

A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications

Zoé Drey¹, Julien Mercadal², and Charles Consel²

¹ Thales / LaBRI-Université de Bordeaux, France

² INRIA / LaBRI-Université de Bordeaux, France
{drey,mercadal,consel}@labri.fr

Abstract. Various forms of pervasive computing environments are being deployed in an increasing number of areas including healthcare, home automation, and military. This evolution makes the development of pervasive computing applications challenging because it requires to manage a range of heterogeneous entities with a wide variety of functionalities. This paper presents Pantagruel, an approach to integrating a taxonomical description of a pervasive computing environment into a visual programming language. A taxonomy describes the relevant entities of a given pervasive computing area and serves as a parameter to a sensor-controller-actuator development paradigm. The orchestration of area-specific entities is supported by high-level constructs, customized with respect to taxonomical information.

We have implemented a visual environment to develop taxonomies and orchestration rules. Furthermore, we have developed a compiler for Pantagruel and successfully used it for applications in various pervasive computing areas, such as home automation and building management.

Key words: Visual Rule-Based Language, Pervasive Computing

1 Introduction

Various forms of pervasive computing environments are being deployed in an increasing number of areas including healthcare [1], home automation [2], building management [3] and military [4]. This trend is fueled by a constant flow of innovations in devices forming ever richer pervasive computing environments. These devices have increasingly more computing power offering high-level interfaces to access rich functionalities. Access to their functionalities can be done remotely because most devices are now networked.

The advent of this new generation of devices enables the development of pervasive computing systems to abstract over low-level embedded systems intricacies. This development is now mainly concerned with the programming of the orchestration of the entities, whether hardware (sensors and actuators) or software (*e.g.*, databases and calendars). Yet, the nature of pervasive computing systems makes this programming very challenging. Indeed, orchestrating networked heterogeneous entities requires expertise in a number of areas, including

distributed systems, networking, and multimedia. There exist middlewares and programming frameworks that are aimed to facilitate this task; examples include Gaia [5], Olympus [6], One.World [7], and Aura [8]. However, these approaches do not fill the semantic gap between an orchestration logic and its implementation because they rely on a general-purpose language and use large APIs. This situation makes the programming of the orchestration logic costly and error-prone, impeding evolutions to match user's requirements and preferences.

To circumvent this problem, visual approaches to programming the orchestration logic have been proposed, based on storyboards and rules [9–11]. These approaches enable the programmers to express the orchestration logic using intuitive abstractions, facilitating the development process. However, this improvement is obtained at the expense of expressivity. Specifically, existing visual approaches are limited to a given area (*e.g.*, CAMP magnetic poetry for the Smart Home domain [10]), a pre-defined set of categories of entities (*e.g.*, in iCap rule-based interactive tool [11]), or abstractions that do not scale up to rich orchestration logic (*e.g.*, Oscar service composition interface [12]).

This paper

This paper presents Pantagruel, an expressive approach to developing orchestration logic that is parameterized with respect to a *taxonomy* of entities describing a pervasive computing environment. Specifically, our approach consists of a two-step process: (1) a pervasive computing environment is described in terms of its constituent entities, their functionalities and their properties; (2) the development of a pervasive computing application is driven by a taxonomy of entities and consists of orchestrating them using high-level constructs.

The environment description allows our approach to be instantiated with respect to a given application area. This description defines the classes of entities that are relevant to the target area. For each class, it specifies an interface to access its functionalities. Because the orchestration logic is written with respect to the environment description, entities are combined in compliance with their description. To facilitate the programming of the orchestration logic, we have developed a visual tool that uses a sensor-controller-actuator paradigm. This paradigm is suitable for programmers with standard skills, or even for novice programmers, as demonstrated by its use in various fields such as computer games (*e.g.*, Blender³) and robots (*e.g.*, Altaira [13] or LegoSheets [14] for Lego Mindstorms⁴). Like a game logic, an orchestration logic collects context data from sensors, combines them with a controller, and reacts by triggering actuators. This paradigm is intuitive and makes programs readable. Furthermore, our visual programming environment offers the developer an interface that is customized with respect to the environment description. Information about the environment entities is exploited to guide the programmer in defining sensor-controller-actuator rules.

³ <http://www.blender.org>

⁴ <http://mindstorms.lego.com/>

Because Pantagruel is a very high-level language, its compilation could be challenging, leading to intricate and error-prone processings. To alleviate this problem, we have developed a compiler for Pantagruel that leverages an architecture description language (ADL) dedicated to distributed systems, named DiaSpec [15]. Given an architecture description, the compiler for this ADL, named DiaGen, generates a dedicated programming framework in Java, which provides extensive support to discover and interact with distributed entities. The compilation process of Pantagruel consists of two stages: (1) an environment description is translated into a DiaSpec description (2) orchestration rules are compiled into Java code, supported by a DiaGen-generated programming framework. Leveraging on DiaSpec enables Pantagruel to gain access to a large number of heterogeneous entities, available in our Lab's smart space. They include hardware entities (*e.g.*, IP phones, webcams, displays, and X10 controllers) and software entities (*e.g.*, calendars, presence agents and instant messaging clients).

The contributions of this paper are as follows.

- *Area-specific approach.* We introduce a novel approach to visual programming of pervasive computing applications that is parameterized with respect to a description of a pervasive computing environment. This makes our approach applicable to a range of areas.
- *Area-specific visual programming.* We extend the sensor-controller-actuator paradigm to allow the programming of the orchestration logic to be driven by an environment description. This extended approach eases programming and enables verifications.
- *Preliminary validation.* We have implemented a compiler and successfully used it for applications in various pervasive computing areas such as home automation and building management.

Outline

To motivate our approach, Section 2 presents an example of a meeting application. Section 3 then introduces our taxonomical approach to defining descriptions of pervasive computing environments. Section 4 presents a visual environment to develop applications that orchestrate entities, defined in a taxonomy. The implementation of Pantagruel is examined in Section 5 and a study of its use is presented in Section 6. The related work is detailed in Section 7. Concluding remarks and future work are provided in Section 8.

2 Working Example

To motivate our approach, we consider as an example a meeting management application. This application area orchestrates various kinds of entities, consisting of RFID readers, presence agents, an LCD display, laptops, IP phones, a shared agenda, and an instant messaging server. An example of a physical layout for an office space is displayed in Figure 1.

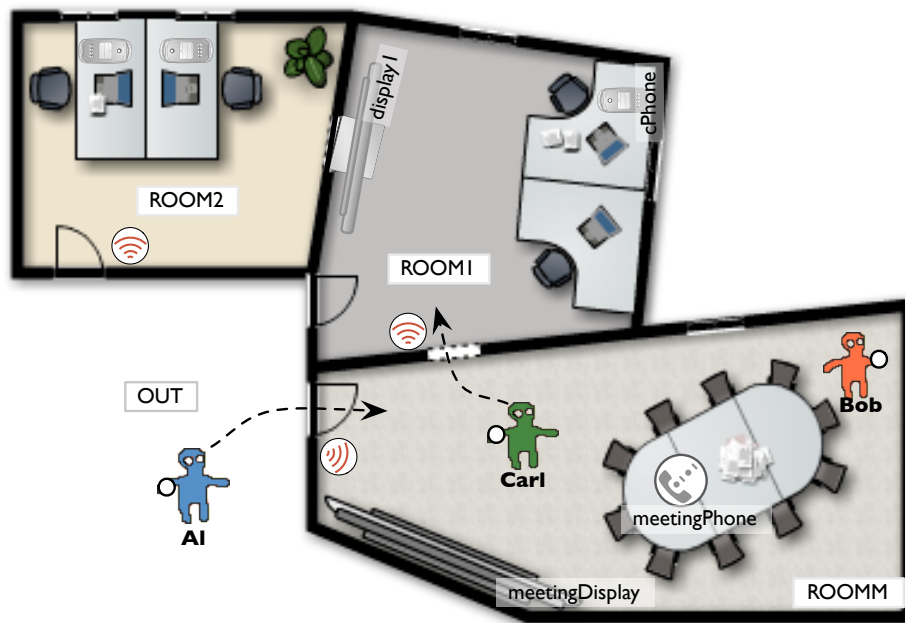


Fig. 1. The physical layout of an office space

A RFID reader is placed in every room (ROOM1, ROOM2, and ROOMM) to detect the location of users (AI, Bob and Carl) wearing an RFID tag. A shared agenda stores each meeting with additional information such as the date, the starting and ending times, and the participants.

