Modelling Digital Avatars: A Tuple Space Approach[⋆]

Alejandro Pérez-Vereda[*], Carlos Canal, Ernesto Pimentel

ITIS Software, Universidad de Málaga, Spain

Abstract

The development of the Internet of Things (IoT) came with the manufacturing of a huge amount of smart things equipped with sensors for making them aware of their environment, and with network connection for allowing remote interaction with them. However, most smart things still lack enough autonomy and context-awareness, hindering them from being people-friendly and actually useful for their users' everyday tasks. IoT devices should take advantage of their sensors and smartness to react automatically to the needs of their users and to provide seamless interactions with them. Within this field, the authors work on the design of Digital Avatars, a mobile computing framework for dynamically programming interactions among smart devices. The framework is based on the virtual profile of the user, which is inferred, stored, and shared by their smartphone. The profile provides a personalized context for running scripts which interact with IoT devices. This way, smartphones become a digital avatar of the user, capable of acting as a personal and seamless interface with their IoT environment. In this work, we present a formalization of Digital Avatars by means of a Linda-based approach with multiple shared tuple spaces. By means of a case study, we show how properties of the systems can be proved, and we briefly describe an implementation of both the Digital Avatars framework and the case study.

Keywords: Digital Avatars, digital avatar, virtual profile, Internet of Things, IoT, Linda, Multiple Tuple Spaces

[⋆]This work has been funded under grants PGC2018-094905-B-I00 (Spanish Government) and UMA18-FEDERJA-180 (Junta de Andalucía/ATech/FEDER).

[*]Corresponding author

Email addresses: `apvereda@uma.es` (Alejandro Pérez-Vereda), `carloscanal@uma.es` (Carlos Canal), `epimentel@uma.es` (Ernesto Pimentel)

1. Introduction

The Internet of things (IoT) is an ecosystem of devices, sensors, objects of any kind, and people connected to the Internet for transferring data and communicating over the network. Among them, smart things [1] are those endowed with sensors and processing capabilities, and which offer interfaces both to access the information they collect, and to configure how they work (e.g. the frequency to pick up the data, or how to reset them) [2]. However, these smart things still require a lot of manual configuration, and this becomes a challenging problem the bigger the number of smart devices we daily interact with. In a desirable scenario, technology should work for the people and not the other way round. Every smart thing should adapt to the needs of the people dynamically.

Taking advantage of the pervasive presence of smartphones, we advocate for using them to learn about their owners, creating and storing virtual profiles describing their preferences and other context information about them. This way, smartphones become seamless and automatic interfaces that can negotiate in the name of their owners, based in the information stored in their virtual profiles, and adapting and configuring the smart things in their surroundings to their wishes. The more comprehensive these virtual profiles are, the better technology may adapt to the people. For that purpose, our proposal complements virtual profiles with interaction mechanisms that allow to configure smart things, and also to complete the virtual profiles themselves with context knowledge obtained from these interactions. These virtual profiles extended with behaviour is what we call *digital avatars*.

With this goal in mind, we are currently developing Digital Avatars, a programming framework which allows to specify the interactions between smartphones and smart things by means of scripts which are downloaded and run on-the-fly [3]. These scripts are executed in the smartphone, making use of the virtual profile stored in it, and then reconfiguring the behavior of the smart things with the context information of the user.

In this paper, we present a formal framework for Digital Avatars. The framework provides a formal description of virtual profiles and of the scripts to execute on them, and it establishes the basis for issues like privacy or security, with secure connections controlling the access to virtual profiles. The formalization is based on a multiple shared tuple spaces model inspired by Linda, which makes possible to ensure the soundness of the framework, and also the analysis of some interesting properties. We illustrate our proposal by means of a case study: a Treasure Hunt in which several participants compete for finding a number of hidden treasures.

This work is an extension of [4], presented at the 17th International Workshop on Coordination an Self-Adaptativeness of Software Applications (FOCLASA 2019). Apart from a thorough reworking of many parts of that publication, in this paper we extend the formal model in [4] with a new primitive (*nrout*), a remote output operation that blocks when the tuple being written is already present in the tuple space, and we extend the transition system of our framework accordingly (Section 5). As it will be shown, the *nrout* primitive helps avoiding some undesirable situations, in particular when an exclusive write operation is required, and it allows to prove additional properties of interest for the case study.

