Denotational Semantics of A User-Oriented, Domain-Specific Language

Julien Mercadal, Zoé Drey, Charles Consel

Denotational Semantics of A User-Oriented, Domain-Specific Language

Julien Mercadal
ASCOLA group
INRIA & Ecole des Mines
Nantes (France)
julien.mercadal@mines-nantes.fr

Zoé Drey
STIC-IDM
Lab-STICC & ENSTA Bretagne
Brest (France)
zoe.drey@ensta-bretagne.fr

Charles Consel
PHOENIX group
INRIA & University of Bordeaux
Bordeaux (France)
charles.consel@inria.fr

This paper presents the formal definition of a domain-specific language, named Pantagruel, following the methodology proposed by David Schmidt for language development. This language is dedicated to programming applications that orchestrate networked entities. It targets developers that are professionals in such domains as building management and assisted living, and want to leverage networked entities to support daily tasks.

Pantagruel has a number of features that address the requirements of the domain of entity orchestration. Furthermore, Pantagruel provides high-level constructs that make it accessible to developers that do not necessarily have programming skills. It has been used to develop a number of applications by non-programmers.

We show how the user-oriented programming concepts of Pantagruel are expressed in the denotational semantics of Pantagruel. This formal definition has been used to derive an interpreter for Pantagruel and to provide a basis to reason about Pantagruel programs.

1 Introduction

The realisation of domain-specific languages (DSLs) contrasts with that of general-purpose programming languages (GPLs) where generality, expressivity and power are expected. Instead, a DSL revolves around a narrow, specific application domain and its users, providing high-level abstractions and constructs tailored towards this domain and the needs of users [4, 12, 6, 7]. Thus, much time and effort must be devoted to the domain analysis and the language design. Methodologies and tools have been proposed to develop DSLs, covering the complete development life-cycle, from design [5, 13] to implementation [17, 21, 22, 23], including formalisation [16, 20].

If such methodologies are well-known in the programming-language community, they hardly have an echo in the end-user community yet. As an illustration, in their state-of-the-art paper about end-user software engineering, Ko *et. al* [15] address the challenge of enabling the end user to produce reliable programs that actually achieve his requirements, by supporting the programming task with visualization and simulation tools. However, they do not mention the need for a formal definition of an end-user language so that the supporting tools be provably consistent with the language definition. A first step towards addressing this issue is to bridge the gap between the domain analysis, taking the requirements into account, and the design of a language, providing the constructs to achieve these requirements.

In this paper, we present the formal definition of a domain-specific language, named Pantagruel [10, 11], following the methodology proposed by David Schmidt for language development [20]. This language is dedicated to programming applications that orchestrate networked entities. It targets end users that are professionals, in areas such as building management and assisted living, and leverages networked

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 229–249, doi:10.4204/EPTCS.129.14

entities to support daily tasks. For example, security guards are assisted by an anti-intrusion system that coordinates motion detectors and surveillance cameras to signal the presence of an intruder on monitoring screens. For another example, a range of assistive devices, like time trackers, task prompters, and motion sensors, are available to support caregivers in assisting people with disabilities in their daily life. Pantagruel offers a number of features that address the requirements of the domain of networked entity orchestration, while making it accessible to users that do not necessarily have programming skills. Specifically, Pantagruel provides two language layers, one for specifying interfaces from which entities are manipulated, and another for orchestrating entities. The orchestration layer offers high-level operators taking advantage of the information defined in the specification layer to discover, select concrete entities, and interact with them via their interfaces.

This paper gives a definition of Pantagruel in the form of a denotational semantics, exhibiting the domain-specific features of this language. In particular, this formal definition makes the key concepts of the orchestration of networked entities (*i.e.,* entity discovery and interactions) explicit. It also brings out some traditional programming language concepts (*e.g.,* the notion of memory and loop, or variable assignment), which have been abstracted away from the developer.

Outline

The rest of this paper is organized as follows. Section 2 presents the specificities and the key concepts of the domain of the orchestration of networked entities. Section 3 gives a tour of our DSL, Pantagruel, illustrated by a working example of a building-automation application. Section 4 motivates the use of the denotational style for our language. Sections 5 and 6 describe the syntax and the denotational semantics of the specification layer and of the orchestration layer of Pantagruel, respectively. Conclusion and future work are given in Section 7.

2 Orchestration of networked entities

A wide variety of networked entities, both hardware and software, are populating smart spaces that become prevalent in a growing number of areas, including supply chain management, building automation, healthcare, and assisted living. These entities have *sensing capabilities*, enabling to collect data (*e.g.,* ambient temperature, or user presence), and/or *actuating capabilities*, enabling to perform actions (*e.g.,* turn on/off an entity, or display a message). Additionally, they are characterized by *attributes* (*e.g.,* a location or a status). Applications need to be developed to exploit the capabilities provided by these entities. Such applications determine the actions to be triggered according to sensed data and values of entity attributes. We refer to such applications as *orchestration applications*. The development of orchestration applications is challenging, requiring to cope with heterogeneity and dynamicity of networked entities.

Heterogeneity. The entities of a smart space are off-the-shelf software components and devices that use a variety of communication protocols and rely on intricate distributed systems technologies. The heterogeneity of these entities and the intricacies of underlying distributed technologies tend to percolate in the code of the orchestration logic, cluttering it with low-level details. To facilitate the development of the orchestration logic, it is necessary to raise the level of abstraction at which entities are manipulated. To do so, an abstraction layer between the entity implementation and the orchestration logic must be defined. We refer to such a layer as a *specification layer*, enabling to describe entities according to their sensing and actuating capabilities and their attributes. This layer is coupled with the *orchestration layer* enabling to write an orchestration application with respect to the entity descriptions from the specification

layer. Specifically, an orchestration application consists of a set of rules that specify which actions on entities need to be performed (the right part of rules) when some data are sensed by entities (the left part of rules).

Dynamicity. An application may interact with a changing set of entities because they may become (1) available after the application is deployed, (2) unavailable due to malfunction (*e.g.,* power loss), or (3) unreachable due to a network failure. The code of the orchestration logic should manage these variations at a high level, abstracting over current entities of a given smart space. To do so, we define an abstraction for interacting with a set of entities sharing some capabilities and attributes, regardless of the current entities of the smart space. We refer to such an abstraction as an *interface*. An interface groups together a set of sensing and actuating capabilities and attributes to which entity implementations must conform. Interfaces may be organized hierarchically, enabling them to inherit capabilities and attributes.

We have built a domain-specific language, named Pantagruel [10, 11], for developing orchestration applications using the methodology for language development proposed by Schmidt [20]. Pantagruel consists of two language layers: a specification layer for describing the entity interfaces, and an orchestration layer for describing the orchestration logic. The syntax and denotational semantics of both layers are presented in the following sections. Notably, Pantagruel relies on a pervasive computing platform, named DiaSuite [3], for dealing with the variety of underlying, low-level technologies.

3 Tour of Pantagruel

The Pantagruel language is introduced using a working example: a building-automation application. This application is responsible for switching on/off lights of a room when a motion is detected in a room. It also helps regulating the temperature of rooms using fans. Although rather simplistic, this working example is sufficient to illustrate the salient features of the Pantagruel language. More elaborate examples can be found elsewhere [8].

A key feature of Pantagruel is the tight coupling of the specification layer with the orchestration layer. It allows users to easily and safely develop their own orchestration applications. Figure 1 gives a specification for our working example. This specification consists of four entity interfaces and eight entity instances. For example, the `MotionDetector` interface declares a sensing capability, named `detected`, enabling to signal the presence or the absence of a motion. The `Light` interface declares an actuating capability, named `switch`, enabling to switch on/off the entity. Both interfaces have a `room` attribute, indicating the location of the entity in the building. For simplicity, the types of all the elements in this specification are either integer or boolean. Next, entity instances are defined, according to the declarations of interfaces. For example, `l10` is an entity whose implementation conforms to the `Light` interface. As a result, this entity can be switched on/off. Moreover, its `room` attribute is initialized to the value `101`.

Given a specification, an orchestration logic is then defined to orchestrate the entities. In Pantagruel, this orchestration logic consists of a set of rules, combining the sensing and actuating capabilities of entities. Figure 2 gives three orchestration rules for our working example. Rule (1) switches the lights on (referred to as `l`) in the rooms where a motion has been detected (instance of motion detector referred to as `m`). Inversely, Rule (2) switches the lights off in the rooms where no motion has been detected. This rule uses the operators **all** and **groupby** to aggregate and group together some data collected by entities. These domain-specific operators are not detailed in this paper. Finally, Rule (3) adjusts the speed of a fan when two conditions are satisfied : (1) a light in the room where the fan is located has been switched on, and (2) the temperature sensed by the `thermo` entity is equal to `30`. This rule illustrates the ability

Figure 1: A specification for a building equipped with motion detectors, lights, fans, and temperature sensors

Interfaces

```
interface MotionDetector {
      attribute room : Integer
      event detected : Boolean }
interface Light {
      attribute room : Integer
      action switch( Boolean ) }
interface Fan {
      attribute room : Integer
      action setSpeed( Integer ) }
interface TemperatureSensor {
      event temperature : Integer }
```

Entities

```
m10:MotionDetector { room : 101 }
m20:MotionDetector { room : 201 }

l10:Light { room : 101 }
l11:Light { room : 101 }
l20:Light { room : 201 }

fan10:Fan { room : 101 }
fan20:Fan { room : 201 }

thermo:TemperatureSensor{}
```

of an orchestration logic to react to the execution of an action (here, a light is switched on). To do so, each action in Pantagruel produces an *implicit event* when it is triggered. This event has the same name as the action and its type consists of the types of the action parameters. These three rules show how the language constructs enable to interact either with a particular entity (*e.g.,* the `thermo` entity in Rule (3)), or with a set of entities via their interfaces. Moreover, the dependencies between the entities within an orchestration rule are made explicit by the **with** clause.