Numerous orchestration scenarios can be defined to manage meetings. Let us examine an example that focuses on starting meetings on time with all participants. For simplicity, we suppose that meetings occur in the same room, say the meeting room. Specifically, if a participant is not present in the meeting room but connected via instant messaging, (s)he receives a reminder message shortly before the meeting time. To make the meeting room available on time for the next scheduled meeting, a message is sent to the LCD display indicating the ending time of the current meeting and the list of participants that need to leave before the next meeting starts. When a new meeting starts, the organizer's laptop switches over to the LCD display of the meeting room, making it possible to share documents and presentations with the participants. Participants attending the meeting from a remote location but connected by instant messaging are invited to start an audio session using their IP phone. Simultaneously, the meeting's presentations are displayed on their laptop.

Many other scenarios for meeting management could be imagined. More scenarios combining meeting management with related activities could be introduced. In fact, for a given environment, it should be possible to define various orchestration scenarios, adapting to users' requirements and preferences, and re-

acting to users' feedback. Because these scenarios can have a tremendous impact on people's life, it is critical to ease the creation and improvement of orchestration logic. In doing so, orchestration logic becomes understandable to the widest audience and close to the users' informal specification.

Also, our example application area illustrates the richness of the entities that are commonly available today, requiring expressivity to combine them. Finally, the office space environment consists of entities for which numerous variations are available, requiring an approach to defining the orchestration logic that abstracts over these differences.

3 Defining a Pervasive Computing Environment

To abstract over the variations of entities, we introduce a declarative approach to defining a taxonomy of entities relevant to a given area. The entity declarations form an environment description that can then be instantiated for a particular setting.

3.1 Environment description

An environment description consists of declarations of entity classes, each of which characterizes a collection of entities that share common functionalities. The declaration of an entity class lists how to interact with entities belonging to this class. The generality of these declarations makes it possible to abstract over a range of variations, enabling the re-use of an environment description.

Furthermore, the declaration of an entity class consists of attributes defining a context and methods accessing the entity functionalities. Entity classes are organized hierarchically, allowing attributes and methods to be inherited. Figure 2 displays a hierarchy of entity classes for the meeting management area in an office space. Starting at the root, this hierarchy breaks down the entity classes into increasingly specific elements. Each successive element adds new attributes and methods.

Entity context. Context awareness is a critical issue in programming a pervasive computing application. If addressed with inappropriate abstractions, it may bloat the application code with conditionals and data structure operations. In our approach, an entity class defines attributes, each of which holds some element of the context of a pervasive computing system. A context element may either be constant, external or applicative. A context element is tested in the sensor part of an orchestration rule. Let us examine each kind of context element.

A *constant* context element is an attribute whose value does not change over time. For example, the `FixedDevice` entity class declares a `room` attribute which is constant for a given setting. As such, an instance of a `Camera` entity class is assumed to have a constant location.

An external context element is aimed to acquire context information from the outside. This kind of context element may correspond to sensors (*e.g.*, a

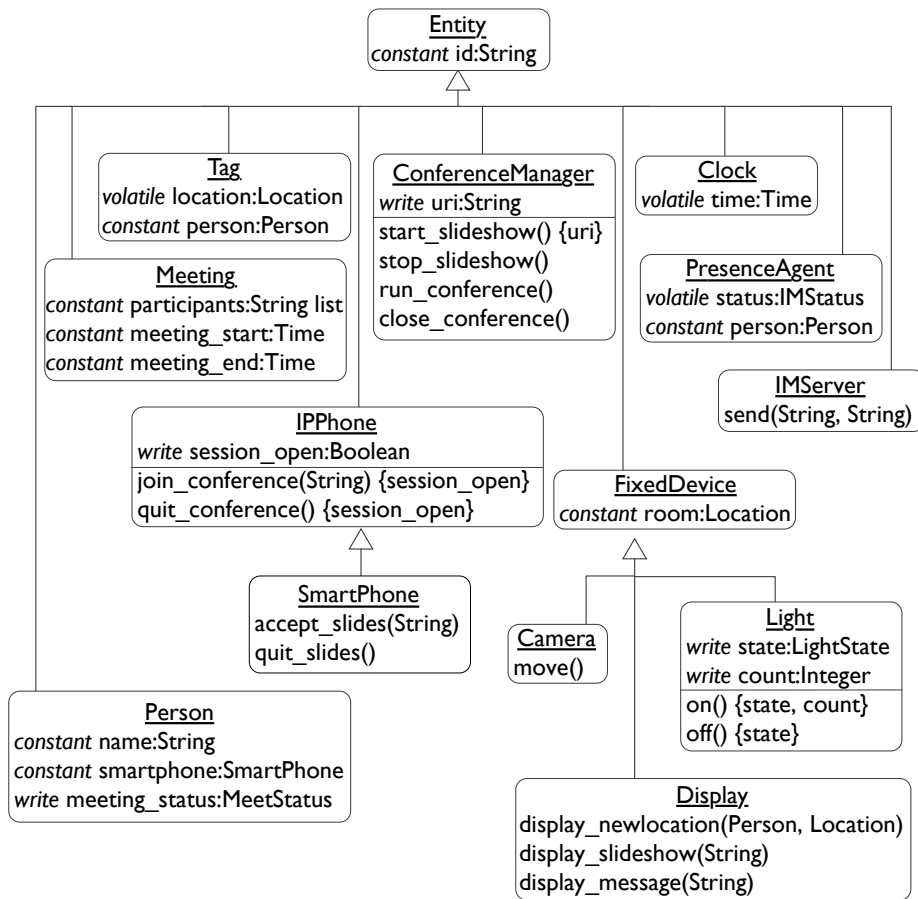


Fig. 2. A hierarchy of classes of entities for our working example

device reporting RFID tag location) or software components (*e.g.*, a calendar reporting meeting events). To model an external context element that varies over time, we introduce the notion of *volatile* attribute. To communicate context information to a Pantagruel program, an external entity updates the value of this attribute. An updated value may then trigger an orchestration rule. As an example of volatile attribute, consider the **Tag** class entity in Figure 2. Each instance of **Tag** holds the **Location** of its owner. This location is updated as the owner moves around and is detected by RFID readers. In our example, the location is an enumeration type over the rooms of the office space. As can be noticed, this information is higher level than the raw information captured by an RFID reader. This level of abstraction requires an expert to wrap the RFID reader such that when a tag is detected, it retrieves its owner and updates the

corresponding Pantagruel object. This wrapper is very similar to a device driver, keeping device intricacies separate from entity orchestration.

