Taxonomy-Driven Prototyping of Home Automation Applications : a Novice-Programmer Visual Language and its Evaluation

Zoé Drey, Charles Consel

Taxonomy-Driven Prototyping of Home Automation Applications: a Novice-Programmer Visual Language and its Evaluation

Zoé Drey[a], Charles Consel[a,b]

[a] *INRIA, Bordeaux Sud-Ouest*
[b] *University of Bordeaux/LaBRI*

Abstract

Home automation environments are dedicated to helping users in their everyday life and are being deployed in an increasing number of areas, including home security, energy consumption, and assisted living. The range of situations to be addressed makes the development of home automation applications challenging: it requires to manage heterogeneous entities with a wide variety of functionalities. Moreover, since this area covers a large spectrum of user needs, it is crucial to ease the development and the evolution of these applications.

This paper presents Pantagruel, an expressive and accessible approach to integrating a taxonomical description of a home automation environment into a visual programming language. A taxonomy describes the relevant entities of a given home automation 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 that integrates a taxonomical approach in the development of orchestration rules. Furthermore, we have developed a compiler for Pantagruel and successfully used it to test applications in various areas related to orchestration development for the domain of home automation. Finally, we have successfully evaluated the usability of Pantagruel through a user study performed with eighteen novice programmers.

Keywords: Visual Rule-Based Language, Home Automation, Entity Orchestration

1. Introduction

Easing the programming of applications for assisting users in their everyday life has been a dominant trend in the home automation research community [1, 2, 3, 4, 5, 6]. These applications aim to address the user needs by automating their daily tasks in various areas of the home automation domain, such as assisted living, home security, and energy management. This goal is facilitated by a constant flow of innovations in devices that enables to form ever richer home automation environments. Furthermore, not only do devices have increasingly more computing power, offering high-level interfaces to their rich functionalities, but they also are now network-enabled, making these functionalities accessible remotely. The advent of this new generation of devices allows the development of applications to abstract over low-level embedded systems intricacies, paving the way for the high-level orchestration of entities, whether actual devices (*e.g.,* temperature sensors, lights) or software components (*e.g.,* databases, calendars).

Because the users of home automation environments are intimately concerned with applications, it is crucial to make the development process usable by the widest audience. To address this challenge, the learning threshold of application programming can be lowered, enabling the user of applications to program them. To do so, end-user approaches have been proposed to prototyping pervasive computing applications. These approaches provide a domain-specific vocabulary (*e.g.,* iCAP [3]) or use a metaphor-based representation (*e.g.,* CAMP [1]). To assess their approaches, researchers conducted user studies [3, 1, 7, 4].

In most cases, usability is obtained at the expense of expressiveness: the proposed approaches are restricted to a specific area, making hard the evolution of applications with regard to the technology advances. Because new devices and functionalities are continually made available, the application areas should be enriched accordingly (for example, cellphones turned into a multi-function device). Furthermore, as user requirements evolve over time, applications should be easily adjusted. This situation is well illustrated in the assisted-living area. For example, applications for assisting an intellectually impaired person necessitate periodic adjustment as his conditions improve or deteriorate.

Therefore, combining usability and expressiveness is crucial for an application development approach in the domain of home automation.

This paper

This paper presents Pantagruel, a high-level language that integrates expressiveness and usability for programming home automation applications. Pantagruel is dedicated to the development of applications, parameterized by the description of a home automation environment. Specifically, our approach consists of a two-step process: (1) the description of a home automation environment takes the form of a *taxonomy* and defines the functionalities and the properties of the environment entities; (2) the development of a home automation application is driven by a taxonomy of entities and consists of orchestrating them using high-level constructs. To support these two steps, the Pantagruel language consists of a taxonomy layer and a visual orchestration layer.

The taxonomy definition 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. Each class 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. The taxonomy definitions make our approach open-ended, adding an expressiveness dimension to cover a range of areas. A first step towards evaluating this expressivenes is reported in Section 7.

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 novice programmers, as demonstrated by its use in various fields such as computer game design (*e.g.,* Blender[1]) and robot control (*e.g.,* Altaira [8] or LegoSheets [9] for Lego Mindstorms[2]). Like a game logic, an orchestration logic collects context data from sensors, combines them with a controller, and reacts by triggering actuators. 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

[1]http://www.blender.org
[2]http://mindstorms.lego.com/

rules. This approach makes Pantagruel usable by novice programmers, as illustrated by our user study, reported in Section 8.

The main contributions of this paper are as follows.

An open-ended approach We introduce an approach to visually prototyping orchestration applications. The novelty in this approach is that it is parameterized with respect to a description of a home automation environment. This makes our approach applicable to a range of areas for the domain of home automation.

A taxonomy-driven visual language We extend the sensor-controller-actuator paradigm to allow the programming of the orchestration logic to be driven by an environment description. This approach eases programming and enables verifications. Moreover, early testing of applications is made possible by leveraging a home automation simulator, named DiaSim [10], in the Pantagruel environment.

Towards an evaluation of Pantagruel expressiveness We show that our taxonomy language ranges over the categories of entities of the home automation domain. Furthermore, we evaluate the ability of the orchestration layer of Pantagruel to express a range of home automation applications.

An evaluation of the orchestration logic usability We validate the usability of the orchestration layer of Pantagruel by a user study performed with eighteen novice-programmer participants. This study is based on existing usability evaluation processes found in the home automation literature.

To motivate our approach, Section 2 presents an example of an assisted-living application. Section 3 examines the requirements for a development approach targeting the domain expert of home automation areas. Section 4 introduces our taxonomical approach to defining descriptions of home automation environments. Section 5 presents a visual environment to develop applications that orchestrate entities, defined in a taxonomy. Section 6 details the development process for building home automation applications using our taxonomy-driven approach. Section 7 presents a study towards validating the expressiveness of our approach. Sections 8 and 9 validate the usability of our approach. The related work is detailed in Section 10. Concluding remarks and future work are provided in Section 11.

2. Working Example

To motivate our approach, we consider as an example an assisted-living application. This example is based on scenarios collected by a thirty-years experienced caregiver, and dedicated to cognitively impaired persons.

This application area orchestrates various kinds of entities, consisting of RFID readers and tags, a calendar, an alarm clock, a mixing valve, and a time tracker to indicate a task status. Figure 1 is an example of a physical layout for an impaired person's apartment.

Our example scenario compiles situations encountered in Svensk's career as a caregiver [11]. More scenarios have been collected and documented in Carmien's thesis [12].

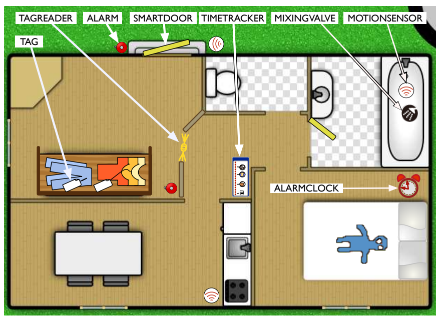


Figure 1: Henrick's apartment

Svensk describes the beginning of a working day for Henrick, a fictitious character with intellectual disabilities, who lives independently in an apartment. He needs to be assisted during the day, from the time he wakes up until he leaves his apartment for work. Specifically, Henrick needs to wake up on time according to his day activity: work or holiday. To help him wake up, his alarm clock is configured to play a music whose genre is determined with respect to the day activity.

Once Henrick is awake, he needs to take a shower, then get dressed, and have breakfast. Because he has difficulties to regulate the water temperature, automation needs to be introduced in the shower to address this problem. Svensk proposes a system such that *"when* [*Henrick*] *closes the door, the water comes on automatically at just the right temperature"*. Such automation enables Henrik to turn the shower activity into an ordinary step of his morning routine. After the shower, Henrick gets dressed. To select the clothes that are appropriate for the weather, he is assisted by a caregiver: he "*gets dressed in the clothes a staff member has laid out on a chair*". To relieve the caregiver from the task of picking clothes for Henrick a Web service and RFID technology can be combined. Specifically, Henrick is being informed about the latest weather report, when he enters the dressing room. To ensure that selected clothes are appropriate, they are attached a RFID tag. These tags, combined with a tag reader, make it possible to warn Henrick in case the selected clothes are not adequate for the weather of the day.

Because Henrick is unable to read the time on a clock, a specific process must be devised to allow him to keep track of time and tasks he needs to perform. In doing so, Henrick's daily tasks can be streamlined, preventing panic caused by a task that is omitted or performed for too long. To do so, software appliances such as a calendar and a time tracker can be used.

