DiaSim: A Simulator for Pervasive Computing Applications

Julien Bruneau, Charles Consel

To cite this version:

Julien Bruneau, Charles Consel. DiaSim: A Simulator for Pervasive Computing Applications. Software: Practice and Experience, Wiley, 2013, 43 (8), 10.1002/spe.2130 . hal-00715745

HAL Id: hal-00715745
https://hal.inria.fr/hal-00715745

Submitted on 18 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DiaSim: A Simulator for Pervasive Computing Applications

Julien Bruneau and Charles Consel
Inria Bordeaux, France
{julien.bruneau, 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.

1 Introduction

Numerous pervasive computing applications coordinate a variety of networked entities collecting data from sensors and reacting by triggering actuators. These entities are either software or hardware. To collect data, sensors process stimuli that are observable changes of the environment (*e.g.,* fire and motion). Triggering 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 have 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.

Various middlewares and programming frameworks have been proposed to ease the development of pervasive computing applications [1, 2, 3, 4]. 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.,* GPS and accelerometer) 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 emulators [5, 6, 7]. 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 emulators gives us a practical basis for identifying the requirements for pervasive computing systems. Let us now examine these requirements.

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 a taxonomy of entities dedicated to a given activity (*e.g.,* cameras, motion detectors and alarms in the building surveillance area). 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 types of entities and stimuli to be introduced easily.

Transparent simulation A key feature of most embedded systems emulators 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.

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.

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 [8, 9, 10]. 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 [11]. 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 [12].

This paper

This paper presents DiaSim, a simulator for pervasive computing applications. DiaSim specifically targets applications which (1) are based on sensors and actuators, (2) are deployed in a physical environment, and (3) involve users. DiaSim enables the simulation of the application logic of such applications but does not simulate the other components of a pervasive computing environment. For instance, it does not provide any support to estimate physical aspects of an environment (*e.g.,* the thermal modeling of a room), to simulate the network traffic between application components, or to model the behavior of application users.

DiaSim is parameterized with respect to a high-level description of the target pervasive computing environment. Such a description defines a taxonomy of entities, whether hardware or software, relevant to a target pervasive computing area. Both simulated and real 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 real environment. The resulting simulated pervasive computing environment enables the testing of 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 real environment. We ensure a functional correspondence between a simulated environment and a real 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 real entities. Hybrid simulation is a key feature to successfully transition to a real environment: it allows real entities 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 [13]. 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 [14].

Outline

Our simulation approach is parameterized with respect to a high-level description of the pervasive computing system to be simulated. This description is used

to generate a programming framework to implement and simulate the described pervasive computing system (Section 2). Our underlying simulation model, dedicated to pervasive computing systems, is introduced by examining its key concepts (Section 3). The simulation support of the generated programming framework is then presented, underlying how it is leveraged by the tester to implement simulation scenarios (Section 4). At runtime, our simulation support allows the tester to execute and monitor a simulation (Section 5). To validate our approach, we have applied DiaSim to a wide range of pervasive computing systems and simulation scenarios. We have also evaluated DiaSim in terms of scalability, performance and usability (Section 6). Finally, we have compared our approach with the related works (Section 7) and drawn conclusions (Section 8).

2 Our Approach

To ease the simulation of a pervasive computing system, we propose a simulation approach that is parameterized with respect to a high-level description of this system. This enables to specialize our simulation support for simulating this particular system, easing the task of the tester. Specifically, the description is used as an input to generate (1) a programming support for application development, (2) building-block implementations of a simulated environment, (3) a development support for simulation scenarios, and (4) configurations for simulation rendering.

2.1 Describing a pervasive computing system

We rely on the DiaSpec language [15, 16] for describing pervasive computing systems. Such systems comprise a pervasive computing environment, as well as pervasive computing applications that interact with this environment. DiaSpec first consists of a taxonomy language dedicated to describing classes of entities that are relevant to the target pervasive computing environment. DiaSpec also provides an Architecture Description Language (ADL) layer that declares the components of a pervasive computing application. Let us introduce the salient features of DiaSpec via excerpts of a heating control system. A detailed description of DiaSpec is presented elsewhere [16].

Taxonomy. A pervasive computing environment for a given area is defined as a taxonomy of classes of entities. An extract of the DiaSpec taxonomy for a heating control system is shown in Figure 1. Each of these classes represents a set of entities sharing common capabilities. Entity classes are introduced by the **device** keyword. `Heater` and `MotionDetector` are examples of entity classes described in the taxonomy of our heating control system. An entity consists of sensing capabilities, producing data, and actuating capabilities, providing actions. Accordingly, an entity description declares a data source for each one of its sensing capabilities. The sensing capabilities of an entity class

```
device LocatedDevice {
  attribute location as Location;
}

device MotionDetector extends LocatedDevice {
  source detection as Boolean;
}

device TemperatureSensor extends LocatedDevice {
  source temperature as Temperature;
}

device Heater extends LocatedDevice {
  action Heat { on(); off(); };
}

structure Location { room as String; }
structure Temperature { value as Float; }

[ ... ]
```

Figure 1: Extract of the heating control system taxonomy. DiaSpec keywords are printed in **bold**.

are declared by the **source** keyword. For example, the `MotionDetector` class defines a `detection` data source. As well, an actuating capability corresponds to a set of method declarations. Actuating capabilities are declared by the **action** keyword. For instance, the `Heater` class declares the `Heat` action. An entity declaration also includes attributes, characterizing properties of entity instances. Attributes are introduced using the **attribute** keyword. For example, the `LocatedDevice` instances are characterized by their location. Finally, entity classes are organized hierarchically, allowing them to inherit attributes, data sources and actions. In our heating control system, the `MotionDetector`, `TemperatureSensor` and `Heater` classes of entities extend the `LocatedDevice` class. Thus, they all inherit the `location` attribute.

Architecture. DiaSpec provides an ADL layer for specifying the architecture of pervasive computing applications. This layer is dedicated to an architectural pattern commonly used in the pervasive computing domain [1, 17, 18]. This architectural pattern is illustrated in Figure 2. Pervasive computing applications are decomposed in two types of components: context and controller. Context components are fueled by sensing entities. These components filter, interpret and aggregate these data to make them amenable to the application needs. Controller components receive application-level data from context components and determine the actions to be triggered on entities.

The ADL layer of DiaSpec is illustrated with our heating control system. A graphical representation of the architecture of this system is displayed in Figure 3. At the bottom of this figure are the entity sources, as described in

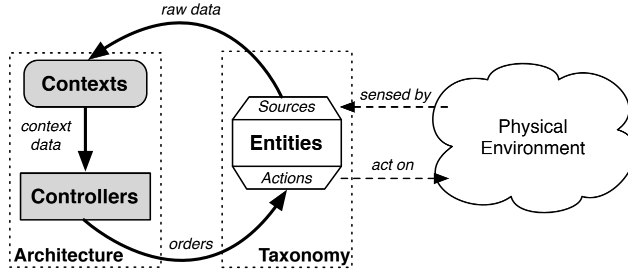

Figure 2: Architectural pattern of a pervasive computing application.