Lastly, an applicative context element corresponds to context data computed by the application. To address applicative context elements, we introduce *write* attributes. These attributes can be written in the actuator part of an orchestration rule. In our example, Figure 2 shows the `meeting_status` attribute that can either be `PRESENT`, `REMOTE` or `ABSENT`, depending on the status of the meeting participant. This attribute is updated depending on the participant's tag location and presence status.

By raising context elements to abstractions in the Pantagruel language, we facilitate their manipulation, improve readability, and enable program verification.

Methods. The functionalities of an entity class correspond to method interfaces. They are typed to further enable verification. Methods are introduced to perform actions that are out of the scope of Pantagruel. For example, Pantagruel is not suited to program the sequence of operations needed to rotate a camera. As a consequence, the `Camera` entity class includes a `move` method signature to access this functionality. This method is invoked in the actuator part of an orchestration rule.

When a method is invoked, it may modify the applicative context. Consider the `Light` entity class. Instances of this class have a limited life expectancy that depends on the number of times they are switched on and off. To allow the programmer to design rules that would control the use of lights, we modeled `Light` with two attributes: `state`, holding the light status (`ON` or `OFF`), and `count`, counting the number of times the light is activated. Additionally, the `Light` entity class includes the `on()` and `off()` method signatures. Turning on/off a light affects the `Light` attributes and thus the applicative context. When this method is invoked in an actuator, the Pantagruel developer needs to be aware of these side-effects to define the orchestration logic. To do so, we require side-effecting methods to declare the list of attributes they may alter. For example, the `on()` method declaration includes the list of attributes that are updated by this method, namely `{state, count}`.

3.2 Instantiating an environment description

Once the environment description is completed, it is used to define concrete environments by instantiating entity classes. Figure 3 gives an excerpt of the concrete entities used in our meeting management example. Each entity has a name and refers to its entity class. For instance, we created the `meetingPhone` entity of the `SmartPhone` class; it is noted `meetingPhone:SmartPhone`. This entity corresponds to the phone that is located in the meeting room. Because it plays a specific role in our meeting management scenario, it needs to be created at this stage to allow the programmer to use it in the orchestration logic.

Figure 3 also includes examples of meeting events registered in the shared agenda; they are instances of the `Meeting` entity class. As such, their attributes

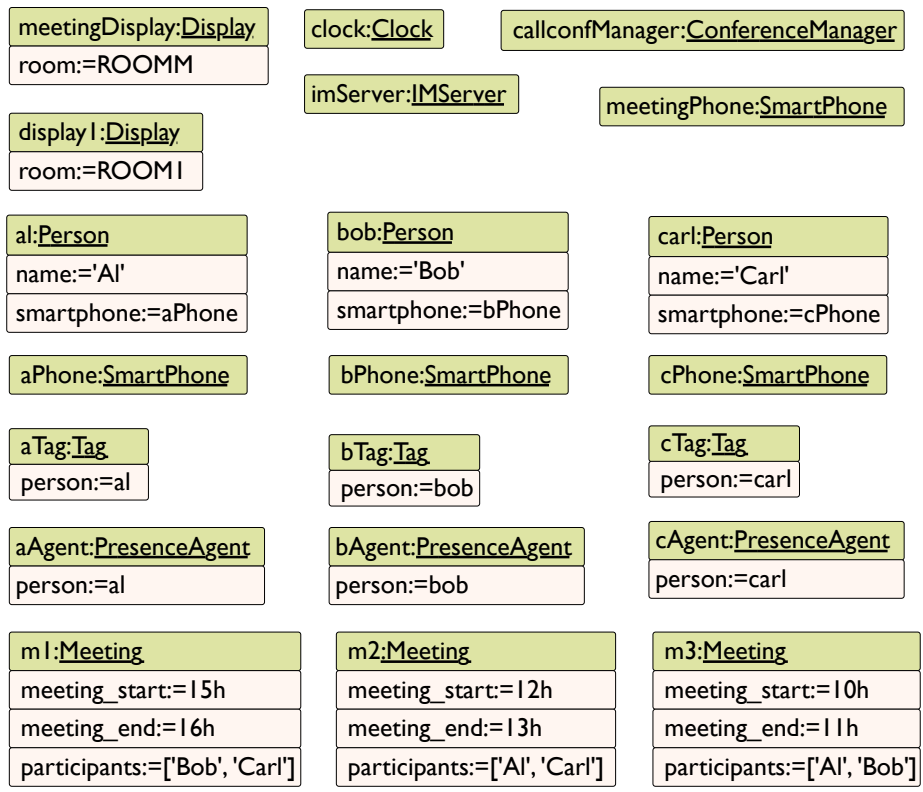


Fig. 3. An excerpt of a concrete environment

are constant. Entity instances can be created dynamically in a given environment (*e.g.*, meeting events and RFID tags). As discussed in the next section, the orchestration logic can be written so as to apply to all instances of an entity class, or to a specific instance (*e.g.*, `meetingPhone`). Notice that applicative and external context elements are initially undefined.

Pantagruel allows simple datatypes to be defined. An example of an enumeration type is given by `Location` in Figure 3; it ranges over the rooms of our example office space.

4 Defining Orchestration Rules

We now present a visual environment dedicated to the development of orchestration rules. Following our paradigm, the Pantagruel development environment offers a panel divided in three columns: sensors, controllers, and actuators. This panel is shown in Figures 4 and 5. These two figures present the orchestration rules of our working example of meeting manager (Section 2).

To develop an application, the programmer starts by defining some conditions on context elements in the sensor column, combining them in the controller column, and triggering actions in the actuator column. For readability, rules are numbered in the controller column (*e.g.*, `R1`). A key feature of our approach is to drive the development of orchestration rules with respect to an environment description. In doing so, the development environment provides the programmer with contextual menus and one-the-fly verification to guide the definition of rules.

4.1 Sections as constituent parts of a rule

To further structure the definition of orchestration rules, the development panel is divided into horizontal sections, one for each entity instance or entity class involved in the application. This is illustrated by the first two sections of the meeting manager application shown in Figure 4. These sections orchestrate the `PresenceAgent` and `Meeting` entity classes. As well, the last section of Figure 4 is defined for the `meetingPhone` instance of the `SmartPhone` entity class.

Within a section, Pantagruel operations are in the scope of the corresponding entity class. For example, the `meeting.start` attribute, defined in the environment description, can be manipulated within the `Meeting` section. In contrast, the `time` attribute needs to be explicitly accessed via the `clock` entity.