Many other scenarios for assisted living could be imagined, combining other related activities, such as assistance for cooking and leaving home on time. In fact, for a given environment, it should be possible to define various orchestration scenarios, adapting to users' requirements and preferences, and reacting 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 expressiveness to combine them. Finally, the assisted-living environment consists of entities for which numerous variations are available, requiring an approach to defining the orchestration logic that abstracts over these differences.

3. Requirements for home automation development tools

The requirements for tools to develop home automation applications strongly depend on the developers targeted for these tools. In this section, we first identify these developers. Then, we examine the needs for such tools. Finally, we examine the needs to make accessible the building blocks of an application area of the home automation domain.

3.1. Identifying the users

The home automation domain involves a variety of roles, from the installer, who equips the house with networked devices, to the occupants, who are the end users of applications. These users can be more or less involved in the development of applications. Let us take the assisted-living area as an example and consider caregivers as the end-user's proxy. Building applications for the apartment of an impaired person involves at least a caregiver and the impaired person. The caregiver is a domain expert of the assisted-living area. Also, he has a good knowledge of the needs of the person he is assisting.

Because they have an extensive knowledge of the end-user needs and the application area, caregivers are most qualified to propose solutions. Specifically, they can picture what kind of application could assist the end user to perform a given task based on his abilities and preferences. This is particularly meaningful because the end user may not be able to directly express his needs [13].

Consequently, an end user, or his proxy, is the domain expert required to develop a home automation application.

3.2. Enabling a high-level description for orchestration

Entity orchestration occurs in a variety of areas, ranging from robot control to game programming and service orchestration. In these fields, the flow-based paradigm [14, 15, 16] as well as the rule-based paradigm [8, 17, 18, 19, 20] have been widely adopted and proved their usability. Our aim is to apply one of these well-established paradigms for the home automation area. Home automation devices are mainly composed of sensors and actuators, where actuators perform actions in reaction to sensed data. There is a direct correspondance between the sensor-controller-actuator paradigm and the rule-based paradigm: sensors represent context data defining a condition; actuators represent actions triggered when this condition holds; and, a controller combines conditions and actuators to form a rule. In contrast to the flow-based paradigm, rules represent elementary reactions to sensor detection; they represent small units of program that can be understood independently of each other, thus easing their visualization inside a window [8].

3.3. Describing the building blocks of an application area

A home automation application interacts with entities (*e.g.,* motion detectors, lights, and calendars) whose heterogeneity and low-level implementation make them unusable by a domain expert. To overcome this problem, it is necessary to offer high-level constructs abstracting over the entities low-level details and factoring entity variations. Such abstractions should provide the relevant data to the domain expert.

4. Defining a networked environment

To abstract over the variations of entities, we introduce a declarative language to defining a taxonomy of entities relevant to a given area. It is declarative in that it aims to specify what the entities do, not how they are implemented. The entity declarations form an environment description that can then be instantiated for a particular setting. Entities are specified by an entity expert developer, based on the needs expressed by the domain expert.

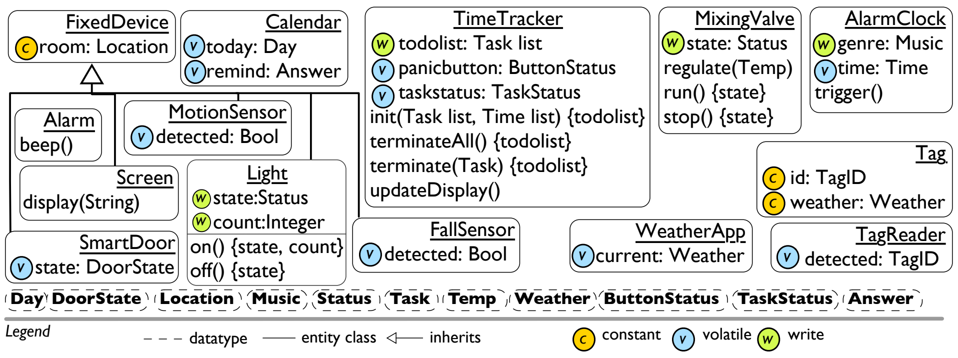


Figure 2: A taxonomy of entities for our working example

4.1. 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 contextual information about the environment, and methods, accessing the entity functionalities. Entity classes are organized hierarchically, allowing attributes and methods to be inherited. Figure 2 displays a UML-based representation of entity classes for the assisted living area in an impaired person's apartment. Entity classes can be organized hierarchically, enabling them to share attributes and methods. For example, the `Light` and the `Alarm` entity classes inherit the `room` attribute from the `FixedDevice` entity class.

The specificity of our approach is that the nature of the context information captured by entities is encapsulated both in their attributes and their methods, as explained next.

Context interface. A class of entities captures context information about a home automation environment. This context information may be fixed (*e.g.,* the location of a light) or vary over time (*e.g.,* a detected badge). In Pantagruel, context information is modeled by the attributes of entity classes.

Context information plays a key role to express the conditions of orchestration rules. If addressed with inappropriate abstractions, it may result in application code bloated with conditionals and data structure operations. Defining a taxonomy thus relies on a thorough analysis of the targeted area. To facilitate this analysis, we propose to classify context information into three categories: constant, external or applicative.

A *constant* context information is an attribute whose value does not change over time. For example, the `FixedDevice` entity class declares a `room` attribute that is constant for a given setting. As such, instances of inherited classes have a constant location.

An external context information is aimed to collect changes in the physical environment. This kind of context information may correspond to sensors (*e.g.,* a device reporting RFID tag location, a temperature sensor) or software components (*e.g.,* a web service reporting the weather of the day). To model an external context information 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 `TagReader` entity class in Figure 2. It defines the `detected` attribute that corresponds to the identifier of the detected outfit worn by Henrick.

Lastly, an applicative context information corresponds to context data computed by the application. To address applicative context information, we introduce *write* attributes. In our example, Figure 2 shows the `genre` attribute of the `AlarmClock` entity class that can either be `JAZZ`, `COUNTRY` or `POP`, depending on the planned activity of the day. This attribute is updated when the calendar daily activity changes.

By making context information to an abstraction of the Pantagruel language, we facilitate its manipulation, improve readability, and enable program verification.

Functionalities. Entity classes offer a variety of functionalities to perform actions according to the context of a home automation environment. To access these functionalities, an entity class declares method interfaces, each of which abstracting over the implementation of a specific action. For example, the `AlarmClock` entity class includes a `trigger` method signature to access the triggering functionality of an alarm clock.

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 control the use of lights, the `Light` class defines two attributes: `state`, holding the light status (`ON` or `OFF`), and `count`, counting the number of times the light is activated. For example, this attribute can be used to remind the user to soon change a light bulb. 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. The side-effects of this method are exposed to allow reasoning when invoked in the orchestration logic. To do so, methods declare the list of side-effected attributes. For example, the `on` method declaration defines the `{state, count}` attributes as side-effected.

4.2. 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 assisted living example. Each entity has a name and refers to its entity class. For instance, we created the `alarmClock` entity of the `AlarmClock` class. This entity corresponds to the alarm clock that is located in the bedroom. Because it plays a specific role in our assisted-living scenario, it needs to be created prior to programming the orchestration logic.

Figure 3 also includes examples of tagged outfits stored in Henrick's wardrobe; they are instances of the `Tag` entity class. As such, their attributes are constant. Entity instances can be created dynamically in a given environment (*e.g.,* 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.,* `alarmClock`).

Pantagruel allows simple datatypes to be defined. For example, the `Location` enumeration type in Figure 3 ranges over the rooms of our example assisted-living apartment.

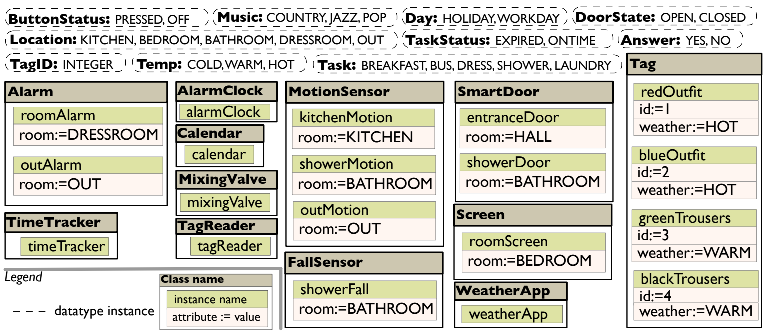


Figure 3: An excerpt of a concrete environment – datatype instances (top), entities (bottom)

5. Defining orchestration rules

We now present our visual language dedicated to the development of orchestration rules. Following our paradigm, the Pantagruel orchestration language offers a panel divided in three columns: sensors, controllers, and actuators. This panel is depicted in Figure 4.