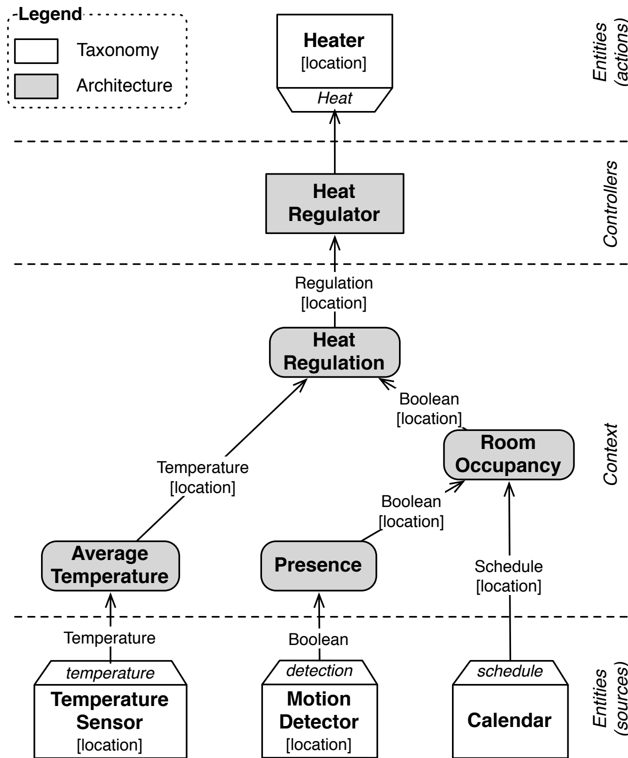

Figure 3: Specification of the heating control system

the taxonomy. The layer above consists of the context components fueled by entity sources. The next layer above gathers the controller components that invoke the top layer of our system, namely, actions of entity actuators. The

```
context AverageTemperature as Temperature indexed by location as Location {
  source temperature from TemperatureSensor;
}

context Presence as Boolean indexed by location as Location {
  source detection from MotionDetector;
}

context RoomOccupancy as Boolean indexed by location as Location {
  source schedule from Calendar;
  context Presence;
}

context HeatRegulation as Regulation indexed by location as Location {
  context AverageTemperature;
  context RoomOccupancy;
}

controller HeatRegulator {
  context HeatRegulation;
  action Heat on Heater;
}
```

Figure 4: Architecture of the heating control system. DiaSpec keywords are printed in **bold**.

DiaSpec description of the architecture of the heating control system is presented in Figure 4. In this application, temperature values are provided to the `AverageTemperature` component, declared using the **context** keyword. This component calculates the average temperature for each room of a building. It processes the average temperature using the `temperature` source provided by the temperature sensors. This is declared using the **source** keyword that takes a source name and a class of entities. To process the average temperature on a per-room basis, this context is declared as indexed by `Location`. In doing so, each calculated average temperature is associated with a location. The `Presence` context determines whether a room is currently occupied from the information provided by motion detectors. The `RoomOccupancy` context determines a more advanced room occupancy than the `Presence` context. This occupancy takes into account the information provided by the `Presence` context, as well as the room schedule given by a calendar. Thus, the information provided by `RoomOccupancy` allows to heat a room prior to being occupied. When the occupancy of a room changes, the `HeatRegulation` context is invoked. Depending on the current temperature in this room, it may order a heat regulation to the `HeatRegulator` controller, declared using the **controller** keyword. The controller acts on `Heater` instances to regulate the temperature as required by the `HeatRegulation` context. This is declared using the **action** keyword.

2.2 Implementing a pervasive computing system

The DiaSpec compiler generates a Java programming framework from both a taxonomy definition and an architecture description. This programming framework provides support for the implementation of the described pervasive computing system. A generated programming framework contains an abstract class for each DiaSpec component declaration (entity, context, and controller). These abstract classes include generated methods to support the implementation (*e.g.,* entity discovery and interactions). The generated abstract classes also include abstract method declarations to allow the developer to program the application logic (*e.g.,* triggering entity actions).

Implementing a DiaSpec component is done by subclassing the corresponding generated abstract class. The developer implements the application logic in each abstract method of these subclasses. As shown in Figure 5, the implementation of the `HeatRegulator` controller extends the generated abstract class `AbstractHeatRegulator`. In doing so, the developer is required to implement the `onHeatRegulation` abstract method to receive a value published by the `HeatRegulation` context. In addition to this value, this method is passed a support object to discover devices and actuate them (`discover`). In this example, when a heat regulation is required, the controller implementation discovers the heaters related to the location of interest and turns them on or off.

```
public class HeatRegulator extends AbstractHeatRegulator {

  @Override
  public void onHeatRegulation(HeatRegulation regulation, Discover discover) {
    HeaterComposite heaters = discover.heatersWhere().location(regulation.
         getLocation());
    if (regulation.getType() == Regulation.START_HEATING)
      heaters.on();
    else if (regulation.getType() == Regulation.STOP_HEATING)
      heaters.off();
  }

}
```

Figure 5: Implementation of the `HeatRegulator` controller.

2.3 Simulating a pervasive computing system

From the high-level description of a pervasive computing system, we also generate a simulation support to test this system before its actual deployment.

Simulated environment. To abstract over distributed systems technologies, the DiaSpec compiler follows a layered architecture for the generated programming frameworks. This architecture has made it possible to easily introduce a simulated environment composed of simulated entities. Our layered architecture

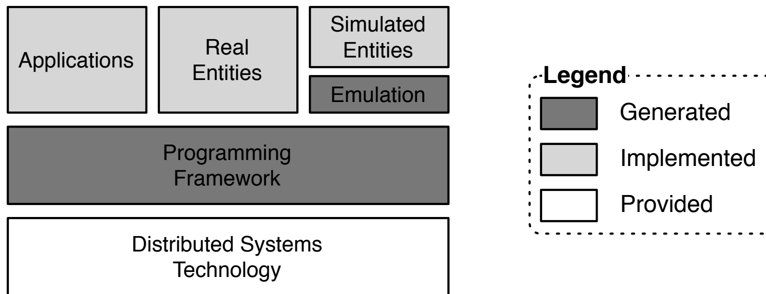

Figure 6: Our layered architecture

is shown in Figure 6. Intuitively, our approach consists of generating an emulation layer between the programming framework and the simulated entities. When the pervasive computing application interacts with a simulated entity, the emulation layer triggers its simulated version. This emulation layer allows an application to interact with a simulated environment transparently, without knowing whether an entity is simulated or real. The simulation logic introduced for a simulated entity replaces the real entity.

Simulation scenarios. Once the entities 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 actions on entities whose coordinated actions achieve specific tasks. For example, a heating control system regulates the building temperature by coordinating temperature sensors, motion detectors, and heaters.

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 [19]. Siafu is parameterized with respect to information automatically generated from the DiaSpec specification of the pervasive computing system.

3 Simulation Model

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

3.1 Stimulus producers

Stimuli are changes of the environment that are observed by the sensors of the pervasive computing environment. From a simulation perspective, emitting environment stimuli may trigger an entity data source (*e.g.,* the detection source of a motion detector) that publishes events, that may eventually trigger actions on actuators (*e.g.,* turning on a light). 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 data sources provided by entities. Additionally, every type of stimulus is associated with a set of rules defining its evolution in terms of space, time and value. Physical environment stimuli are often modeled by mathematical definitions (*e.g.,* with ordinary/partial differential equations). Such a definition is typically provided by experts of the application area or the literature in related fields. For example, temperature stimuli required for testing a heating control system can be modeled with heat transfer formulas described in any thermodynamics books (*e.g.,* [20]).

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 [21]. 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 entity data sources, stimulus producers are decoupled from the simulated entities.

So far, we described stimuli as being directly processed by entities. 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*.

3.2 Simulated entities

A simulated environment consists of stimulus producers and simulated entities. Like a real entity, a simulated entity interacts with a simulated environment by processing stimuli, performing actions, and exchanging data with pervasive computing applications. An entity has two kinds of facets, each one playing

a key role in simulation: data source and action. The simulated version of a data source mimics the behavior of a real data source, 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.

