser and compiler to handle DiaSpec amount for 2,800 LOC, and the code generators have 2,300 LOC. DiaGen was tested by sample applications and approximately 100 unit-tests. The distribution library has 10,000 LOC.

DiaSim[6], our pervasive computing simulator, uses Siafu [Martin and Nurmi 2006], an open source context simulator for the 2D rendering of DiaSpec-simulated pervasive computing applications. DiaSim amounts for 15,000 LOC (without counting Siafu). In addition to the simulation renderer (Figure 12), DiaSim comprises a visual editor (Figure 13) to set-up and configure a pervasive computing environment and pre-defined behavior patterns for simulated devices and entities (persons, *etc*.).

Figure 13. DiaSim simulated environment editor for an health care scenario. The middle frame displays a 2D view of a home environment. The devices extracted from a DiaSpec taxonomy are displayed on the left frame and can be placed via drag-and-drop operations in the simulated environment. The right frame summarizes devices deployed in the environment.

7. Related Work

In this section, we review existing approaches for the development of pervasive computing applications. As well, we briefly describe how our generative strategy leverages existing approaches.

Middleware. Middleware is a widely used approach to developing distributed applications. Standardized middlewares (*e.g.,* CORBA [OMG 1995], DCOM [Sessions 1998], and Web Services [Consortium 2004]) support the development of distributed applications via dedicated mechanisms to perform operations such as event subscription and device discovery. Existing middlewares often provide a variety of interface definition languages (IDLs), mainly to define the type signature of individual entities. These definitions are used to generate communication support, as illustrated by the RPC paradigm [Sun Microsystem 1988]. Our approach leverages this basic support by providing a high-level programming framework encompassing the architectural aspects of the pervasive computing system.

Olympus is a programming framework on top of the Gaia middleware, which is dedicated pervasive computing[Román et al. 2002, Ranganathan et al. 2005]. It provides high-level programming interfaces to implement active spaces. However, the framework remains general purpose and thus provides little guidance to the developer and still requires boilerplate code to be written.

Component-oriented programming languages. ComponentJ [Seco and Caires 2000], ACOEL [Sreedhar 2002] and ArchJava [Aldrich et al. 2002] build a bridge between ADLs and programming languages by adding syntactic constructs to a mainstream programming language such as Java to model architectures. Because these approaches mix architecture declarations with application code, architecture changes can be difficult to make. In contrast, a DiaSpec specification is processed prior to component programming, allowing dedicated programming support to be generated. Of these approaches, ArchJava addresses the distributed setting by allowing users to introduce new connectors [Aldrich et al. 2003]. However, this approach imposes additional burden on the developer because it requires implementing the new connectors. In contrast, DiaGen provides complete support for built-in connectors, including mappings to various distributed systems technologies.

Architecture Description Languages. Architecture description languages model distributed and non-distributed systems to ensure different properties at compile time and at run time. Some of them also provide a generic framework on top of which developers can implement components and connectors [Garlan et al. 2000, Allen 1997, Moriconi and Riemenschneider 1997, Binns et al. 1996, Magee and Kramer 1996, Luckham et al. 1995, Shaw et al. 1995]. DiaSpec is an ADL-inspired DSL for modeling pervasive computing systems; it relies on a tool chain for the generation of a dedicated programming framework.

Domain-specific languages. In previous work, we have developed the domain-specific coordination language Pantaxou, which includes a language layer for describing the entities relevant to an application area, and another for implementing services coordinating these entities [Mercadal et al. 2008]. The former language layer is inspired by DiaSpec but not as dedicated to the pervasive computing domain. Furthermore, our approach brings the idea of declaring the interactions allowed between distributed components from a domain-specific language to a mainstream programming language, Java.

8. Conclusion and Future Work

In this paper, we have presented DiaSpec, a DSL for the development of pervasive computing systems. We have shown that the DiaSpec language facilitates the description of pervasive computing applications. We have also demonstrated that our generative programming approach covers the entire development cycle of a pervasive computing system.

The DiaSpec language eases the description of pervasive computing environments by enforcing an architectural pattern commonly used in this domain. DiaSpec provides abstractions to model (1) the process of gathering data from devices, (2) their transformation into context information relevant to making decisions, and (3) the triggering of actions to carry out decisions.