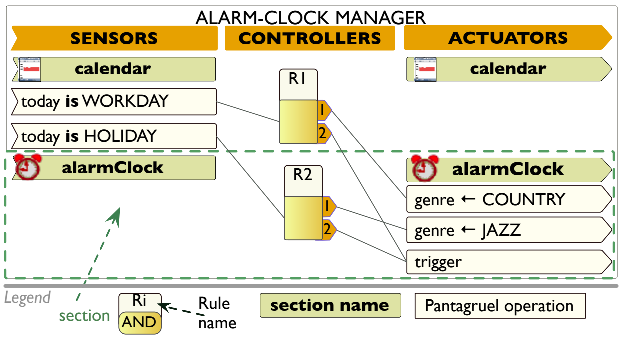
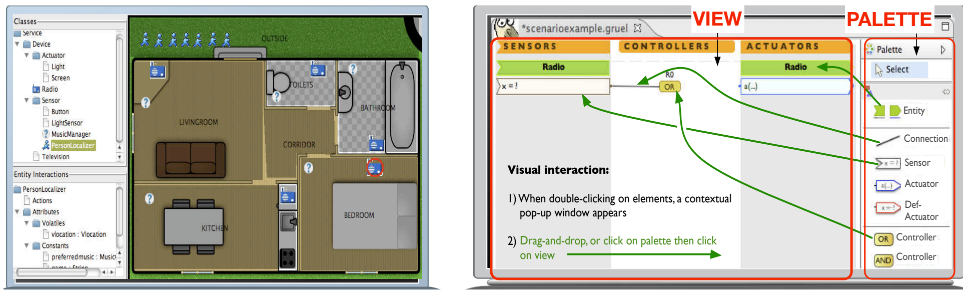


Figure 4: An example Pantagruel program : managing the tasks of the current day

It is integrated into a visual development editor, as illustrated in the right part of Figure 5. This visual editor is composed of two parts : the view, containing orchestration rules, and the palette, offering a tool for each Pantagruel visual element. The Pantagruel editor is connected to a 2D editor and simulator called DiaSim [10], offering a 2D model of a concrete environment, defined in the taxonomy language (Figure 5, left part).

Figures 4, 6, 7 and 8 present the orchestration rules of our working example of assisted living. The first scenario (Figure 4) configures the alarm clock of the impaired person: whether he has to wake up for a work day or for a holiday. The second scenario (Figure 6) handles the management of the person's tasks, using a time tracker, a calendar, an alarm clock, and a fall sensor. The third scenario (Figure 7) offers assistance to the clothing task of the person, and the fourth scenario (Figure 8) manages the apartment lights.

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



Visual interaction:

1) When double-clicking on elements, a contextual pop-up window appears

2) Drag-and-drop, or click on palette then click on view

Figure 5: DiaSim 2D model and the Pantagruel editor

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 editor provides the programmer with contextual menus and on-the-fly verification to guide the definition of rules. We further examine the development process using the Pantagruel editor in Section 6.

5.1. Sections as constituent parts of a rule

To visually structure the definition of orchestration rules, the development panel is divided into horizontal sections, one for each entity instance involved in the application.

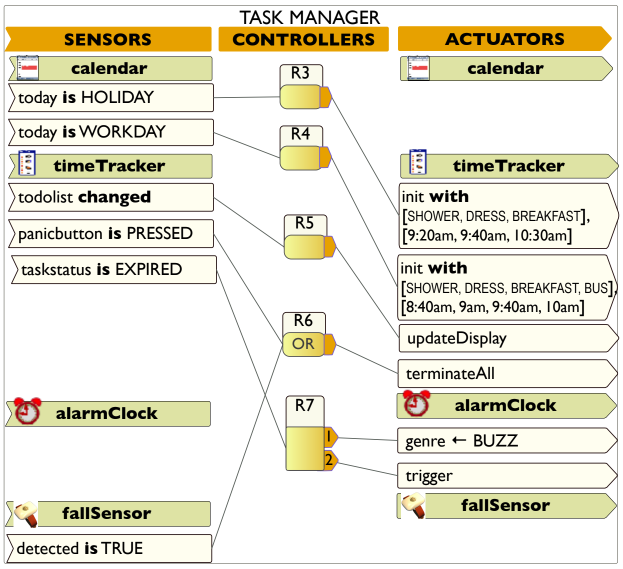


Figure 6: An example Pantagruel program : configuring the alarm clock

For example, the second section of Figure 4 is defined for the alarmClock entity instance. Within a section, Pantagruel operations are in the scope of the corresponding entity instance.

For example, the `todolist` attribute, defined in the `TimeTracker` class of the environment description (Figure 2), can be manipulated within the `timeTracker` section. A section is also required to trigger an action on this entity, *e.g.,* updating the task list display (see Figure 6). In contrast, when an attribute is out of the scope of a section, it needs to be explicitly accessed; this is done with the `current` attribute of the `weatherApp` entity (Figure 7) using the notation "`attribute of` *`entity`*".

5.2. Refering to entity classes

As described above, Figure 4 illustrates a section for the `alarmClock` entity instance, represented with a single icon. This is needed to operate this specific alarm clock when Henrick has to wake up on time. When a large range of entities are deployed in a home automation environment, orchestration rules as defined so far do not scale up; they need to range over all entities of a class. Moreover, when actual entities are dynamically created (*e.g.,* a new `Tag` instance appearing in an environment, corresponding to a new outfit), they cannot be explicitly refered to in a rule. To address these issues, we allow a section to refer to all entity instances of an entity class. The name of such a section is composed of the entity class name (illustrated with a double icon). When an orchestration rule includes an operation (sensor or actuator) coming from such a section, it is executed over all the instances of the corresponding entity class. For example, Figure 7 includes the `Tag` section. One of the conditions defined in this section determines whether the outfit worn by Henrick is adequate for the current weather (*i.e.,* `weather is current of` *`weatherApp`*). This condition enables a rule to apply to any tagged outfit stored in the wardrobe.

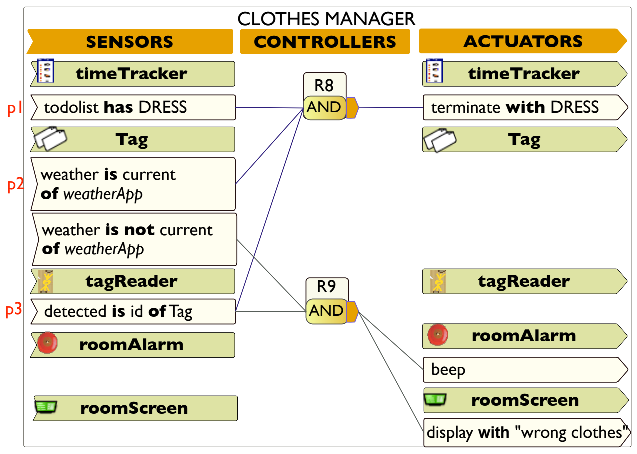


Figure 7: A example Pantagruel program to assist clothing

5.3. 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.

5.3.1. Condition operators

Values of context elements can be tested with comparison operators (*e.g.,* <, >, `is` and `isnot`) and set operators (*e.g.,* `has` and `in`). When a sensor operates on an entity class instead of a specific entity instance, it acts as a filter on the instances of this class. For example, in Figure 8 at the `Light` section, the sensor `room is room of` *`MotionSensor`* collects the lights that are present in the room where Henrick is detected.

A specific construct called `changed` operates on an external or applicative attribute (see Figure 6) . This construct yields true whenever an attribute value changes. As a result, the orchestration rules are completely insulated from the implementation details of the context change; they focus on the logic to react to a context change.

5.3.2. Flow combination

As we collect entities from an entity class, we sometimes need to further refine the filtering and trigger some actions. For example, when Henrick's outfit is detected by the tag reader, we further need to test whether the outfit is suited for the current weather. To do so, we define an additional condition over the `tagReader` section that only filters relevant outfit tag(s). Specifically, a tag instance is collected if (1) it corresponds to an outfit matching the current weather (the p_2 condition in the `Tag` section), and (2) Henrick is currently wearing the tagged outfit (the p_3 condition in the `tagReader` section). Filtering clauses can be combined by the `AND` and `OR` controllers.

