DiaSim: A Parameterized Simulator for Pervasive Computing Applications

Julien Bruneau*, Wilfried Jouve, Charles Consel
INRIA Bordeaux, *France*
{julien.bruneau, wilfried.jouve, charles.consel}@inria.fr

***Abstract*—Pervasive computing applications involve both software concerns, like any software system, and integration concerns, for the constituent networked devices of the pervasive computing environment. This situation is problematic for testing because it requires acquiring, testing and interfacing a variety of software and hardware entities. This process can rapidly become costly and time-consuming when the target environment involves many entities.**

This paper introduces DiaSim, a simulator for pervasive computing applications. To cope with widely heterogeneous entities, DiaSim is parameterized with respect to a description of a target pervasive computing environment. This description is used to generate both a programming framework to develop the simulation logic and an emulation layer to execute applications. Furthermore, a simulation renderer is coupled to DiaSim to allow a simulated pervasive system to be visually monitored and debugged.

DiaSim has been implemented and used to simulate various pervasive computing systems in different application areas, demonstrating the generality of our parameterized approach.

I. Introduction

Numerous pervasive computing applications coordinate a variety of networked entities collecting context data from sensors and reacting by triggering actuators. To collect context data, sensors process stimuli that are observable changes of the environment (*e.g.,* fire and motion). Trigerring actuators is assumed to change the state of the environment. Developing a pervasive computing application requires to address a number of issues such as entity heterogeneity, physical constraints, and types of stimuli present in the target environment. Also, such an application needs to implement strategies to manage a variety of scenarios *e.g.,* fire situations, intrusions, and crowd emergency-escape plans. Consequently, in addition to the challenges of developing any software system, a pervasive computing system needs to validate the environment entities both individually and globally, to identify potential conflicts. For example, a fire manager and an entrance manager could issue contradicting commands to a building's door to respectively enable evacuation and ensure security. In practice, the many parameters to take into account for the development of a pervasive computing application can considerably lengthen this process. Not only does this situation has an impact on the application code, but it also involves changes to the physical layout of the target environment, making each iteration time-consuming and error-prone.

*Current affiliation: Thales Airborne Systems.

Various middlewares and programming frameworks have been proposed to ease the development of pervasive computing applications [1], [2], [3]. However, they require a fully-equipped pervasive computing environment for an application to be run and tested. As a result, an iteration process is still needed, involving the physical setting of the target environment and the application code.

In fact, the development of a pervasive computing system is very similar to the development of an embedded system. Like a pervasive computing system, an embedded system coordinates a number of heterogeneous hardware components that can be viewed as sensors (*e.g.,* microphones and buttons) and actuators (*e.g.,* displays and speakers). Some embedded systems are capable of discovering components dynamically, such as a smartphone detecting bluetooth components. As in the pervasive computing domain, embedded systems developers need to anticipate as wide a range of usage scenarios as possible to program their support. Despite similarities, the embedded systems domain differs from the pervasive computing domain in that it provides approaches and tools to facilitate software development for a system under design. Indeed, embedded systems applications can be tested and debugged using simulators [4]. Hardware components are *simulated* via software components that faithfully duplicate their observable behavior. And, the embedded systems application is *emulated*, executing as if it relied on hardware components, without requiring any code change. The study of embedded systems simulators gives us a practical basis for identifying the requirements for pervasive computing systems. Let us now examine these requirements.

a) Area-specific simulator: Like embedded systems, pervasive computing systems target a variety of application areas, including home automation, building surveillance and assisted living. Each area corresponds to specific pervasive computing environments, consisting of classes of entities dedicated to a given activity (*e.g.,* a light sensor, a motion detector or a wireless heart rate monitor). Correspondingly, the related stimuli drastically vary with respect to the target area. As a consequence, a simulation tool for the pervasive computing domain is required to deal with different application areas, enabling new classes of entities and stimuli to be introduced easily.

b) Transparent simulation: A key feature of most embedded systems simulators is that they emulate the execution of an application without requiring any change in the application

code. As a result, when the testing phase is completed, the application code can be uploaded as is and its logic does not require further debugging. The same functionality should be provided by a simulator for pervasive computing applications.

c) Testing a wide range of scenarios: Some pervasive computing applications address scenarios that cannot be tested because of the nature of stimuli involved (*e.g.,* fire and smoke). In other situations, the scenarios to be tested are large scale in terms of stimuli, entities and physical space they involve. These situations would benefit from a simulation phase to refine the requirements on the constituent entities of the environment, before acquiring them. Regardless of the nature of the target pervasive computing system, its application logic is best tested on a wide range of scenarios, while the system is under design. This strategy allows improvements to be made as early as possible in both its architecture and logic.

d) Simulation renderer: Like an embedded systems simulator, one for pervasive computing systems needs to visualize the simulation of scenarios. This simulation renderer needs to take into account various features of the pervasive computing domain. Specifically, it should support visual representations for an open-ended set of entities and stimuli, visual support for scenario monitoring, and debugging facilities to navigate in scenarios in terms of time and space.

Some existing approaches propose to visualize the simulation of pervasive computing applications [5], [6]. However, these approaches are limited because they require significant programming effort to address new pervasive computing areas. Furthermore, they do not provide a setting to test applications deterministically. The Lancaster simulator addresses this issue but does not support scenario definition [7]. The PiCSE simulator provides a comprehensive simulation model and generic libraries to create new scenarios. However, users have to manually specialize the simulator for every new application area [8].

This paper

This paper presents DiaSim, a simulator for pervasive computing applications based on sensors and actuators. This simulator is parameterized with respect to a high-level description of the target pervasive computing environment. Such a description defines the classes of entities, whether hardware or software, relevant to a target pervasive computing area. Both simulated and actual environments must conform to the same environment description, ensuring a functional correspondence between the two. Furthermore, the environment description is used to generate an emulation layer to run pervasive computing applications and a simulation programming framework for developing the simulation logic. Our approach makes it possible for the same application code to be simulated or executed in the actual environment. The resulting simulated pervasive computing environment enables to test the application logic against the full range of scenarios corresponding to the environment description. This simulation phase allows the pervasive computing system to be refined in terms of application logic and environment entities. DiaSim includes a simulation renderer enabling the developer to visually monitor and debug a pervasive computing system, navigating in terms of time and space in a simulation.

