Formal Modeling and Verification of Services Managements for Pervasive Computing Environment

Hai Wan, Zoé Drey, Zhiyang You, Liu Liu

Formal Modeling and Verification of Services Managements for Pervasive Computing Environment

Hai Wan[1], **Zoe Drey**[2], **Zhiyang You**[3], **Liu Liu**[4]

[1]Department of Computer Science and Technology, Tsinghua University, China (wanh03@mails.tsinghua.edu.cn)
[2]Thales/LaBRI, France (zoe.drey@labri.fr)
[3]Department of Computer Science and Technology, Tsinghua University, China (youzy05@mails.tsinghua.edu.cn)
[4]School of Software, Tsinghua University, China (lliu07@mails.tsinghua.edu.cn)

ABSTRACT

Various forms of pervasive computing environments are being deployed in an increasing number of areas including hospitals, homes and military settings. Entities in this environment provide rich functionalities (*i.e.* services). How to organize these heterogeneous and distributed entities to deliver user-defined services is challenging. Pantagruel is an approach to integrate a taxonomical description of a pervasive computing environment into a visual programming language. A taxonomy describes the relevant entities of a given pervasive computing area and serves as a parameter to a sensor-controller-actuator development paradigm. The orchestration of area-specific entities is supported by high-level constructs, customized with respect to taxonomical information. Pantagruel is also a language that describes and manages services. Further more, Pantagruel can be viewed as a high level service contract between the service designer and the program implementer. This paper presents a formalization of Pantagruel, both its syntax and semantics. Four kinds of static properties are stated based on the formalization. Predicate abstraction based algorithms are designed to verify the properties.
Keywords: service management, pervasive computing, formal modeling and verification, predicate abstraction

1. Introduction

Various forms of pervasive computing environments are being deployed in an increasing number of areas including heathcare, home automation and building management. This trend is fueled by a constant flow of innovations in devices forming ever richer pervasive computing environments. These devices have increasingly more computing power offering high-level interfaces to access rich services.

The advent of this new generation of devices enables the development of pervasive computing systems to abstract over low-level embedded systems intricacies. This development is now mainly concerned with the programming of the orchestration of the entities to provide certain services. Yet, the nature of pervasive computing systems makes this programming very challenging. Indeed, orchestrating networked heterogeneous entities requires expertise in a number of areas, including distributed systems, networking, and multimedia. There exist middlewares and programming frameworks that are aimed to facilitate this task; examples include Gaia [1] and Olympus [2]. However, these approaches do not fill the semantic gap between an orchestration logic and its implementation because they rely on a general-purpose language and use large APIs. This situation makes the programming of the orchestration logic costly and error-prone, impeding evolutions to match user's requirements and preferences.

To circumvent this problem, visual approaches to programming the orchestration logic have been proposed, based on storyboards and rules [3, 4]. These approaches enable the programmers to express the orchestration logic using intuitive abstractions, facilitating the development process. However, this improvement is obtained at the expense of expressivity. Specifically, existing visual approaches are limited to a given area (*e.g.* CAMP magnetic poetry for the Smart Home domain [3]), a pre-defined set of categories of entities (*e.g.* in iCap rule-based interactive tool [4]), or abstractions that do not scale up to rich orchestration logic (*e.g.* Oscar service composition interface [5]).

In [6], we proposed Pantagruel, an expressive approach to developing orchestration logic that is parameterized with respect to a *taxonomy* of entities describing a pervasive computing environment. Specifically, our approach consists of a two-step process: (1) a pervasive computing environment is described in terms of its constituent entities, their functionalities and their attributes; (2) the development of a pervasive computing application is driven by a taxonomy of entities and consists of an orchestration logic which manipulates them using a high-level rule-based language.

Because the semantics of Pantagruel solely involves domain-specific aspects, Pantagruel can be reasoned about in a direct and simple way. The verification process therefore becomes high level and amenable to formal verification tools. Our goal is to verify Pantagruel against a set of properties. Namely, we want to show how Pantagruel semantics can guarantee the conflict-free property, termination property, dead-rule property and deadlock property of any pantagruel program. Because Pantagruel has a particular semantics , verifying these properties require a specific treatment (a re-ecrire). Specifically, Pantagruel does not specify the complete behaviors of methods: the information on a method is limited to the set of variables of the taxonomy that it may modify, when executed. Second, it introduces a specific constraint that relies on context change. Particularly, a Pantagruel rule is executed only if the evaluation of its conditions change from *false* to *true* between two discrete times. To take into account these execution characteristic, we propose an adaptation of the predicate abstraction principles [7] to verify general-purpose properties based on Pantagruel semantics.