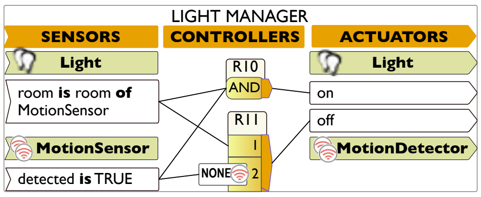


Figure 8: Light manager application

Beyond logical operators, filtering an entity class requires to express clauses that apply to all its instances, or none of them. This filtering is done by the `ALL` and `NONE` controllers, prefixed with an entity class or its icon representation (*e.g.,* NONE). This is illustrated in the `R11` rule in Figure 8, where `Lights` are switched off when none of the motion sensors of a room has detected a presence. These sensors are collected using the `MotionSensor` entity class. In this case, a flow ordering is applied, first selecting lights and motion sensors, then testing if none of the collected sensors is activated.

5.4. Defining actions

When a controller evaluates to true, either a unique action (*e.g.,* a method) is executed, as in the `R3` rule, or several actions are performed. In the latter case, the actions may be executed in any order (the `R9` rule, Figure 7) or sequentially (the `R7` rule, Figure 6). Actions may correspond to attribute assignments, typically needed to update an entity status, as in the `alarmClock` section in Figure 4. Only write attributes can be assigned a value. The 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 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 terminate method is invoked on timeTracker, it may set the todolist write attribute to the empty list (the R6 rule, Figure 6), indicating that no more tasks need to be achieved.

The use of write attributes enables to depict a causal relationship between rule actions. For example, the display of the time tracker is updated only if its todo list has changed (the R5 rule, Figure 6). This status is enabled by the R3, R4, R6, and R8 rules.

5.5. Executing the application : key concepts

Pantagruel is a reactive programming language in that it constantly interacts with the environment, by *reacting* to context changes. We have defined a simplified model of computation, facilitating the orchestration steps of a program. Our model relies on the following key concepts: (1) context-centric model, (2) discrete time and parallel mode and (3) non-interference. We now describe these concepts and we relate them to the Pantagruel visual paradigm and its usability.

Context-centric model. To represent the reactive nature of Pantagruel, we leverage the context information provided by the taxonomy, focusing on what data are available at each orchestration step, instead of how these data are acquired. These data, representing the runtime state of a program, are made easy to manipulate via the sensor column of the orchestration panel.

Discrete time and parallel mode. Discrete time is modeled as clock ticks. Each tick corresponds to an *orchestration step* where all the rules are evaluated within a tick. Parallel mode assumes that all the rules, whose conditions are satisfied at a given step, are executed with the same state, making this execution simultaneous. As a result, there are no rule dependencies within a step; the user can observe the intended method invocations in the right part of the visual layer by selecting specific sensors in the left part.

Non-interfering execution. We say that execution is non-interfering when two rules can be executed independently from each other because their side-effects are disjoint. The side-effect declarations of methods enable to detect potential interferences. We use this language property to guide the user when he defines orchestration rules with side-effecting methods. A conflict can be displayed on the visual layer by highlighting interfering rules. This process forces deterministic behavior, thus increasing the confidence of the domain expert when developing orchestration rules.

6. Development process

Following our programming language presentation, we now explain an adapted programming process for a given application area. This programming process is integrated in an overall process summarized in Figure 9. Initially, the end user expresses his requirements that are analyzed by the domain expert. This step results into (1) goals, which consist on an informal, decomposed description of the expected behavior of the application, and (2) entities, further specified into a Pantagruel taxonomy. Entities are specified by the entity expert, based on the needs expressed by the domain expert. Once specified, the entities are used as the building blocks to be orchestrated by Pantagruel. Application development is ensured by the domain expert, using the Pantagruel visual development editor. We now examine how the editor eases the domain expert in the development task ; this process is illustrated by creating a rule similar to the R9 rule of Figure 7.

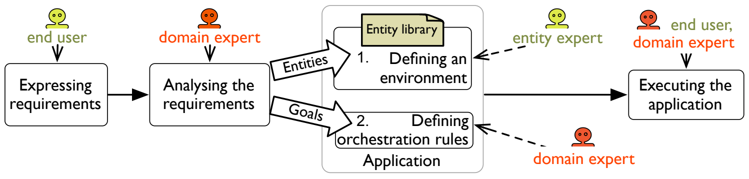


Figure 9: Overall development process of a Pantagruel application

The visual editor is parameterized with respect to an environment description, loaded as an XML file. Doing so enables to guide the domain expert in the creation of rules. This process is decomposed in conformance with the Pantagruel visual paradigm: first, the domain expert selects the entities relevant to the application to be defined (Figure 10). To do so, he selects the entity tool in the palette, which opens a contextual window listing the entity classes and their entity instances, available in the target environment description. The domain expert can first define a program that manipulates specific entities. To generalize rules, he can replace specific entities with an entity class, by simply selecting the generic entity, as specified by the `ANY` keyword in the pop-up window.

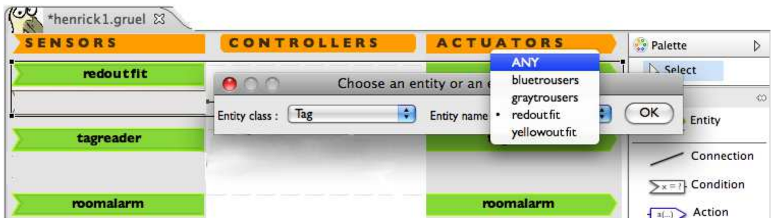


Figure 10: Selecting the relevant entities for the application

Second, he extracts context information from the selected entities (Figure 11). To guide the domain expert, another pop-up window lists the available attributes within a section, as well as their possible values, according to the data types defined in the concrete environment. In doing so, the programming process prevents syntax and type errors by contruction.

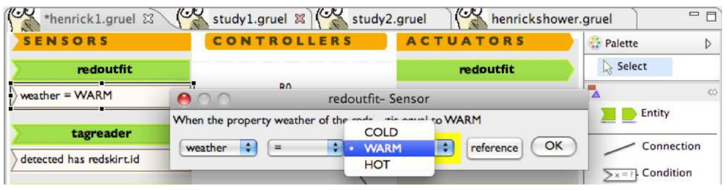
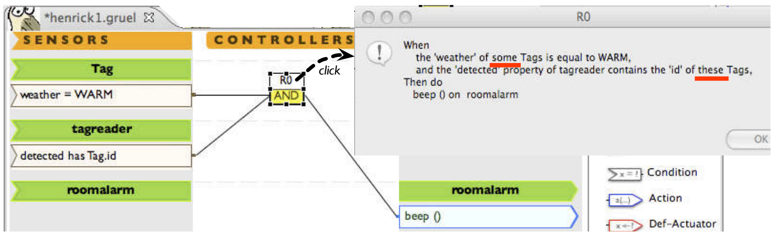


Figure 11: Selecting the relevant context information among the available attributes

Third, he selects the actions that achieve the expected behavior from a pop-up window, similar to the attribute window. Finally, the domain expert connects the conditions and actions with a controller selected in the palette; a phrasing of the rule appears on the controller pop-up window (Figure 12).



```
When
  the 'weather' of some Tags is equal to WARM,
  and the 'detected' property of tagreader contains the 'id' of these Tags,
Then do
  beep () on  roomalarm
```

Figure 12: Connecting context information and actions

7. Towards an expressiveness study

A first step towards evaluating the expressiveness of Pantagruel is to study its ability (1) to model a range of entities that are found in the home automation domain, and (2) to model a range of applications related to home automation. We now present this study.

7.1. Study context

To conduct our study, we have decomposed the requirements for applications of the home automation domain into increasingly specific goals This decomposition has resulted into elementary goals. For example, the “managing energy” requirement was decomposed into subgoals. One of them was “controlling lights”, which itself was further decomposed into three subgoals, “detecting presence”, “measuring luminosity”, “adjusting light intensity”.

This decomposition process exposes the context information and the functionalities needed to express a range of applications in a given area. In doing so, each elementary goal is readily mapped into one or more objects, selected from a library of entities, according to the functionalities provided by these objects. For example, “measuring luminosity” was mapped to a light sensor. In our study, this type of mapping resulted in a taxonomy for each application area.