The contributions of this paper are as follows.

- *Parameterized simulator.* We present a simulator that is parameterized with respect to a high-level description of a pervasive computing environment.
- *Transparent simulation.* Our approach makes it possible for the same code to be simulated or executed in the actual environment. We ensure a functional correspondence between a simulated environment and an actual one by requiring both implementations to be in conformance with the pervasive computing environment description.
- *Hybrid environments.* An application can be executed in a hybrid environment, combining simulated and actual services. Hybrid simulation is a key feature to successfully transition to an actual environment: it allows actual services to be added incrementally in the simulation, as the implementation and deployment progress.
- *Generated simulation support.* A pervasive computing environment description is used to generate both an emulation layer, to execute applications, and a simulation programming framework, to develop simulated entities.
- *Simulation renderer.* We present a simulation renderer that enables the developer to visually monitor and debug a pervasive computing system.
- *Validation.* Our approach has been implemented in a tool called DiaSim. The generality of our parameterized approach has been demonstrated by simulating applications in a variety of pervasive computing areas. The practicality of DiaSim has been shown on a large-scale simulation of an engineering school.

Outline

Section II describes our approach to simulating pervasive computing systems. Section III presents our simulation model. Section IV describes how simulated scenarios are defined. Section V examines how applications are developed, emulated and tested. The implementation and validation of our simulation tool are described in Section VI. Finally, we discuss the related work in Section VII and conclude in Section VIII.

II. Our Approach

Because of the heterogeneity of entities in a pervasive computing system, we propose an approach to simulation that is parameterized with respect to a description of a pervasive computing environment. This description is used to generate support to cover the main development stages of a pervasive computing system. Specifically, a description of a pervasive computing environment is used as an input to generate (1) programming support for application development, (2) building-block implementations of a simulated environment, (3) development support for simulation scenarios, and (4) configurations for simulation rendering.