4.2 Referring to entity instances

Because they orchestrate actual entities, orchestration rules need to refer to instances of entity classes deployed in a given environment. Doing so requires the ability to discover entities over which rules are defined. Rules thus need to be defined over and applied to all instances of a given entity class. Each of these instances may require rules to further refine the processing. As well, specific

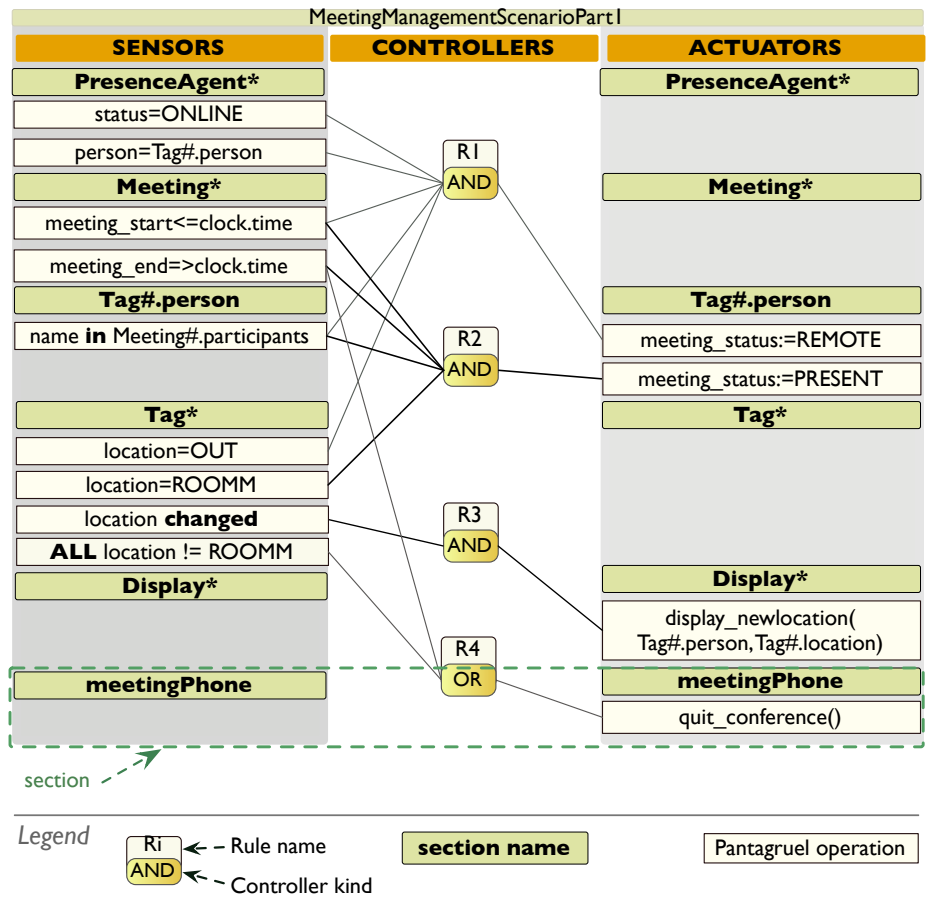


Fig. 4. An example program in the Pantagruel development environment (Part 1)

rules may be required for specific entity instances (*e.g.*, the phone of the meeting room).

To address these issues, we introduce notations to define a section that refers to all entity instances of an entity class. The name of such a section is composed of the entity class name (that starts with an uppercase letter) followed by the ‘*’ symbol. When an orchestration rule includes an element (sensor or actuator) coming from such a section, it is executed over all the instances of the corresponding entity class. For example, Figure 4 includes the **Tag*** section. One of the conditions defined in this section determines whether the tag of a person is located in the meeting room (*i.e.*, **location=ROOMM**), acting as a filter towards collecting the set of meeting participants that are present. As we collect this set, we sometimes need to define rules that manipulate sub-parts of instances being collected. This is typically required to further refine the filtering of instances and

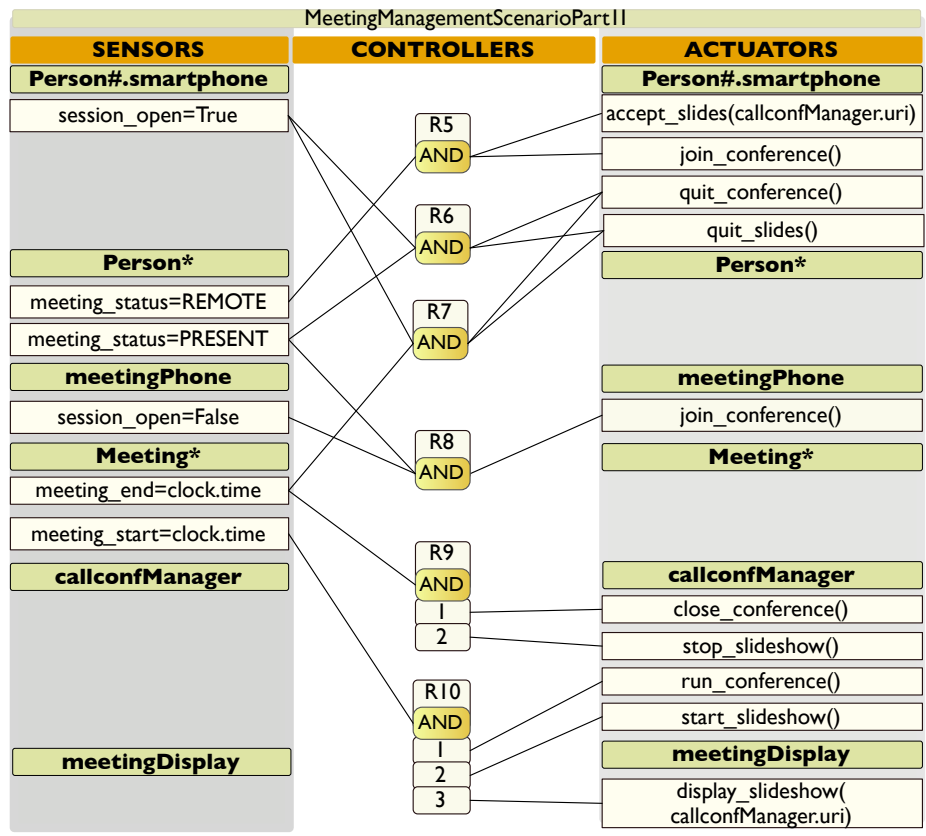


Fig. 5. An example program in the Pantagruel development environment (Part 2)

trigger some actions. For example, when a person is present, we further want to test whether (s)he is participating to an upcoming meeting. To do so, we define a section over a sub-part of a tag entity that holds information about its owner. Specifically, the `Tag#.person` section of Figure 4 refers to the `person` sub-part of a tag being collected in the `Tag*` section. A tag instance is collected if (1) it is present in the meeting room (2nd condition in the `Tag*` section), (2) the person's name belongs to the list of participants of a meeting (condition in the `Tag#.person` section), (3) the meeting is occurring (both conditions in the `Meeting*` section). As illustrated in the `Tag#.person` section, the '#' notation is used to introduce a section that refers to each filtered element of an entity class. Elements of a class are filtered in a '*' section, as exemplified by the `Tag*` section. In the `Tag#.person` section, the '#' notation is also used to check the current person against the list of participants of the current meeting (*i.e.*, `Meeting#.participants`), collected in the `Meeting*` section.

Lastly, Figure 4 illustrates a section for the `meetingPhone` entity instance (whose name starts with a lowercase letter). This is needed to operate this spe-

cific phone, when the meeting is over. Notice that, `meetingPhone` is started by the action of Rule R8 in Figure 5.

4.3 Defining context conditions

Sensors consist of conditions defined over context elements, whether constant, external or applicative. Pantagruel provides notations and operators to easily express conditions. Values of context elements can be tested with comparison and set operators. Attributes of entities are accessed using the conventional dot notations. When filtering an entity class, a condition may need to hold for all of its instances. Such condition is expressed by prefixing a predicate by the keyword **ALL** (**NONE** for the inverse). This keyword is illustrated in Rule R4 in Figure 4, where `meetingPhone` is closed when all tag owners have left the meeting room.

A specific construct called **changed** deals with external and applicative context elements. This construct turns true when the value of such attribute changes. As a result, the orchestration rules are completely insulated from the implementation details of the context change. They only focus on the logic to react to a context change.