To illustrate the behavior of a Pantagruel program, Figure 3 shows an excerpt of an execution trace for the orchestration rules given in Figure 2. This trace illustrates the behavior of our working example when a person enters the room `101` which was empty until then. It is represented as a sequence of states, where each state consists of both the values of data collected by entities and the implicit events. The transitions between states correspond either to an update of collected data, or to a rule execution. Consider Rule (1) given in Figure 2. When a person is detected in the room `101`, the `m10` entity updates its `detected` data, leading to the state `s1`. This situation then satisfies the execution conditions of Rule (1) for `m = m10`, triggering the `switch` action on the `l10` and `l11` lights. This leads to the state `s2`, which contains the updated values for the implicit event `switch` of these lights. At this state, the execution conditions of Rule (3) are verified for `thermo` (*i.e.,* `temperature` value is 30), setting the speed of the `fan10` entity. Note that the value of an implicit event is initially undefined (`undef`), and after each update caused by an action execution, it becomes again undefined at the next state change. From the state `s3`, the temperature sensed by the `thermo` entity is updated twice. Note that Rule (1) is triggered only once in the trace, *i.e.,* as long as the `detected` attribute of the motion detector remains true. The execution continues indefinitely, either waiting for updates of data collected by entities, or the appearance of new entities.

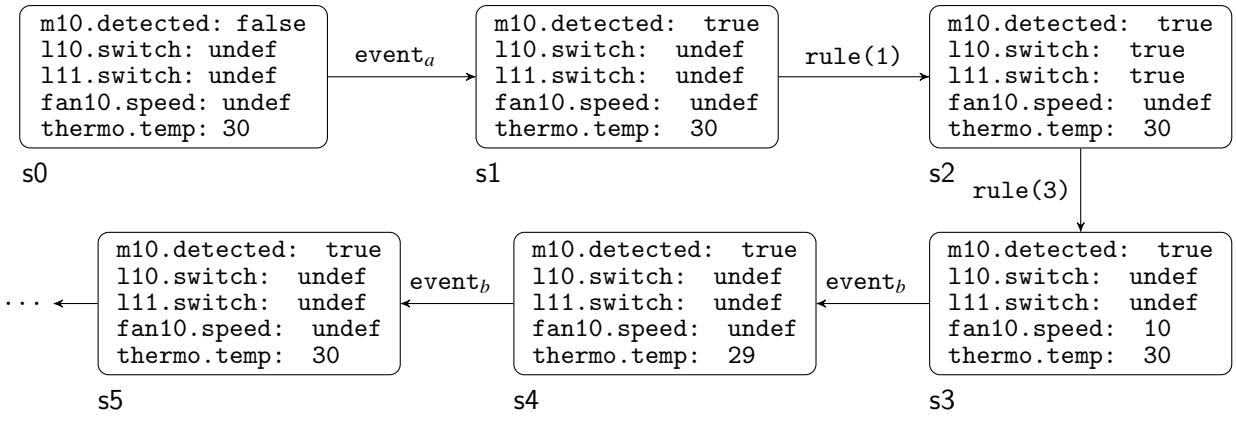
Figure 2: An orchestration logic for a building-automation application

```
rules
(1) when
        event detected from m:MotionDetector value = true
     trigger
        action switch(true) on l:Light with room = m.room
    end

(2) when
        all event detected from m:MotionDetector value = false groupby room
     trigger
        action switch(false) on l:Light with room = m.room
    end

(3) when
        event switch from l:Light value = true
        and event temperature from thermo value = 30
    trigger
        action setSpeed(10) on f:Fan with room = l.room
    end
end
```

Figure 3: An example of an execution trace of a Pantagruel program



Legend	event$_a$ is an update-event on m10.detected event$_b$ is an update-event on thermo.temperature, abbreviated thermo.temp

4 Pantagruel in the denotational style

Defining the semantics of a language usually relies on one of the following three methodologies: operational, axiomatic, or denotational [19]. Denotational semantics focuses on the mathematical relation between input data and output data, and makes explicit the domain definition of these data. In our approach, the structure and characteristics of the data related to the entities are central to the dynamic semantics of our language. Furthermore, because the denotational semantics uses a functional style, a definition can easily be mapped into an implementation. In fact, we have implemented in the OCaml language an interpreter based on the semantics of Pantagruel.

Additionally, the functional style of the denotational semantics enables us to generalize the definition of orchestration rules, from an entity to a class of entities. Specifically, we leverage higher-order functions to express the rule-evaluation process as a pattern that is applied on a set of entities, similar to map-like functions in higher-order languages. In contrast, an operational definition would make explicit these generalization steps. We illustrate this feature in Section 6.3 with the valuation functions of rules, events and actions.

5 The specification layer

In this section, we present the specification layer of the Pantagruel language: its semantic algebras, its abstract syntax and its valuation functions. Let us first introduce some notations used thereafter.

5.1 Some notations

This section presents the notations used to define the specification layer and the orchestration layer of Pantagruel. We use the following conventions, inspired by Schmidt's methodology [20]:

$\langle a,b \rangle$	a tuple of two elements a and b
$[x \mapsto v]\rho$	x maps to v in ρ
$x{\downarrow_i}, i \in \mathbb{N}^*$	projection on i^{th} element of tuple x, where i is a non-null integer
$[\![X]\!]$	denotation of a syntactic fragment
$T^\omega = \{t_i \in T\}_{i \in \mathbb{N}}$	T^ω is the set of infinite sequences of elements of set T
$T = [\ \tau_0, \tau_1, ...\ ]$	T is an infinite sequence

The last notation is borrowed from [1, 2, 14]. Following Schmidt's methodology, we define the semantics of the specification layer of Pantagruel in three steps : first, the semantic algebras are specified, representing the semantic domains of the language; second, the abstract syntax of the language is defined; third, the valuation functions define the meaning of programs, by mapping the abstract syntax to the semantic domains.

5.2 Semantic algebras

The semantic domains for the specification layer of Pantagruel are listed in Figure 5. They show that the specification layer is mainly understood in terms of environments mapping identifiers of the language to some value. The environments are defined in the usual form of a *Map* domain as illustrated in Figure 4, where X stands for the type of the value stored by the environment. For clarity, the *Errvalue* domain used for ill-typed values has been omitted from Figure 5. Let us examine the domains of the specification layer.

Figure 4: Generic algebra for the definition of environment domains

Domain $m \in Map = Id \rightarrow (X + Errvalue)$
  where $Errvalue = Unit$
Operations
$empty_m$: Map
  $empty_m = \lambda i.\text{in}Errvalue()$
$access_m$: $Id \rightarrow Map \rightarrow (X + Errvalue)$
  $access_m = \lambda i.\lambda m.m(i)$
$update_m$: $Id \rightarrow X \rightarrow Map \rightarrow Map$
  $update_m = \lambda i.\lambda x.\lambda m.[i \mapsto \text{in}X(x)]m$

Interface abstraction. Interfaces are values from the *Interface* domain (part III of Figure 5). To discriminate between entities, an interface declares typed attributes stored in the *Interface-attribute* domain. An interface similarly declares its sensing capabilities in the *Interface-event* domain. Finally, actuating capabilities are defined as method signatures, in the *Interface-action* domain, which maps a method identifier to a pair: (1) the type of the method parameter and (2) a procedure, defined by the *Action-struct* domain, to update a store with the value of an implicit event (see Section 5.3). For conciseness, methods have only one formal parameter. Interfaces are stored in an *Env-interface* environment (part IV of Figure 5) mapping an identifier to an interface. This environment is constant for a given Pantagruel program.

Figure 5: Semantic algebras for the specification layer

I. Basic domains : truth values, natural numbers, identifiers, and types (operations omitted)
Domain $t \in Type =$ Type-structure
Domain $n \in Nat = \mathbb{N}$
Domain $b \in Tr = \mathbb{B}$
Domain $i \in Id =$ Identifier

II. Values
Domain $x \in Value = Nat + Tr + Undefinedvalue,$ where $UndefinedValue = Unit$

III. Interfaces
Domain $f \in Interface = Interface\text{-}attribute \times Interface\text{-}event \times Interface\text{-}action$
  where $p \in Interface\text{-}attribute = Id \rightarrow Type$
    $s \in Interface\text{-}event = Id \rightarrow Type$
    $a \in Interface\text{-}action = Id \rightarrow (Type \times Action\text{-}struct)$
    $u \in Action\text{-}struct = Id \rightarrow Value \rightarrow Store \rightarrow Store$

IV. Environments of interfaces
Domain $e \in Env\text{-}interface = Id \rightarrow Interface$

Dynamic context data. Entities are the elements that hold the runtime information of a Pantagruel program. Entities are explicitly represented in the semantics as named objects from the *Entity* domain; their implementation conforms to one or more interfaces. For conciseness, we only define entities referring to a unique interface. Each entity is identified with its current context data, *i.e.*, its attributes and

events: they conform to the attribute declarations and to the sensing capabilities of the entity interface, respectively. Both context data are defined by the *Entity-attribute* and the *Entity-event* domains.