The contributions of this paper are summarized as follows.

Contributions

- *A formal description of Pantagruel properties.* We propose to formalize four properties for reactive programming adapted to Pantagruel runtime semantics. These properties enable the verification of programs prior to their execution.
- *Verification algorithms that adapt predicate abstraction to Pantagruel.* Our definition of properties relies on the predicate abstraction principle. We propose an original customization of the predicate abstraction principle to the characteristics of Pantagruel, namely its method specification and the context change principle.

The paper is organized as follows. In Section 2, we give a brief introduction of Pantagruel and an informal description of the static properties that we are interested in. Section 3 presents some preliminaries for the sequent formalization. In Section 4, we give a formal presentation of Pantagruel's syntax and semantics, and based on them give the formal definitions of the static properties. Section 5 presents how to verify the properties. Section 6 concludes the paper.

2. Introduction of Pantagruel

Pantagruel is a visual domain specific language. Defining a Pantagruel programs consists of two steps: first, we define an environment, which abstracts over the variations of entities. In this step, we introduce a declarative approach to define a taxonomy of entities relevant to a given area. The entity declarations form an environment description that can then be instantiated for a particular setting. Then, we define an orchestration logic of the entities of a given instance of the taxonomy. Each rule collects context data from sensors, combines them with a controller, and reacts by triggering actuators. The sensors represent the condition part of the rules, and the actuators represent the actions that are triggered when the conditions are evaluated to true.

2.1. Defining an Environment

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. Furthermore, the declaration of an entity class consists of attributes defining a context and methods accessing the entity functionalities. Entity classes are organized hierarchically, allowing attributes and methods to be inherited (as shown in Fig. 1).

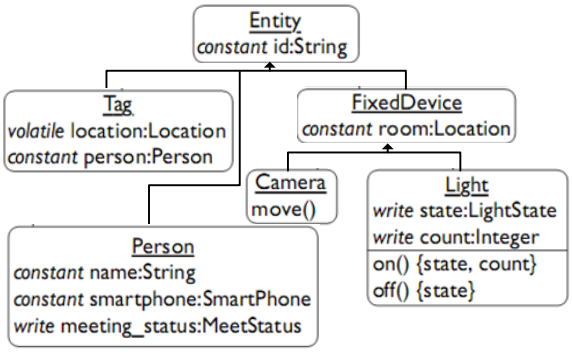

Fig. 1: A hierarchy of classes of entities

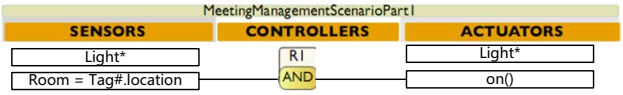

Fig. 2: Part of a scenario

- **Attributes.** An entity attribute may either be constant, writable or volatile. A *constant* attribute is a variable whose value does not change over time. For example, the attribute `room` of entity `Light` (note that `room` is inherited from `FixedDevice`). A *volatile* attribute is aimed to acquire information from the outside. This kind of attributes may correspond to sensors (*e.g.* a device reporting RFID tag location) or software components (*e.g.* a calendar reporting meeting events). A volatile attribute is read-only to the program. Lastly, a *writable* attribute corresponds to the data computed by the application. A writable attribute can be manipulated by the program.
- **Methods.** The functionalities of an entity class correspond to method interfaces. They are typed to further enable verification. For example, the `Light` entity class contains the `on()` method interface which intuitively is used to turn on the light. In Pantagruel, the interface of method only describes the set of writable attributes the method may modify. The method behavior, including the way it modifies the writable variables is not specified and is left to the method implementer.
- **Instantiating an environment description**

  Once the environment description is completed, it is used to define concrete environments by by instantiating the entity classes.

2.2. Defining Orchestration Rules

An orchestration logic is a set of rules. Each rule has a condition, a controller and an actuator. As demonstrated in Fig. 2, rule `R1` intuitively means that for all lights if there is a `Tag`, whose location is the same as the light location, then the method `on()` is triggered to turn on the light. For more information and detailed example, please refer to [6].