Application areas	**Taxonomy**		
	classes	attributes	methods
Light/music	10	10	5
Plantcare	11	9	5
Assisted living	15	15	9
Intercom	9	19	11
School information	10	16	7
School security	10	10	6
Meeting management	15	19	15

Table 1: Taxonomy case studies

Application area	Orchestration application
Light and music management	follow-me light
	dim lightning control
	radio configuration
	follow-me music
Plantcare	humidity control
	assisted manual lightning
	automated lightning
Assisted living	wake-up assistance
	clothing assistance
	shower assistance
	leaving home on time
	task management
Intercom	presence manager
	phone call control
	follow-me conversation
Others (school information and security, meeting management)	door manager
	fire manager
	schedule manager
	remote display manager
	intrusion manager

Table 2: Orchestration application case studies

Each taxonomy definition consisted of 9 to 15 entity classes as reported in Table 1. In total, the three areas involved 50 entity classes. Some of them were shared among the areas (*e.g.,* the presence detectors, the lights, and the calendars). For each area, we developed 3 to 6 applications, each consisting of 2 to 7 rules. Out of 20 applications, 15 of them have been deployed on the DiaSim simulator. These applications are reported in Table 2. We consider school management as a generalization of the home automation domain to large-scale buildings. Applications reported for the assisted living area were demonstrated at Percom'10 [21].

7.2. Entity design space

Writing the taxonomy definitions for the domain of home automation exercised key dimensions of our language expressiveness. Each dimension addresses the following facets of entities: physical and virtual sensing, configuring and monitoring, and processing. An entity class may be an arbitrary combination of these facets.

Physical and virtual sensing. Home automation applications highly depend on the periodical changes of the surrounding environment. For example, an application that manages energy needs to react to temperature changes. The varying nature of an environment is captured by physical and virtual sensors. Unlike physical sensors, virtual sensors capture a status related to a computer activity. For example, an instant-messenging client detects whether the owner of a computer is online or away. The `Calendar` entity class of our example application, sending events according to a day planning, can also be modeled as a virtual sensor. Such varying information can be represented by volatile attributes, whether coming from physical or virtual devices.

Configuring and monitoring. A fundamental requirement of home automation is to be able to configure an environment according to sensed information and user preferences. We call components performing such activities, configuration components. For example, consider a music management application deployed in a house that relies on a radio installed in every room. Each radio is configured according to the preferences of the user entering the room. The configurable parts of this entity class are modeled as write attributes, such as a `musicGenre` attribute on `Radio`. The preferences of a person can be gathered in a specific entity class (*e.g.,* `PersonProfile`), defining constant attributes (*e.g.,* a `preferredMusic` attribute).

Entity classes monitoring actions also declare specific write attributes. In the shower example, the `run` method of `MixingValve` is monitored by adding a `showerStatus` write attribute on the `MixingValve` methods.

Processing. When they involve numerous entities, home automation applications often consist of gathering and processing data sources. Such processing is typically defined in terms of input/output dependencies. To illustrate this approach, let us consider an energy management application. This application needs to configure heaters by processing sensed temperatures and other sources such as time and room occupancy. An example of such component is the `HeatManager` entity class that collects the people locations and working hours, and determines a target temperature for the heaters. The target temperature is captured by a write attribute (*e.g.,* `avheat`); the processing is defined by the `calculateHeat` method, declaring the `avheat` in its side effects.

Summary. In this section, we identified three facets of entities, covering a range of application areas, captured by our taxonomy language. Our preliminary expressiveness study provides an illustration of the relevance of the Pantagruel taxonomy abstractions (*e.g.,* attribute kinds and side-effect declarations).

7.3. Orchestration language expressiveness

The taxonomy language provides the necessary building blocks to model a range of home automation entities. This is put to practice by the following analysis that conforms with the iCAP process [3] and covers Schilit's four applications categories [22].

Proximate selection. Proximate selection applications enable to access resources according to their proximity. The filtering mechanism of the Pantagruel orchestration layer generalizes this selection process to any context information beyond location: it provides a concise means to prototype a range of applications requiring resource selection. For example, it is used to select a tagged outfit according to the current weather (see the `Tag` section of Figure 7). Such applications involve at least sensing components, as defined in our entity design space (see Section 7.2).

Automatic contextual reconfiguration. Applications of this category can detect the dynamic change of resources, that is, appearance or disappearance of components, or new connections between components. These applications include the follow-me applications, where the user interface of the application can follow the user as he moves around [23]. For example, a light follow-me application written in Pantagruel is displayed in Figure 8. Because Pantagruel leverages an entity discovery mechanism provided by its underlying platform [24], applications can manage appearing and disappearing entities. Such applications combine configuring and sensing components.

Contextual information and commands. Applications in this category execute commands parameterized by the contextual information captured by entities. We support these applications through the use of typed attributes and parameterized methods. Attributes, representing the context information may serve as parameters on the entity methods. For example, consider an application such that when a person enters a room, his smart phone displays information customized with respect to his new location; this operation would correspond to an actuator invoking a `displayLocation` method on a smart phone entity class, with the location as an argument. Such applications combine configuring and processing components.

Context-triggered actions. Applications that define such actions extend the previous category by allowing context-triggered commands to be "*invoked automatically according to previously specified rules*" [22]. In the Pantagruel orchestration layer, this feature is enabled by the write attributes, as defined in monitoring components. Specifically, a write attribute depicts a dependency between two rules when (1) it is accessed in the sensor part of a rule, and (2) it is modified in the actuator part (*i.e.,* an assignment or a side-effecting method) of another rule. This rule dependency is illustrated in our working example (Figure 6): on the one hand, the `R5` rule includes in its sensor part the `todolist` attribute of the `TimeTracker` entity class; on the other hand, the `R3` and `R4` rules set the value of `todolist` in their actuator part, through the `init` method invocation; therefore, triggering `R5` depends on the previous execution of either `R3` or `R4`.

Summary. Through this study, we experimented and illustrated the ability of Pantagruel concepts and paradigm to embrace the four categories of applications proposed by Schilit *et al.*, in conformance with their meaning.

8. User Study

We evaluated whether non-programmer users could create orchestration rules for a predefined taxonomy, using the Pantagruel visual programming environment. In particular, we evaluated the following aspects of our language:

- *Intuitiveness of the Pantagruel visual paradigm.* We evaluated whether a development process driven by the entities was suitable for a non-programmer.
- *Accessibility of the Pantagruel abstractions.* We evaluated whether a non-programmer could create class-based orchestration rules for a range of entities.

We conducted a user study in two phases. The first phase involved 12 participants, and aimed to collect feedback for improving the Pantagruel tool. The second phase involved 6 participants, and aimed to assess an improved version of the Pantagruel tool, as reported in Section 9.3. Both phases were conducted with the same study context and process. We now describe the material of this user study.

8.1. Study context

The user study was composed of the following material: a pre-questionnaire aimed to collect the computer science background (programming language or visual software) of the participants. After the participants were presented a short tutorial of our tool, we observed them expressing a set of 6 rules (see Tables 3 and 4) using our tool. Finally, a post-questionnaire, based on the System Usability Scale [25], gathered their overall acceptance of the Pantagruel tool. As shown in Figure 5 of Section 5, two screens were set up for each session of our user study: a screen displayed the Pantagruel editor; the other screen displayed a 2D-model of a house. It served as a visual support for choosing the entities to orchestrate. Initially, the Pantagruel visual editor was empty.

To accompany each participant during the study session, we recruited two test observers in our research group. One of them helped the participant if he felt confused for too long (*i.e.,* hesitating for more than 3 minutes) or if he had problems with the graphical interface (*e.g.,* how to use the tool palette). The other observer noted down the behavior of the participant (*e.g.,* an invalid user-interaction with the editor, or when he thought aloud) and the interventions of the first observer. Interventions mainly consisted in showing how to interact with the editor, and showing elements of the solution (*e.g.,* pointing at the `ANY` keyword on the entity pop-up window).

8.2. Participants

We evaluated our system with 18 volunteer students (age range from 18 to 21, 4 females, 14 males) who majored in maths and physics in high school, and just started their undergraduate studies. At that time, they had not attended any course related to Pantagruel, neither to home automation nor programming. According to the pre-questionnaire, the students had little or no exposure to programming languages or visual editors. Specifically, only 8 students had a small experience of visual editors for music/video/graphism in undergraduate school, and 2 students had some programming experience (PHP and Visual Basic). However, whether or not they had prior exposure to programming did not impact their approach to the Pantagruel editor. Additionally, none of the students were familiar with the idea of home automation.