The *Store* domain (part VI of Figure 6) holds the set of available entities indexed by their name, corresponding to the store variables. When new entities are discovered at runtime, new variables corresponding to the name of these entities are introduced in the store. To cumulate the results produced by each rule and, within a rule, by each action, the *Store* domain further defines the *join* operation to join two partial stores which are produced either by two rules or by two actions. The *join* operation is associative, assuming stores have disjoint domains, thus suggesting noninterfering parallelism. As suggested by Schmidt [20] (Chapter 5), *join* combines the effects of the execution of actions without overwriting them. This is achieved by an appropriate definition of a *combine* operation, enabling partial stores to be combined; it is left unspecified here.

The values of the data sensed by entities, their attributes, and the actual parameter of actions, are specified by the *Value* domain (part II of Figure 5). The *Undefinedvalue* domain allows undefined values to be stored. For simplicity, only natural numbers and truth values (part I of Figure 5) are considered.

Figure 6: Semantic algebras for the specification layer (continued)

V. Entities
Domain $o \in \mathit{Entity} = \mathit{Id} \times \mathit{Entity\text{-}attribute} \times \mathit{Entity\text{-}event}$
  where $q \in \mathit{Entity\text{-}attribute} = \mathit{Id} \to \mathit{Value}$
    $t \in \mathit{Entity\text{-}event} = \mathit{Id} \to \mathit{Value}$

VI. Stores
Domain $\sigma \in \mathit{Store} = \mathit{Id} \to \mathit{Entity}_\perp$
Operations
newstore: $\mathit{Id} \to \mathit{Entity}_\perp$
 newstore= $\lambda i.\perp$
join: $\mathit{Store} \to \mathit{Store} \to \mathit{Store}$
 join= $\lambda\sigma_1.\lambda\sigma_2.\ (\lambda i.\sigma_1(i)\ \mathit{combine}\ \sigma_2(i))$
join:* $\mathscr{P}(\mathit{Store}) \to \mathit{Store}$
 *join**= $\lambda S.\ \sigma_1\ \mathit{join}\ ...(\sigma_{n-1}\ \mathit{join}\ \sigma_n)$, where $\sigma_i \in S, i < |S|$ and S is finite
access-action: $\mathit{Id} \to \mathit{Id} \to \mathit{Env\text{-}interface} \to \mathit{Store} \to \mathit{Action\text{-}struct}$
 access-action= $\lambda i_a.\lambda i_o.\lambda e.\lambda\sigma.\ (\mathit{access}\ i_a\ (\mathit{access}\ (\sigma\ i_o)\downarrow_1\ e)\downarrow_3)\downarrow_2$
access-event: $\mathit{Id} \to \mathit{Id} \to \mathit{Store} \to \mathit{Value}$
 access-event= $\lambda i_e.\lambda i_o.\lambda\sigma.((\sigma\ i_o)\downarrow_3\ i_e)$
update-event: $\mathit{Id} \to \mathit{Id} \to \mathit{Value} \to \mathit{Store} \to \mathit{Store}$
 update-event= $\lambda i_e.\lambda i_o.\lambda p.\lambda\sigma.[i_o \mapsto \langle(\sigma\ i_o)\downarrow_1, (\sigma\ i_o)\downarrow_2, [i_e \mapsto p](\sigma\ i_o)\downarrow_3\rangle]\sigma$
Note : *access-interface*, *access-attribute* and *update-attribute* are handled similarly

5.3 Definition of the specification layer

The abstract syntax and the valuation functions for the specification layer appear in Figures 7 and 8. A specification S consists of a set of interfaces F and an initial set of entities O. Note that, at runtime, these entities can be removed and new entities can be deployed.

The **S** valuation function produces a pair of two environments named *Env-interface* and *Store*, corresponding to a mapping from an identifier to an interface, and a mapping from an identifier to an entity,

Figure 7: Abstract syntax for the specification layer

S ∈ Specification
F ∈ Interface-declaration
P ∈ Attribute
E ∈ Sensing-capability
A ∈ Actuating-capability
O ∈ Entity-declaration
G ∈ Assignment

T ∈ Type-structure
X ∈ Value

I ∈ Identifier
N ∈ Numeral

S ::= F**;**O
F ::= F_1**;**F_2 | **interface** I {P **;** E **;** A}
P ::= P_1**;**P_2 | **attribute** I **:** T
E ::= E_1**;**E_2 | **event** I **:** T
A ::= A_1**;**A_2 | **action** I_1(T)
O ::= O_1**;**O_2 | **entity** I_o **:** I_f {G}
G ::= G_1**;**G_2 | I **=** X

T ::= **nat** | **bool**
X ::= N | **true** | **false**

respectively. This valuation function produces the environment of entities of a given Pantagruel program by invoking the **F** function to produce an environment of interfaces that is passed as an argument to the **O** function. In doing so, the entities can be checked against the interface that they implement. The **P**, **E**, **A** and **O** functions are composed similarly to the **F** function when involving a sequence (*e.g.*, F_1;F_2). For conciseness, we have omitted them. These three functions are responsible for enriching environments. The **A** function defines a procedure of the *Action-struct* domain. Since an action is invoked on an entity, this entity is referred to as the i_o parameter. Following the procedure specification suggested by Schmidt [20] (Chapter 8), it also takes an x formal parameter, which is assumed to be of type **T**. When it is invoked through an entity action, this procedure generates an implicit event of the name of the concerned action. Finally, the **O** function states that the values of sensed data are initially undefined, and the **G** function initializes the attributes to some value.

In our definitions, type checking is not performed yet. To check that the value assigned to an attribute is consistent with the interface of the concerned entity, the functionality of the **O** function must be modified to take in account an erroneous *Entity-attribute* construction. To do so, we use the same definition as the one proposed by Schmidt for the *Store* domain (see [20], Chapter 7): the *Entity-attribute* domain is tagged to indicate an attribute initialization failure, and a *check* operation prevents further assignments in case of an error.

Domain *Post-entity-attribute* $\in p = OK + Err$

where $OK =$ *Entity-attribute* *and* $Err =$ *Entity-attribute*.

Operations

check: (*Entity-attribute* $\rightarrow$ *Post-entity-attribute*)$\rightarrow$ (*Entity-attribute* $\rightarrow$ *Post-entity-attribute*)

$check\, f = \lambda p.$cases (p) of is$OK(s) \rightarrow (f\ s)$ [] is$Err(s) \rightarrow p$ end

Assignment is then adjusted to check type consistency by calling the *check* operation :

G: Assignment$\rightarrow$ *Interface-attribute* $\rightarrow$ *Entity-attribute* $\rightarrow$ *Post-entity-attribute*

Figure 8: Valuation functions for the specification layer

S: Specification $\to$ *Env-interface* $\times$ *Store*

$\mathbf{S}[\![\text{F O}]\!] = \mathsf{let}\ e = \mathbf{F}[\![\text{F}]\!]\mathit{empty}\ \mathsf{in}\ \langle e, \mathbf{O}[\![\text{O}]\!]e\ \mathit{empty}\rangle$

F: Interface-declaration $\to$ *Env-interface* $\to$ *Env-interface*

$\mathbf{F}[\![\text{F}_1;\text{F}_2]\!] = \mathbf{F}[\![\text{F}_2]\!] \circ \mathbf{F}[\![\text{F}_1]\!]$

$\mathbf{F}[\![\textbf{interface}\ \text{I}\ \{\ \text{P}\ ;\ \text{E}\ ;\ \text{A}\ \}]\!] = \lambda e.\mathit{update}[\![\text{I}]\!]\ \langle \mathbf{P}[\![\text{P}]\!]\mathit{empty}, \mathbf{E}[\![\text{E}]\!]\mathit{empty}, \mathbf{A}[\![\text{A}]\!]\mathit{empty}\rangle\ e$

P: Attribute $\to$ *Interface-attribute* $\to$ *Interface-attribute*

$\mathbf{P}[\![\textbf{attribute}\ \text{I}\ \textbf{:}\ \text{T}]\!] = \lambda p.\mathit{update}[\![\text{I}]\!]\ \mathbf{T}[\![\text{T}]\!]\ p$

E: Sensing-capability $\to$ *Interface-event* $\to$ *Interface-event*

$\mathbf{E}[\![\textbf{event}\ \text{I}\ \textbf{:}\ \text{T}]\!] = \lambda s.\mathit{update}[\![\text{I}]\!]\ \mathbf{T}[\![\text{T}]\!]\ s$