Furthermore, we have also included in Section 4 an implementation of the case study that uses Eddystone beacons for representing smart things. The implementation consists in an Android application that downloads and runs scripts in the context of the virtual profile stored in the smartphone. The implementation shows how one single and generic application is able to handle dynamic and personalized interactions with the IoT in a seamless way, encouraging us to engage in a deeper development of the Digital Avatars framework in order to explore in full the possibilities that it offers.

The rest of this paper is structured as follows. In Section 2 we present Digital Avatars and define the concepts necessary to reason on them. In Section 3, we formalize the interactions that take place in the framework and formally demonstrate interesting properties of the system. Then, Section 4 presents the case study as a proof of concept of our proposal, and it analyzes several of its properties by means of the formal framework. Next, Section 5 extends the framework with the new primitive already mentioned, and provides a refined implementation of the case study, where additional properties can be proved. In Section 6 we discuss some related works. Finally, Section 7 draws the conclusions of the paper and briefly outlines some future works.

2. Digital Avatars

In order to define a formal framework for reasoning on Digital Avatars, we formally introduce the notion of virtual profile together with a number of related concepts, and we describe how virtual profiles can be exploited under the Digital Avatars framework.

2.1. Definitions

The key issue for taking into account the user in an IoT environment is her virtual profile. It contains information about user preferences, habits, relations, or places visited. According to our proposal, all this information is only stored in the user's device (e.g. a smartphone), where it can be offered as a service to third parties. The definition below formalizes this notion.

Definition 1. A *virtual profile* P is a multiset of entities, where each entity is a tuple $t = <name, v_1, \ldots, v_i, \ldots, v_n>$, where $name \in Name$ represents the name of the entity, and $v_1, \ldots, v_i, \ldots, v_n$, each $v_i \in Value$, is a sequence of values with a structure which depends on the entity itself. We will denote by T the set of tuples, and by $\mathcal{P}$ the set of virtual profiles.

The entities of a virtual profile are structured in nested sections, for a better organization of the information contained in the profile. Although the model does not depend on how sections and entities are defined, we commonly consider a predefined structure characterized as follows:

`personal` It consists in personal and contact information of the user (`personal` $\in$ *Name*), containing a collection of entities like `name`, `phone`, `address`, or `email`, all of them also in *Name*, with the corresponding information.

`relations` It provides information on how users are related to each other, including subsections identified as `family`, `friends`, `colleagues`, or `acquaintances`. These are collections of entities representing people related to the user and information about them: certificate hash fingerprints, user profiles in social networks, etc.

`places` It defines information concerning locations (home, office, etc.) and places visited. Thus, entities such as `home` or `work`, both in *Name*, are stored in this section.

These sections can be further tailored for specific domains, for which additional sections can be defined, too. Moreover, the actual implementation of entities in virtual profiles will typically include additional elements, such as type information, or a timestamp for recording the time when it was added to the virtual profile. However, here we omit all this additional information as it is not relevant for the formalisation presented in this work.

Virtual profiles can be accessed and/or modified by means of processes executing appropriate actions. In order to formalize this idea, we are inspired by Linda [5], a coordination language [6] consisting of a set of inter-agent communication primitives, which can be virtually added to any programming language. The primitives in Linda allow processes to add, read, and delete tuples from a shared *tuple space*. Tuple spaces are a convenient approach to represent information shared by concurrently running processes, and a virtual profile will be represented by a multiset of tuple entities encapsulated in a device.

In particular, we adopt a multiple tuple space model, where each tuple space represents a virtual profile and contains tuples with relevant information about its owner (e.g. `<personal.name,Pérez-Vereda,Alejandro>`). In addition, virtual profiles also store tuples with auxiliary information for profile management, like read/write access rights to the tuples. For instance, a tuple `<access,personal.name/2,read>` would grant reading access to the entity `name` (containing two value arguments) in the `personal` section of the virtual profile. Access rights and other security issues will be further explained in Section 2.2.

Following other approaches [7, 8], we shall consider a process algebra $\mathcal{L}$ including the Linda communication primitives and the usual concurrency connectives, parallel and non-deterministic choice. The primitives permit to add a tuple (*out*), to remove a tuple (*in*), and to check the presence (or absence) of a tuple (*rd*, *nrd*) in a given profile (tuple space).