N°	Simple rule sentences	G1	G2
R1	If Bruno or Anna is in the living-room, put the music genre of the radio to JAZZ.	5.9	5.0
R2	If it is daylight outside, switch off the outside light.	4.7	4.2
R3	If Anna enters the kitchen, switch on the kitchen lights.	7.7	4.3

Table 3: Simple rule sentences and average rule completion time (group 1 and 2), in minutes

N°	Advanced rule sentences	G1	G2
R4	If a person enters a room, then switch on the lights of this room.	8.7	4.7
R5	If a person is in a room, then change the music genre of the radio located in this room to this person's favorite genre.	8.5	6.0
R6	If nobody is in a room, then switch off the lights of this room.	8.7	6.7

Table 4: Advanced rules sentences and average rule completion time (groups 1 and 2), in minutes

8.3. Study Sessions

Each participant came to our lab for a 60-minute session. Participants were first presented a 15-minutes tutorial (included in the session). The tutorial started with an introduction to the domain of home automation. Then, we demonstrated the usage of the Pantagruel editor through an example that covered the language concepts.

Finally, sessions were conducted individually, using the Pantagruel editor parameterized by a concrete environment. This environment was composed of various entities: lights (*i.e.,* defined with a `Light` entity class), person tags (*i.e.,* `PersonLocalizer`), light sensors (*i.e.,* `LightSensor`), and radios (*i.e.,* `Radio`).

Each participant was asked to create 6 rules with varying complexity for orchestrating these entities. Specifically, they were asked to define each rule without guidance but by carefully following the development process as described in Section 6. We classified these rules in 2 sets shown in Tables 3 and 4. The first set of rules aims to assess the visual paradigm of Pantagruel, and necessitates to reason about specific entities. The second set of rules aims to evaluate Pantagruel abstractions, and necessitates to use entity classes, thus introducing a level of abstraction. The `R4` rule generalizes the `R3` rule; it requires to (1) create a filtering sensor on an entity class (*i.e.,* `Light`), (2) express a dependency between 2 classes of entities (*i.e.,* `PersonLocalizer` and `Light`, combined with a condition on their `location` attribute). The `R5` rule requires to define an action whose parameter is an attribute of an entity class (*i.e.,* the `preferredmusic` attribute of `PersonLocalizer`). Finally, the `R6` rule requires to express a condition that has to match on *all* entities of a given class (*i.e.,* `PersonLocalizer`).

9. User Study Results

Let us now expose the opinions of participants, gathered by our post-questionnaire. We will then analyze the result of the study sessions with regard to the task performance of the first set of participants. Finally, we report the lessons learned from the second set of participants.

9.1. Subjective results

To evaluate the overall opinion of the Pantagruel editor, we used the System Usability Scale questionnaire [25]. It is a ten-statement questionnaire, each having a five-point scale ranging from *strongly disagree* to *strongly agree*. The questionnaire score ranges from 0 to 100. The global score of our questionnaire, including the 18 students, is 70 (Standard Deviation=11.8, min=47.5, max=90.0). This rate ranks the usability of Pantagruel as acceptable according to Bangor *et al.* [26, 27]. The separate scores were 68.5 and 73.8 for the first and the second set of students, respectively. The latter score illustrates the efficiency of the improved version, which preserved the visual concepts of the language.

Specifically, 15 participants reported that they would like to use frequently this tool to prototype other home automation applications (agree or strongly agree). Though nobody disagreed with the statement "*I thought the system was easy to use*", 7 of them were undecided about this statement (*don't know* answer). 1 participant did not "*feel very confident using the system*", against 12 of them who either agreed or strongly agreed with this statement. To sum up, opinion for each statement was different than *don't know*, except for the statements (1) "*I think I would need the support of a technical person to be able to use this system*" and (2) "*I imagine that most people would learn to use this system very quickly*", for which as many people agreed and disagreed.

The questionnaire was completed with open remarks. It is worth noting that a participant found it "*fun and easy to program a house*". However, 2 of the participants reported that they needed a 2 or 3 hour-training before feeling confident with the Pantagruel editor.

9.2. First phase: analysis of task performance

We now report our analysis on the first phase of our study, of the evaluation of the visual paradigm with the first set of rules (Table 3), and of the language abstractions with the second set of rules (Table 4).

9.2.1. Intuitiveness of the visual paradigm

Successes. For the first rule, we reminded individually each participant through a two-minute tutorial how to use the user interface (using different entities than the ones required for the rule). All of them then naturally applied the rule creation process associated with the visual paradigm; it consisted of selecting and placing the relevant elements from the editor palette to the editor view. As an illustration, 10 of the 12 defined the `R2` rule in 3 to 6 minutes; 7 of them defined the `R3` rule without an intervention in 3 to 8 minutes.

Drawbacks. For the first rule (`R1`), 8 of the 12 participants required help with the user interface to position the appropriate graphical elements on the editor view. For example, one participant attempted to directly connect the left part of a section with an actuator using the wire element.

9.2.2. Accessibility of Pantagruel abstractions

Successes. The `R3` rule (Table 3) involved, but did not require, the use of entity classes. 4 participants naturally used a class to define it (with the sensor `location = KITCHEN` on the `Light` section). The `R4` rule (Table 4) is similar to the `R10` rule of Figure 8. For this rule, half of the participants needed to be reminded of the notion of dependency between classes; such a dependency is depicted by both sensors of the `R10` rule, related to each other by the `AND` controller and the use of the `MotionSensor` class. Then, 8 participants were able to create the `R4` rule, and 10 for the `R5` rule, without any help.

Drawbacks. As illustrated by the Tables 3 and 4, the second set of rules required more time than the first one. A participant did not feel confident with the filtering effect of conditions on the `Light` entity class of the `R4` rule: although he intuitively created a correct rule, he was not convinced of its meaning while reading it. Finally, all participants required the experimenter to explain how the NONE quantifier, required for the R6 rule, changes the meaning of its sensors. At the time of the study, the quantifier was placed on the sensor part.

9.3. Second phase: lessons learned and improvements

From our first study, we collected the feedback of the participants. This feedback helped us to learn lessons regarding the improvement of the Pantagruel tool. We now report these lessons, as well as the results of a preliminary improvement, from which the second set of students benefited.

Lessons learned. Pantagruel visual paradigm seems easy to grasp, although its entity-centric structure is not as natural as we expected. Its intuitiveness may be increased with a stronger connection between the 2D-model of the environment and the Pantagruel editor to emphasize its entity-centric paradigm. For example, 2 of the participants at first wanted to drag-and-drop the entities to orchestrate from the 2D-model to the Pantagruel editor. This seems an interesting software improvement of our tool. Such an improvement would enable the user to create orchestration rules through *direct manipulation* [28] of the entities via the 2D-model, instead of manually creating sections in the Pantagruel editor.

Preliminary improvement. Pantagruel abstraction concepts require further training or user-interface support. This observation motivated us to improve the Pantagruel tool with a rough natural-language translation of rules that pops-up when double-clicking on a controller element. This translation exposes the filtering effect of the rule conditions and the dependencies between classes by combining the words *these*, *any* or *all*. Such a translation is illustrated in Figure 12. As a result, 5 of the 6 participants who benefited from this assistance completed the rules R1 to R5 without any help, 2 minutes faster than the others. The difficulties encountered on the ALL/NONE quantifier leaded us to represent it as shown in this paper, making explicit its application to a class. This improvement was approved by the test observers. However, further evaluation is necessary for validation.

9.3.1. Summary

The overall observations and the acceptable subjective results show that the Pantagruel tool is accessible to novice programmers. However, the time needed by the participants to create rules showed that the Pantagruel abstractions are not intuitive enough: compared to iCAP [3], which also has a rule-based paradigm, participants needed twice as much time to resolve similar rules (*e.g.,* R4). Pantagruel and iCAP are further compared in Section 10, and some usability improvements are mentioned in Section 11.

10. Related Work

Our contributions can be measured with respect to four dimensions for developing orchestration applications. Let us examine existing approaches sharing these dimensions.

10.1. Taxonomy-driven approaches

Taxonomy-inspired approaches [29, 30, 31, 32, 33, 16, 17] have been used to address a range of areas, such as pervasive computing, web services, graphical computing, and robot control. For example, Olympus [32] is a programming model developed over a pervasive computing middleware, enabling to define an ontology of services to be orchestrated. However, Olympus does not provide an interaction model to specify the context information exposed by these services.