4.4 Defining actions

Context conditions are grouped with AND/OR operators, forming a controller. When a controller evaluates to true, either a unique action (*i.e.*, a method) is executed, as in Rule R1, or several actions. In the latter case, the actions may be executed in any order (Rule R5) or sequentially (Rule R9). Actions may correspond to attribute assignments, typically needed to update an entity status, as in the `Tag#.person` section in Figure 4. Only write attributes can be assigned values. Volatile attributes have their value updated externally to Pantagruel. The action part of a rule may also involve method invocations, as required to operate entities from Pantagruel. These invocations have to conform to the type signature declared in the environment description. When the method of an instance is invoked, it may update one of its attributes as declared in the environment description. For example, when the `join_conference` method is invoked on `meetingPhone`, it may set the `session_open` to True, indicating that `meetingPhone` has joined an audio session.

5 Implementation

We now present an overview of the implementation of Pantagruel. This implementation is based on the denotational semantics of the language. This formal definition has been used to precisely specify the semantics of the orchestration rules with respect to a taxonomy of entities. As well, it is a basis to formulate various verifications of Pantagruel programs; we briefly discuss this topic at the end of this section.

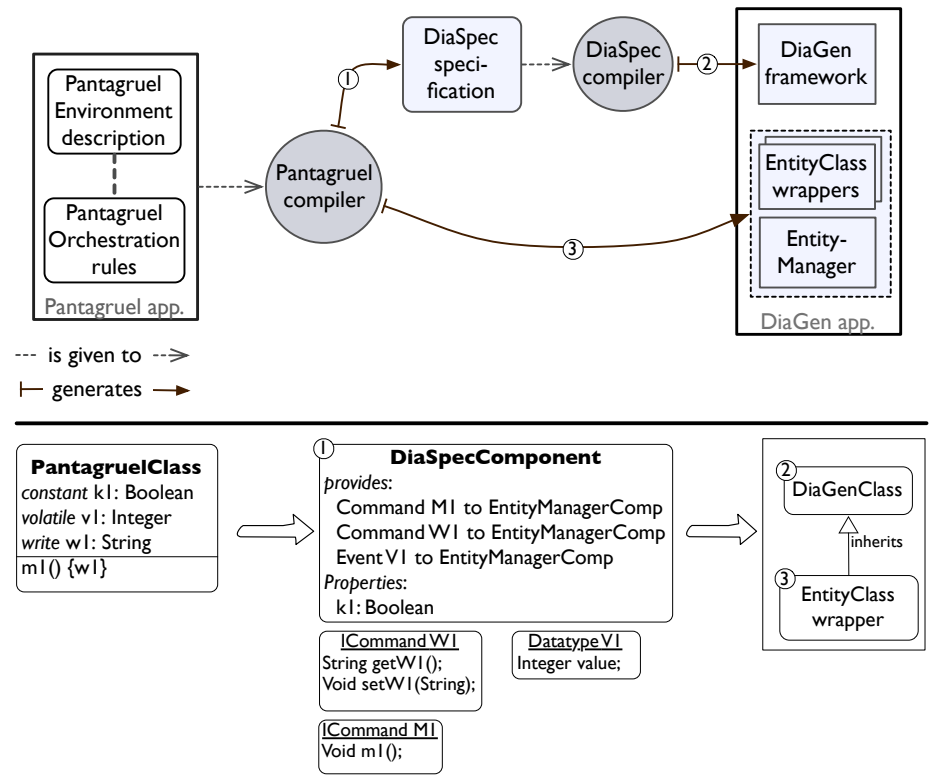


Fig. 6. Pantagruel compilation. Upper part: an overview of the compilation process. Lower part: example of compilation steps

Because of the nature of Pantagruel, its compilation is rather challenging, requiring to bridge the gap between a high-level language and some underlying middleware. This situation usually leads to an intricate compilation process, involving various stages to gradually map high-level abstractions into general-purpose ones.

In contrast, our strategy consists of leveraging a high-level software development tool. Specifically, our approach relies on DiaSpec [16], a general-purpose architecture description language (ADL), targeting distributed systems. This ADL integrates a component-and-connector architecture into Java [17]. It comes with DiaGen, a generator that produces programming frameworks, customized with respect to an ADL description. DiaSpec and its generator enable us to abstract over distributed systems technologies and mechanisms to interact with networked entities. A Pantagruel environment description is thus mapped into a DiaSpec description.

More specifically, the interaction modes offered by Pantagruel to communicate with entities are mapped into DiaSpec connectors for distributed com-

ponents. To do so, volatile attributes are expressed in terms of DiaSpec events. That is, when an entity notifies an event, the volatile attribute of the corresponding Pantagruel entity is updated. Method invocations of Pantagruel entities are compiled into DiaSpec commands, which can be viewed as remote method invocations. Write attributes are mapped into getters and setters, implemented as DiaSpec commands.

Compilation. The overall compilation process of Pantagruel is displayed in Figure 6 (upper part). The first step consists of mapping a Pantagruel environment description into a DiaSpec description (Figure 6 – lower part). An entity class is compiled into a component declaration, mapping Pantagruel interaction modes into DiaSpec ones. When this mapping is completed, the ADL description is passed to DiaGen to produce a programming framework, dedicated to the input taxonomy (step 2). This framework includes support for the declared events and commands; it is generated once and for all for a given taxonomy. The third step of the compilation process consists of generating Java code from the Pantagruel rules, given the customized programming framework. Only this code needs to be regenerated and compiled when the Pantagruel rules change; the customized programming framework is re-used as is. The implementation of a Pantagruel application is completed by introducing wrappers for the environment entities. A wrapper implements the interface required by a Pantagruel entity class. It corresponds to a Java class implementing (1) methods that receive data from some entities and publish them as events, (2) methods that access entity functionalities, and (3) methods managing write attributes.

To generate the wrappers, we allow the taxonomy to be annotated with implementation declarations. These declarations map the taxonomy elements to existing DiaSpec components and their interaction modes (that is, events or commands). During the compilation of a Pantagruel program, the generated DiaSpec specification is connected to the existing DiaSpec components. In doing so, we leverage the existing DiaSpec specifications for which wrapper implementations already exist.

Execution. Once compiled into Java, a Pantagruel application is deployed in the target pervasive computing environment. Executing an application amounts to evaluating rules given the current store, modeling the context of the pervasive computing environment, to produce a new store, reflecting the actions performed. A store consists of volatile attributes and write attributes. A detailed view of the execution process is depicted in Figure 7. At the center lies the `RuleEvaluation` thread that loops over the evaluation of the orchestration rules. At each iteration, the rules are executed with an initial store. When the predicates of a rule hold, its actions are triggered and their effects performed on the new store. Volatile attributes that change during an iteration are handled by the `EntityManager` component. This component uses the new store to update their value as their changes get notified by events. When the iteration is completed, the new store becomes the initial store of the next iteration. This dual-store strategy allows