2.3. Compiling Pantagruel programs

The environment part of a Pantagruel program is compiled into a DiaSpec [8] (DiaSpec is an architecture description language dedicated to distributed systems) description; the orchestration rules part is compiled into Java code, supported by a DiaGen-generated programming framework. Leveraging on DiaSpec enables Pantagruel to gain access to a large number of heterogeneous entities, available in our Lab's smart space. Note that all the methods of entities in Pantagruel are compiled to empty methods in the resulting Java code. The Java code is then given to the implementers, who will implement all the empty methods.

2.4. Static Properties of Pantagruel

In this section, we informally present four static properties of Pantagruel, *i.e.* conflict free, termination, dead rule and deadlock. We will give formal definitions of these properties in Section 4.

- **Conflict free** In Pantagruel, we may consider some rule can not be triggered at the same time. These rules are said to be in conflict with each other. For example, suppose there are two rules: one is used to turn on the light and the other is used to turn off the light. Then these rules cannot be triggered at the same cycle. In Pantagruel, we can define a set of conflict-rule sets.
- **Termination** Termination means that the program won't be executed for ever. This informal description is proper for non-interactive systems but not proper for interactive systems, such as Pantagruel. In Pantagruel termination is defined as: if the volatile attributes do not change (this change is caused by the environment) after some time, then the system will eventually stops, *i.e.* no rule is triggered. The changes of volatile attributes is the reason why the system starts to trigger rules. The effects of the execution of rules may modify some writable attributes, which in turn may lead to another round of execution of rules. If the environment doesn't change volatile attributes after some time but the system doesn't finally stop – this is not a well-defined system behavior. For example, this may lead to an infinite alternated `on()/off()` operations of `light`.
- **Dead rule** A dead rule is a rule that cannot be triggered at all. Dead rules are not allowed in Pantagruel, since every rule should represent some intended behavior of the system.
- **deadlock** A system that has a deadlock may reach a stable state where no rule can be triggered anymore, whether the values of volatile attributes change or not. deadlock makes the system responseless to any stimuli of the environment. deadlock is not allowed in the system.

3. Preliminaries

Let V (C, resp.) be a set of *variables* (*constants*, resp.). Each variable $v \in V$ (constant $c \in C$, resp.) has its *domain*, which is denoted by D_v (D_c, resp.). A special value $\perp$ (*i.e. undefine*) is added to every domain D. The result domain is denoted by $D_\perp$. There are two predefined domains: $Int_\perp$ for integers and $Bool_\perp$ for Boolean. Let O be a set of *operators*. In this paper, we only consider unary and binary operators. *Terms* can be built on V, C and O. Given $o \in O$ $x\ y \in V \cup C$, if o is unary then $(o\ x)$ is a term; if o is binary then $(x\ o\ y)$ is a term. We use t_V to denote the variables used in term t.

When $o \in \{<, \leq, \geq, >, =, \neq\}$ (these operators have the traditional meanings), term $(x\ o\ y)$ is also called a *predicate*. Given a valuation of variables in V, the value of predicate $(x\ o\ y)$ can be calculated to *true* or $false$. A predicate p is *satisfiable* if there is a valuation on which p is calculated to *true*. A set of predicates $\mathbb{P}$ is *satisfiable* if there is a valuation on which all predicates in $\mathbb{P}$ is calculated to *true*. Predicate p_1 *implies* p_2, denoted by $p_1 \rightarrow p_2$, if p_2 is *true* on the valuations on which p_1 is *true*. Predicate set $\mathbb{P}$ *implies* predicate p, denoted by $\mathbb{P} \rightarrow p$, if p is *true* on the valuations on which $\mathbb{P}$ is *true*. A predicate p is *decidable under* predicate set $\mathbb{P}$ if $\mathbb{P} \rightarrow p$ or $\mathbb{P} \rightarrow \neg p$ holds. A set of predicates $\mathbb{P}$ can be also represented by $\mathbb{P}(x_1, \cdots, x_n)$ where $x_i (i \in [1, n])$ is used in $\mathbb{P}$.

Given a set S, $|S|$ denotes the cardinality of S and 2^S denotes the powerset of S.