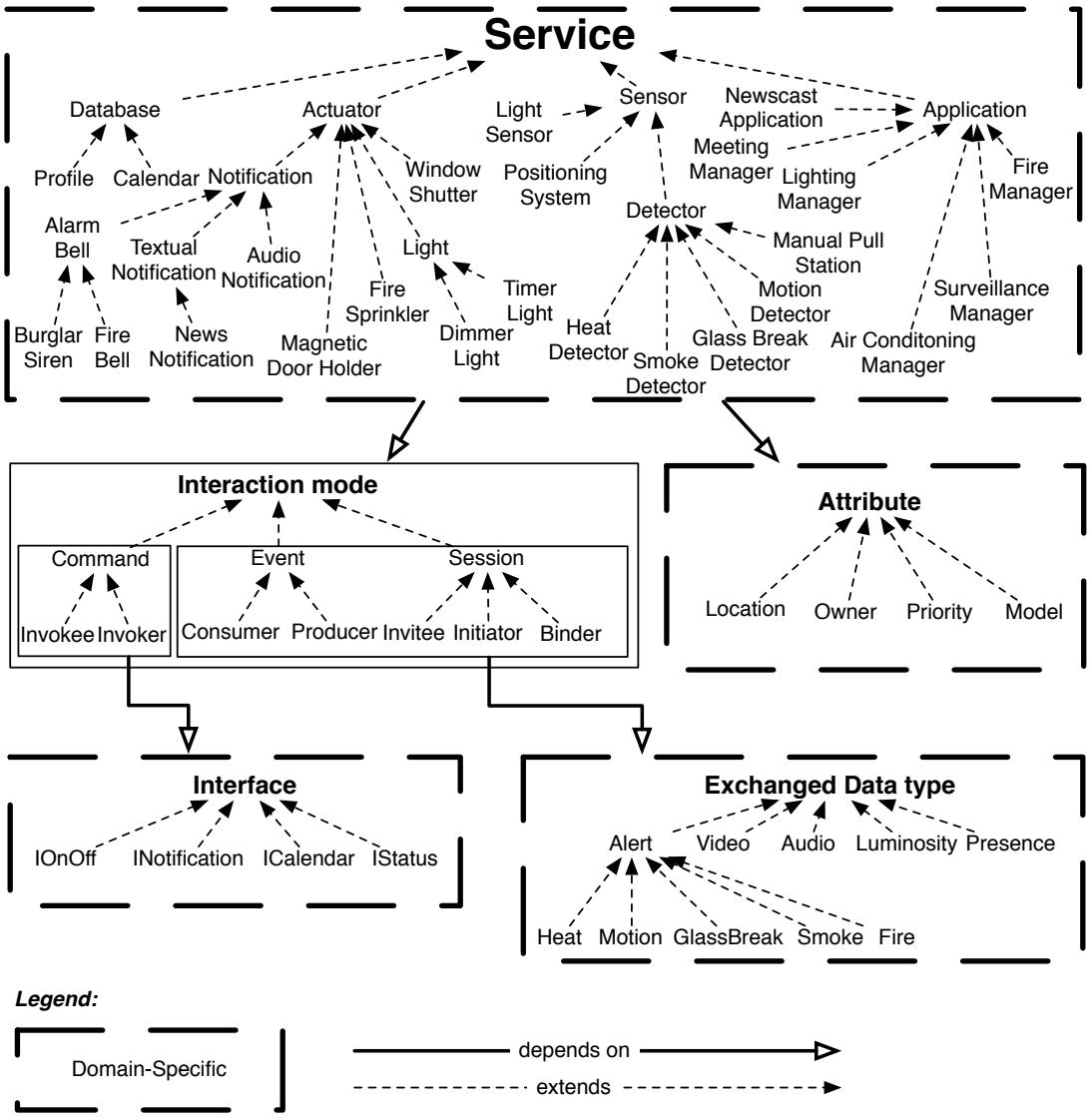

Fig. 1. Excerpt of the building management area definition

A. Describing a pervasive computing environment

A pervasive computing environment for a given area is defined as a collection of *service classes*, each of which represents a set of entities sharing common functionalities. Elements of a service class are referred to as *services*. Associated with each service class is a collection of *attributes*, characterizing the non-functional properties (*e.g.,* the location) of the associated services, and a collection of *functionalities*, specifying how the associated services may interact with other entities. These functionalities rely on one of the three proposed *interaction modes*: *commands* (one-to-one synchronous, RPC-like interactions), *events* (one-to-many asynchronous interactions), and *sessions* (one-to-one interactions involving the exchange of information over a period of time). Each interaction mode is combined with a Java type. For command, the Java type is an interface listing the relevant methods, *e.g.,* the `IOnOff` interface contains methods for activating or deactivating `Light` services. For event or session, the Java type indicates the type of the data that are exchanged. For an event, this Java type describes a type of context data, exchanged between event producers (*e.g.,* sensors) and event consumers. These interaction modes have been carefully designed to be easily mapped into existing middlewares such as CORBA, and Java RMI. Finally, service classes and attribute values are organized hierarchically, permitting more specific instances to be used where less specific instances are required (polymorphism).

To illustrate an area definition, consider the taxonomy of services for building management, shown in Figure 1. Starting at the root node, the hierarchy breaks down the set of possible entities of this area into increasingly specific service classes. This taxonomy consists of service classes supporting commands (*e.g.,* `Burglar Siren`), events (*e.g.,* `Light Sensor`) and sessions (*e.g.,* `Audio Notification`).

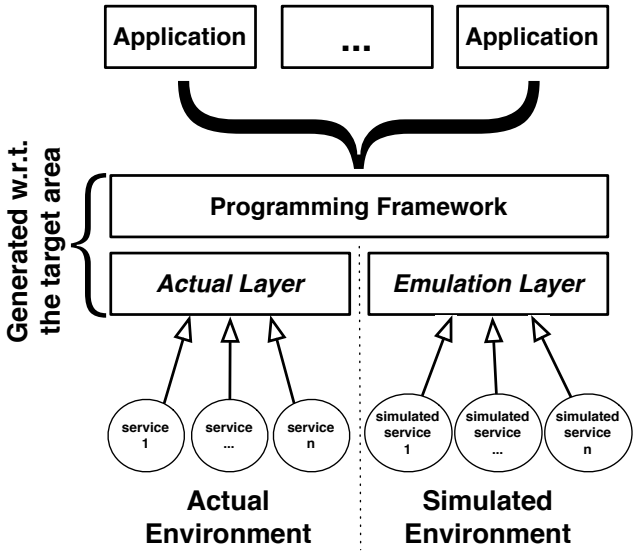

Fig. 2. Our layered architecture

The definition of a pervasive computing area also specifies what kinds of data can be exchanged between services, analogous to an interface definition language. For example, the `Light Sensor` service class produces events of type `Luminosity`.

In practice, a description of a pervasive computing environment is written in a specification language called DiaSpec [9], [3]. A DiaSpec specification is passed to a compiler, DiaGen, to produce a customized programming framework. This programming framework provides developers with high-level abstractions to discover services and to communicate with them. DiaSpec and its compiler have been used for a number of applications, including advanced telephony and home automation. A detailed description of DiaSpec, its compiler, the generated programming support, and specific verifications are presented elsewhere [9], [3].

B. Simulated environment

To abstract over distributed systems technologies, DiaGen follows a layered architecture for the generated programming frameworks. This architecture has made it possible to introduce a simulated environment as just another technology underlying the programming framework. Our layered architecture is shown in Figure 2. Intuitively, our approach consists of replacing the actual execution layer (bottom left of Figure 2) underneath the programming framework by the emulation layer (bottom right of Figure 2). When, the pervasive computing application invokes a *primitive service*, that is, a device or an existing software component, the emulation layer triggers its simulated version. The simulation logic introduced for a primitive service replaces the actual entity.

C. Simulation scenarios

Once the primitive services are simulated, we define simulation scenarios to test the pervasive computing system. A simulation scenario is defined for a given spatial layout of entities. It consists of a set of initial stimuli and a set of evolution rules for the environment stimuli. As a simulation scenario unfolds, the environment context changes, triggering services whose coordinated actions achieve specific tasks. For example, a surveillance manager detects and sends intrusion notifications by coordinating services such as motion detectors, location sensors, glass break detectors, sirens and lights.

D. Simulation renderer

Because of the number of entities involved in a pervasive computing system, a simulation scenario rapidly becomes complicated to follow. To circumvent this problem, we have coupled DiaSim with an existing visualization tool: the Siafu open source context simulator [10]. Siafu is parameterized with respect to information automatically generated from the DiaSpec specification of the pervasive computing environment.

III. Simulation Model

Let us now describe the key concepts involved in our approach to simulating a pervasive computing system.

A. Stimulus producers

Stimuli are changes of the environment that are observed by the sensors of the pervasive computing environment. From a simulation perspective, emitting stimuli may trigger sensors (*e.g.,* a motion detector) that publish events, that may in turn trigger more services (*e.g.,* a webcam). Emitters of stimuli are called *stimulus producers*; they are dedicated to a type of stimulus.