A: Actuating-capability $\to$ *Interface-action* $\to$ *Interface-action*

$\mathbf{A}[\![\textbf{action}\ \text{I}\ (\ \text{T}\ )]\!] = \lambda a.\mathit{update}[\![\text{I}]\!]\langle \mathbf{T}[\![\text{T}]\!], \lambda i_o.\lambda x.(\mathit{update\text{-}event}[\![\text{I}]\!]\ i_o\ x)\rangle\ a$

O: Entity-declaration $\to$ *Env-interface* $\to$ *Store* $\to$ *Store*

$\mathbf{O}[\![\textbf{entity}\ \text{I}_o : \text{I}_f\ \{\ \text{G}\ \}]\!] = \lambda e.\lambda\sigma.\mathit{update}[\![\text{I}_o]\!]\ \langle [\![\text{I}_f]\!], (\mathbf{G}[\![\text{G}]\!](\mathit{access}[\![\text{I}_f]\!]\ e)\downarrow_1\ \mathit{empty}), \mathit{empty}\rangle\ \sigma$

G: Assignment $\to$ *Interface-attribute* $\to$ *Entity-attribute* $\to$ *Entity-attribute*

$\mathbf{G}[\![\text{G}_1;\text{G}_2]\!] = \lambda p.(\mathbf{G}[\![\text{G}_2]\!]p) \circ (\mathbf{G}[\![\text{G}_1]\!]p)$

$\mathbf{G}[\![\textbf{I = X}]\!] = \lambda p.\lambda q.\mathit{update\text{-}attribute}[\![\text{I}]\!]\ \mathbf{X}[\![\text{X}]\!]\ q$

$\mathbf{X}[\![\text{X}]\!]$: *Value* (operations omitted; X is assumed to be a constant numeral or boolean here)

$\mathbf{G}[\![\mathrm{G}_1;\mathrm{G}_2]\!] = \lambda p.\ (\mathit{check}\ (\mathbf{G}[\![\mathrm{G}_2]\!]p)) \circ (\mathbf{G}[\![\mathrm{G}_1]\!]p)$
$\mathbf{G}[\![\mathrm{I} = \mathrm{X}]\!] = \lambda p.\lambda q.\mathsf{cases}\ (\mathit{access}\ [\![\mathrm{I}]\!]\ p)\ \mathsf{of}$
is*Nat-type*$() \rightarrow$ cases x of
is*Nat*$(n) \rightarrow$ in*OK*$((\mathit{update}[\![\mathrm{I}]\!]\ (\mathbf{X}[\![\mathrm{X}]\!])\ q))\ []$ is*Tr*$(b) \rightarrow$ in*Err*(q) end
is*Tr-type*$() \rightarrow$ ... similar processing as above ... end

The introduction of the *Post-entity-attribute* implies adjusting the functionality of the **O** function accordingly. The same modifications can be applied to the formal parameter of actions (not shown here).

6 The orchestration logic

In this section, we present an overview of the reactive computation model of the orchestration layer. We then formalize it by defining its semantic algebras, its abstract syntax and its valuation functions.

6.1 A simple reactive computation model

Pantagruel is a reactive programming language in that a program constantly interacts with the outside, reacting to *context changes* that are observed by changes occurring on the sensed data and the attributes provided by entities. A program execution therefore corresponds to an infinite loop. Each iteration is called an *orchestration step*, where each step consists of executing actions on entities according to some conditions on the sensed data or the attributes of these entities. Each orchestration step produces a new state reflecting the actions performed.

To facilitate reasoning about the orchestration steps of a program, we assume :

i) *Discrete time and atomic execution*. Discrete time is modeled as clock ticks. Each tick corresponds to an orchestration step, where all the rules are evaluated within a tick. Invoked actions are executed until completion. In doing so, an orchestration step is executed with respect to a logical cycle, abstracting over time as well as implementation of the actuating capabilities of entities.

ii) *Noninterfering parallelism*. Parallelism assumes that all the rules whose conditions are satisfied at a given orchestration step, are executed with the same state, making this execution simultaneous. Furthermore, we assume a noninterfering parallelism, as defined by Schmidt [20] (Chapter 5): two rules are executed independently of each other within an orchestration step, producing two disjoint, partial states. At the end of the execution, states are joined into a final state without deleting each other's effects. Checking conflicting states that would occur in interfering parallelism is left for another paper.

6.2 Semantic algebras

Interface-centric behavior. The interface abstraction is used to select entities on which an action needs to be triggered. Such an abstraction also enables applications to deal with the disappearance or appearance of new entities while they are executed. In doing so, dynamicity is taken in account by our language. The interface-based selection mechanism is ensured by the use of entity variables involved in rules. For example, Rule (1) of Figure 2 declares an `m` entity variable of `MotionDetector` interface. To use interface abstraction in rules, we define the *Env-entity* domain as in Figure 9, part VIII: it maps the identifier of an entity variable to a reference *Reference* of an entity. The *Reference* domain is a disjoint union of an *Interface*, reflecting the interface of an entity that may match a rule, and of an *Instance*, reflecting the entity on which a rule eventually applies. Entity variables can only be mapped to current entities after

events and actions are evaluated (we examine this point in Section 6.3.2). As a consequence, to evaluate events in a rule with respect to interface abstraction, we define the intermediate domain $\mathbb{B}$-*function* of boolean functions, as in Figure 9, part IX. Specifically, it needs an environment *Env-entity* to evaluate a boolean operation. Boolean operators and_ρ and or_ρ are defined accordingly.

Entity variables are inside the scope of a rule; this makes our language be a block-structured language, as formalized by Schmidt [20], Chapter 7: a rule is a block, which scope is represented by the *Env-entity* domain. Our approach is similar to the *Location*-based semantics (Chapter 3): a reference has the same role as a location, holding the denotation of variable identifiers. However, in contrast to the *Location* domain, the space of values covered by entity variables in a rule is constant and restricted to the elements present in the global store of entities, hence the use of the specific *Reference* domain.

Context change. To represent the reactive nature of Pantagruel, we leverage the context data provided by the entities of a specification, focusing on *what* data are available at each orchestration step, in order to abstract over the way these data are acquired. The events produced by entities, along with their attributes, represent the runtime state of a Pantagruel program, as defined by the *Store* domain (part VI of Figure 5); orchestration steps are enabled by changes in this state. To express changes due to new events, two stores must be involved, enabling to compare event values between two iterations of the reactive loop. We make this dual-store strategy explicit with the compound domain *DualStore*, illustrated in Figure 9, part VII .

Figure 9: Semantic algebras for the orchestration layer

VII. Dual stores
Domain $\delta \in DualStore = Store \times Store$

VIII. References and environments of entities
Domain $f \in Reference = Interface + Instance$
where $Interface = Id$ and $Instance = Id$
Domain $\rho \in Env\text{-}entity = Id \to Reference$
Operations
$instantiate: Store \to Env\text{-}entity \to \mathscr{P}(Env\text{-}entity)$
$instantiate = \lambda\sigma.\lambda\rho.\{\lambda i.((\rho\ i)\ \ equals\ \ \text{in}Interface((\sigma\ j)\downarrow_1)) \to\ (\text{in}Instance(j))\,[]\,(\rho\ i) \mid j \in Id\}$

IX. Boolean functions
Domain $b_\rho \in \mathbb{B}\text{-}function = Env\text{-}entity \to Tr$
Operations
$and_\rho: \mathbb{B}\text{-}function \to \mathbb{B}\text{-}function \to Env\text{-}entity \to Tr$
$and_\rho = \lambda b_1.\lambda b_2.\lambda\rho.\ (b_1\ \rho)\ and\ (b_2\ \rho)$ (same for or_ρ)

6.3 Definition of the orchestration layer

The syntax of the orchestration layer is given in Figure 10. An orchestration program P is given as a specification S and a set K of rules R, relating events W to action calls C. We examine the language constructs through the valuation functions, described in the following sections.

Figure 10: Abstract syntax for the orchestration language

P	$\in$	Program	I	$\in$	Identifier
K	$\in$	Rule-block			
R	$\in$	Rule			
W	$\in$	Event			
C	$\in$	Action-call			
D	$\in$	Declaration			
B	$\in$	Boolean-expression			
F	$\in$	Filter			
X	$\in$	Expression			

P	::=	S**;** K
K	::=	**rules** R **end**
R	::=	R_1 **;** R_2 | **when** W **trigger** C **end**
W	::=	W_1 **or** W_2 | W_1 **and** W_2 | **event** I_e **from** D (**with** F)$^?$ B
C	::=	C_1 || C_2 | C_1 **,** C_2 | **action** I_a (X) **on** D (**with** F)$^?$
D	::=	I_o **:** I_c | I_o
B	::=	**value changed** | **value =** X
F	::=	I_p **=** X
X	::=	N | I_o**.** I_x

6.3.1 Top-level evaluation : the reactive loop

The **P** valuation function of a Pantagruel program is depicted at the top of Figure 11. We represent the reactive loop as a function from the domain of infinite traces of input stores ($Store^{\omega}$) to its output counterpart ($Store^{\omega}$). An input store σ^{in} contains event values pulled from the outside and captured by the available entities as well as new entities deployed at runtime.