4. Formal Representation of Pantagruel

In order to reason about Pantagruel formally, we propose here a formal representation of Pantagruel's syntax and semantics. We only consider portions of Pantagruel which are related to the verification. Some parts of Pantagruel, such as the hierarchy of entity classes, is abstracted away, since they are static structure which do not change during the execution. In the sequel, we first give the syntax, then the semantics, finally, the formal definitions of the four static properties are given.

4.1. Syntax

There are 6 *operators* in Pantagruel: $<$, $>$, $=$, $\neq$, $\in$ and ! (*i.e.* changed). The set of operators is denoted by O.

Scenario A *scenario* is a tuple $\Psi = (V, C, V_w, T, R)$, where V is a finite set of variables (*i.e.* attributes in Pantagruel), C is a set of constants, $V_w \subseteq V$ is the set of writable variables, T is a finite set of terms built on V, C and O, and R is a finite ordered set of rules. We use $V_v \triangleq V - V_w$ to denote the volatile variables.

A *rule* is of one of the following forms, where $P \subseteq T$ and $E \subseteq V_w$: 1) $(\bigwedge_{t \in P} t) \rightarrow E$; 2)$(\bigvee_{t \in P} t) \rightarrow E$

Given a rule r, the part before the arrow is called the *condition* of r, and the part after the arrow is called the *effected set* of r (*i.e.* the set of writable variables that may be modified by r). We use r_P, r_E and r_T to denote the condition of r, the effected set of r, and the set of terms used in r respectively.

From now on, we fix a scenario Ψ in our sequent discussions.

4.2. Semantics and Concrete Model

As described in Section 2, a Pantagruel program is compiled to a host language (such as Java) with all the methods unimplemented. This resulted program in the host language is called an *empty-framework*. After the implementers realize all the methods, we get a *completed-framework*. Note that, first, both empty and completed framework can be executed, but the empty framework is useless, since methods in an empty-framework do nothing; second, there are only one empty-framework but infinite completed-frameworks.

The pseudocode of the execution algorithm of a framework is demonstrated in Algorithm 1. The execution of a framework is an infinite loop. In each iteration, the changes of the environment are sampled and polled into the volatile variables (line 3), then the set of enabled rules are cal-

culated based on the previous and current states (line 6), finally, all the enabled rule are triggered one by one and the result is stored in state s' (lines 7–9). Note that in an empty-framework, the executions of methods just leave all the variables unchanged. There are two concepts in the algorithm needing further explanations: states and enabled rule.

Algorithm 1 The execution algorithm of Pantagruel

1: Let s, s' and s_tmp be three states. Initially, $s = s' = s\_tmp$ and $\forall v \in V, s(v) = \bot$.
2: **while** true **do**
3: $s' \leftarrow update\_volatile(V_v, s')$
4: $s \leftarrow copy(s\_tmp)$
5: $s\_tmp \leftarrow s'$
6: $Rs \leftarrow$ the set of enabled rules at (s, s')
7: **for all** $r \in Rs$ **do**
8: $s' \leftarrow eval\_rule(r, s')$
9: **end for**
10: **end while**

States A *state* is a type consistent valuation of variables in V. Let $\mathbb{S}$ denote the set of states. $s(v)$ denotes the value of variable v at state s.

Enables Rules Given a term t and two states s_1 and s_2, $(s_1, s_2) \models t$ denotes t is evaluated to *true* at s_1 and s_2. $(s_1, s_2) \models t$ is defined as follows (note that given a state s, if x is a constant, in order to make the representation concise we use $s(x)$ to represent the value of the constant):

- $(s_1, s_2) \models (x_1 < x_2)$ iff $s_2(x_1) < s_2(x_2) \wedge s_2(x_1) \neq \bot \wedge s_2(x_2) \neq \bot$;
- $(s_1, s_2) \models (x_1 > x_2)$ iff $s_2(x_1) > s_2(x_2) \wedge s_2(x_1) \neq \bot \wedge s_2(x_2) \neq \bot$;
- $(s_1, s_2) \models (x_1 = x_2)$ iff $s_2(x_1) = s_2(x_2) \wedge s_2(x_1) \neq \bot \wedge s_2(x_2) \neq \bot$;
- $(s_1, s_2) \models (x_1 \neq x_2)$ iff $s_2(x_1) \neq s_2(x_2) \wedge s_2(x_1) \neq \bot \wedge s_2(x_2) \neq \bot$;
- $(s_1, s_2) \models (x \in D)$ iff $s_2(x) \in D \wedge s_2(x) \neq \bot$;
- $(s_1, s_2) \models (!x)$ iff $s_1(x) \neq s_2(x) \wedge s_2(x) \neq \bot$.