An action provided by an entity typically modifies the state of this entity as well as the observable environment context. For example, invoking an action on a 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, entities need to be deployed. For example, the simulated `Light` entity needs to be instantiated as many times as required to mimic the real environment. In doing so, entity instances may be assigned specific attribute values such as their location and luminosity intensity in the light example.

As for context and controller components, they are insensitive to whether or not entities are simulated. For example in our heating control system, the same implementation of a `HeatRegulator` controller operates `Heater` instances, regardless of whether or not they are simulated. As well, the `Presence` context will not interact any differently with a simulated or a real `MotionDetector` instance.

3.3 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 entity instance whose real version may impact the physical 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.* Entity 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 7. Stimulus producers emit stimuli of various types according to a scenario. The values of the stimuli can either be read from logs of measurements or can be computed from an approximated physical model. In place of data sources of real entities, data sources of simulated entities process these stimuli and produce events. The

unchanged application reacts to these events by invoking entity actions. In turn, actions change the simulated environment, triggering stimulus producers.

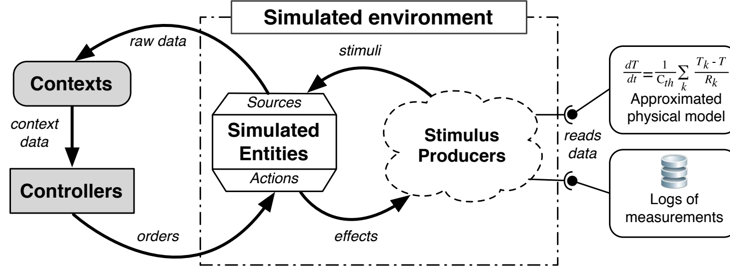

Figure 7: Simulation model

4 Developing A Simulated Environment

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

4.1 Developing simulated entities

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

Besides an area-specific programming framework to develop real entities, the DiaSpec compiler generates a simulation programming framework to develop simulated entities. For each entity class, a set of Java classes is generated for programming real and simulated entities, as depicted in Figure 8: real entities (*e.g.,* $\mathcal{R}_1$) extend the $\mathcal{C}$ abstract class of the real programming framework, whereas simulated entities (*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 real programming framework and adds simulation-specific functionalities. For instance, it enables entities to receive simulated stimuli and to trigger stimulus producers. Figure 9 shows a generated abstract class that is used for implementing simulated motion detectors.

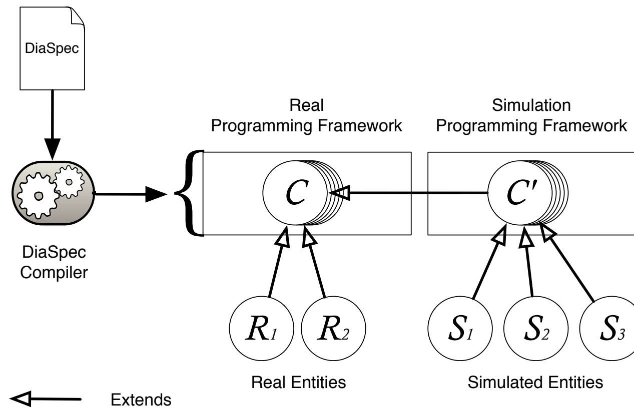

Figure 8: Correspondence between real and simulation programming frameworks

To implement the simulated version of an entity, the tester subclasses the corresponding abstract class. For instance, Figure 10 shows the implementation of a simulated motion detector named `MySimulatedMotionDetector`. The related `SimulatedMotionDetector` abstract class contains an abstract method to receive simulated detection events (`receive`) and a concrete method to publish `MotionDetection` events (`publish`).

As illustrated by Figure 10, the implementation of a simulated entity is often trivial. It only forwards the received stimuli. Thus, to simplify the tester task, the simulation layer of the generated programming framework provides such implementations for all the simulated entities. However, nothing prevents the tester from implementing more sophisticated behaviors by extending the corresponding abstract class.

```
public class SimulatedMotionDetector extends
                    AbstractMotionDetector implements SimulatedEntity {

  public SimulatedMotionDetector(Location location) {
     super(location);
  }

  @Override
  public void receive(Stimulus stimulus) {
     if (stimulus.getName().equals("detection")
        receive((Boolean)stimulus.getEvent());
  }

  public abstract void receive(Boolean detection);
}
```

Figure 9: Implementation of the generated `SimulatedMotionDetector` class.

```
public class MySimulatedMotionDetector extends
                SimulatedMotionDetector {

  public MySimulatedMotionDetector(Location location) {
     super(location);
  }

  @Override
  public void receive(Boolean detection) {
     publish(detection);
  }
}
```

Figure 10: Implementation of a simulated `MotionDetector` entity

4.2 Developing hybrid environments

Our approach permits real entities to be used in a simulated environment, whenever desirable. This key feature enables real entities to be incrementally added to the simulated environment, facilitating the transition to a real environment. Also, this strategy enables to improve the rendering of a simulation by mixing real entities. For example, a real LCD screen can be introduced in a simulation to display messages that future users will read.

To examine how real entities are integrated in a simulated environment, recall our inheritance strategy, as illustrated in Figure 8. Because of this strategy, when a controller looks up a given entity type, it receives the real entities, as well as the simulated ones. Similarly, when a context subscribes to a data source, it can interact with both real and simulated data sources. This approach allows applications to be executed in a hybrid environment. Furthermore, real and simulated entities can be added dynamically, as the simulation of a pervasive computing system runs.

4.3 Developing stimulus producers

The development of stimulus producers is facilitated by a simulation programming framework. This programming support provides a generic `StimulusProducer` class that the tester can use to create his own stimulus producers. Classes of stimulus are defined from types of data sources defined in DiaSpec. For example, the building management area includes stimuli of temperature and motion detection. Several stimulus producers can be attached to the same class of stimulus. For example, if a room contains two heaters, each one has its own producer of temperature stimuli. A stimulus producer 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.

In our heating control system, we use this simulation programming framework to produce motion events when simulated people move in the range of a

```
public class MyAgentModel extends DiaSimAgentModel implements AgentListener {     1

  private static int RANGE = 5;                                                    3
  private StimulusProducer stimulusProducer;                                       4

  public MyAgentModel(World world) {                                               6
    super(world);                                                                  7
    Source motionDetectionSource = new Source("MotionDetector", "detection",      8
         "Boolean");
    stimulusProducer = new StimulusProducer(motionDetectionSource);                9
  }                                                                                10

  @Override                                                                        12
  public List<DiaSimAgent> createAgents() {                                        13
    List<DiaSimAgent> agents = super.createAgents();                               14
    AgendaStimulusProducer studentAgenda = new                                     15
         AgendaStimulusProducer(``resources/studentagenda.xml'');
    AgendaStimulusProducer teacherAgenda = new                                     16
         AgendaStimulusProducer(``resources/teacheragenda.xml'');
    for (DiaSimAgent a : agents) {                                                 17
      agent.addAgentListener(this);                                                18
      if (agent.getType().equals(``Student'')                                      19
        agent.setAgendaStimulusProducer(studentAgenda);                            20
      else if (agent.getType().equals(``Teacher'')                                 21
        agent.setAgendaStimulusProducer(teacherAgenda);                            22
    }                                                                              23
    return agents;                                                                 24
  }                                                                                25

  @Override                                                                        27
  public void agentMoved(Agent agent, String location) {                           28
    for (DiaSimDevice d : getDevices()) {                                          29
      int distance = agent.distanceFrom(d.getPosition());                          30
      if (d.getType().equals("MotionDetector")                                     31
             && distance < RANGE)                                                  32
        stimulusProducer.publish(true,location);                                   33
    }                                                                              34
  }                                                                                35
}                                                                                  36
```

Figure 11: Implementation of the MyAgentModel class used in the heating control system simulation. This class is responsible for publishing motion detection events when simulated people come within a range of a motion detector.

motion detector. To illustrate the use of the simulation programming framework, Figure 11 presents the implementation of the class that publishes motion events. This class extends `DiaSimAgentModel`. The `DiaSimAgentModel` class is provided by the simulation programming framework and provides programming support for handling the simulated people of the simulation. In this example, it is used to be notified when a simulated agent moves in the detection area of a motion detector. A stimulus producer is created in this class: `stimulusProducer` (Figure 11, line 9). The simulation programming framework allows to be notified when an agent moves by implementing the `AgentListener` interface. When an agent moves, the `agentMoved` method is called (Figure 11, line 28). Finally, when an agent moves in the detection area of a motion detector, a motion detection stimulus is published (Figure 11, line 33).