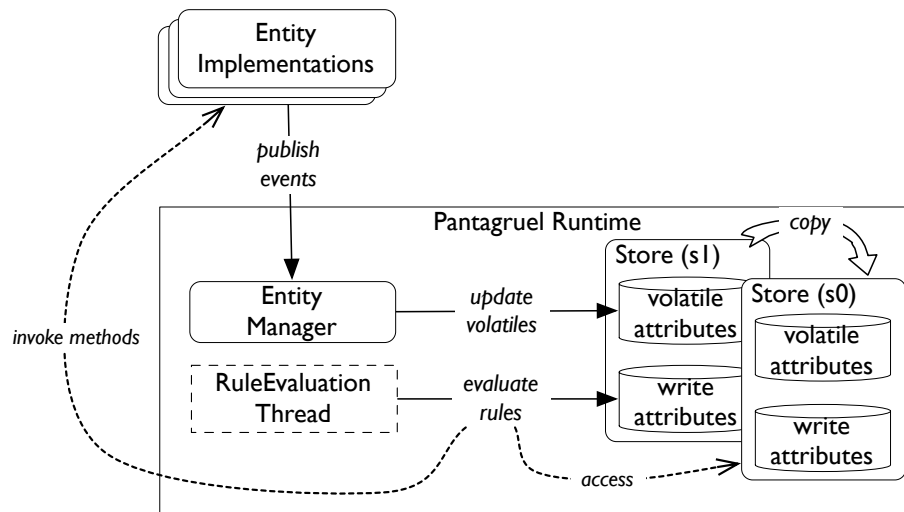


Fig. 7. Execution process

rules to be evaluated independently of the others, making Pantagruel programs easier to understand for the developers.

As can be noticed, DiaSpec enables the compilation of Pantagruel to be rather high level. As well, our approach makes it possible for Pantagruel to benefit from the numerous entities supported by DiaGen and installed in our Lab's smart space. Indeed, it gives Pantagruel access to a range of hardware and software entities.

To facilitate testing of Pantagruel programs, we leverage another tool from DiaSpec: a simulator for pervasive computing applications, called DiaSim [18], which has been developed by our research group. This simulator is coupled with a 2D-renderer. It allows the developer to create a visual layout of a physical environment, including representations of entities. The execution of a compiled Pantagruel program, equipped with simulated entities, is visualized and driven by a set of stimuli and their evolution rules. Existing simulated entities can be re-used by a Pantagruel program, as is done with wrapper implementations of actual entities.

Verification. Verification of Pantagruel programs is based on the formal definition of the language and facilitated by its dedicated nature. Specifically, Pantagruel programs can be represented as finite-state systems, making them amenable to model checking tools. Safety and liveness properties can be specified and proved for a Pantagruel program. To do so, we define a straightforward compilation process for Pantagruel programs into the TLA+ specification language [19] and use the TLC model checker [20] to verify properties. The semantics of TLA formulas is based on sequences of states, similar to the Pantagruel semantics. In essence, our verification process consists of three phases. Firstly, we

generate a TLA+ specification from a Pantagruel program. Secondly, we provide abstraction functions that abstract over the possible states of write and volatile attributes to prevent an explosion of the state-space. Thirdly, we express a set of domain-specific properties as TLA+ formulas. As an example, a property for our meeting scenario can be specified as follows: a conference that was open on a smartphone (`join_conference`) must eventually be closed (`quit_conference`). In other words, we can formulate properties to ensure that, if some sensor conditions hold, then some actuators will eventually be triggered. Specifying and verifying such properties can greatly help developers to ensure the safety of their orchestration logic.

6 Preliminary evaluation

We now present a preliminary evaluation of Pantagruel. This evaluation relies on a prototype of a graphical editor that we developed in the Eclipse IDE⁵. Figure 8 gives a snapshot of the Pantagruel graphical editor. It implements the coupling between the taxonomy and the orchestration logic, providing a graphical guide to build rules that conform to a taxonomy. Using this editor, we defined simple scenarios for three application areas: home automation, surveillance management, and professional communication management.

This experimental work is a first step towards determining Pantagruel's applicability. In particular, we want to assess Pantagruel's ability to fulfill four key requirements: (1) to model a variety of application areas, using the concepts of our taxonomy-based approach; (2) to model a range of applications for a given area; (3) to easily create variations of an application, driven by user preferences, as discussed in our working example (Section 2); (4) to validate the insulation of Pantagruel programs from the heterogeneous implementations of entity classes. The latter requirement was partly addressed by the generation of wrappers presented in the previous section.

To evaluate the first three requirements, we developed three taxonomies for the above-mentioned application areas. These taxonomies factorize the knowledge on three different pervasive application areas. Their differences give a preliminary measure of the expressivity of our declarative language for taxonomies (requirement 1). Each taxonomy specifies the building blocks needed to develop a range of applications within an area (requirement 2). In the rest of this section, for each application area, we present a taxonomy and describe two different applications. We also give a flavor of the ease to adapt one of these applications to specific needs (requirement 3).

6.1 Home automation

We introduce an application to manage the air conditioning and one to manage lights. They involve typical devices found in the home automation area.

⁵ <http://www.eclipse.org> - Eclipse Interface Development Environment

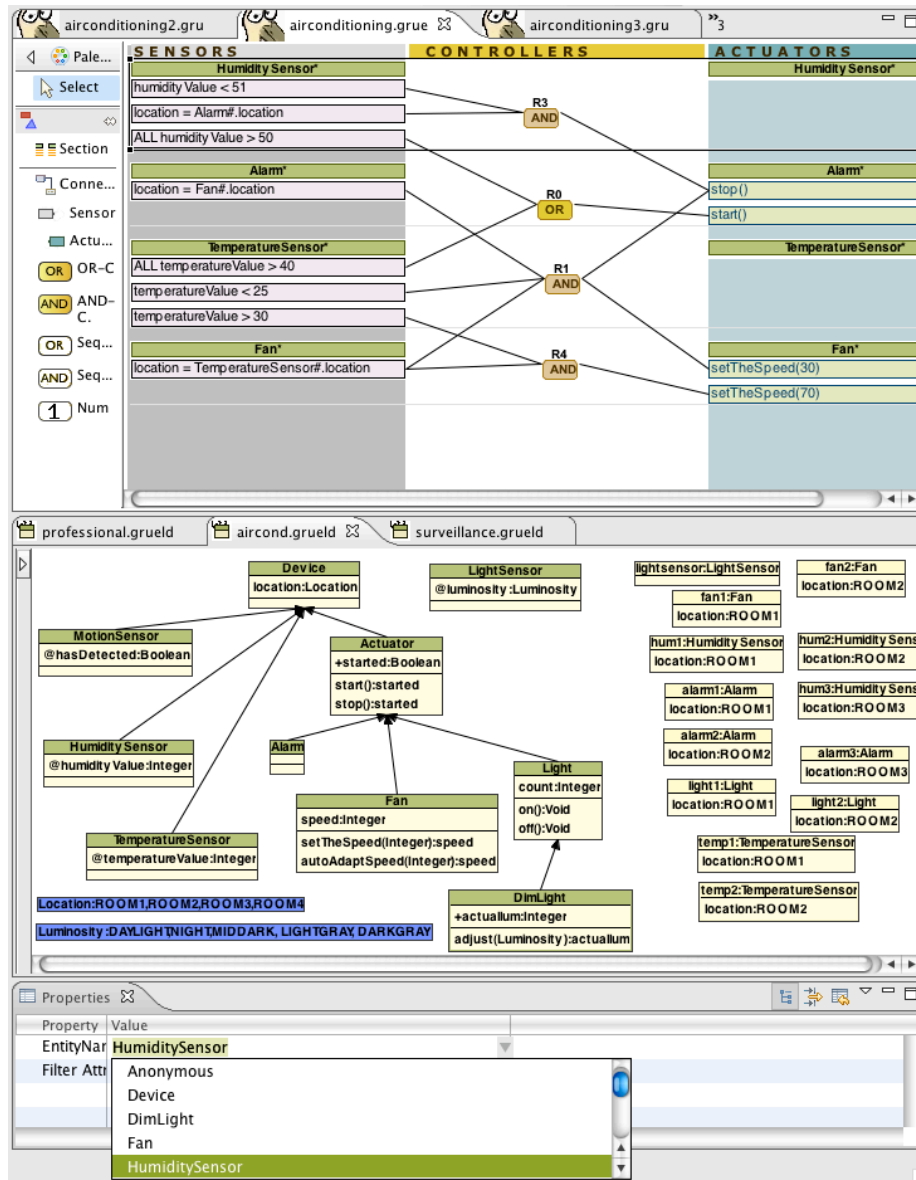


Fig. 8. A screenshot of the Pantagruel graphical editor

These applications imply a set of sensors that capture the external conditions, and a set of actuators to react to these conditions to regulate indoor ambience, including temperature, luminosity, or humidity. A taxonomy for this aspect of the home automation area is composed of seven classes, as shown in Figure 8: MotionSensor with the has_detected volatile, TemperatureSensor with