A term t is *disabled* at states (s_1, s_2) if $\exists v \in t_V, s_2(v) = \bot$. It is obvious that if t is disabled at (s_1, s_2) then $(s_1, s_2) \not\models t$.

$(s_1, s_2) \models P_r$, which means rule r's condition is true at states (s_1, s_2), is defined as: 1) $(s_1, s_2) \models \bigwedge_{t \in r_T} t$ iff $\forall t \in r_T, (s_1, s_2) \models t$; 2) $(s_1, s_2) \models \bigvee_{t \in r_T} t$ iff $\exists t \in r_T, (s_1, s_2) \models t$.

From now on, for simplicity we only consider integers and their associated operations *i.e.* $\{<, >, =, \neq\}$ on them. The "$\in$" operator can be expressed using "=" based on a mapping from the domain to integers. Boolean can be dealt in a similar manner.

A term $t \in T$ is *active* at states pair (s_1, s_2) if:

- $(s_1, s_2) \models t$, where t is of form "$!x$";
- $s_1 \not\models t \wedge s_2 \models t$, otherwise.

A rule r is *enabled* at states (s_1, s_2) if $(s_1, s_2) \models r_P \wedge \exists t \in r_T, t$ is active at (s_1, s_2).

Traces and Runs A *trace* is an infinite sequence of states: $s_0 s_1 \cdots$. $s[i]$, $s[\cdots i]$ and $s[i \cdots]$ denotes the i-th state, the prefix $s_0 \cdots s_i$ and the suffix $s_i s_{i+1} \cdots$ of a trace respectively. Two traces τ_1 and τ_2 are equal if $\forall i \in \mathbb{N}, \tau_1[i] = \tau_2[i]$. Two prefixes $\tau_1[\cdots i_1]$ and $\tau_2[\cdots i_2]$ are equal if $\exists k \in \mathbb{N}, i_1 = k \wedge i_2 = k \wedge \forall i \leq k, \tau_1[i] = \tau_2[i]$.

A trace can be derived from a real execution of a framework: in each iteration, one state – state "s" in the 4-th line of algorithm 1– is sampled to form the trace. Here we want to get all the traces that can be produced by a framework (*i.e. runs*).

In order to get a uniform representation, implicitly there is a state $\sigma[-1]$, in which all variables are set to $\bot$.

Given a framework f of scenario Ψ, σ is a *run* of f if the following conditions hold:

- for each $v \in V_w, \sigma[0](v) = \bot$; (this means that all writable variables are $\bot$ at the first state.)
- for all $i \in \mathbb{N}$, let Rs be the set of enabled rules at $(\sigma[i-1], \sigma[i])$, execute rules in Rs one by one on $\sigma[i]$ cumulatively, and the result $\sigma[i]$ agrees with $\sigma[i+1]$ on each writable variable.

The set of runs of framework f is denoted by Δ_f.

Given a run σ, two functions are deployed to describe the behaviors of the environment and the Pantagruel system:

- $E_\sigma : \mathbb{N} \to Bool$: $E_\sigma(i) = true$ iff $\exists v \in V_v, \sigma[i-1](v) \neq \sigma[i](v)$. Intuitively, $E_\sigma(i)$ expresses whether the environment changes the volatile variables during the i-th iteration.
- $RT_\sigma : \mathbb{N} \to 2^R$: $RT_\sigma(i)$ is the set of enabled rules at states $(\sigma[i-1], \sigma[i])$.

The behaviors of a framework f is described by Δ_f. The states and runs form the *concrete model* of a framework.

4.3. Formal Definitions of Static Properties

In this section, we give the formal definitions of the four static properties based on the definitions given in previous sections.