An output store σ results from the evaluation step of a rule set, as described below. The initial store σ_0 is defined from the specification and has no preceding store (σ_{-1} is the empty store). This representation using traces conforms to usual definitions of reactive languages, such as encountered in [1, 2, 14].

The evaluation of a program is as follows. The **S** valuation function first builds a specification, consisting of an interface environment e and a store σ_0 containing the initial values of the entity attributes. They are given as parameters of the rule valuation function **K** along with two store snapshots: σ_{t-1} at time $t-1$, and σ'_t at time t, including events values pulled from the outside and new entities. These changes from the external environment are abstracted by the *update-external* function. The evaluation of **R** results in a new store, updating the current store σ'_t to produce the store at $t+1$ with the *update-internal* function (its definition is not shown here). This function is also in charge of setting back to the undefined value all the implicit events that were present in the current store σ'_t.

6.3.2 Rule evaluation

The **R** function requires an environment ρ and a dual store δ as illustrated in Figure 12. Rule events are evaluated with the **W** function with respect to the dual-store strategy. Their evaluation produces an entity environment ρ_e, mapping variables, declared in the event definition, to a reference (*Reference*) to an current entity. Event evaluation also produces a boolean function whose execution is done once

Figure 11: Valuation function of the top-level program

$$\begin{array}{l}
\textbf{P}\text{: Program} \to \mathit{Store}^{\omega} \to \mathit{Store}^{\omega} \\
\quad \mathbf{P}[\![\mathrm{S};\mathrm{K}]\!] = \lambda\Sigma.\mathsf{let}\ \langle e, \sigma_0\rangle = \mathbf{S}[\![\mathrm{S}]\!] \\
\qquad \mathsf{in}\ \{(\mathit{update\text{-}internal}\ e\ (\mathbf{R}[\![\mathrm{R}]\!]e\ \langle\sigma_{t-1}, \sigma'_t\rangle)\ \sigma'_t) \mid \sigma_t^{\mathtt{in}} \in \Sigma, \sigma'_t = (\mathit{update\text{-}external}\ \sigma_t^{\mathtt{in}}\ \sigma_t)\}_{t\in\mathbb{N}} \\
\quad \mathsf{where}\ \sigma_{-1} = \mathit{newstore} \\
\quad \mathsf{and}\ \mathit{update\text{-}external} : \mathit{Store} \to \mathit{Store} \to \mathit{Store} \\
\quad \mathsf{and}\ \mathit{update\text{-}internal} : \mathit{Env\text{-}interface} \to \mathit{Store} \to \mathit{Store} \to \mathit{Store} \\
\textbf{K}\text{: Rule-block} \to \mathit{Env\text{-}interface} \to \mathit{DualStore} \to \mathit{Store} \\
\quad \mathbf{K}\ [\![\textbf{rules}\ \mathrm{R}\ \textbf{end}]\!] = \mathbf{R}[\![\mathrm{R}]\!]
\end{array}$$

the ρ_e environment is fully defined. Actions are then evaluated with the **C** function, taking ρ_e, and updating it into ρ_a with new variables appearing in actions. Both ρ_e and ρ_a are in the scope of the rule block **when** ... **end**.

Rule as a pattern. At this stage of the rule evaluation process, variables declared in the ρ_a environment do not refer to current entities yet, preventing the action evaluation to complete. We thus need to "instantiate" ρ_a by mapping each of its variables to a current entity of the specification. In fact, more than one entity for each variable of an entity environment may be concerned by an event (an action, respectively). As a result, more than one instantiation of environment may match the events (the actions, respectively). We thus need to construct a set of such entity environments, mapping entity variables to identifiers of current entities. This is done by defining the set P: given an entity variable i_v declared with interface i_f, it builds all the environments, such that for each environment an entity of the same interface i_f is mapped to i_v. Each resulting environment is then given as an argument to the partly evaluated actions, which return a store. Stores produced by the actions are finally joined with join^*.

Figure 12: Valuation function of a rule

$$\begin{array}{l}
\textbf{R}\text{: Rule} \to \mathit{Env\text{-}interface} \to \mathit{DualStore} \to \mathit{Store} \\
\quad \mathbf{R}[\![\mathrm{R}_1;\mathrm{R}_2]\!] = \lambda e.\lambda\delta.(\mathbf{R}[\![\mathrm{R}_1]\!]e\ \delta)\ \mathit{join}\ (\mathbf{R}[\![\mathrm{R}_2]\!]e\ \delta) \\
\quad \mathbf{R}[\![\textbf{when}\ \mathrm{W}\ \textbf{trigger}\ \mathrm{C}\ \textbf{end}]\!] = \lambda e.\lambda\langle\sigma_{t-1},\sigma_t\rangle.\ \mathsf{let}\ \langle\rho_e, b\rangle = (\mathbf{W}[\![\mathrm{W}]\!]\langle\sigma_{t-1},\sigma_t\rangle\ \mathit{empty}\ (\lambda\rho.\mathit{true}))\ \mathsf{in} \\
\qquad \mathsf{let}\ \langle\rho_a, a\rangle = (\mathbf{C}[\![\mathrm{C}]\!]e\ \sigma_t\ \rho_e\ (\lambda\rho.\mathit{newstore}))\ \mathsf{in} \\
\qquad \mathsf{let}\ P = \mathit{instantiate}\ \sigma_t\ \rho_a\ \mathsf{in} \\
\qquad \mathit{join}^*\{((b\ \rho_{\mathit{inst}}) \to (a\ \rho_{\mathit{inst}})\ []\ \mathit{newstore}) \mid \rho_{\mathit{inst}} \in P\}
\end{array}$$

6.3.3 Event evaluation

The **W** function evaluates events with a dual store δ (in "read-only" mode), an environment ρ, and a boolean function b. The functionality of **W** shows that an event produces a pair of functions: an environment and a boolean function. The environment is updated by the **D** function, and the boolean function is defined by the **B** function. Specifically, the **D** function is interpreted as a variable declaration when it involves an entity variable, augmenting an environment. The **B** function evaluates the B test condition on an event, which is either of the form **value =** X or **value changed**. Note that other comparison operators can be envisioned. We omit them for simplicity.

The evaluation of B depends on an entity environment which is completed after the evaluation of all the combined W events. As a result, **B** produces an intermediate evaluation of the test condition, waiting for an environment to complete. This intermediate evaluation is specified by a $\mathbb{B}$-*function*, requiring an environment to produce a boolean value. The **B** function also depends on the dual store: when B is of the form **value =** X, it yields true if the value of event $[\![\mathrm{I}_e]\!]$ equals $\mathbf{X}[\![\mathrm{X}]\!]$ at time t and not at $t-1$. When B is of the form **value changed**, it yields true as often as the event value changes, *i.e.*, when its value at t is different from its value at $t-1$.

Finally, to be concerned by an event, an entity may further need to satisfy an extra condition. This condition is evaluated by the **F** function, which is similar to **B**: instead of testing an event value, it tests an attribute value, and produces a $\mathbb{B}$-*function* denoting this test. We ignored the case where the "**with** F" term does not appear, for it is equivalent to setting $(\mathbf{F}[\![\mathrm{F}]\!] i\ \sigma)$ to the constant value $(\lambda\rho.\mathrm{true})$.

Figure 13: Valuation functions for events

W: Event $\to$ *DualStore* $\to$ *Env-entity* $\to$ $\mathbb{B}$-*function* $\to$ (*Env-entity* $\times$ $\mathbb{B}$-*function*)

$\mathbf{W}[\![\mathrm{W}_1 \textbf{ and } \mathrm{W}_2]\!] = \lambda\delta.\lambda\rho.\lambda b.$
  let $\langle \rho_1, b_1 \rangle = (\mathbf{W}[\![\mathrm{W}_1]\!]\delta\ \rho\ b)$ in $(\mathbf{W}[\![\mathrm{W}_2]\!]\delta\ \rho_1\ b_1)$

$\mathbf{W}[\![\mathrm{W}_1 \textbf{ or } \mathrm{W}_2]\!] = \lambda\delta.\lambda\rho.\lambda b.$
  let $\langle \rho_1, b_1 \rangle = (\mathbf{W}[\![\mathrm{W}_1]\!]\delta\ \rho\ b)$ in
  let $\langle \rho_2, b_2 \rangle = (\mathbf{W}[\![\mathrm{W}_2]\!]\delta\ \rho_1\ b)$ in $\langle \rho_2, (or_\rho\ b_1\ b_2) \rangle$

$\mathbf{W}[\![\textbf{event } \mathrm{I}_e \textbf{ from } \mathrm{D} \textbf{ with } \mathrm{F}\ \mathrm{B}]\!] = \lambda\delta.\lambda\rho.\lambda b.$
  let $\langle v, \rho' \rangle = \mathbf{D}[\![\mathrm{D}]\!]\rho$ in
  let $b_\rho = \lambda\rho.$cases $(access\ v\ \rho)$ of
    is*Instance*$(i_o) \to$ let $x_e = \lambda\sigma.(access\text{-}event[\![\mathrm{I}_e]\!]\ i_o\ \sigma)$ in
      $(and_\rho\ (\mathbf{F}[\![\mathrm{F}]\!] i_o\ \delta{\downarrow_1})\ (and_\rho\ (\mathbf{B}[\![\mathrm{B}]\!]\ x_e\ \delta)\ b)\ \rho)$
    [] is*Interface*$(i_f) \to$ *false* end —– "no entity was found $\to$ no event is caught"
  in $\langle \rho', b_\rho \rangle$