Pervasive computing systems often interact with people. For instance, our heating control system relies on the detection of motion. To help introducing the behavior of simulated people, we provide a class for defining the movements of the simulated agents during the simulation: `AgendaStimulusProducer`. This class is parameterized by an agenda describing where a simulated agent will be located during the simulation (Figure 11, lines 15 and 16). This agenda allows to define time slots during which the agent is in a specific location. This agenda is defined in XML. Figure 12 presents an extract of the `studentagenda.xml` file used in the `MyAgentModel` class (Figure 11, line 15). A simulated agent can be associated with an `AgendaStimulusProducer` object (see for example Figure 11, lines 20 and 22). Thus, this simulated agent will automatically move during the simulation with respect to this agenda. Though using an agenda to model the human behavior is very limited, we can test a wide range of pervasive computing applications with this basic support. The large-scale simulation of engineering school presented in Section 6 simulates 200 people with this simple support.

```
<?xml version="1.0" encoding="UTF-8"?>
<agenda>
  <item>
    <location>I 112</location>
    <startTime>11/04/2011 10:30:00 GMT</startTime>
    <endTime>11/04/2011 11:50:00 GMT</endTime>
  </item>
  <item>
    <location>Hall</location>
    <startTime>11/04/2011 11:50:00 GMT</startTime>
    <endTime>11/04/2011 12:00:00 GMT</endTime>
  </item>
  [ ... ]
</agenda>
```

Figure 12: Extract of the `studentagenda.xml` XML file.

To summarize the relationships between the classes introduced in this section, Figure 13 presents a class diagram of the implementation of the heat regula-

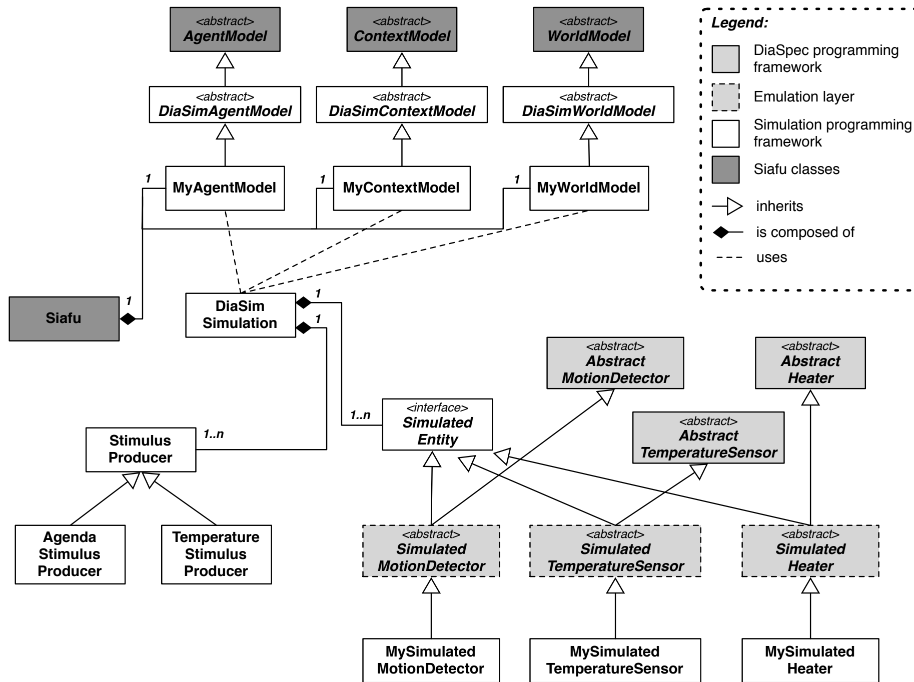

Figure 13: Class diagram of the implementation of the heat regulator simulation.

tor simulation. In this example, every class except `TemperatureStimulusProducer` is provided to the tester either by the generated DiaSpec framework, the simulation programming framework, the generated emulation layer, or Siafu. The tester may modify the simulated entity implementations (*e.g.,* `MySimulatedHeater`) if he needs a more sophisticated behavior as the one provided by default. He may also modify the `MyAgentModel` class if he needs to send simulated stimuli triggered by simulated agents. For instance, sending a simulated detection stimulus when an agent is in the scope of a motion detector would be implemented in the `MyAgentModel` class. It is important to notice that the simulated entity implementations are independent from the stimulus producers. The communication between the stimulus producers and the simulated entities is handled by the `DiaSimSimulation` class. For instance, it is possible to modify the implementation of `TemperatureStimulusProducer` without modifying the `MySimulatedTemperatureSensor` implementation. Thus, this independence between stimulus producers and simulated entities would allow to compute temperature values from a thermal physical model instead of reading from logs of measurements without any impact on the simulated entity implementations.

5 Testing Applications

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

5.1 Transparent simulation

A programming framework generated by the DiaSpec compiler provides applications with an abstraction layer to discover entities. This entity discovery support is based on the taxonomy definition. In particular, it includes methods to select any node in the entity taxonomy. The result of this selection is the set of all entities corresponding to the selected node and its sub-nodes. The developer can further narrow down the entity discovery process by specifying the desired values of the attributes. This situation is illustrated in Figure 5. When a heat regulation is required in a particular location, the `HeatRegulator` controller implementation discovers the heaters located in this location and turns them on. The `discover` parameter is used to achieve this entity discovery. Using the value of `regulation.getRegulation()`, only the heaters in this particular location are discovered.

Because of this abstraction layer, simulation is achieved transparently: the same application code discovers and interacts with entities, whether or not simulated. This transparent simulation applies to all aspects of a pervasive computing application. For another example, simulated data sources can be added to a pervasive computing system, without requiring any change in the application code.

5.2 Simulator architecture

The overall architecture of DiaSim is displayed in Figure 14. 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.

5.2.1 Executing simulation scenarios

An environment simulator generates stimuli as a given simulation scenario unfolds. It consists of stimulus producers and a scenario manager that dispatches stimuli to the relevant entities. The *scenario manager* is a mediator, periodically querying the stimulus producers to feed the data sources of simulated entities. For example, the scenario manager collects stimuli of outdoor luminosity and passes them to outside light sensors.

Actions can create changes to the simulated environment. To do so, entities register new stimulus producers to the scenario manager. 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.