Definition 1 (Static properties) *Here are the definitions of the four static properties:*

- Conflict free *Let $X \subseteq 2^R$ be the conflict sets, a framework f is conflict free if* $\forall C \in X\ r_1 \in C\ r_2 \in C\ \sigma \in \Delta_f\ i \in \mathbb{N}, r_1 \neq r_2 \to \neg(r_1 \in RT_\sigma(i) \wedge r_2 \in RT_\sigma(i))$.
  A scenario is conflict free if all its frameworks are conflict free.
- Dead rule *A rule r is a dead rule of a framework f if* $\forall \sigma \in \Delta_f\ i \in \mathbb{N}, r \notin RT_\sigma(i)$.
  A rule is a dead rule of a scenario if it is a dead rule of all the frameworks of the scenario.
- Termination *A framework f terminates if* $\forall \sigma \in \Delta_f\ i \in \mathbb{N}, (\forall j \in \mathbb{N}, i \leq j \to E_\sigma(j) = false) \to (\exists k \in \mathbb{N}, i \leq k \wedge RT_\sigma(k) = \emptyset)$.

A scenario terminates if all its frameworks terminates.

- deadlock free *A framework f has a deadlock if* $\exists\sigma \in \Delta_f\ i \in \mathbb{N}, (\forall\sigma' \in \Delta_f, \sigma[\cdots i] = \sigma'[\cdots i] \rightarrow \forall j \in \mathbb{N}, i \leq j \rightarrow RT_{\sigma'}(j) = \emptyset)$.
  A scenario is deadlock free if all its framework have not a deadlock.

5. Verification of Properties

From the property definitions in section 4.3 we know that there is a link between a scenario and its frameworks. There is a universal quantifier in all definitions. Hence, a scenario has a property if and only if all its frameworks also have the property. For instance, if we can verify that a scenario is deadlock free, then we need to verify all its frameworks to show they that are deadlock free. Since the set of frameworks is infinite (*i.e.* the number of concrete models is infinite), it is impossible to verify the properties of a scenario by checking all its frameworks. Furthermore, when verifying a framework (*i.e.* concrete model), its state space is also infinite, since we have integer data type in our program. Thus an abstract model is needed to do the verification. The abstract model should be finite and sound and complete according to the static properties. In this section, We first introduce the abstract model, then give definitions of the static properties, finally describe the verification algorithms.

5.1. Abstract Model of a Scenario

Predicate abstraction [7] is deployed to construct an abstract model, in which a state is represented by a set of predicates and related to a (finite or infinite) set of states in the concrete model. The idea using a set of predicates to represent a set of concrete is inspired by [9]. Before giving the formal definition of abstract states, some preprocessing and definitions are needed.

Preprocessing and Definitions The preprocessing is divided into two phrases. Recall that T is the set of terms in scenario Ψ and O is the set of operators.

1. Let $\bar{T} \triangleq \{t \in T |$ t is not of form "$!x''\} - \bar{T}$ only contains terms of the form $x\ o\ y$ ($o \in \{<, >, =, \neq\}$);
2. extend $\bar{T}$ to T' such that T' is the smallest set that contains T and for each term $x\ o\ y \in T$, $|T' \cap \{x = y, x < y, x > y\}| \geq 2$, where $o \in O$.
3. if there is no constant used in T then skip this step.
   (a) let *max* and *min* be the maximal and minimal constants used in T respectively;
   (b) extend T' to T^e such that T^e is the smallest set that contains T' and for each variable x, $|T^e \cap \{x = max, x < max, x > max\}| \geq 2$ and $|T^e \cap \{x = min, x < min, x > min\}| \geq 2$.

States in Abstract Model In the concrete model, the enabled rules are calculated based on two adjacent states. We want the enabled rules can be calculated based on one abstract state, hence we need to add some auxiliary information to an abstract state, *i.e.* , the $true/false$ value of terms at the previous cycle (the $\mathbb{T}$ part in an abstract state) and how the value of a variable changes (*i.e.* increase, decrease, stay the same or undefine yet) (the $\mathbb{C}$ part in an abstract state). A state in the abstract model consists of:

- a function $\mathbb{T}$ of type $\bar{T} \rightarrow Bool$;
- a function $\mathbb{C}$ of type $V \rightarrow \{\uparrow, \downarrow, \rightarrow, \perp\}$;
- a set $\mathbb{G}$ of predicates. There are two cases:
  - if there is no constant used in T, then each element in $\mathbb{G}$ is of one of the following forms: $x = y$ and $x < y$, where $x\ y \in V$;
  - otherwise, each element in $\mathbb{G}$ is of one of the following forms: $x = y$, $x < y$, $x = c$, $x < c$, $c < x$, and $x + c < y$, where $x\ y \in V$ and c is a constant in $[min, max]$.