Processes in $\mathcal{L}$ provide a convenient way to model scripts which can be downloaded from a smart thing and run on a smart device. Thus, the syntax of $\mathcal{L}$ is formally defined as follows:

$$\begin{array}{rcl} S \in \mathcal{L} & ::= & 0 \;\mid\; \alpha.S \;\mid\; S + S \;\mid\; S \parallel S \;\mid\; S(\tilde{t}) \\ \alpha \in \mathit{Act} & ::= & \mathit{rd}(t) \;\mid\; \mathit{nrd}(t) \;\mid\; \mathit{in}(t) \;\mid\; \mathit{out}(d, t) \end{array}$$

where 0 denotes the empty process, $d \in D$ a device identifier, and t denotes a tuple. The process $S(\tilde{t})$ denotes a procedure call where the procedure definition will be given by a script template $S(\tilde{x})$ (where $\tilde{x}$ is a sequence of variables instantiated by a sequence of tuples $\tilde{t}$). In order to simplify the definition of the rules modelling the primitive actions in $\mathcal{L}$ in Subsection 3.1, we will assume, as it is usual in Linda-based languages, that reading a tuple does not imply the evaluation of operations (e.g. arithmetic operations), nor the instantiation of variables. This assumption does not imply any loss of generality of the proposal.

Notice that the primitives allow accessing, adding, and removing tuples from a local tuple space (i.e. the virtual profile stored in the device executing the primitives). Although we could have also considered accessing and deleting tuples from remote tuple spaces, for our purposes we only need to add tuples remotely. For this reason, only the *out* primitive includes as a parameter the device on which adding the tuple. That is, *rd*, *in* and *nrd* actions will only be made locally. The same considerations were made in [8]. As it will be shown later, remote adding of tuples will only affect the smart artefact where the script was downloaded from, thus we will not allow arbitrary remote adding of tuples. This asymmetric treatment of read and out primitives is one of the distinctive features of the Digital Avatars model: local access is only allowed to the device itself, and remote changes can only be made on artifacts providing the scripts being run.

In our framework, we distinguish two kinds of artifacts: *smart devices* and *smart things*. The difference between them is that smart devices exhibit computing capabilities, and therefore they can download and execute scripts, whereas smart things only provide a (link to a) script.

Formally, we define an artifact as a pair consisting of a virtual profile and a process corresponding to the execution of one or several scripts. We assume that D is a set of artifact identifiers. Every artifact, with a unique identifier d, has also associated a script definition $S_d(\tilde{x})$ (a process with parameters $\tilde{x}$) which can be downloaded by other artifacts with computing capabilities (i.e. smart devices). Before downloading a script from an artifact, its parameters will be conveniently instantiated by information coming from the artifact's virtual profile. The following definition specifies what an artifact is.

Definition 2. An *artifact* $d \in D$ is characterized by a pair $\langle P : S \rangle_d$, including a virtual profile P and a process $S \in \mathcal{L}$, corresponding to the scripts running on the artifact. In addition, an artifact may contain a script definition $S_d(\tilde{x})$. We will denote by $S_d(P)$ the script instantiated by the specific tuples in the profile P. We will represent by $\mathcal{D} = \mathcal{P} \times \mathcal{L} \times D$ the set of artifacts.

A smart thing will be characterized by having only a profile; that is, it is deprived of computing capabilities, and thus its process is always the empty process 0. A typical example of smart thing would be a beacon broadcasting a Bluetooth signal which encodes the URL of a script file to be downloaded from a server. On the other hand, typical smart devices are smartphones, tablets, or any other device with computing capabilities. Both kinds of artifacts —smart things and smart devices— store information in a virtual profile.

2.2. Security in Digital Avatars

The actions executed over the virtual profile of a smart device may emerge from internal processes of the device, or they may be part of a script downloaded from another artifact, such as a beacon broadcasting a link to a script file as mentioned above. In this case several conditions must be fulfilled: the smart device is close enough to the beacon, the beacon artifact is registered, its script code is trusted, etc. To manage the conditions in which a given action can be executed in a smart device we define three mappings governing the different scenarios we may find. A *links* mapping which informs about the feasibility of establishing a connection between two given artifacts, a *certify* mapping to avoid running untrusted scripts, and an *accept* mapping to establish when a particular action is acceptable for a script running over a certain virtual profile. All these mappings are implemented by means of auxiliary tuples stored in the virtual profiles themselves.