We have shown that the DiaSpec generative approach leverages existing technologies to cover the development cycle of pervasive computing systems. At design time, DiaSpec proposes high-level constructs to describe a pervasive computing system. A DiaGen-generated framework abstracts over the specific features of distributed systems technologies and proposes programming support dedicated to a taxonomy definition and architecture declarations. Moreover, DiaGen makes it possible for applications to easily integrate existing components from various technologies (*e.g.,* Web Services, CORBA, SIP). Finally, our approach allows incremental testing and deployment by generating configuration code for the DiaSim simulation environment.

This work is being actively expanded in various directions; we briefly mention two of them. Because applications are written in Java, the developer can bypass the restrictions imposed by the generated programming framework. To complement the restrictions imposed by the Java language, we plan to annotate a generated

[6] `https://diasim.bordeaux.inria.fr/`

framework to enable a model checker to ensure that the implementation is compliant to the DiaSpec specification. In the longer term, we want to widen the scope of DiaSpec by introducing non-functional architectural information (*e.g.,* security and fault tolerance).

References

Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581365.

Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In Luca Cardelli, editor, *Proceedings ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102, Darmstadt, Germany, 2003. Springer. doi: 10.1007/b11832. URL `http://dblp.uni-trier.de/rec/bibtex/conf/ecoop/AldrichSCN03`.

Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997. URL `http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/rallen_thesis.htm`.

Pam Binns, Matt Engelhart, Mike Jackson, and Steve Vestal. Domain-specific software architectures for guidance, navigation, and control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996. doi: 10.1142/S0218194096000107.

Julien Bruneau, Wilfried Jouve, and Charles Consel. Diasim, a parameterized simulator for pervasive computing applications. In *Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous'09)*, Toronto, CAN, jul 2009. ICST/IEEE.

World Wide Web Consortium. Web services architecture, 2004. http://www.w3.org/TR/ws-arch/.

Anind K. Dey, Gregory D. Abowd, and Daniel 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.

Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998. ISBN 0764580434.

Torbjörn Ekman and Görel Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3): 14–26, 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.02.003.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995. ISBN 0-201-63361-2-(3).

David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.

Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and Practice of Programming in Java*, pages 189–197, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-223-8. doi: 10.1145/1411732.1411758.

David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. URL `ftp://pavg.stanford.edu/pub/Rapide-1.0/tse94.ps.Z`.

Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, San Francisco, CA, USA, 1996. ACM. ISBN 0-89791-797-9. doi: 10.1145/239098.239104.

Miquel Martin and Petteri Nurmi. A generic large scale simulator for ubiquitous computing. In *Third Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, 2006 (MobiQuitous 2006)*, San Jose, CA, USA, July 2006. IEEE Computer Society. doi: 10.1109/MOBIQ.2006.340388. Poster.

Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. ISSN 0098-5589. doi: 10.1109/32.825767.

Julien Mercadal, Nicolas Palix, Charles Consel, and Julia Lawall. Pantaxou: a domain-specific language for developing safe coordination services. In *Proceedings of the Seventh International Conference on Generative Programming and Component Engineering (GPCE)*, pages 149–160, Nashville, TN, USA, October 2008. Acm Press. doi: 10.1145/1449913.1449936.

Mark Moriconi and Robert A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Sri-csl-97-01, SRI International, mar 1997.

OMG. The common object request broker: Architecture and specification. Technical Report 2.0, Object Management Group, 1995.

Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2299-8. doi: 10.1109/PERCOM.2005.26.

Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002. ISSN 1536-1268. doi: 10.1109/MPRV.2002.1158281.

Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler. SIP: Session Initiation Protocol. Technical report, RFC 3261, August 2002. http://www.ietf.org/rfc/rfc3261.txt.

João Costa Seco and Luís Caires. A basic model of typed components. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, United Kingdom, 2000. Springer-Verlag. ISBN 3-540-67660-0. doi: 10.1007/3-540-45102-1_6.

Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998. ISBN 0-471-19381-X.

Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995. ISSN 0098-5589. doi: 10.1109/32.385970.

Vugranam C. Sreedhar. Mixin'up components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 198–207, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: 10.1145/581339.581366.

Sun Microsystem. RPC: Remote procedure call protocol specification, version 2. Technical report, Sun Microsystem, 1988.

Eric. Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute Grammar-Based Language Extensions for Java. *Lecture Notes in Computer Science*, 4609:575, 2007.

X10. Standard and extended x10 code protocol, 1993. URL `http://software.x10.com/pub/manuals/xtdcode.pdf`.