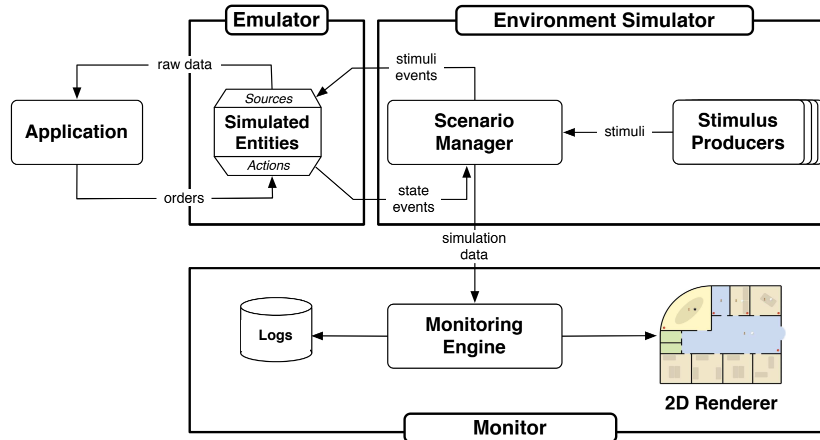

Figure 14: DiaSim architecture

When the application deactivates the fire sprinkler, the water stimulus producer is stopped by the scenario manager.

5.2.2 Monitoring simulation

The scenario manager receives simulation data from stimulus producers and simulated entities to keep track of the simulated environment state. The scenario manager 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.

5.3 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 simulated entities (*e.g.,* alarm ringing, motion detection) 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 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 [22]. A simulation log contains information about interactions between entities (*i.e.,* time, source, destination,

interaction parameters) and between stimuli and entities (*i.e.,* time, source, destination, class of stimuli, stimuli parameters). Replays help to isolate bugs but do 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.

5.4 An integrated approach to simulation

Exploring simulation in the context of DiaSpec enables simulation to coevolve with DiaSpec. Indeed, the wider the scope of DiaSpec, the more simulation aspects can be investigated. Concretely, recent works in our research group has expanded the scope of DiaSpec to cover non-functional concerns, extending the DiaSpec language and its compiler. These extensions include (1) handling access conflicts to resources of a pervasive computing system [23], (2) modeling entity failures at the declaration level, enforcing their treatment at the programming level [24], and (3) declaring performance constraints, ensuring them at compile time and run time [25]. Each of these non-functional concerns introduce opportunities to expand the scope of simulation. In fact, we successfully applied our simulation approach to the avionics domain [26]. Specifically, we developed an aircraft guidance system, using stimulus producers for simulating entity failures. We are planning to extend DiaSim with the simulation of non-functional concerns that are now available in DiaSpec.

6 Implementation and Validation

The components of DiaSim, detailed in Figure 14, are implemented in Java, and consist of 15,000 lines of code.

To validate DiaSim, we simulated various applications in the building management area, such as the heating control system depicted in Figure 3. 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 a telecommunications company. For the sake of conciseness, we do not detail further this application area.

6.1 Applications

The ENSEIRB school is a three-floor building of 13,500 m^2, consisting of several lecture halls, labs and recreation rooms for students. ENSEIRB hosts up to 900 occupants, including students from various countries and faculty members.

[1] http://www.enseirb.fr
[2] http://diasuite.inria.fr/documentations/resources

Using the generated Java programming framework, we have 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.

6.2 Setting up simulation scenarios

To test applications, we simulated three main scenarios: a working day, a weekend day and an open house day. Each scenario is simulated using variations, including entity failures, number and location of entities, and number of users. Scenarios are defined using a Java GUI: the *scenario editor* (Figure 15). From the DiaSpec definitions, simulated entities are either graphically defined using a wizard, or developed using the simulation programming framework generated by the DiaSpec compiler. In the first case, attributes are defined by filling in a form and the location attribute is set by dragging and dropping entity icons in the simulated space. The behavior of the simulated entity is then defined by graphically selecting and parameterizing the appropriate class of behaviors. For example, an `AudioNotification` action provided by loudspeakers is simulated with a text-to-speech library. For another example, the action to turn on alarms is 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 real entities in the simulation, either to ease the scenario definition (*e.g.,* calendars and the profile database) or to validate their behavior (*e.g.,* `NewsNotification` entities 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).

6.3 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 [19]. Our simulator interfaces with Siafu to use its rendering

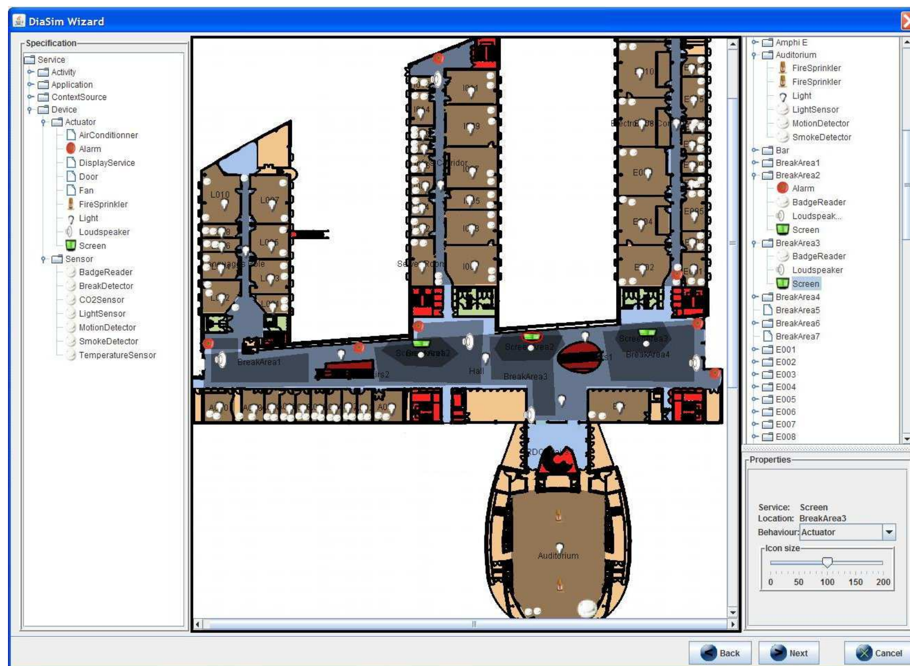

Figure 15: DiaSim scenario editor. The DiaSim editor is parameterized by an entity taxonomy. The entities defined in the taxonomy are displayed on the left panel of the graphical user interface. The entities can be dragged and dropped on the central panel to add simulated entity instances into the simulated environment.

and time-control functionalities. On top of a picture of the simulated space, the simulation renderer displays entities and stimuli, as shown in Figure 16.

The simulation renderer shows the state of the simulated entities, by displaying a bubble of raw text above entities (*e.g.,* when a data source publishes events) and/or modifying the visual representation of the entity (*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. Real entities 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 real `NewsNotification` entities. We also used enriched views for simulated entities, *e.g.,* loudspeakers are rendered using audio streams.