the `temperatureValue` volatile, `Fan`, `HumiditySensor`, `LightSensor`, `Light`, `DimLight` that refines `Light`, and `Alarm`. We also provide two datatypes: `Location` that specifies the location of sensors and lights, and `Luminosity` that gives a measure of the luminosity.

Our first application controls the fan speed according to the temperature measured by the temperature sensors whose room is given by the `Location` attribute, inherited from the `Device` class. Also, temperature sensors trigger alarms in rooms if the captured temperature exceeds a given threshold. This triggering is done by testing each instance (*e.g.*, `temperatureValue > 50`) of the `TemperatureSensor*` entity class.

The second application controls the lights of the rooms, according to the outside luminosity measured by a light sensor. For example, when the luminosity changes, the light intensity is adjusted accordingly. We also have a rule that controls the lights of a room according to its luminosity and whether presence is detected.

These applications define some basic tasks that involve entities sensing external events and actuating devices, like lights. The sensors are conveniently mapped into a unique volatile attribute. The rules are simple and readable in that they combine two or three sensors to trigger an actuator.

6.2 Building surveillance

The building surveillance detects various forms of intrusions. The taxonomy for the building surveillance area consists of the following entity classes: `Clock`, `Light`, `Camera`, `Alarm`, `MailServer` and `RestrictedLocation`. We also reuse an entity class from the previous application area, namely `MotionSensor`. The camera takes pictures of an intruder. The `RestrictedLocation` class is associated to each room, whose location is defined by the `loc` constant attribute. It defines the `is_restricted` write attribute that can be set to true to restrict the access of a given room.

The intrusion management application sends a message to the security guard when a sensor detected a motion outside the working hours. As well, it triggers the alarms, takes pictures and turns all the lights on. Doing so relies on two rules that respectively activate and deactivate the motion sensors depending on previously defined working hours.

The second surveillance manager monitors restricted rooms at specific times during the day. In case a restricted room is entered by an employee at a prohibited time, a message is sent to the guard. When accessible, the `is_restricted` attribute of a restricted room is set to false.

In this application, by appropriately defining the entity classes, we enable the definition of variations of applications. For example, instances of the `RestrictedLocation` entity class can be parameterized to change the restrictions of the rooms. The `Clock` sensor can be configured to define new time intervals.

6.3 Professional communication management

We defined a taxonomy towards managing professional communications, namely, call transfer, when somebody is unavailable, and reminder, when a meeting is upcoming. In contrast with the previous taxonomies, this one essentially involves entities corresponding to software components. Our taxonomy contains the following set of classes: `PresenceAgent` defines the `status` volatile, indicating the user availability; `SmartPhone` consists of the `availability` write attribute, the `callreceived` volatile, and a method to control the phone ring; `TelephonyServer` has a forward-call method; `SMSServer` sends SMS messages; and, `Agenda` defines a `meeting` volatile, indicating a meeting status: upcoming, ongoing or completed. All instances of these entity classes, except the servers, are associated with a user (*e.g.*, Bob).

The first application is composed of three rules. One rule changes the availability status of a person to false if she is attending a meeting or her presence agent has a `BUSY` status. Another rule does the inverse when her agent status is `ONLINE` and no meeting is planned. A third rule invokes the `forwardSecretary` method of the telephony server with the caller name the original callee.

A second application sends a reminder to the participant of an upcoming meeting, using the `meeting` volatile of her agenda. It also turns off the smart phone ring when the meeting has started, and turns it back on at the end of the meeting.

6.4 Discussion

Expressivity. From this preliminary study we draw the following conclusions. Pantagruel fulfills the first requirement in that it makes it possible to define entities ranging from physical sensors, like a temperature sensor, to software components, like an agenda. Because of our model of entity interaction, the orchestration logic abstracts over the nature of the attributes (*i.e.*, volatile, write or constant) that define the rule contexts, raising further the level of programming. We examined the Pantagruel programs written for the target application areas and observed that they had a small number of rules and did not illustrate any limitations. However, our applications remain simple; larger scale developments are needed to further assess Pantagruel's expressivity and scalability.

Benefits of a visual language. Visual programming languages and textual languages essentially differ in that visual languages offer a multi-dimensional representation of programs [21]. For example, one particular dimension is represented by spatial relationships between objects. In Pantagruel, we offer a spatial structuring of entities, rules, sensors, and actuators that is customized to the domain of pervasive computing. Carrying out a multi-dimensional representation of a program requires a close integration between a visual language and its programming environment. This integration facilitates the management of the constituent elements of a program, improving the development process.

Because it is driven by a taxonomy, the rule editor of Pantagrue allows to directly identify which entities are involved in an orchestration logic, and what purpose they serve. We plan to further emphasize this benefit by improving our graphical selection mechanisms to filter entities, rules, sensors and actuators.

Finally, the visual nature of Pantagrue should contribute to making this language accessible to novice programmers. In the future, we plan to explore the combination between the Pantagrue language and the DiaSim simulator to provide programmers with a better visual feedback of the behavior of their orchestration rules.

7 Related Work

Frameworks or middlewares have been proposed to separate the description of the environment entities from the application (*i.e.*, orchestration) logic. Kurkani *et al.* [22] describe entities using an XML-based representation, and develop the orchestration logic using a specification language. However, the relationship between the entities and the application logic is achieved via references to XML files. This approach is low level and makes verification difficult. In our approach the orchestration logic is driven by the entity class descriptions, which enable the verification of Pantagrue programs. In the Olympus programming model [6], the orchestration logic relies on an ontological description of entities. However programming the orchestration logic is done in a general-purpose programming language, which is not constrained by the ontological descriptions and may lead to programs that do not conform to these descriptions.

Visual prototyping tools are gaining interest within the pervasive computing community. The visual nature of these tools enables a simplified, intuitive representation of the orchestration logic. However, to the best of our knowledge, none of the tools reported in the literature propose a taxonomy-based approach to developing orchestration logic. As a consequence, existing approaches are either limited to a specific pervasive computing area or cannot be customized with respect to a given area. For example, CAMP [10] is an end-user, rule-based tool. It is based on the magnetic poetry metaphor, proposing to visually compose sentences. However, its vocabulary is restricted to the home automation area. Another related tool is iCap [11], it provides a fixed set of generic entity classes: objects, activities, time, locations and people. Unlike our approach, these pre-defined entity classes cannot be extended with new attributes and methods. Moreover, iCap does not provide a uniform means to express rules over a group of entities besides a group of persons. In contrast, our approach provides syntactic abstractions to define filters on all instances of an entity class, as well as specific entities, enabling the orchestration logic to address a dynamically changing environment, as well as static entities.