In web services, WSDL [33] is a language for describing the interaction interface of web services. It declares the functionalities and the data provided by a service. However, it does not provide any information about the data beyond its type. In contrast, Pantagruel entity classes expose the nature of the context information captured by their attributes, whether external, applicative or constant. This strategy helps the developer understand entities and their context of usage.

In the field of robot control, Prograph CPX [16] enables to define a hierarchy of object classes that compose a robot. Similarly, AgentSheets [17, 9] is a toolkit for defining domain-oriented environments composed of agents, interacting with each other, or with a user. AgentSheets enables to model a domain by defining agent classes, and tasks defining the behavior of agents. However, neither Prograph CPX nor AgentSheets provides any information about the nature of the data that can be manipulated in a program or the effect of a method invocation. In contrast, the entity interaction interfaces of Pantagruel offer useful information that guide the development and the verification of applications.

10.2. Visual rule-based orchestration languages

The rule-based paradigm is widely spread in visual programming fields dedicated to robot control (*e.g.,* to orchestrate the sensors and effectors of a robot) [8, 9, 34] or game, agent-based programming (*e.g.,* to orchestrate agents or moving objects) [35, 17, 19].

To avoid rule explosion, AgentSheets [17] proposes a mechanism to easily specify *analogies* [36] between the agents, resulting in the description of concise rules for a range of agents. In contrast, Pantagruel leverages the class abstraction, enabling rules to be defined on a range of entities, instead of specific entities.

KidSim [35] is another visual rule-based language that enables children to create games by defining interactions rules between agents and/or their properties, through the use of pictures that represent the agents. However, KidSim and AgentSheets do not offer a visual structuring of rules centered on the entities. In contrast, our three-panel, section-based representation can be naturally combined with selection mechanisms to visualize subsets of rules according to various criteria: entities, sensors, or actuators.

Our paradigm extends the rule-based paradigm of the Blender Game Engine [19]. While Blender is prototype-based, the Pantagruel orchestration language relies on classes. Our approach allows programs to be reusable over a range of concrete environments.

Ladder Diagrams is a rule-based language [37] to program logic controllers. It is limited to manipulate low-level data, making large programs hard to read or analyze [38]. In contrast, our approach enables the user to manipulate high-level information captured by entities, thus guiding the development of orchestration rules.

Other paradigms have been proposed to develop applications for orchestrating objects. Examples include the puzzle paradigm, used by Scratch [39], and the storytelling paradigm, used by Alice [40]. However, neither Scratch nor Alice provide domain-specific abstractions to represent the sensing or actuating nature of objects. This situation makes it difficult to examine the context information used for a given program. In contrast, the visual structure of Pantagruel facilitates the reasoning about the context and actions used in a program.

10.3. Visual paradigms for programming home automation applications

There exist visual approaches to develop home automation applications. These approaches have been shown to provide an intuitive representation for the orchestration logic, bridging the gap between end-user requirements and a program. However, to the best of our knowledge, none of the tools reported in the home automation literature proposes an open-ended approach to developing applications, enabling an extensible set of entities to be orchestrated.

For example, CAMP [1] is a rule-based tool which uses the magnetic poetry metaphor to define sentences by visual word composition. However, its vocabulary is restricted to a specific area of home automation. Another related tool is iCAP [3], it provides a fixed set of generic entity classes: objects, activities, time, locations and people. Its elements are visual representations of these classes. Composition of conditions is achieved by visual arrangement of rectangles on the screen. However, iCAP does not provide a uniform approach to expressing rules over a group of entities besides a group of persons. In contrast, our approach provides syntactic support to define filters over all instances of an entity class, as well as specific entities, enabling generic orchestration rules to be described.

The visual language VisualRDK [41] contrasts with the previous tools in that it targets a range of programmers, novice as well as experienced ones. VisualRDK is a programmatic approach, offering language-inspired constructs such as processes, conditional cascades and signals. These numerous visual constructs mimic conventional programming, without specifically targeting the domain-specific aspects of home-automation orchestration logic. Pantagruel differs from this approach in that rules are driven by (1) the entities and (2) connectors to orchestrate them.

Other visual prototyping tools like the rule-based language OSCAR [7] and Jigsaw [2], target domestic spaces and propose an approach to discovering, connecting and controlling services and devices. However, they offer limited expressiveness to access the functionalities of entities.

10.4. End-user development for home automation

Other approaches for enabling end users to "program" their own homes have been proposed to reduce the end-user programming burden while proposing rich applications based on abstractions. For example, the Media Cubes language [5] enables to program applications by composing cubes, representing abstract operations. This approach is based on a cognitive model [42] that enables the end user to get familiar with abstractions. Our approach could benefit from this model to improve the usability of entity classes. MAPS [43] is another approach to design assisted-living applications using a design by composition approach. However, it is limited to handheld prompter applications.

Our approach follows the lines of end-user software engineering proposed by Mørch *et al.* [6]. Specifically, Pantagruel is based on a compositional approach, where the user connects components (*i.e.,* , sensors, actuators, and controllers) together to form an application. Moreover, constraints are integrated in the Pantagruel visual editor to guarantee correct composition and connection of the visual elements.

11. Conclusion and Future Work

Conclusion. Home automation concerns 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 home automation 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.

We explored the expressiveness of our taxonomy-based approach by defining orchestration applications for a range of application areas that go beyond home automation. We developed applications for these areas, and tested most of them on a home automation simulator called DiaSim [10]. The simulator enabled us to explore the expressiveness of Pantagruel programs in areas that would otherwise be out of reach. These studies resulted in an entity design space that is covered by the Pantagruel taxonomy language. We further studied the expressiveness of the Pantagruel orchestration language, parameterized by a taxonomy.

We conducted a usability study of Pantagruel orchestration language. This study has showed that it is accessible and intuitive to novice programmers. Still, improvements are needed to increase the usability of Pantagruel abstractions, as well as the efficiency of users while creating rules.

Future work. Our user study has showed interesting research directions towards improving usability. One of them would be to seamlessly integrating Pantagruel in the 2D editor, enabling rules to be created by directly selecting and connecting together the entities represented in the 2D model of the environment. Doing so could later lead to a programming-by-example development process [44]. This research direction could also leverage recent advances in end-user software engineering [45]. These works introduce various techniques (*e.g.,* interactive testing and adapted debugging tools) to guide the end-user in writing correct applications using visual tools. For example, one could provide support to visually render the execution process of a rule subset prior to a complete program execution, provided a user-defined input of test values.

A direction towards end-user usability would be to define various layers over Pantagruel, making it more user-friendly (*e.g.,* providing a natural language-based programming layer such as CAMP [1]) and more adapted to the problem vocabulary of a domain expert (*e.g.,* a caregiver, who would build assisted-living applications, is more comfortable when reasoning by means of “actions” rather than entities). This approach could still leverage the DiaSpec [24] platform while offering area-specific programming metaphors.

Finally, we have developed various analyses for Pantagruel programs to guarantee properties such as safety, liveness, and non-interference of orchestration rules. These verifications could be integrated in the Pantagruel development environment, to drive the developer in writing correct orchestration logic.

[1] K. N. Truong, E. M. Huang, G. D. Abowd, A magnetic poetry interface for end-user programming of capture applications for the home, in: 6th Int’l Conference on Ubiquitous Computing (UbiComp), Springer, 2004, pp. 143–160.

[2] J. Humble, A. Crabtree, T. Hemmings, K.-P. Åkesson, B. Koleva, T. Rodden, P. Hansson, "Playing with the Bits" user-configuration of ubiquitous domestic environments, in: 5th Int’l Conference on Ubiquitous Computing (UbiComp), Vol. 2864, Springer, 2003, pp. 256–263.

[3] A. K. Dey, T. Sohn, S. Streng, J. Kodama, iCAP: Interactive prototyping of context-aware applications, in: 4th Int’l Conference on Pervasive Computing (Pervasive), Springer, 2006, pp. 254–271.

[4] Y. Li, J. I. Hong, J. A. Landay, Topiary: a tool for prototyping location-enhanced applications, in: 17th Symposium on User Interface Software and Technology (UIST), ACM, 2004, pp. 217–226.

[5] A. F. Blackwell, End-user developers at home, Commun. ACM 47 (2004) 65–66.

[6] A. I. Mørch, G. Stevens, M. Won, M. Klann, Y. Dittrich, V. Wulf, Component-based technologies for end-user development, Commun. ACM 47 (2004) 59–62.

[7] M. W. Newman, A. Elliott, T. F. Smith, Providing an integrated user experience of networked media, devices, and services through end-user composition, in: 6th Int’l Conference on Pervasive Computing (Pervasive), Springer, 2008, pp. 213–227.