DiaSpec supports several back-ends including Java RMI, SIP [27] and Web Services. A back-end defines the communication protocol used by the DiaSpec components 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 real entities and to distribute the workload over several

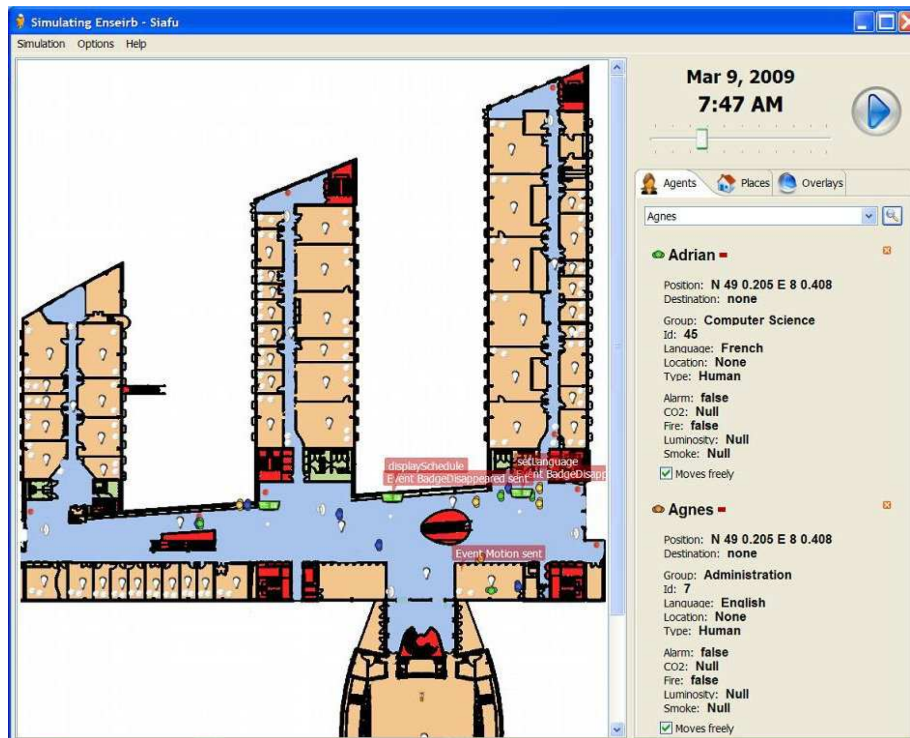

Figure 16: DiaSim simulation renderer. The simulated environment is displayed in the left part of the graphical user interface. The red popups transparently displayed above the simulated entities indicate that the entity has realized an interaction. More information about the simulated people and simulated entities can be found on the right of the graphical user interface.

different hosts when numerous simulated entities are in play.

6.4 Evaluation

We now conduct an evaluation of DiaSim. To do so, we explore three aspects. We first discuss the *scalability* of our simulation tool. We then study the *usability* of DiaSim, before evaluating its *performance*.

6.4.1 Scalability.

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

Task	**Completion**		**Avg.**
	full	**part**	**time**
DiaSpec specification	100%	0%	2h
Implementation	60%	40%	5h
DiaSim simulation	30%	0%	1h

Table 1: Results of a lab involving 60 Master's level students.

In the ENSEIRB experiment, the simulation allowed us to validate the coordination logic at a large scale, combining 110 entities, 6 stimulus producers, 200 people and 6 applications. Some entities were coordinated and shared by several applications (*e.g.,* `Calendar` and `MotionDetector`). 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 entities 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.

6.4.2 Usability.

We have been using DiaSim as part of a course on software architectures for three years. This course includes an 8-hour lab consisting of twenty groups of three master's level students. These students have only followed an introductory course on Java before our course and have basic knowledge of software design and no exposure to the domain of pervasive computing. The goal of our lab is to develop a Newscast application as defined previously. It requires devices to broadcast messages (*e.g.,* loudspeakers and screens), and devices to identify users (*e.g.*, RFID badge readers). The students have to (1) design the application with DiaSpec, (2) implement it, and (3) simulate it with DiaSim.

The results of this lab are presented in Table 1. Due to the short duration of the lab, last year, only 30% of the students completed their implementation and had enough time to simulate it with DiaSim. The students had to instantiate simulated screens, simulated loudspeakers and simulated badge readers using the DiaSim editor. They also had to create several simulated people to the simulation. Then they had to create a stimulus producer that sends simulated badge detection stimuli when a simulated agent is getting close to a badge reader. We provided them with an online tutorial to help them create their simulation[3]. It is interesting to notice that these students only required on average 1 hour to simulate their application using DiaSim. Though the simulation was simple, it allowed us to get feedback on the usability of DiaSim. In particular, during this lab, simulated people were added programmatically to the simulation. The creation of the simulated people was complicated for the students. Because of that feedback, we modified the DiaSim graphical editor to

[3] `http://diasuite.inria.fr/documentations/tutorial/`

allow simulated agents to be added graphically to the simulated environment.

This experience demonstrates that students with modest knowledge in software engineeing are able to efficiently use DiaSim in a short period of time. However, because the lab is short, the students were only able to achieve a simple simulation. It would be interesting to do another lab focusing especially on the simulation. This would allow to request a more complicated simulation to the students, giving us a more thorough evaluation of the usability of DiaSim.

6.4.3 Performance.

To study the overhead caused by DiaSim, we evaluated its performance during the engineering school simulation. Our goal is to collect measurements when DiaSim is applied to two different simulation workloads: low activity and high activity. The simulation has been executed on a laptop with a CPU Intel Core 2 Duo 2.80 GHz and with 4 GB of RAM. The operating system used by the laptop was Windows XP. The measurements were realized with the JProfiler software.

CPU usage. We first evaluated the CPU usage during the simulation. The results of this evaluation are presented in Figure 17. The CPU usage has first been evaluated when the activity is low during the simulation. A low simulation activity is typically during the night or when students are sitting in the classrooms. Then, we evaluated the CPU usage during high activity periods. There is a high activity during the breaks when students are moving in the school. The CPU usage was evaluated with respect to the simulation speed, which ranges from twice as fast as the real time (simulation speed number 1 in Figure 17) to 360 times as fast as the real time (simulation speed number 11 in Figure 17).

This evaluation shows that simulating at a low speed uses less than 20% of CPU. We can also see that simulating at a higher speed requires more CPU. This is due to the graphical rendering that requires to update more often its rendering. To use less CPU, it is possible to disable the graphical rendering and only log the simulation data. It prevents the tester from monitoring the simulation graphically, but it allows him to execute the simulation at a much higher speed while using less CPU.

Memory usage. To run the simulation, we allocated 1.4 GB of maximum memory to the JVM. The memory is fully used during the simulation. This memory is mainly used by Siafu (approximately 1.2 GB to store information concerning the motion of the simulated agents. It uses this information to quickly compute a path to graphically move a simulated agent from one point to another. Thus, simulating fewer people enables to use less memory.

Thread distribution. We studied the threads used during the simulation. In particular, we studied the thread distribution between DiaSpec and DiaSim. Overall, the vast majority of the threads are related to the DiaSpec runtime,

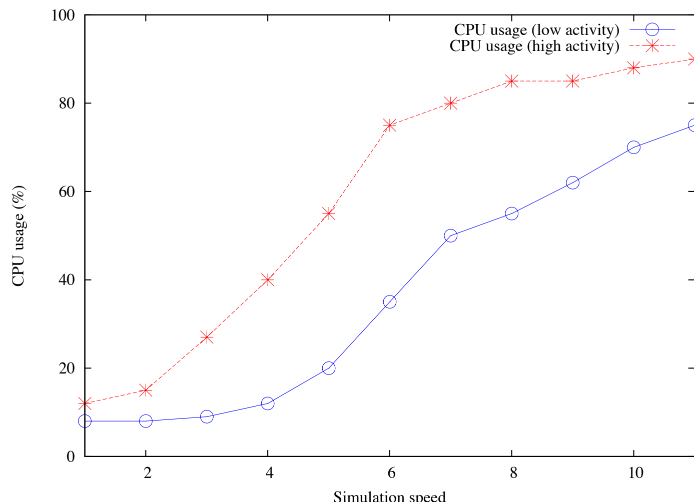

Figure 17: This graph represents the average CPU usage with respect to the simulation speed. The CPU usage has been evaluated during a period of low activity for the simulated agents and during a period of high activity.