Every stimulus has a type that matches the type of one or more sensors. Additionally, every type of stimulus is associated with a set of rules defining its evolution in terms of space, time and a notion of intensity. A type of stimulus can often be modeled by a mathematical function. This function can be periodic (*e.g.,* cars going to workplaces each day) or discrete (*e.g.,* people moving from one room to another). Such a function is typically provided by experts of the application area or the literature in related fields (*e.g.,* luminosity in Graphics [11]). Other types of stimulus can be introduced by replaying logs of measurements collected in an actual environment. For example, to design zero-energy building, extensive measurements are carried out to log the variations in temperature, light and wind over a one-year period [12]. This line of work contributes to building a rich library of measurements, facilitating simulation without compromising accuracy.

However, measurement logs are not available in general for simulation (*e.g.,* fire simulation), requiring the definition of some model to approximate an actual environment, as accurately as necessary. To achieve this goal, our approach is to define an approximation model with respect to each type of stimulus managed by the sensors of an environment. For example, the simulation of location-related sensors can be defined as processing Cartesian coordinate stimuli. If location-related sensors report location information at the granularity

of a room, coarse-grain information can be generated by the stimulus producers (*e.g.,* a unique Cartesian coordinate stimulus per room).

Because a type of stimulus can be consumed by different sensors, stimulus producers are decoupled from the simulated sensors.

So far, we described stimuli as being directly processed by sensors. However, a type of stimulus can also influence the evolution of other types of stimulus; such a type of stimulus is called a *causal stimulus*. For example, fire could be declared as a causal stimulus if we needed to model its resulting action on the temperature stimulus. When a stimulus does not impact others, it is called *simple stimulus*.

B. Simulated services

A simulated environment consists of stimulus producers and simulated services. Like an actual entity, a simulated service interacts with a simulated environment by processing stimuli, performing actions, and exchanging data with other services. Two kinds of services play a key role in simulation: sensors and actuators. The simulated version of a sensor mimics the behavior of the actual sensor, reacting to stimuli generated by the stimulus producers. For example, the simulated version of a motion detector, when turned off, ignores coordinate stimuli. Otherwise, when the motion detector is on and receives coordinate stimuli matching its room, a motion event is published with its room identifier.

As for actuators, they typically accept commands that modify their state as well as the observable environment context. For example, invoking a service of `Light` to turn it on, changes the light state and locally increases the luminosity. The simulated version of a light thus needs to maintain its state (on/off) and to create a stimulus producer to increase luminosity with respect to an intensity specific to the light.

In addition to defining their simulated versions, services need to be deployed. For example, the simulated `Light` service needs to be instantiated as many times as required to mimic the actual environment. In doing so, service instances may be assigned specific attribute values such as their location and luminosity intensity in the light example.

As for the callers of services, they are insensitive to whether or not they are simulated. For example, the same implementation of a `Lighting Manager` service operates `Light` service instances, regardless of whether or not they are simulated.

C. Physical space

To complete the simulation of an environment, we need to model the physical space (*e.g.,* an office space, an apartment, a building or a campus) and to make it evolve as the simulation scenario unfolds. A simulated space allows us to model stimulus propagation, according to pre-defined rules. As well, it is annotated with the location for each service instance whose actual version may impact the actual environment, whether they are fixed, mobile and dynamically appearing.

The model of a physical space is decomposed into polygon-shaped regions. This decomposition is hierarchical, breaking down a physical space into increasingly narrow regions. For example, a building consists of floors, each of which has corridors and rooms, *etc*. Service instances are positioned in the simulated space, in accordance to the desired (or existing) physical setting to be simulated. As an approximation, the intensity of a stimulus is assumed to be uniform within a region.

Our overall simulation model is depicted in Figure 3. To summarize, stimulus producers emit stimuli of various types according to a scenario. In place of actual sensors, simulated ones process these stimuli and produce events. The unchanged application reacts to these events by invoking actuator commands, for example. In turn, actuators change the simulated environment, triggering stimulus producers.

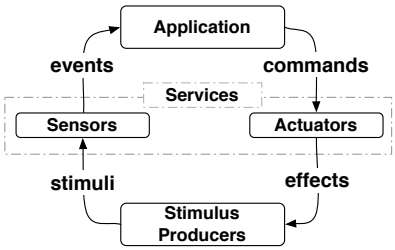

Fig. 3. Simulation model

IV. Developing A Simulated Environment

Given the simulation model presented earlier, we are now ready to develop the simulated version of primitive services and stimulus producers, forming a simulation scenario.

A. Developing simulated services

To develop a service, the programmer first determines the service class it should belong to. The declarations of the selected service class then provide the programmer with a area-specific programming framework for implementing all the facets of the service, ranging from its attributes to its functionalities. This dedicated programming framework is generated from the DiaSpec specification on top of a generic middleware, which includes service and event brokers. To access these brokers, developers call high-level operations to safely (1) register and lookup other services via the service broker or (2) publish, subscribe and receive events via the event broker.

Besides an area-specific programming framework to develop actual services, DiaGen generates a simulation programming framework to develop simulated services. For each service class, a set of classes is generated for programming actual and simulated services, as depicted in Figure 4: actual services (*e.g.,* $\mathcal{A}_1$) extend the $\mathcal{C}$ abstract class of the actual programming framework, whereas simulated services (*e.g.,* $\mathcal{S}_2$) extend the $\mathcal{C}'$ abstract class of the simulation programming framework. A simulation programming framework inherits support provided by the related actual programming

framework and adds simulation-specific functionalities. For instance, it enables sensors to receive simulated stimuli, and actuators to trigger stimulus producers. Moreover, service interactions are automatically logged for monitoring purposes. Figure 5 shows the implementation of a simulated service named `MySimulatedMotionDetector`. The related `SimulatedMotionDetector` abstract class contains an abstract method to receive stimulus events (`receive`) and a concrete method to publish `MotionDetection` events (`publish`).