  Given an abstract state $s = (\mathbb{T}, \mathbb{C}, \mathbb{G})$, let $\mathbb{T}^e_s$ denote $\{t \in T^e | \forall v \in t_V, \mathbb{C}(v) \neq \perp\}$. Intuitively $\mathbb{T}^e_s$ contains all the terms in T^e that are not disabled. Given a function f of type $\mathbb{T}^e_s \rightarrow Bool$, let $\mathbb{P}_f$ denote $\{x < y | f(x < y) = true\} \cup \{x = y | f(x = y) = true\} \cup \{y < x | f(x < y) = false \wedge f(x = y) = false\}$. Let $\mathbb{V}_s = \{v | \mathbb{C}(v) \neq \perp\}$. Let $\mathbb{R}_s$ be the enabled rules at s_1. Let $\mathbb{E}_s \triangleq (\exists v \in V_v, \mathbb{C}(v) \neq \perp \wedge \mathbb{C}(v) \neq \rightarrow)$ indicates whether the environment changes the volatile variables.
  An abstract state satisfies : 1) all predicates in $\mathbb{T}^e_s$ are decidable under $\mathbb{G}$; 2) if $x + c < y \in \mathbb{G}$, then $\mathbb{G}$ implies $min \leq x$, $x \leq max$, $min \leq y$ and $y \leq max$; 3) $\mathbb{G}$ only involves variables in $\mathbb{V}_s$.

Let $\mathbb{A}$ be the set of abstract states. It is not hard to prove the following theorem:

Theorem 1 $|\mathbb{A}|$ *is finite.*

Initial States An initial state $s = (\mathbb{T}, \mathbb{C}, \mathbb{G})$ satisfies:

- $\forall t \in \bar{T}, \mathbb{T}(t) = false$;
- $\forall v \in V_w, \mathbb{C}(v) = \perp$;
- There exists a function f of type $\mathbb{T}^e_s \rightarrow Bool$ such that $\mathbb{P}_f$ is satisfiable and $\mathbb{G} = \mathbb{P}_f$.

Next-state Function A next-state function nsf is of type $\mathbb{A} \rightarrow 2^{\mathbb{A}}$. Given a states $s_1 = (\mathbb{T}_1, \mathbb{C}_1, \mathbb{G}_1)$, a next state $s_2 = (\mathbb{T}_2, \mathbb{C}_2, \mathbb{G}_2)$ can be constructed following:

- $\mathbb{T}_2$ can be calculated based on $\mathbb{G}_1$;
- Let $E_w = \bigcup_{r \in \mathbb{R}_{s_1}} r_E$ be the writable variables that may be modified by rules in Rs. $\mathbb{C}_2$ only need to satisfy that $\forall v \in E_w \cup \mathbb{V}_{s_1}, \mathbb{C}_2(v) \neq \perp$. It is not hard to verify that $\mathbb{V}_{s_1} \subseteq \mathbb{V}_{s_2}$ hold. In order to distinguish the variables in s_1 and s_2, all variables in s_1 are primed.
- Let f be a function of type $\mathbb{T}^e_{s_2} \rightarrow Bool$, let $\mathbb{G}'$ be $\mathbb{P}_f \cup \mathbb{G}_{s_1} \cup \{x' < x | \mathbb{C}_2(x) = \uparrow\} \cup \{x' = x | \mathbb{C}_2(x) = \rightarrow\} \cup \{x < x' | \mathbb{C}_2(x) = \downarrow\}$. It can be proved that $\mathbb{G}'$ can be reduced to $\mathbb{G}$ only involving variables in $\mathbb{V}_{s_2}$. Let $\mathbb{G}_2 = \mathbb{G}$.

It can be proved that following these steps, we indeed get an abstract state.

5.2. Abstract Trace and Run

Following the same manner described in Section 4.2, we have the *abstract traces*. An *abstract run* is a trace σ such that *sigma*[0] satisfies the initial condition and $\forall i, \sigma[i+1] \in nsf(\sigma[i])$. The set of abstract runs is denoted by Λ.