First, for validating the identity of trusted artifacts from which to download scripts, we assume a Certificate Authority capable of ensuring the trustfulness of an artifact d, and a Boolean mapping *certify* which provides this information in such a way that $\mathit{certify}(d, P)$ is true when the emitter d has been authenticated by the profile P. In order to consider the authentication process as part of the tuple space framework, we will assume that when a device accesses a certified artifact d, a tuple `<certified,d>` is added to the profile of that device. Thus, the formal definition of *certify* is:

$$\mathit{certify} : D \times \mathcal{P} \longrightarrow \{\mathit{true}, \mathit{false}\}$$

in such a way that:

$$\mathit{certify}(d, P) = \mathit{true} \;\; \mathit{iff} \;\; <\!\mathit{certified}, d\!> \in P \tag{1}$$

In addition, in order to guarantee that the actions executed while running a script on a smart device are permitted (which may depend, for instance, on the level of privacy defined on each entity in its virtual profile), we assume a Boolean mapping

$$\mathit{accept} : \mathit{Act} \times \mathcal{P} \longrightarrow \{\mathit{true}, \mathit{false}\}$$

that restricts the action primitives that are enabled for a given profile, in such a way that $\mathit{accept}(\alpha, P)$ is true when the action α is acceptable on the profile P. Formally,

$$accept(\alpha, P) = \begin{cases} true & if\ \alpha = rd(t) \wedge <access, t, read> \in P \\ true & if\ \alpha = nrd(t) \wedge <access, t, read> \in P \\ true & if\ \alpha = out(d,t) \wedge <access, t, write> \in P \\ false & otherwise \end{cases} \quad (2)$$

The definition of *accept* above depends on `access` tuples stored in the virtual profile of the artifact. It is worth noting that the access rights for a given entity are either granted or forbidden to anyone. More precise access rights could be considered by including a *device* parameter in the *accept* mapping, and an additional value in `access` tuples for precising the scope or level of the access right. This would be similar to the definition of the visibility of attributes and methods in object-oriented languages (i.e. considering public, private, and friend access levels).

An additional consideration on security comes from restricting who is allowed to remotely add tuples to a given profile. As mentioned in the previous section, the *out* primitive is used for adding tuples to both local and remote virtual profiles. Although the model imposes no limitations on which profiles can be remotely modified, for security reasons the artifact identifier d used in a remote *out(d,t)* in a script $S_d(\tilde{x})$ can only be that of the artifact d itself. Thus, an artifact's profile may only be remotely modified by running a script previously downloaded from this same artifact. These protects profiles from unwanted writing actions coming from unknown sources.

Apart from certificates and access rights, and restricting remote addition of tuples, we also need to consider some technical issues about the feasibility of the connection itself. Indeed, whereas *certify* provides a third-party declaration about the trust of an artifact for downloading scripts from it, and *accept* controls which actions are permitted inside an artifact once a script has been downloaded, we need a way to detect when two artifacts are actually able to communicate with each other, from a technical point of view.

For instance, consider a scenario where a smartphone storing a virtual profile P approaches a smart thing d which provides a script $S_d(\tilde{x})$. For downloading the script from d and running it in the smartphone, we assume a mapping, *links*, which provides a feasible connection to the smart thing. Thus, the *links* function associates to the profile P (representing the smartphone which is going to download the script) and the smart thing d, the feasibility of establishing a connection between them. This may depend on different factors, like the availability to download, the closeness between both artifacts, a stable signal strength, etc. Indeed, this mapping guarantees the existence of a tuple representing a link (e.g., a URI or a Bluetooth connection) to provide a way to access the artifact d. If there are no links, or the profile P does not accept downloading the script offered by d, the returning result will be undefined ($\perp$).

Given a profile P and an artifact d, for defining $links(P, d)$ we will consider information coming from two different sources. On the one hand, auxiliary tuples locally stored in the profile P, representing connections previously established between the artifact d and the device having the profile P. These connections, if any, will be stored in P by means of tuples `<link,d,L>`, where L represents the URI or Bluetooth connection to access the services offered by d.

On the other hand, the mapping $links(P, d)$ also takes into account information provided by the artifact d. We assume that this information is stored in a header section of the script associated to d (i.e. $S_d(\bar{x})$) as metadata to be used during its execution. The header may contain any relevant information about the script, such as authoring, versioning, warranty statements, request permissions, etc. In particular, for formalization purposes, we consider that at least it declares the number of times or the rate at which the script can be executed: only once, a given number of times, every fifteen minutes, etc. This avoids repeatedly running a script every time the corresponding artifact is detected, and in particular prevents simultaneous runs of the same script in parallel.