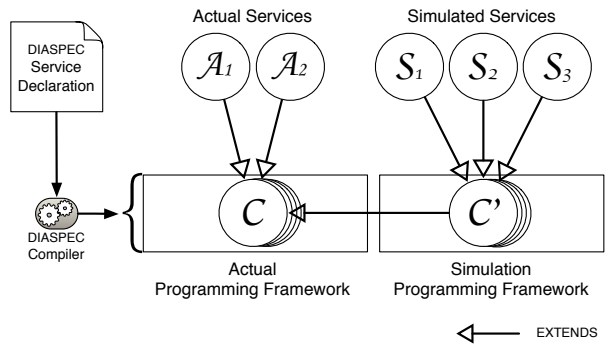

Fig. 4. Correspondence between actual and simulation programming frameworks

```
public class MySimulatedMotionDetector extends
                SimulatedMotionDetector {

  public MySimulatedMotionDetector() {
     [...]
     location.set("Hall");
  }

  public void receive(MotionDetectionStimulus stimulus) {
     [...]
     publish(stimulus.getEvent());
     [...]
  }

}
```

Fig. 5. Implementation of a simulated MotionDetector service

B. Developing hybrid environments

Our approach permits actual services to be used in a simulated environment, whenever desirable. This key feature enables actual services to be incrementally added to the simulated environment. In doing so, it facilitates the transition to an actual environment. Also, it enables to improve the rendering of a simulation by mixing actual entities. For example, an actual LCD screen can be introduced in a simulation to display messages that future users will see.

To examine how actual services are integrated in a simulated environment, recall our inheritance strategy, as illustrated in Figure 4. Because of this strategy, when an application looks up a given service type, it receives the actual services, as well as the simulated ones. Similarly, when an application subscribes to a specific type of event, it receives events from both actual and simulated services. This approach allows applications to be executed in a hybrid environment. Furthermore, actual and simulated services can be added dynamically, as the simulation of a pervasive computing system runs.

C. Developing stimulus producers

The development of stimulus producers is supported by classes of generic stimulus producers for both simple and causal stimuli. A generic stimulus producer is then specialized with respect to a class of stimulus. Classes of stimulus are defined from types of context data defined in DiaSpec, *e.g.,* the building management area includes stimuli of `Luminosity` and `Presence`. Several stimulus producers can be attached to the same class of stimulus. For example, if a room consists of two lights, each one has its own producer of the luminosity stimulus. A stimulus producer must implement a method `getStimuli(Time time)`, which returns a set of pairs (intensity, location). The logic of this method defines the evolution of a source of stimuli. For example, to simulate fire gaining intensity, a stimulus producer gradually increases the intensity of the emitted fire stimulus.

V. Testing Applications

We now detail how applications are tested in the DiaSim simulator. DiaSim executes simulation scenarios, monitors simulations and supports application debugging.

A. Transparent simulation

A programming framework generated by DiaGen provides applications with an abstraction layer to look up and invoke services. In particular, it includes methods to select any node in a service class hierarchy. The result of this selection is a set of all services corresponding to the selected node and its sub-nodes. The developer can further narrow down the service lookup process by specifying the desired values of the attributes. This situation is illustrated in Figure 6. A parameter to a service lookup request is created by invoking `AlarmFilter`. Its location attribute is then set to the coordinates where a motion was detected (`event.getLocation()`). Instances of a node in the service classs hierarchy are looked up by calling `getAlarms` with the request parameter (`filter`).

Because of this abstraction layer, simulation is achieved transparently: the same application code looks up and invokes services, whether or not simulated. This transparent simulation applies to all aspects of a pervasive computing application. For another example, simulated event producers and consumers can be added to a pervasive computing system, without requiring any change in the application code.

B. Simulator architecture

The overall architecture of DiaSim is displayed in Figure 7. It consists of an emulator to support the execution of pervasive computing applications and a simulator of context to manage stimuli. The simulator of context communicates the simulation data to the monitor for rendering purposes.

```
public class MySurveillanceManager extends
                SurveillanceManager {

  public void receive(MotionDetection event) {
     [...]
     AlarmFilter filter = new AlarmFilter();
     filter.setAttribute("location", event.getLocation());

     Alars alarms = getAlarms(filter);
     alarms.on();
  }
}
```

Fig. 6. Service lookup and invocation

1) Executing simulation scenarios: A simulator of context generates stimuli as a given simulation scenario unfolds. It consists of stimulus producers and a scenario controller that dispatches stimuli to the relevant services. The *scenario controller* is a mediator, periodically querying the stimulus producers to feed the simulated sensors. The generality of our approach enables it to be implemented as a DiaGen service, declared as a producer of stimulus events in the DiaSpec specification. Correspondingly, simulated sensors are declared as consumers of stimulus events. For example, the scenario controller collects stimuli of outdoor luminosity and passes them to outside light sensors.

Actuators can create changes to the simulated environment. To do so, they register new stimulus producers to the scenario controller. For example, when fire is detected, a fire sprinkler discharges water on a given region. Because water is declared as a causal stimulus with respect to fire, it reduces the fire intensity. When the application deactivates the fire sprinkler, the water stimulus producer is disposed of by the scenario controller.

2) Monitoring simulation: The scenario controller receives simulation data from stimulus producers and primitive services to keep track of the simulated environment state. The scenario controller passes simulation data to the monitoring engine that graphically renders simulation scenarios. The monitoring engine also accepts live user interactions, to pause the simulation or modify the scenario on-the-fly (*e.g.,* by adding new stimulus producers). Beyond the visual rendering of a simulation, we propose additional functionalities to DiaSim to further assist the user, as presented next.