	Nb. of threads	Percentage
DiaSpec	244	96.06%
DiaSim	10	3.94%

Table 2: Distribution of the threads executed during the ENSEIRB simulation.

DiaSim accounts for less than 4%. The results of this study are presented in Table 2.

To conclude, it is important to notice that this simulation has been executed on a three-year old laptop, not very powerful compared to today's computers. We would get much better performance results if the simulation were run on a recent computer. Nevertheless, it is possible to execute a large-scale simulation comprising 200 simulated people and 110 simulated entity instances with a modest computer.

6.5 Discussion

We now examine pragmatic issues involved in developing and using a simulated environment. We start by discussing the potential pitfalls of our approach. Then, we investigate the performance issues involved in running large-scale simulations. Finally, we discuss how our simulation approach can be generalized to other application domains.

6.5.1 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 real 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 entity data sources, 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 entity actions 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 entity actions 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 light sensors. This merging operation is also user-defined; to be meaningful its definition needs to rely on domain-specific knowledge.

As one can see, taking into account the simulation requirements and developing stimulus producers in Java can be laborious. It often requires to encode in Java mathematical formulas describing the stimulus producer evolution. To reduce this complexity, we are actively working on easing simulation of natural phenomena [28]. To do so, we leverage Acumen [29], a Domain-Specific Language (DSL) for describing differential equations. The differential equations defined with Acumen describe physical phenomena. With Acumen, we use off-the-shelf physical environment models and formulas that are available in textbooks and the research literature. Their correctness is extensively documented and well established. Leveraging a physical environment modeling language such as Acumen allows us to both reduce the stimulus producer implementation complexity and ensure the correctness of our stimulus producers.

Entities are emulated so that applications interact with them without code modification. To be faithful, an emulated entity should have an observable behavior that is equivalent to its real counterpart. To do so, the data source of an emulated entity can be programmed such that, for a given input, it produces the same output as its real counterpart.

6.5.2 Performance

The simulation of physical spaces may involve lots of entities, accurate simulation models, and rich simulation logic. This situation calls for a scalable simulator.

To support compute-intensive simulation, DiaSpec enables to distribute simulated entities and stimulus producers. This distribution is naturally achieved using DiaSpec because DiaSpec components communicate via a distributed systems technology. Our implementation of DiaSpec supports several distributed systems technologies including a local software bus, Java RMI, SIP and Web Services. The selection of this distributed technology is done at deployment time and does not affect DiaSpec component implementation. When the simulation back-end used by DiaSim is Java RMI, the workload can be distributed over several different hosts, enabling numerous entities and stimulus producers to be introduced. A distributed technology also makes it possible to perform hybrid simulation by integrating distributed, real entities.

6.5.3 Generalization to other application domains

In DiaSim, the graphical rendering and the stimulus producers are specific to the pervasive computing domain. However, our simulation approach can be generalized to other application domains. In a recent work, we applied our simulation approach to the avionics domains [26]. We developed a flight guidance system using the DiaSpec approach and simulated it with our simulation approach. To address the avionics domain, we replaced Siafu by the FlightGear flight simulator for graphically rendering the simulation. We have also interfaced simulated entities with the aircraft physical model provided by FlightGear, enabling it to produce the simulated stimuli. This strategy allows our simulated entities to receive realistic stimuli from the physical model defined by FlightGear (*e.g.,* current heading or altitude) and act on the realistically-simulated aircraft components (*e.g.,* the ailerons to control the aircraft heading).

Applying our simulation approach to avionics has demonstrated that our key concepts are general enough to tackle a drastically different domain. Futhermore, this work has shown that the architecture of DiaSim can easily accommodate another rendering layer and be composed of arbitrarily complex physical models.

7 Related Work

In this section, we study other existing pervasive computing simulators. We also study simulators from two simulation fields related to the pervasive computing domain: context simulators and networked entities simulators.

Pervasive Computing Simulators

Few simulators are dedicated to the testing of pervasive computing applications [8, 9, 10, 11, 12]. Ubiwise [8] and Tatus [9] are built upon 3D game graphics engines, respectively Quake III Arena and Half-Life. By providing a first person view of the simulated pervasive computing 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.

The Lancaster simulator enables deterministic testing conditions and emulation to test location-based applications [11]. However, libraries of actuators and sensors are not provided and the development of new types of sensors and actuators is not supported. The PiCSE simulator addresses the problem of extensibility by providing generic libraries to create sensors and actuators [12]. However all these approaches do not propose an emulation framework to incrementally integrate real entities in a simulated system.

Contrary to the other existing approaches, UbiREAL [10] provides an emulation framework that allows to combine simulated and real entities. It also provides a 3D graphical renderer to simulate pervasive computing applications. However users have to manually specialize the simulator for every new application area. In contrast, DiaSim relies on DiaSpec to automatically customize the simulation tools (*i.e.,* the editor and renderer).

Context Simulators

Some simulators focus on the simulation of context [30, 31, 19]. The Generic Location Event Simulator publishes location information, which can be used by location-based applications [30]. However, it is limited to location information. SimuContext [31] and Siafu [19] 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 entities and applications.

Networked Entities Simulators

Various approaches propose to simulate sensor networks [32, 33, 34, 35] 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.

MATLAB/Simulink [36] and Ptolemy [7] are used for modeling networked entities. These modeling tools provide libraries of computation models that the user can compose for modeling and simulating devices. The use of these tools is also complementary to our approach in that they allow simulated entities to be modeled instead of being programmed. Network emulators that focus on network-related issues have been proposed [37, 38] and could also complement our approach.

8 Conclusion and Future Work

In this paper, we have presented a novel approach to simulating pervasive computing applications. This approach has been implemented in a simulator named DiaSim. DiaSim is parameterized with respect to a description of a pervasive computing environment, relevant to a given area. This description is defined using the DiaSpec language and is processed by the DiaSpec compiler to produce a dedicated programming framework.

We have extended this approach by generating a simulation programming framework and an emulation layer. The generated emulation layer makes it possible for the same application to be emulated or executed in a real environment. This emulation layer also enables to have an application interact with both real and simulated entities in an hybrid environment. Hybrid simulation allows real entities to be incrementally added in the simulation, as the implementation and deployment progress. The generated simulation programming framework provides support for developing the simulation logic. This logic comprises the simulated entities and the producers of simulated stimuli. A 2D graphical environment is provided to the user to define his simulated environment, simulation scenarios, and to monitor and debug a simulated pervasive computing system. This approach has been implemented in the DiaSim tool, and validated on a large-scale simulation of an engineering school. Finally, we have evaluated DiaSim with respect to its scalability, usability and performance.

Ongoing and future work

This work is being expanded in various directions.

Easing the physical environment simulation

Simulating natural phenomena like heat propagation can be quite complex as they involve mathematical equations. We are working on easing the simulation of these phenomena by leveraging Acumen [29], a DSL for describing and executing differential equations. The differential equations representing the physical phenomena are defined with Acumen.

Adding human behavior modeling

Numerous pervasive computing applications surround people in their life. To help simulating this kind of applications, we plan on connecting human behavior models to DiaSim. For instance, human behaviors could be implemented in a multi-agent simulation toolkit such as MASON [39]. Each agent would represent a simulated person. Coupling DiaSim with human behavior would allow the simulation of a fire alarm system in a building, as well as the behavior of the building occupants for example. The simulation of the building occupants would enable to observe their behaviors in case of fire.

Applying DiaSim to other application areas

We have been applying our approach to the building and home automation areas. However, DiaSim can address numerous other application areas, such as agriculture, industry, and traffic flow. To demonstrate the usability of our approach, we are working on simulating applications in each of these application areas.