The visual language VisualRDK [23] contrasts with the previous tools in that it targets novice or experienced programmers. VisualRDK is a programmatic approach, offering language-inspired constructs such as processes, conditional cascades and signals. These constructs mimic conventional programming, obscuring

the domain-specific aspects of entity orchestration. Pantagruel differs from this approach in that rules are driven by the entities, connecting sensors to actuators, as reflected by the three-column panel of its development environment. Other visual prototyping tools like OSCAR [12] and Jigsaw [24] propose an approach to discovering, connecting and controlling services and devices, dedicated to domestic spaces. However, they offer limited expressivity to access the functionalities of the entities. Finally, tools based on the storyboard paradigm like Topiary [9] express the orchestration logic as control flows. However, Topiary does not allow scenarios where actions do not have a visual representation.

8 Conclusion and Future Work

Pervasive computing is reaching an increasing number of areas, creating a need to factorize knowledge about the entities that are relevant to each of these areas. This paper presents Pantagruel, an approach and a tool that integrate a taxonomical description of a pervasive computing environment into a visual programming language. Rules are developed using a sensor-controller-actuator paradigm, parameterized with respect to a taxonomy of entities. We have used Pantagruel to develop a number of scenarios, demonstrating the benefits of our taxonomy-based approach.

In the future, we will further evaluate our taxonomy-based approach by widening the scope of the target pervasive computing areas. To do so, we are collecting coordination scenarios that have been reported in the literature. Specifically, we plan to study the management of hospital patients [1], technological assistance to persons with intellectual deficiencies [25], and museum tour guide systems [3]. These studies will further assess how much is factorized by our taxonomy-based approach and how expressive is our rule-based language. While developing new scenarios, we intend to refine the syntax and semantics of our language.

Tackling these scenarios will go beyond the mere definition of a taxonomy and applications in these areas. In fact, we will test our Pantagruel programs by leveraging DiaSim, our pervasive computing simulator [18]. In doing so, we are also able to explore the expressivity of Pantagruel programs in areas that would otherwise be out of reach.

Another direction for future work is to conduct a usability study of Pantagruel. The goal is to determine what skills are needed to define a taxonomy in a given area and whether it can be defined by a domain expert without a programming background. Similarly, we want to determine what training is required to define coordination rules and whether these rules can be defined by non-programmers.

Finally, we are developing various analyses for Pantagruel programs to guarantee properties such as termination and non-interference of coordination rules. This work builds upon the formal semantics of Pantagruel. These verifications will be integrated in the Pantagruel development environment, providing early feedback to the developer.

Acknowledgments

The authors would like to thank Alex Consel for introducing them to Game Blender, a sub-application of Blender dedicated to make games. Alex shared his enthusiasm for Game Blender and answered numerous questions about it. This tool has been a major source of inspiration for Pantagruel.

This work was partially funded by the *Conseil Régional d'Aquitaine*.

References

1. Boris Shulman. RFID for Patient Flow Management in Emergency Unit. Technical report, IBM Corporation, <http://www.ibm.com/news/fr/fr/2006/03/cp1851.html>, 2006.
2. W. Keith Edwards and R. E. Grinter. At home with ubiquitous computing: Seven challenges. In *3th Int'l Conference on Ubiquitous Computing (UbiComp)*, pages 256–272. Springer-Verlag, 2001.
3. T. Kuflik, J. Sheidin, S. Jbara, D. Goren-Bar, P. Soffer, O. Stock, and M. Zancanaro. Supporting small groups in the museum by context-aware communication services. In *12th Int'l Conference on Intelligent User Interfaces (IUI)*, pages 305–308. ACM, 2007.
4. M. Adkins, J. Kruse, and R. Younger. Ubiquitous computing: Omnipresent technology in support of network centric warfare. In *35th Hawaii Int'l Conference on System Sciences (HICSS)*, page 40. IEEE Computer Society, 2002.
5. M. Roman and R. H. Campbell. Gaia: enabling active spaces. In *9th ACM SIGOPS European Workshop*, pages 229–234. ACM, 2000.
6. 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 *3rd Int'l Conference on Pervasive Computing and Communications (PerCom)*, pages 7–16. IEEE Computer Society, 2005.
7. R. Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.
8. D. Garlan, D. P. Siewiorek, and P. Steenkiste. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1:22–31, 2002.
9. Y. Li, J. I. Hong, and J. A. Landay. Topiary: a tool for prototyping location-enhanced applications. In *17th Symposium on User Interface Software and Technology (UIST)*, pages 217–226. ACM, 2004.
10. K. N. Truong, E. M. Huang, and G. D. Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In *6th Int'l Conference on Ubiquitous Computing (UbiComp)*, pages 143–160. Springer, 2004.
11. A. K. Dey, T. Sohn, S. Streng, and J. Kodama. iCAP: Interactive prototyping of context-aware applications. In *4th Int'l Conference on Pervasive Computing (Pervasive)*, pages 254–271. Springer, 2006.
12. M. W. Newman, A. Elliott, and T. F. Smith. Providing an integrated user experience of networked media, devices, and services through end-user composition. In *6th International Conference on Pervasive Computing (Pervasive)*, pages 213–227. Springer, 2008.
13. J.J. Pfeiffer Jr. Altaira: A rule-based visual language for small mobile robots. *Journal of Visual Languages and Computing*, 9(2):127–150, 1998.

14. J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning. LEGOsheets: a rule-based programming, simulation and manipulation environment for the LEGO programmable brick. In *11th Symposium on Visual Languages (VL)*, pages 172–179. IEEE Computer Society, 1995.
15. W. Jouve, J. Lancia, N. Palix, C. Consel, and J. Lawall. High-level programming support for robust pervasive computing applications. In *6th Int'l Conference on Pervasive Computing and Communications (PerCom)*, pages 252–255, 2008.
16. W. Jouve, N. Palix, C. Consel, and P. Kadionik. A SIP-based programming framework for advanced telephony applications. In *2nd Int'l Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–20. Springer-Verlag, 2008.
17. N. Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
18. W. Jouve, J. Bruneau, and Consel C. DiaSim: A parameterized simulator for pervasive computing applications. Technical report, INRIA/Labri, 2008.
19. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
20. Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, pages 54–66, London, UK, 1999. Springer-Verlag.
21. Margaret M. Burnett and David W. McIntyre. Visual programming - guest editors' introduction. *IEEE Computer*, 28(3):14–16, 1995.
22. D. Kulkarni and A. Tripathi. Generative programming approach for building pervasive computing applications. In *1st Int'l Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments (SEPCASE)*, page 3. IEEE Computer Society, 2007.
23. T. Weis, M. Knoll, A. Ulbrich, G. Muhl, and A. Brandle. Rapid prototyping for pervasive applications. *IEEE Pervasive Computing*, 6(2):76–84, 2007.
24. J. Humble, A. Crabtree, T. Hemmings, K-P. Åkesson, B. Koleva, T. Rodden, and P. Hansson. "Playing with the Bits" user-configuration of ubiquitous domestic environments. In *5th Int'l Conference on Ubiquitous Computing (UbiComp)*, volume 2864, pages 256–263. Springer, 2003.
25. Svensk A. Design for cognitive assistance. In *Human Factors and Ergonomics Society Europe Annual Meeting (HFES)*, 2003.