C. Application testing support

Monitoring a simulation requires measuring, collecting and rendering a stream of simulation data. Because of its volume, simulation data often require to be approximated in order to be rendered. To do so, the simulated environment is approximated in space and time. Space approximation provides an idealized map of the physical space, rendering the evolution of primitive services (*e.g.,* alarm ringing, event publishing) and stimuli (*e.g.,* fire spreading, people moving). Environments are also approximated in time, decoupling the rendering time from the real time. As a result, the user often cannot follow the simulation in real time. To focus on the sequence of events leading to an error, the monitoring engine of DiaSim provides

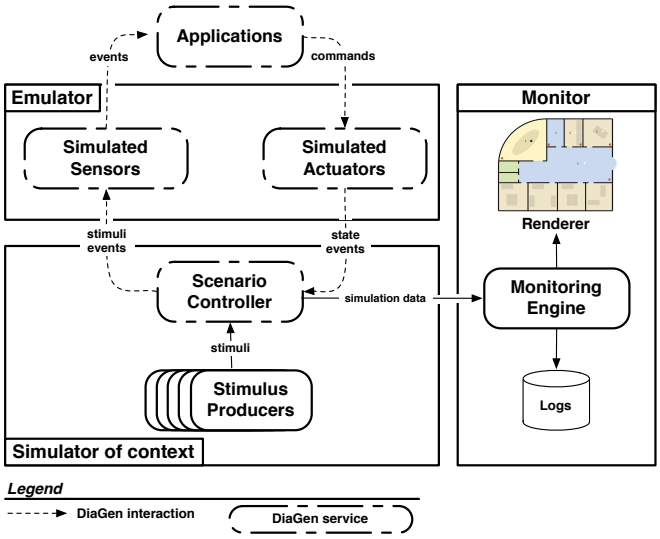

Fig. 7. The DiaSim simulator architecture

time shifting functionalities, to replay part of a simulation. Raw data from the simulation log can be directly browsed by DiaSim, like network traces by network analyzers [13]. A simulation log contains information about interactions between entities (*i.e.,* time, source, destination, type of interactions, interaction parameters) and between stimuli and entities (*i.e.,* time, source, destination, class of stimuli, stimuli parameters). Replays help to isolate bugs but does not ensure applications have been fixed correctly. Reproducing exact testing conditions is required to validate a new version of an application. To do so, a simulation scenario completely defines the simulated environment and its behavior, making testing conditions deterministic and reproducible.

VI. Implementation and Validation

The components of DiaSim, detailed in Figure 7, are implemented in Java, consist of 15,000 lines of code, and were developed over one year.

To validate DiaSim, we simulated various applications in the building management area, depicted in Figure 1. DiaSim validated the logic of these applications and the feasibility of such a deployment in ENSEIRB[1], an engineering school to which the authors are affiliated. Videos reporting various simulation scenarios are available on our site.[2]

Aside from the ENSEIRB environment, we simulated and deployed home automation applications. This was done as part of a project in collaboration with telecommunications company. For the sake of conciseness, we do not detail further this application area.

A. Applications

The ENSEIRB school is a three-floor building of 13,500 m^2, consisting of several lecture halls, labs and recreation rooms

[1]http://www.enseirb.fr
[2]https://diasim.bordeaux.inria.fr

for students. ENSEIRB hosts up to 900 occupants, including students from various countries and faculty members.

Using the generated Java programming framework, we developed various applications. Let us briefly introduce some of them. The newscast application displays news and teaching schedules on school LCD screens and adapts the contents with respect to the department affiliation and the nationality of the people surrounding the screens. The surveillance manager alerts security personnel when an intrusion or a theft is detected. The meeting manager notifies users about their meetings if they are not present in the corresponding meeting room. It also displays information about meetings on the school LCD screens when they involve a group of students from a department. The lighting manager controls lights based on outside luminosity, the school calendar and school occupancy.

B. Setting up simulation scenarios

To test applications, we simulated three main scenarios: a working day, a week-end day and an industry day. Each scenario is simulated using variations, including primitive service failures, number and location of primitive services, and number of users. Scenarios are defined using a Java GUI: the *scenario editor* (Figure 8). From the DiaSpec definitions, simulated services are either graphically defined using a wizard, or developed using the simulation programming framework generated by DiaGen. In the first case, attributes are defined by filling in a form and the location attribute is set by dragging and dropping service icons in the simulated space. The behavior of the simulated service is then defined by graphically selecting and parameterizing the appropriate class of behaviors. For example, `Audio Notification` services embedded in loudspeakers are simulated with a class of text-to-speech services. For another example, `Burglar Siren` services are simulated with a class of audio file renderer parameterized by an audio file. We simulated a variety of sensors, notification services and lights. Furthermore, we integrated actual services in the simulation, either to ease the scenario definition (*e.g.,* calendars and the profile database) or to validate their behavior (*e.g.,* `News Notification` services embedded in LCD screens).

The second part of the scenario definition is the configuration of stimulus producers. The scenario editor supports the definition of stimulus producers and their behavior, by allowing the user to define stimulus intensities in areas of the simulated space at specific moments in time. For example, a producer of motion stimuli simulates a user moving in a school hallway at a given time. Alternatively, stimulus producers are defined by a modeling function (*e.g.,* a function defining the outside luminosity for 24-hour period) or previously logged measurements (*e.g.,* class schedules or statistics on class attendance).

C. Monitoring the simulation

A scenario is saved as an XML file that can later be modified by the scenario editor. The XML file configures the DiaSim simulator with the defined scenario. DiaSim includes a simulation renderer, which is based on Siafu in our current implementation [10]. Our simulator interfaces with Siafu to use its rendering and time-control functionalities. On top of a picture of the simulated space, the simulation renderer displays services and stimuli, as shown in Figure 9.