5.3. Definitions of Abstract Properties

Definition 2 (Abstract properties) *Based on the abstract model, we define the abstract counterparts for the static properties:*

- Conflict free *Let* $X \subseteq 2^R$ *be the conflict sets, the abstract model is conflict free if* $\forall C \in X\ r_1 \in C\ r_2 \in C\ \sigma \in \Lambda\ i \in \mathbb{N}, r_1 \neq r_2 \rightarrow \neg(r_1 \in \mathbb{R}_{\sigma[i]} \wedge r_2 \in \mathbb{R}_{\sigma[i]})$.
- Dead rule *A rule r is a dead rule of the abstract model if* $\forall \sigma \in \Lambda\ i \in \mathbb{N}, r \notin \mathbb{R}_{\sigma[i]}$.
- Termination *The abstract model terminates if* $\forall \sigma \in \Lambda\ i \in \mathbb{N}, (\forall j \in \mathbb{N}, i \leq j \rightarrow \mathbb{E}_{\sigma[j]} = false) \rightarrow (\exists k \in \mathbb{N}, i \leq k \wedge \mathbb{R}_{\sigma[k]} = \emptyset)$.
- deadlock free *The abstract model has a deadlock if* $\exists \sigma \in \Lambda\ i \in \mathbb{N}, (\forall \sigma' \in \Lambda, \sigma[\cdots i] = \sigma'[\cdots i] \rightarrow \forall j \in \mathbb{N}, i \leq j \rightarrow \mathbb{R}_{\sigma'[j]} = \emptyset)$.

The following theorem states that the verification problem of a scenario is reduced to the verification problem of an abstract model, which is feasible.

Theorem 2 *A scenario is conflict free iff its abstract model is conflict free. A scenario terminates iff its abstract model terminates. A scenario is deadlock free iff its abstract model is deadlock free. A rule r is a dead rule of a scenario iff r is a dead rule of its abstract model.*

5.4. Verification Algorithms

By theorem 1, we know the set of abstract states is finite. Hence, we can construct a graph using abstract states and the next-state function. Based on the graph, we design algorithms to do the verification for the static properties. The main operations of these algorithms are traversing a graph and finding a cycle in a graph. Take the verification of termination property for example, the main steps are: 1) remove all the states s such that $\mathbb{E}_s = true$ and their associated edges; 2) if there exists a cycle which can be reached from the initial states, then the termination property does not hold. The other algorithms for the rest properties are similar.

6. Concluding Remarks

Pervasive computing is reaching an increasing number of areas, creating a need to factorize knowledge about the entities that are relevant to each of these areas. Pantagruel is an approach and a tool that integrate a taxonomical description of a pervasive computing environment into a visual programming language. Rules are developed using a sensor-controller-actuator paradigm, parameterized with respect to a taxonomy of entities. In this paper, we give a formalization of Pantagruel, both its syntax and semantics. Transition systems are adopted to describe the behaviors of a Pantagruel program. Based on that, four static properties are formally defined. Since the state space of the transition system is infinite, in order to verify the properties, according to the static properties a sound and complete abstract model is derived. Algorithms for the verification are designed. A tool realizing the algorithm is now under development.

REFERENCES

[1] M. Roman and R. H. Campbell, "Gaia: enabling active spaces," in *9th ACM SIGOPS European Workshop*. ACM, 2000, pp. 229–234.

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

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

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

[5] M. W. Newman, A. Elliott, and T. F. Smith, "Providing an integrated user experience of networked media, devices, and services through end-user composition," in *6th International Conference on Pervasive Computing (Pervasive)*. Springer, 2008, pp. 213–227.

[6] Z. Drey, J. Mercadal, and C. Consel, "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*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 78–99.

[7] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM New York, NY, USA, 2002, pp. 191–202.

[8] W. Jouve, J. Lancia, N. Palix, C. Consel, and J. Lawall, "High-level programming support for robust pervasive computing applications," in *6th Int'l Conference on Pervasive Computing and Communications (PerCom)*, 2008, pp. 252–255.

[9] A. Miné, "A new numerical abstract domain based on difference-bound matrices," in *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*. London, UK: Springer-Verlag, 2001, pp. 155–172.