D: Declaration $\to$ *Env-entity* $\to$ (*Id* $\times$ *Env-entity*)

$\mathbf{D}[\![\mathrm{I}_v{:}\mathrm{I}_f]\!] = \lambda\rho.\langle [\![\mathrm{I}_v]\!], (update\ [\![\mathrm{I}_v]\!]\ \text{in}Interface([\![\mathrm{I}_f]\!])\ \rho) \rangle$

$\mathbf{D}[\![\mathrm{I}_s]\!] = \lambda\rho.\langle [\![\mathrm{I}_s]\!], ([\![\mathrm{I}_s]\!] \in \mathsf{domain}(Store) \to update[\![\mathrm{I}_s]\!]\ \text{in}Instance([\![\mathrm{I}_s]\!])\ \rho\ []\ \rho \rangle$

B: Boolean-expression $\to$ (*Store* $\to$ *Value*) $\to$ *DualStore* $\to$ $\mathbb{B}$-*function*

$\mathbf{B}[\![\textbf{value = } \mathrm{X}]\!] = \lambda x.\lambda\delta.$ let $b_\rho = \lambda\sigma.\lambda\rho.(x\ \sigma)\ eq\ (\mathbf{X}[\![\mathrm{X}]\!]\sigma\ \rho)$ in
  $not(b_\rho\ \delta{\downarrow_1})\ and_\rho\ (b_\rho\ \delta{\downarrow_2}))$

$\mathbf{B}[\![\textbf{value changed}]\!] = \lambda x.\lambda\delta.\lambda\rho.((x\ \delta{\downarrow_1})\ neq\ (x\ \delta{\downarrow_2}))$

F: Filter $\to$ *Id* $\to$ *Store* $\to$ $\mathbb{B}$-*function*

$\mathbf{F}[\![\mathrm{I}_p \textbf{ = } \mathrm{X}]\!] = \lambda i_o.\lambda\sigma.\lambda\rho.(access\text{-}attribute[\![\mathrm{I}_p]\!]\ i_o\ \sigma)\ eq\ \ (\mathbf{X}[\![\mathrm{X}]\!]\sigma\ \rho)$

X: Value $\to$ *Store* $\to$ *Env-entity* $\to$ *Value*

$\mathbf{X}[\![\mathrm{N}]\!] = \lambda\sigma.\lambda\rho.\mathbf{N}[\![\mathrm{N}]\!]$

$\mathbf{X}[\![\mathrm{I}_v.\mathrm{I}_x]\!] = \lambda\sigma.\lambda\rho.$cases $(access[\![\mathrm{I}_v]\!]\ \rho)$ of
  is*Instance*$(i_o) \to\ ([\![\mathrm{I}_x]\!] \in \mathsf{domain}(Entity\text{-}event) \to\ access\text{-}event[\![\mathrm{I}_x]\!]\ i_o\ \sigma\ []\ access\text{-}attribute[\![\mathrm{I}_x]\!]\ i_o\ \sigma)$
  [] is*Interface*$(i_f) \to\ Undefinedvalue$ end

6.3.4 Action evaluation

The semantic equations for the **C** valuation function are defined in Figure 14. **C** requires an environment of interfaces, an environment of entities, and a store. The functionality of **C** shows that an action produces a pair of functions: an environment and a function waiting for an environment to produce a store. The definition of **C** mirrors the one of **W**: similarly to events, actions can declare variables, which are evaluated by the **D** function, enriching the environment given as a parameter to **C**. Similarly to **W**, **C** depends on an environment which is completed after the evaluation of all the combined C actions; the invoked action identified by $[\![\mathrm{I}_a]\!]$ produces an intermediate evaluation of this statement, waiting for an environment argument to complete. This intermediate evaluation is specified by the intermediate domain *Env-entity* $\rightarrow$ *Store* (mirroring the $\mathbb{B}$-*function* domain for events), requiring an environment to produce an updated store, reflecting the effect of the $[\![\mathrm{I}_a]\!]$ action. In our language semantics, we abstract over the implementation details through the action parameter, by making its actual value corresponds to the value of the produced effect. Specifically, it corresponds to the value of the implicit event generated by the invocation of the action. The store is thus updated by mapping this actual parameter to an event of the name of the action, as specified by the **A** function (Figure 8).

When actions are executed in parallel, the final store is built by joining the partial stores produced by these actions (see Figure 9 for the *join* operation). In contrast, a sequential execution builds the argument of the second action by cumulating the partial store produced by the first action and the current store.

Note that this process assumes that the scope of a sequence of actions is bound to the environment in which the concerned actions apply. In other words, the order of actions defined sequentially over an entity variable is preserved only for the actions of the current entity to which the variable is mapped.

Figure 14: Valuation functions of actions

$$
\begin{array}{l}
\mathbf{C}: \text{Action} \rightarrow \textit{Env-interface} \rightarrow \textit{Store} \rightarrow \textit{Env-entity} \rightarrow (\textit{Env-entity} \rightarrow \textit{Store}) \rightarrow (\textit{Env-entity} \times (\textit{Env-entity} \rightarrow \textit{Store})) \\
\quad \mathbf{C}[\![\mathrm{C}_1 \parallel \mathrm{C}_2]\!] = \lambda e.\lambda\sigma.\lambda\rho.\lambda f_\sigma. \\
\qquad \mathsf{let}\ \langle \rho_1, f_{\sigma_1} \rangle = (\mathbf{C}[\![\mathrm{C}_1]\!] e\ \sigma\ \rho\ f_\sigma)\ \mathsf{in} \\
\qquad \mathsf{let}\ \langle \rho_2, f_{\sigma_2} \rangle = (\mathbf{C}[\![\mathrm{C}_2]\!] e\ \sigma\ \rho_1\ f_\sigma)\ \mathsf{in}\ \langle \rho_2, \lambda\rho.(f_{\sigma_2}\ \rho)\ \textit{join}\ (f_{\sigma_1}\ \rho) \rangle \\
\quad \mathbf{C}[\![\mathrm{C}_1\ \mathbf{,}\ \mathrm{C}_2]\!] = \lambda e.\lambda\sigma.\lambda\rho.\lambda f_\sigma. \\
\qquad \mathsf{let}\ \langle \rho_1, f_{\sigma_1} \rangle = (\mathbf{C}[\![\mathrm{C}_1]\!] e\ \sigma\ \rho\ f_\sigma)\ \mathsf{in}\ (\mathbf{C}[\![\mathrm{C}_2]\!] e\ \sigma\ \rho_1\ f_{\sigma_1}) \\
\quad \mathbf{C}[\![\mathbf{action}\ \mathrm{I}_a\ (\ \mathrm{X}\ )\ \mathbf{on}\ \mathrm{D}\ \mathbf{with}\ \mathrm{F}]\!] = \lambda e.\lambda\sigma.\lambda\rho.\lambda f_\sigma. \\
\qquad \mathsf{let}\ \langle v, \rho' \rangle = (\mathbf{D}[\![\mathrm{D}]\!]\rho)\ \mathsf{in} \\
\qquad \mathsf{let}\ \sigma_\rho = \lambda\rho.\mathsf{cases}\ (\textit{access}\ v\ \rho)\ \mathsf{of} \\
\qquad\quad \mathrm{is}\textit{Instance}(i_o) \rightarrow\ \mathsf{let}\ a_\rho = (\textit{access-action}[\![\mathrm{I}_a]\!]\ i_o\ e\ \sigma)\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ f'_\sigma = \lambda\rho.(a_\rho\ i_o\ (\mathbf{X}[\![\mathrm{X}]\!]\rho\ \sigma))\ (f_\sigma\ \rho)\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{F}[\![\mathrm{F}]\!]\ i_o\ \sigma\ \rho \rightarrow\ (f'_\sigma\ \rho)\ []\ (f_\sigma\ \rho)) \\
\qquad\quad [] \mathrm{is}\textit{Interface}(i_f) \rightarrow\ f_\sigma\ \rho\ \ \mathsf{end} \qquad\qquad \text{-- ``no entity was found} \rightarrow \text{no store update''} \\
\qquad \mathsf{in}\ \langle \rho', \sigma_\rho \rangle
\end{array}
$$

6.4 Applying the semantics to an example

The static semantics analysis of Rule (1) in the program example of Figure 2 is presented in Figures 15 and 16. This example demonstrates that the intended behavior of a Pantagruel program is covered by its semantics. In particular, as illustrated in Figure 16, the store σ_{R1} produced by the execution of Rule (1),

contains two events, reflecting the execution of the `switch` action on the `l10` and `l11` lights, located in room `101`, where the `m10` motion detector detected an event from σ_1 to σ_2. Most `cases` constructs for handling *Reference* elements are skipped in our example, in order to keep the clarity of the illustration.