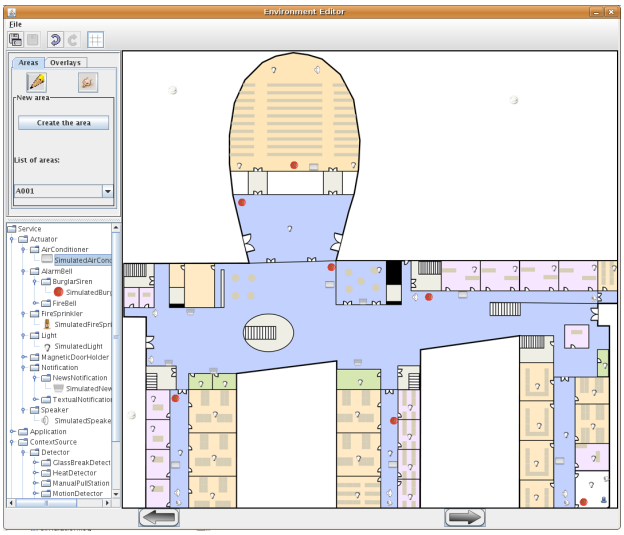

Fig. 8. The scenario editor (ENSEIRB)

The simulation renderer shows the state of the primitive services, by displaying a bubble of raw text above services (*e.g.,* when sensors publish events) and/or modifying the visual representation of the service (*e.g.,* a yellow light is displayed when turned on). To complement these macroscopic views, we enriched Siafu's rendering functionalities with Java and Web interfaces, and audio streams. Actual services greatly benefit from these enriched views as most of them require more than just raw text to display information. In the ENSEIRB simulation, clicking on school LCD screens runs the Web interface of the corresponding actual `News Notification` services. We also used enriched views for simulated services, *e.g.,* loudspeakers are rendered using audio streams.

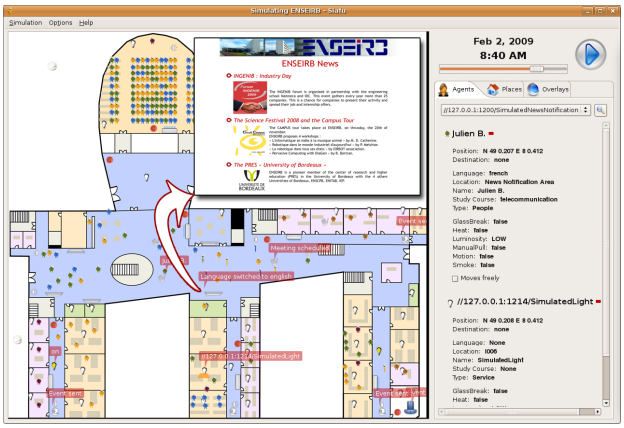

Fig. 9. The simulation renderer (ENSEIRB)

DiaGen supports several back-ends including Java RMI, SIP [14] and Web Services. A back-end defines the communication protocol used by the services to communicate with each other. The simulation back-end used by DiaSim is derived from the Java RMI back-end. This strategy allows us to integrate remote actual services and to distribute the workload over several different hosts when numerous services are in play.

D. Assessment

Target environments and simulation scenarios were successfully defined and simulated in the DiaSim simulator. In the home environment, actual services were added incrementally in the simulation. At the end of the development process, all services were actual and the emulation layer was only used for monitoring purpose, demonstrating the flexibility of our emulation layer.

In the ENSEIRB experiment, the simulation allowed us to validate the coordination logic at a large scale, combining 75 services, 48 stimulus producers, 200 people and 6 applications. Some services were coordinated and shared by several applications (*e.g.,* `Notification`, `Calendar` and `Positioning System`). It was thus essential to ensure the usability of these applications by preventing potential conflicts. We also checked that the application behavior met its requirements when the context of deployment or execution changes (*e.g.,* disappearing primitive services and moving individuals). For example, we improved the newscast application by making it less sensitive to people that do not stop long enough in front of school LCD screens. We also optimized the air conditioning consumption by combining information about the building occupancy and class schedules.

E. Discussion

We now examine pragmatic issues involved in developing and using a simulated environment. We start by discussing the potential pitfalls of this development. Then, we investigate the performance issues involved in running large-scale simulations.

1) Pitfalls: A simulation consists of tested applications and the simulated environment. The output of the simulated environment is the input of the tested applications and vice versa. The complexity of the simulated environment depends on the characteristics of the actual environment and how accurately it needs to be modeled. These issues go beyond the scope of our generated simulation support that is aimed to facilitate the programming of the simulated environment. Producing faithful stimuli and defining meaningful simulation logic are left to the developer.

Specifically, the values generated by a stimulus producer need to be faithful to some simulation model. The simulation model must provide an accuracy that matches the granularity of the situations to be tested. To define a stimulus producer, one option is to replay data logged from actual sensors, whether or not verbatim. Another option is to define a stimulus producer using some domain-specific modeling function. Issues about the correctness of the stimulus producer arise when either the logged data are transformed or a domain-specific modeling function is introduced. Beyond stimulus producers, emulated actuators may have an effect on the simulated environment (*e.g.,* a light impacts the luminosity). As a result, the stimulus producers need to subscribe to all actuator events that may have an effect on the values they generate.

To illustrate these issues consider the sun luminosity. It can simply be defined by a mathematical function. However, its impact on a building is difficult to model as it depends on the number, size and location of windows, and the building structure. Our approach does not help in defining an accurate model of this situation; this is left to the simulation developer that must take into account the simulation requirements.

Another source of inaccuracy may be created by the operations that merge stimulus intensities produced by the same region of the physical space. For example, consider the luminosity in a hall coming from the luminosity of the surrounding rooms. These luminosity intensities are sent to the luminosity producer of the hall, which merges them and passes the new intensity to the hall sensors. This merging operation is also user-defined; to be meaningful its definition needs to rely domain-specific knowledge.

Primitive services, such as sensors, are emulated so that applications interact with them without code modification. To be faithful, an emulated sensor should have an observable behavior that is equivalent to its actual counterpart. To do so, an emulated sensor must be programmed such that, for a given input, it produces the same output as its actual counterpart.

2) Performance: The simulation of physical spaces may involve lots of services, accurate simulation models, and rich simulation logic. This situation calls for a scalable simulator.