Enhancing the system monitoring

A pervasive computing system may involve a large number of entities and applications. Monitoring such a system rapidly becomes excessively complicated. In particular, large-scale simulations in which numerous events occur at the same time are hard to monitor, even with a graphical rendering and logs. To enhance monitoring, we would like to add contracts to DiaSpec in the form of pre- and post-conditions to entities, controllers, and contexts. These contracts would drive the rendering of a simulation by drawing the tester's attention when they are violated.

Enhancing the graphical renderer

The 2D graphical rendering provided by DiaSim allows to easily monitor the simulated applications. However, the user experience of these simulated applications would be improved with a 3D graphical rendering. Indeed, users would be able to test applications immersed in a simulated 3D physical environment. We plan to render simulations in 3D using Blender [40], an authoring tool for creating 3D animations and video games.

References

[1] Dey AK, Abowd GD, Salber D. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction* 2001; **16**(2):97–166.

[2] Román M, Hess C, Cerqueira R, Ranganathan A, Campbell R, Nahrstedt K. A Middleware Infrastructure for Active Spaces. *Pervasive Computing, IEEE* 2002; **1**(4):74–83.

[3] Ranganathan A, Chetan S, Al-Muhtadi J, Campbell RH, Mickunas MD. Olympus: A High-Level Programming Model for Pervasive Computing Environments. *PERCOM'05: Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications*, 2005; 7–16.

[4] Grimm R. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing* 2004; **3**(3):22–30.

[5] iOS SDK, `http://developer.apple.com/technologies/iphone`.

[6] Android SDK, `http://developer.android.com`.

[7] Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y. Taming heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE* 2003; **91**(1):127–144.

[8] Barton JJ, Vijayaraghavan V. UBIWISE, A Ubiquitous Wireless Infrastructure Simulation Environment. *Technical Report*, Hewlett Packard 2002.

[9] O'Neill E, Klepal M, Lewis D, O'Donnell T, O'Sullivan D, Pesch D. A Testbed for Evaluating Human Interaction with Ubiquitous Computing Environments. *TRIDENTCOM '05: Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2005; 60–69.

[10] Nishikawa H, Yamamoto S, Tamai M, Nishigaki K, Kitani T, Shibata N, Yasumoto K, Ito M. UbiREAL: Realistic Smartspace Simulator for Systematic Testing. *UBICOMP'06: Proceedings of the 8th International Conference on Ubiquitous Computing*, 2006; 459–476.

[11] Morla R, Davies N. Evaluating a Location-Based Application: A Hybrid Test and Simulation Environment. *IEEE Pervasive Computing* 2004; **3**(3):48–56.

[12] Reynolds V, Cahill V, Senart A. Requirements for an Ubiquitous Computing Simulation and Emulation Environment. *InterSense'06: Proceedings of the 1st International Conference on Integrated Internet Ad hoc and Sensor Networks*, 2006.

[13] Bruneau J, Jouve W, Consel C. DiaSim, A Parameterized Simulator for Pervasive Computing Applications. *Mobiquitous'09: Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2009.

[14] Jouve W, Bruneau J, Consel C. DiaSim: A Parameterized Simulator for Pervasive Computing Applications (Demo). *PERCOM'09: Proceedings of the 7th IEEE International Conference on Pervasive Computing and Communications*, 2009.

[15] Cassou D, Bertran B, Loriant N, Consel C. A Generative Programming Approach to Developing Pervasive Computing Systems. *GPCE'09: Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, 2009; 137–146.

[16] Cassou D, Bruneau J, Consel C, Balland E. Towards a Tool-based Development Methodology for Pervasive Computing Applications. *IEEE Transactions on Software Engineering* 2011; .

[17] Chen G, Kotz D. Context Aggregation and Dissemination in Ubiquitous Computing Systems. *WMCSA'02: Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications* 2002; :105.

[18] Cassou D, Balland E, Consel C, Lawall J. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. *ICSE'11: Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[19] Martin M, Nurmi P. A Generic Large Scale Simulator for Ubiquitous Computing. *Mobiquitous'06: Proceedings of the 3rd International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2006.

[20] Kuehn T. *Fundamentals: 2005 ASHRAE Handbook*. American Society of Heating, 2005.

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

[22] Wireshark: A Network Protocol Analyzer, `http://www.wireshark.org`.

[23] Jakob H, Consel C, Loriant N. Architecturing Conflict Handling of Pervasive Computing Resources. *DAIS'11: 11th IFIP International Conference on Distributed Applications and Interoperable Systems*, 2011.

[24] Mercadal J, Enard Q, Consel C, Loriant N. A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. *OOPSLA'10: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[25] Gatti S, Balland E, Consel C. A Step-wise Approach for Integrating QoS throughout Software Development. *FASE'11: Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering*, 2011.

[26] Bruneau J, Enard Q, Gatti S, Balland E, Consel C. Design-driven Development of Safety-critical Applications: A Case Study In Avionics. *Technical Report*, Phoenix Research Group, INRIA Bordeaux, France 2011.

[27] Jouve W, Palix N, Consel C, Kadionik P. A SIP-based Programming Framework for Advanced Telephony Applications. *IPTComm'08: Proceedings of the 2nd LNCS Conference on Principles, Systems and Applications of IP Telecommunications*, 2008.

[28] Bruneau J, Consel C, O'Malley M, Taha W, Hannourah WM. Preliminary Results in Virtual Testing for Smart Buildings (Poster). *Mobiquitous'10, Proceedings of the 7th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2010.

[29] Zhu Y, Westbrook E, Inoue J, Chapoutot A, Salama C, Peralta M, Martin T, Taha W, O'Malley M, Cartwright R, *et al.*. Mathematical Equations as Executable Models of Mechnical Systems. *ICCPS'10: Proceedings of the 1st International Conference on Cyber-Physical Systems*, 2010.

[30] Sanmugalingam K, Coulouris G. A Generic Location Event Simulator. *UBICOMP'02: Proceedings of the 4th International Conference on Ubiquitous Computing*, 2002; 308–315.

[31] Broens T, van Halteren A. SimuContext: Simply Simulate Context. *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, 2006; 45.

[32] Sundresh S, Kim W, Agha G. SENS: A Sensor, Environment and Network Simulator. *Proceedings of the 37th Annual Simulation Symposium*, 2004; 221–228.

[33] Levis P, Lee N, Welsh M, Culler D. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. *SenSys '03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, 2003; 126–137.

[34] Titzer BL, Lee DK, Palsberg J. Avrora: Scalable Sensor Network Simulation With Precise Timing. *IPSN'05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, 2005; 477–482.

[35] Polley J, Blazakis D, McGee J, Rusk D, Baras JS, , Karir M. ATEMU: A Fine-Grained Sensor Network Simulator. *SECON'04: Proceedings of the 1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[36] Riederer P. MATLAB/Simulink for Building and HVAC Simulation - State of the Art. `http://www.docjax.com/ajax/view.shtml?id=1290132`.

[37] NS-2 Network Simulator, `http://www.isi.edu/nsnam/ns/`.

[38] D'Aprano F, de Leoni M, Mecella M. Emulating Mobile Ad-hoc Networks of Hand-Held Devices: the OCTOPUS Virtual Environment. *MobiEval'07: Proceedings of the 1st International Workshop on System Evaluation for Mobile Platforms*, 2007; 35–40.

[39] Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G. MASON: A Multiagent Simulation Environment. *Simulation* 2005; **81**(7):517.

[40] Blender, `http://www.blender.org`.