Figure 15: Semantics applied on Figure 2

Let D_0 = `m:MotionDetector`
D_1 = `l:Light`
B_0 = **value =** `true`
F_0 = `room = m.room`
W_0 = **event** `detected` **from** D_0 B_0
C_0 = **action** `switch(true)` **on** D_1 **with** F_0
and an initial store $\sigma_0 = [\![\texttt{l10}]\!] \mapsto \langle [\![\texttt{Light}]\!], \langle [[\![\texttt{room}]\!] \mapsto 101] \mathit{empty}, \mathit{empty} \rangle ... \rangle$ containing the entities of Figure 1. Moreover, in this example, we assume that a motion was detected from σ_1 to σ_2.

$$\mathbf{R}[\![\textbf{when } W_0 \textbf{ trigger } C_0]\!]\ e\ \langle \sigma_1, \sigma_2 \rangle$$
$$= \mathsf{let}\ \langle \rho_e, b \rangle = (\mathbf{W}[\![W_0]\!])\langle \sigma_1, \sigma_2 \rangle\ \mathit{empty}\ (\lambda \rho.\ \mathit{true})\ \mathsf{in}$$
$$\quad \mathsf{let}\ \langle \rho_a, a \rangle = (\mathbf{C}[\![C_0]\!] e\ \sigma_2\ \rho_e\ (\lambda \rho . \mathit{newstore}))\ \mathsf{in}$$
$$\quad \mathsf{let}\ P = \mathit{instantiate}\ \sigma_2\ \rho_a\ \mathsf{in}\ \mathit{join}^*\{((b\ \rho_{\mathit{inst}}) \rightarrow (a\ \rho_{\mathit{inst}})\ [\ ]\ \mathit{newstore}) \mid \rho_{\mathit{inst}} \in P\}$$

Where :

(1) $\mathbf{W}[\![W_0]\!]\langle \sigma_1, \sigma_2 \rangle \mathit{empty}\ (\lambda \rho . \mathit{true})$
$= \mathbf{W}[\![\textbf{event } \texttt{detected} \textbf{ from } D_0\ B_0]\!]\langle \sigma_1, \sigma_2 \rangle \mathit{empty}\ (\lambda \rho.\ \mathit{true})$
$= \mathsf{let}\ \langle v, \rho' \rangle = \mathbf{D}[\![\texttt{m:MotionDetector}]\!] \mathit{empty}\ \mathsf{in}$

$$\langle [\![\texttt{m}]\!], \underbrace{(\mathit{update}\ [\![\texttt{m}]\!]\ (\mathrm{in}\mathit{Interface}([\![\texttt{MotionDetector}]\!]))\ \mathit{empty})}_{\rho' \text{ equals } [[\![\texttt{m}]\!] \mapsto \mathrm{in}\mathit{Interface}([\![\texttt{MotionDetector}]\!])]\mathit{empty}} \rangle\ \mathsf{in}$$

$$\mathsf{let}\ b_\rho = \lambda \rho.\ ... \begin{cases} \mathsf{let}\ x_e = \lambda \sigma . (\underbrace{\mathit{access\text{-}event}[\![\texttt{detected}]\!]\ (\mathit{access}\ [\![\texttt{m}]\!]\ \rho)\ \sigma}_{\text{let this be } \lceil \text{m.detected} \rceil \sigma})\ \mathsf{in} \\ \\ \mathit{and}_\rho\ (\mathbf{B}[\![B_0]\!]\ x_e\ \langle \sigma_1, \sigma_2 \rangle)\ (\lambda \rho . \mathit{true})\ \rho\ {}^{(note)} \end{cases}$$

$$\mathbf{B}[\![\textbf{value =} \text{true}]\!]$$
$$= \begin{cases} \mathsf{let}\ b_\rho = \lambda \sigma . \lambda \rho .\ \underbrace{(x_e\ \sigma)}_{\lceil \text{m.detected} \rceil \sigma}\ \ \mathit{eq}\ \underbrace{(\mathbf{X}[\![\text{true}]\!] \sigma\ \rho)}_{\mathit{true}}\ \mathsf{in} \\ \\ \mathit{not}(b_\rho\ \sigma_1)\ \mathit{and}_\rho\ (b_\rho\ \sigma_2) \end{cases}$$
$$= \mathit{not}(\lceil \text{m.detected} \rceil \sigma_1)\ \mathit{and}_\rho\ (\lceil \text{m.detected} \rceil \sigma_2)$$

$$= \langle \underbrace{[[\![\texttt{m}]\!] \mapsto \mathrm{in}\mathit{Interface}(\texttt{MotionDetector})] \mathit{empty}}_{\rho_e}, \underbrace{\mathit{not}(\lceil \text{m.detected} \rceil \sigma_1)\ \mathit{and}_\rho\ (\lceil \text{m.detected} \rceil \sigma_2)}_{b} \rangle$$

(note) no F filter in W_0 is equivalent to the "**with** true" filter, so we skipped it here

Figure 16: Semantics applied on Figure 2 (continued (2))

(2) $\mathbf{C}[\![\mathrm{C}_0]\!]e\ \sigma_2\ \ \rho_e\ (\lambda\rho'.newstore)$
$= \mathbf{C}[\![\textbf{action}\ \texttt{switch(true)}\ \textbf{on}\ \mathrm{D}_1\ \textbf{with}\ \mathrm{F}_0]\!]$
$=$ let $\langle v, \rho' \rangle = \mathbf{D}[\![\texttt{l:Light}]\!]\rho_e$ in
$\langle [\![\texttt{l}]\!], \underbrace{update[\![\texttt{l}]\!]\ (\text{in}Interface([\![\texttt{Light}]\!]))\ \rho_e}_{[[\![\texttt{l}]\!] \mapsto \text{in}Interface([\![\texttt{Light}]\!])]\rho_e} \rangle$ in
let $\sigma_\rho = \lambda\rho.\ ...$ [let $a_\rho = \underbrace{(access\text{-}action[\![\texttt{switch}]\!]\ (access\ [\![\texttt{l}]\!]\ \rho)\ e\ \sigma)}_{(\lambda i_o.\lambda x.(update\text{-}event[\![\texttt{switch}]\!]\ i_o\ x))(access\ [\![\texttt{l}]\!]\ \rho)}$ in
let $f'_\sigma = \lambda\rho.\underbrace{(a_\rho\ (access\ [\![\texttt{l}]\!]\ \rho)\ (\mathbf{X}[\![\mathrm{X}]\!]\rho\ \sigma))\ (f_\sigma\ \rho)}_{[(access\ [\![\texttt{l}]\!]\ \rho) \mapsto \langle ..., [[\![\text{switch}]\!] \mapsto true]empty\rangle]newstore, \text{and let this be } \sigma'_2}$
$\mathbf{F}[\![\mathrm{F}_0]\!]\ [\![\texttt{l}]\!]\ \sigma_2\ \rho \rightarrow\ (f'_\sigma\ \rho)\ []\ newstore$
$\mathbf{F}[\![\texttt{room == m.room}]\!](access\ [\![\texttt{l}]\!]\ \rho)\ \sigma_2\ \rho$
$\underbrace{access\text{-}attribute[\![\texttt{room}]\!]\ (access\ [\![\texttt{l}]\!]\ \rho)\ \sigma_2}_{\text{let this be} \lceil \text{l.room} \rceil \rho\ \sigma_2}\ eq\ \mathbf{X}[\![\texttt{m.room}]\!]\rho\ \sigma_2$
(3)– see below

$= \langle \underbrace{[[\![\texttt{l}]\!] \mapsto \text{in}Interface([\![\texttt{Light}]\!])]\rho_e}_{\rho_a}, \underbrace{\lambda\rho.((\lceil \text{l.room} \rceil \rho\ \sigma_2)\ eq\ \cdots(3)\cdots) \rightarrow \sigma'_2\ []\ newstore)}_{a} \rangle$

where :

(3) $\mathbf{X}[\![\texttt{m.room}]\!]\rho\ \sigma_2$
$= \text{cases}\ access[\![\texttt{m}]\!]\ \rho\ \text{of}\ \text{in}Instance(i_o) \rightarrow\ (access\text{-}attribute[\![\texttt{room}]\!]\ i_o\ \sigma_2)$
$[]\ \text{in}Interface([\![\texttt{MotionDetector}]\!]) \rightarrow\ Undefinedvalue$

Where :

(4) $instantiate\ \sigma_2\ \rho_a$
$=$ let $P = \{[\![\texttt{m}]\!] \mapsto \text{in}Instance(j),$ with $j \in \{\texttt{m10, m20}\},$ and $(\sigma_2\ j)\downarrow_1 = [\![\texttt{MotionDetector}]\!]$
$[\![\texttt{l}]\!] \mapsto \text{in}Instance(k),$ with $k \in \{\texttt{l10, l11, l20}\},$ and $(\sigma_2\ k)\downarrow_1 = [\![\texttt{Light}]\!]\}$ in
$join^*\{((b\ \rho_{inst}) \rightarrow (a\ \rho_{inst})\ []\ newstore) \mid \rho_{inst} \in P\},$