We will denote by $header(d)$ this metadata information. In order to simplify the definition of the mapping *links* we will only consider three possible rate values in $header(d)$: *once*, *ever*, or $wait(n)$, where n is a natural number representing milliseconds. If needed, this information will be used to prevent further connections to the artifact d: *once* constrains the connection to occur only once, *ever* allows successive connections to the same device without any restriction (i.e. every time a connection to it is feasible), and $wait(n)$ forces to wait for n time units before connecting again to d.

To support the meaning intended for $wait(n)$ we assume a global clock providing the current time of the system, that will be denoted by $clock.ct()$.

Under all these premises, *links* is defined as follows:

$$links : \mathcal{P} \times D \longrightarrow T \cup \{\perp\}$$

where:

$$links(P,d) = \begin{cases} \perp & \text{if} \quad \nexists l.\ <\!link,d,l\!> \in P \lor <\!once,d\!> \in P \lor \\ & \qquad <\!wait,t,n,d\!> \in P \ with\ t+n > clock.ct() \\ <\!once,d\!> & \text{if} \quad \exists l.\ <\!link,d,l\!> \in P \land <\!once,d\!> \notin P \land \\ & \qquad <\!wait,t,n,d\!> \notin P \text{ for } t+n > clock.ct() \land \\ & \qquad once \in header(d) \\ <\!wait,t,n,d\!> & \text{if} \quad \exists l.\ <\!link,d,l\!> \in P \land <\!once,d\!> \notin P \land \\ & \qquad <\!wait,t',n',d\!> \notin P \ for\ t`+n' > t \land \\ & \qquad t = clock.ct() \land \\ & \qquad wait(n) \in header(d) \lor \\ & \qquad ever \in header(d) \implies n = 0 \end{cases} \tag{3}$$

Intuitively, when the profile P already contains a tuple `<once,d>`, or an *active* `wait` tuple on d is present, then no link is established for connecting both artifacts. We consider that a tuple `<wait,t,n,d>` $\in P$ is still active when $t+n$ is greater than the current time. When a `wait` tuple expires, we assume that it is automatically removed from the profile. Alternatively, when the profile P does not include neither a `once` tuple nor an active `wait` tuple on d, then a new tuple is returned as a result. The two possible results are the tuples `<once,d>` or `<wait,t,n,d>`, depending on $header(d)$, where t is the current time, and n represents the time units to wait ($n = 0$ when $ever \in header(d)$). In the last two cases, a link should have been previously established to connect d, that is, `<link,d,l>` $\in P$.

The benefits of defining a rate for precising how often a script must be run can be better understood by means of an example:

Example: a smart air conditioning system.
Let us suppose an air conditioning system that automatically adjusts to the comfort temperature of the people present in a room at any given time. If we implement such a system by means of Digital Avatars, the air conditioning artifact would provide a script (shown in Code 1) for reading the comfort temperatures stored in the virtual profiles of each person in the room. Their smartphones would be constantly downloading and running this script, and continuously notifying the air conditioning about their desired temperature. However, this continuous reminders of presence are not necessary, and they would also consume both a lot of battery and computing resources. Consequently, the script of the air conditioning artifact includes information to

run the script just once every few minutes. This execution rate is stated in the `<header>` section of the script (see Code 1, line 3). Thus, the *links* mapping will return a `wait` tuple (in this case `<wait,t,120000,d>`, `t` being the time when the script has been downloaded, and `d` the identifier of the air conditioning artifact). As we will show when remote synchronization is described (see Table 2), this tuple will be stored in the virtual profile of the smartphones, preventing a new run of the script until two minutes have passed. ■

We assume that all the notions introduced in this subsection (*accept*, *certify*, and *links*) are conveniently implemented, and we will make use of them to define the formal framework in the next section.

3. Formal framework

Now that we have defined the main elements and concepts of our framework, we formalize the interactions between artifacts by means of a transition system with in-device and remote operations. Then, we will show how some interesting properties like bisimilarity and congruence are accomplished by the model.

3.1. In-device transition system

The operational semantics of $\mathcal{L}$ is modeled by the following labelled transition system:

$$\dot{\longrightarrow} \subseteq \mathcal{D} \times \Lambda \times \mathcal{D}$$

defined by the rules [1] of Table 1, where $\mathcal{D} = \mathcal{P} \times \mathcal{L} \times D$ and $\Lambda = \{t, \bar{t}, \underline{t} : t \in T\} \cup \{\tau\}$.