To support computing-intensive simulation, DiaSpec enables to distribute simulated services and stimulus producers. This distribution is naturally achieved using DiaSpec because entities communicate via a software bus that abstracts over the underlying communication protocol. Our implementation of DiaSpec supports several software buses including a local software bus, Java RMI, SIP [14] and Web Services. The selection of the software bus is done at deployment time and does not affect service implementation. When the simulation back-end used by DiaSim is Java RMI, the workload can be distributed over several different hosts, enabling numerous services and stimulus producers to be introduced. A distributed software bus also makes it possible to perform hybrid simulation by integrating distributed, actual entities.

VII. Related Work

Few simulators are dedicated to the testing of pervasive computing applications [5], [6], [7], [8]. Ubiwise [5] and Tatus [6] are built upon 3D game graphics engines, respectively Quake III Arena and Half-Life. By providing a first person view of the simulated pervasive environments, they both allow the user to experience these simulated environments. However the game graphics engine becomes a burden

when it comes to define new scenarios; users can neither add their own actuators and sensors, nor simulate arbitrary context data. Moreover, the same scenario cannot be run multiple times. In contrast, the Lancaster simulator enables deterministic testing conditions and emulation to test location-based applications [7]. However, actuators and sensors are not explicitly provided and the development of new simulation environments is not supported. The PiCSE simulator addresses the problem of extensibility by providing generic libraries to create sensors and actuators [8]. However, users still have to manually specialize the simulator for every new application areas. In contrast, DiaSim relies on DiaGen to automatically customize simulation tools. Finally, existing approaches do not propose an emulation framework to incrementally integrate actual entities in a simulated system.

Some simulators focus on the simulation of context [15], [16], [10]. The Generic Location Event Simulator publishes location information, which can be used by location-based applications [15]. However, it is limited to location information. SimuContext [16] and Siafu [10] are two other context simulators that go one step further, enabling to define any context types. Siafu also graphically renders simulated environments. However, as a context simulator, Siafu does not provide any support to simulate services and applications.

Various approaches propose to simulate sensor networks [17], [18], [19], [20] and could complement our approach. These simulators provide a more comprehensive support for the simulation of sensors compared to previous approaches. However, they do not consider issues of application development and testing. Network emulators that focus on network-related issues have been proposed [21], [22] and could also complement our approach.

VIII. Conclusion and Future Work

In this paper, we have presented a novel approach to simulating pervasive computing applications. DiaSim is parameterized with respect to a description of a pervasive computing environment, relevant to a given area. This description automatically generates an emulation layer to run the application code unchanged. Also, a simulation programming framework is generated to allow the development of the simulation logic for primitive services. A graphical environment is provided to the user to define a simulated space, simulation scenarios, and to monitor and debug a simulated pervasive computing system.

We are extending DiaSim to simulate session-based services, to handle multimedia streams. We also plan to render simulations in 3D using Blender, an authoring tool for creating 3D animations and video games.

Acknowledgments

The authors would like to thank the reviewers for their insightful comments. Also, thanks to the Phoenix group for many fruitful discussions.

References

[1] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PERCOM'05*, pages 7–16, 2005.

[2] R. Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[3] W. Jouve, J. Lancia, N. Palix, C. Consel, and J. Lawall. High-level programming support for robust pervasive computing applications. In *Proceedings of the 6th IEEE Conference on Pervasive Computing and Communications (PERCOM'08)*, pages 252–255, mar 2008.

[4] iPhone SDK, http://developer.apple.com/iphone/program/download.html.

[5] J. J. Barton and V. Vijayaraghavan. Ubiwise, a ubiquitous wireless infrastructure simulation environment. Technical report, Hewlett Packard, 2002.

[6] E. O'Neill, M. Klepal, D. Lewis, T. O'Donnell, D. O'Sullivan, and D. Pesch. A testbed for evaluating human interaction with ubiquitous computing environments. In *TRIDENTCOM '05. Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, 2005.

[7] R. Morla and N. Davies. Evaluating a location-based application: A hybrid test and simulation environment. *IEEE Pervasive Computing*, 3(3):48–56, Juil-Sep 2004.

[8] V. Reynolds, V. Cahill, and A. Senart. Requirements for an ubiquitous computing simulation and emulation environment. In *InterSense '06. Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks*, 2006.

[9] W. Jouve, N. Palix, C. Consel, and P. Kadionik. A SIP-based programming framework for advanced telephony applications. In *The 2nd LNCS Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, jul 2008. Awarded best student paper.

[10] M. Martin and P. 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, California, USA, July 2006. IEEE Computer Society.

[11] A. J. Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100. ACM Press/Addison-Wesley Publishing Co., 1999.

[12] R. E. Frechette III and R. Gilchrist. Towards zero energy, a case study: Pearl River Tower, Guangzhou, China. In *CTBUH: Proceedings of the Council on Tall Buildings and Urban Habitat's 8th World Congress*, pages 7–16, 2008.

[13] Wireshark: A Network Protocol Analyzer. http://www.wireshark.org/.

[14] Rosenberg, J. et al. SIP : Session Initiation Protocol. RFC 3261, IETF, June 2002.

[15] K. Sanmugalingam and G. Coulouris. A generic location event simulator. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 308–315. Springer-Verlag, 2002.

[16] T. Broens and A. van Halteren. Simucontext: Simply simulate context. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 45. IEEE Computer Society, 2006.

[17] S. Sundresh, W. Kim, and G. Agha. Sens: A sensor, environment and network simulator. In *37th Annual Simulation Symposium (ANSS37)*, 2004.

[18] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM, 2003.

[19] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 477–482, 2005.

[20] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, , and M. Karir. Atemu: A fine-grained sensor network simulator. In *SECON'04: First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[21] Information Sciences Institute. NS-2 network simulator. Software Package, 2003. http://www.isi.edu/nsnam/ns/.

[22] F. D'Aprano, M. de Leoni, and M. Mecella. Emulating mobile ad-hoc networks of hand-held devices: the octopus virtual environment. In *MobiEval '07: Proceedings of the 1st international workshop on System evaluation for mobile platforms*, pages 35–40. ACM, 2007.