Now, the b and a functions built in (1) and (2) can be fully evaluated with *e.g.*,:
i) $P = \{\rho_1, \rho_2, ...\},$ such that
$\rho_1 = \{[\![\texttt{m}]\!] \mapsto \text{in}Instance(\texttt{m10}), [\![\texttt{l}]\!] \mapsto \text{in}Instance(\texttt{l10})\},$
$\rho_2 = \{[\![\texttt{m}]\!] \mapsto \text{in}Instance(\texttt{m10}), [\![\texttt{l}]\!] \mapsto \text{in}Instance(\texttt{l11})\}, etc.$

ii) $\sigma_1 = [[\![\texttt{m10}]\!] \mapsto [[\![\texttt{room}]\!] \mapsto [\![\texttt{101}]\!][[\![\texttt{detected}]\!] \mapsto \texttt{false}]empty]$
$[[\![\texttt{l10}]\!] \mapsto [[\![\texttt{room}]\!] \mapsto [\![\texttt{101}]\!]]empty]\ newstore$
$[[\![\texttt{l11}]\!] \mapsto [[\![\texttt{room}]\!] \mapsto [\![\texttt{101}]\!]]empty]\ newstore$
$\sigma_2 = [[\![\texttt{m10}]\!] \mapsto [[\![\texttt{room}]\!] \mapsto [\![\texttt{101}]\!][[\![\texttt{detected}]\!] \mapsto \texttt{true}]empty]$ ...same as σ_1...$]newstore$

As a result, the store σ_{R1}, resulting from the application of $join^*$ equals :
$\sigma_{R1} = [[\![\texttt{l10}]\!] \mapsto \langle [\![\texttt{Light}]\!], \cdots entity\text{-}attributes \cdots, [[\![\texttt{switch}]\!] \mapsto true]empty\rangle][$
$[\![\texttt{l11}]\!] \mapsto \langle [\![\texttt{Light}]\!], \cdots entity\text{-}attributes \cdots, [[\![\texttt{switch}]\!] \mapsto true]empty\rangle]newstore$

7 Conclusion and Future Work

In this paper, we presented the realisation of a user-oriented domain-specific language, named Pantagruel, aimed to ease the development of entity orchestration applications. The design of Pantagruel is supported by a thorough analysis of the entity orchestration domain, bringing out the domain requirements and the needs of users. In addition, we used the formal methodology for language development proposed by David Schmidt [20]. It identifies the key concepts in language design and semantics. Throughout the realisation of Pantagruel, we shown how the denotational semantics of our language reflects the domain analysis, expressing the user-oriented concepts of the domain as first class domains of the semantics. Finally, we hope the example of Pantagruel serves as a basis to help and motivate language developers to provide a proper semantic definition of their DSL.

Once the semantics of our language is formally defined, it can be used both to derive implementations (*i.e.*, interpreters, compilers) and to reason about programs by expressing various program analyses. An interpreter for Pantagruel has been developed in OCaml; it is a straightforward mapping of the formal definition. An implementation of a compiler towards a Java platform has also been developed based on the semantics, and was used for a demonstration of Pantagruel at a conference [9].

As for future work, we will use the semantics of Pantagruel to perform static program analysis, such as conflict detection when rule execution affects the state of the same entities. To do so, we can leverage Nakata's work on the While reactive language [18]. Nakata uses the Coq proof assistant to perform property verifications on the While language, based on a denotational-style definition. We are also studying an approach to allowing end users to express area-specific properties, describing the intended behavior of their orchestration application, in a user-friendly paradigm. Because they are most qualified to determine what area-specific properties should be satisfied by orchestration applications, it is essential to provide them with high-level abstractions to make accessible the description of area-specific properties. To support the verification of orchestration applications against these properties, this work will build upon the formal semantics of Pantagruel. Another interesting direction based on the denotational semantics of Pantagruel, is to develop tools such as debuggers and profilers, and to make them accessible to end users.

Acknowledgment

The formal definition of a programming language can be intricate and dense. David Schmidt has made landmark contributions on this topic, providing researchers with a practical methodology to formalize a programming language definition without sacrificing underlying mathematical foundations. His approach makes the design of a language definition systematic, rigorous and tasteful. Resulting definitions are easier to understand by implementers, to use for formal reasoning, and to leverage to introduce non-standard semantics. This paper is a testimony to the fact that the appealing nature of David Schmidt's methodology is strong across generations of researchers in programming languages.

References

[1] Albert Benveniste, Paul Le Guernic, Yves Sorel & Michel Sorine (1992): *A Denotational Theory of Synchronous Reactive Systems*. *Information and Computation* 99(2), pp. 192–230, doi:10.1016/0890-5401(92)90030-J.

[2] Antonio Brogi & Jean-Marie Jacquet (1997): *Modeling Coordination via Asynchronous Communication*. In: *Coordination '97, LNCS*, Springer-Verlag, pp. 238–255, doi:10.1007/3-540-63383-9_84.

[3] Damien Cassou, Julien Bruneau, Charles Consel & Emilie Balland (2012): *Towards a Tool-based Development Methodology for Pervasive Computing Applications*. *IEEE Transactions on Software Engineering* 38(6), pp. 1445–1463, doi:10.1109/TSE.2011.107.

[4] Charles Consel (2004): *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, chapter From A Program Family To A Domain-Specific Language, pp. 19–29. *Lecture Notes in Computer Science* 3016, Springer-Verlag, London, UK, doi:10.1007/978-3-540-25935-0_2.

[5] Charles Consel & Renaud Marlet (1998): *Architecturing software using a methodology for language development*. In: *Proceedings of the 10 th International Symposium on Programming Language Implementation and Logic Programming, number 1490 in Lecture Notes in Computer Science*, pp. 170–194, doi:10.1007/BFb0056614.

[6] Arie van Deursen & Paul. Klint (1998): *Little Languages: Little Maintenance*. *Journal of Software Maintenance* 10(2), pp. 75–92, doi:10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

[7] Arie van Deursen, Paul Klint & Joost Visser (2000): *Domain-specific Languages: an annotated bibliography*. *SIGPLAN Notices* 35(6), pp. 26–36, doi:10.1145/352029.352035.

[8] Zoé Drey (2010): *Vers une méthodologie dédiée à l'orchestration d'entités communicantes*. Ph.D. thesis, PHOENIX - INRIA Bordeaux - Sud-Ouest.

[9] Zoé Drey & Charles Consel (2010): *A visual, open-ended approach to prototyping ubiquitous computing applications*. In: *PerCom Workshops*, pp. 817–819, doi:10.1109/PERCOMW.2010.5470549.

[10] Zoé Drey & Charles Consel (2012): *Taxonomy-driven prototyping of home automation applications: A novice-programmer visual language and its evaluation*. *Journal of Visual Langages and Computing* 23(6), pp. 311–326, doi:10.1016/j.jvlc.2012.07.002.

[11] Zoé Drey, Julien Mercadal & Charles Consel (2009): *A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications*. In: *DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, Springer-Verlag, Berlin, Heidelberg, pp. 78–99, doi:10.1007/978-3-642-03034-5_5.

[12] Martin Fowler (2010): *Domain Specific Languages*, 1st edition. Addison-Wesley Professional.

[13] Paul Hudak (1996): *Building Domain-Specific Embedded Languages*. *ACM Computing Surveys* 28, doi:10.1145/242224.242477.

[14] Gilles Kahn (1974): *The Semantics of Simple Language for Parallel Programming*. In: *IFIP Congress*, pp. 471–475.

[15] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw & Susan Wiedenbeck (2011): *The state of the art in end-user software engineering*. *ACM Computing Surveys* 43(3), pp. 21:1–21:44, doi:10.1145/1922649.1922658.

[16] Sheng Liang & Paul Hudak (1996): *Modular Denotational Semantics for Compiler Construction*. In: *European Symposium on Programming*, Springer-Verlag, pp. 219–234, doi:10.1007/3-540-61055-3_39.

[17] Marjan Mernik, Jan Heering & Anthony M. Sloane (2005): *When and how to develop domain-specific languages*. *ACM Computing Surveys* 37(4), pp. 316–344, doi:10.1145/1118890.1118892.

[18] Keiko Nakata (2011): *Resumption-based big-step and small-step interpreters for While with interactive I/O*. In: *DSL '11: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, pp. 226–235, doi:10.4204/EPTCS.66.12.

[19] David A. Schmidt: *An introduction to Programming-Language Semantics*. http://people.cis.ksu.edu/~schmidt/705s13/Lectures/chapter.pdf. A revision of the article in the CRC/ACM Computer Science Handbook, 2d ed., 2004.

[20] David A. Schmidt (1986): *Denotational semantics: a methodology for language development*. Allyn and Bacon, Boston, London.

[21] Diomidis Spinellis (2001): *Notable design patterns for domain-specific languages*. Journal of Systems and Software 56(1), pp. 91–99, doi:10.1016/S0164-1212(00)00089-3.

[22] Laurence Tratt (2008): *Domain specific language implementation via compile-time meta-programming*. ACM Transactions on Programming Languages and Systems 30(6), pp. 31:1–31:40, doi:10.1145/1391956.1391958.

[23] Qian Wang & Gopal Gupta (2005): *Rapidly prototyping implementation infrastructure of domain specific languages: a semantics-based approach*. In: SAC, pp. 1419–1426.