[8] J. J. Pfeiffer Jr., Altaira: A rule-based visual language for small mobile robots, Journal of Visual Languages and Computing 9 (2) (1998) 127–150.

[9] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, A. Repenning, Legosheets: A rule-based programming, simulation

and manipulation environment for the leg0 programmable brick, in: Proceedings of the 11th Int'l IEEE Symposium on Visual Languages (VL '95), 1995, pp. 172–179.

[10] J. Bruneau, W. Jouve, C. Consel, Diasim, a parameterized simulator for pervasive computing applications, in: Proceedings of the 6th Int'l Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous'09), ICST/IEEE, Toronto, CAN, 2009.

[11] A. Svensk, Design for cognitive assistance, Certec, 2001.

[12] S. P. Carmien, Socio-technical environments supporting distributed cognition for persons with cognitive disabilities, Ph.D. thesis, University of Colorado at Boulder, Boulder, CO, USA, aAI3239390 (2006).

[13] S. Carmien, M. Dawe, G. Fischer, A. Gorman, A. Kintsch, J. F. Sullivan, JR., Socio-technical environments supporting people with cognitive disabilities using public transportation, ACM Trans. Comput.-Hum. Interact. 12 (2005) 233–262.

[14] S. A. White, Business Process Modeling Notation, V. 1.0., `http://bpmi.org` (May 2004).

[15] Microsoft Corporation, The Microsoft Visual Programming Language, `http://msdn.microsoft.com/en-us/library/bb483088.aspx`.

[16] S. B. Steinman, K. G. Carver, Visual Programming with Prograph CPX, Manning Publications Co., Greenwich, CT, USA, 1995.

[17] A. Repenning, Agentsheets: a tool for building domain-oriented dynamic, visual environments, Ph.D. thesis, University of Colorado at Boulder, Boulder, CO, USA (1993).

[18] D. C. Smith, A. Cypher, L. G. Tesler, Novice programming comes of age, Commun. ACM 43 (3) (2000) 75–81.

[19] J. van Gumster, Blender as an educational tool, in: SIGGRAPH Educators Program, 2003, p. 1.

[20] C. Neumann, R. A. Metoyer, M. M. Burnett, End-user strategy programming, Journal of Visual Languages and Computing 20 (1) (2009) 16–29.

[21] Z. Drey, C. Consel, A visual, open-ended approach to prototyping ubiquitous computing applications, in: Workshops of the 8th IEEE Int'l Conference on Pervasive Computing and Communications, 2010, pp. 817–819.

[22] B. Schilit, N. Adams, R. Want, Context-aware computing applications, in: Proceedings of the Workshop on Mobile Computing Systems and Applications, IEEE, 1994, pp. 85–90.

[23] A. Harter, A. Hopper, P. Steggles, A. Ward, P. Webster, The anatomy of a context-aware application, in: MobiCom '99: Proceedings of the 5th annual ACM/IEEE Int'l conference on Mobile computing and networking, ACM, New York, NY, USA, 1999, pp. 59–68.

[24] D. Cassou, B. Bertran, N. Loriant, C. Consel, A generative programming approach to developing pervasive computing systems, in: GPCE'09: Proceedings of the 8th Int'l Conference on Generative Programming and Component Engineering, ACM, Denver, CO, USA, 2009, pp. 137–146.

[25] J. Brooke, J. sus-a quick and dirty usability scale, Jordan, P., Thomas, B. and Weerdmeester, B. (eds.). Usability Evaluation in Industry.

[26] J. R. Lewis, J. Sauro, The factor structure of the system usability scale, in: Proceedings of the 1st Int'l Conference on Human Centered Design, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 94–103.

[27] A. Bangor, P. T. Kortum, J. T. Miller, An empirical evaluation of the system usability scale, Int'l Journal of Human-Computer Interaction 24 (6) (2008) 574–594.

[28] M. M. Burnett, D. W. McIntyre, Visual Programming, John WIley & Sons Inc., 1999.

[29] F. Paterno, Model-based design of interactive applications, Intelligence 11 (4) (2000) 26–38.

[30] D. Fogli, L. P. Provenza, A meta-design approach to the development of e-government services, Journal of Visual Languages and Computing 23 (2) (2012) 47 – 62.

[31] C. Ardito, B. R. Barricelli, P. Buono, M. F. Costabile, R. Lanzilotti, A. Piccinno, S. Valtolina, An ontology-based approach to product customization, in: Proceedings of the Third international conference on End-user development, IS-EUD'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 92–106.

[32] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, M. D. Mickunas, Olympus: A high-level programming model for pervasive computing environments, in: 3rd Int'l Conference on Pervasive Computing and Communications (PerCom), IEEE, 2005, pp. 7–16.

[33] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web service definition language (`http://www.w3.org/TR/wsdl`), Tech. rep., W3C (March 2001).

[34] P. T. Cox, C. C. Risley, T. J. Smedley, Toward concrete representation in visual languages for robot control, Journal of Visual Languages and Computing 9 (2) (1998) 211–239.

[35] D. C. Smith, A. Cypher, J. C. Spohrer, Kidsim: Programming agents without a programming language, Commun. ACM 37 (7) (1994) 54–67.

[36] A. Repenning, Bending the rules: steps toward semantically enriched graphical rewrite rules, in: Proceedings of the 11th Int'l IEEE Symposium on Visual Languages (VL '95), IEEE, Washington, DC, USA, 1995, p. 226.

[37] D. Pessen, Ladder-diagram design for programmable controllers, Automatica 25 (3) (1989) 407–412.

[38] S. S. Peng, M. C. Zhou, Ladder diagram and petri-net-based discrete-event control design methods, Systems, Man, and Cybernetics IEEE Trans. on 34 (4) (2004) 523 –531.

[39] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, Y. Kafai, Scratch: programming for all, Commun. ACM 52 (2009) 60–67.

[40] C. Kelleher, R. Pausch, Using storytelling to motivate programming, Commun. ACM 50 (2007) 58–64.

[41] T. Weis, M. Knoll, A. Ulbrich, G. Muhl, A. Brandle, Rapid prototyping for pervasive applications, IEEE Pervasive Computing 6 (2) (2007) 76–84.

[42] A. Blackwell, First steps in programming: a rationale for attention investment models, in: Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on, 2002, pp. 2 – 10.

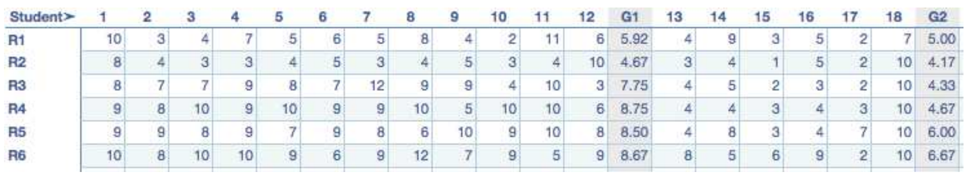
[43] S. Carmien, End user programming and context responsiveness in handheld prompting systems for persons with cognitive disabilities and caregivers, in: CHI '05 extended abstracts on Human factors in computing systems, ACM, 2005, pp. 1252–1255.

[44] D. C. Smith, A. Cypher, L. Tesler, Programming by example: novice programming comes of age, Commun. ACM 43 (3) (2000) 75–81.

[45] M. M. Burnett, C. Cook, G. Rothermel, End-user software engineering, Commun. ACM 47 (9) (2004) 53–58.

Appendix

The following table shows the detailed time spent by each participant for defining the rules of Tables 3 and 4 of Section 8.



Student>	1	2	3	4	5	6	7	8	9	10	11	12	G1	13	14	15	16	17	18	G2
R1	10	3	4	7	5	6	5	8	4	2	11	6	5.92	4	9	3	5	2	7	5.00
R2	8	4	3	3	4	5	3	4	5	3	4	10	4.67	3	4	1	5	2	10	4.17
R3	8	7	7	9	8	7	12	9	9	4	10	3	7.75	4	5	2	3	2	10	4.33
R4	9	8	10	9	10	9	9	10	5	10	10	6	8.75	4	4	3	4	3	10	4.67
R5	9	9	8	9	7	9	8	6	10	9	10	8	8.50	4	8	3	4	7	10	6.00
R6	10	8	10	10	9	6	9	12	7	9	5	9	8.67	8	5	6	9	2	10	6.67

Figure 13: Detailed development times (in minute) spent by the participants of our user study