Rule $\textsc{Out}_1$ describes how the output operation proceeds as an internal move (represented by label τ) which adds the tuple t to the profile P (comma is used to represent the multiset union). Rule $\textsc{Out}_2$ shows that a tuple t is ready to offer itself to the artifact/device by performing an action labelled $\bar{t}$. Rules $\textsc{In}$ and $\textsc{Read}$ describe the behavior of the prefixes $in(t)$ and $rd(t)$ whose labels are $\underline{t}$ and t. The difference on the effect of both actions is made explicit in rules $\textsc{Sync}_1$ and $\textsc{Sync}_2$.

In fact, rule $\textsc{Sync}_1$ is the standard rule for the synchronization between the complementary actions t and $\bar{t}$. It models the effective execution of an $rd(t)$

[1]For the sake of simplicity we will consider only finite processes here.

(Out$_1$)	$\dfrac{accept(out(d,t),P)}{\langle P : out(d,t).S\rangle_d \xrightarrow{\tau} \langle P, t : S\rangle_d}$
(Out$_2$)	$\dfrac{}{\langle P, t : S\rangle_d \xrightarrow{\bar{t}} \langle P : S\rangle_d}$
(Read)	$\dfrac{accept(rd(t),P)}{\langle P : rd(t).S\rangle_d \xrightarrow{t} \langle P : S\rangle_d}$
(In)	$\dfrac{accept(in(t),P)}{\langle P : in(t).S\rangle_d \xrightarrow{t} \langle P : S\rangle_d}$
(NRead$_1$)	$\dfrac{accept(nrd(t),P)}{\langle P : nrd(t).S\rangle_d \xrightarrow{\neg t} \langle P : S\rangle_d}$
(Sum)	$\dfrac{\langle P : S_1\rangle_d \xrightarrow{\alpha} \langle P' : S_1'\rangle_d}{\langle P : S_1 + S_2\rangle_d \xrightarrow{\alpha} \langle P' : S_1'\rangle_d}$
(Sync$_1$)	$\dfrac{\langle P : S_1\rangle_d \xrightarrow{t} \langle P : S_1'\rangle_d \quad \langle P : S_2\rangle_d \xrightarrow{\bar{t}} \langle P' : S_2\rangle_d}{\langle P : S_1 \parallel S_2\rangle_d \xrightarrow{\tau} \langle P : S_1' \parallel S_2\rangle_d}$
(Sync$_2$)	$\dfrac{\langle P : S_1\rangle_d \xrightarrow{t} \langle P : S_1'\rangle_d \quad \langle P : S_2\rangle_d \xrightarrow{\bar{t}} \langle P' : S_2\rangle_d}{\langle P : S_1 \parallel S_2\rangle_d \xrightarrow{\tau} \langle P' : S_1' \parallel S_2\rangle_d}$
(NRead$_2$)	$\dfrac{\langle P : S_1\rangle_d \xrightarrow{\neg t} \langle P : S_1'\rangle_d \quad \langle P : S_1\rangle_d \not\xrightarrow{\bar{t}}}{\langle P : S_1\rangle_d \xrightarrow{\neg t} \langle P : S_1'\rangle_d}$
(Par$_1$)	$\dfrac{\langle P : S_1\rangle_d \xrightarrow{\alpha} \langle P' : S_1'\rangle_d}{\langle P : S_1 \parallel S_2\rangle_d \xrightarrow{\alpha} \langle P' : S_1' \parallel S_2\rangle_d}$

Table 1: Transition system for smart devices.

operation. Notice that the resulting profile is left unchanged, since the read operation $rd(t)$ does not modify it. Rule $\textsc{Sync}_2$ defines the synchronization between two processes performing transitions labelled with $\underline{t}$ and $\overline{t}$, respectively. It models the effective execution of $in(t)$ actions.

Rule $\text{N}\textsc{Read}_1$ describes the prefix action $nrd(t)$, and the transition is labelled with $\neg t$. The effect of this rule is modelled by rule $\text{N}\textsc{Read}_2$ which proceeds when no transition $\overline{t}$ progresses from the current profile, i.e. the tuple t is not in the profile P.

All these rules modelling local actions require that the virtual profile P involved accepts them.

It is worth noting that we do not include in the transition rules any kind of evaluation nor variable instantiation when reading or adding tuples. In this way, tuples are supposed to be fully instantiated (that is, ground terms). However, the examples will assume the usual instantiation mechanisms on free variables, and also the evaluation of expressions when tuples are managed. These mechanisms could be easily represented in the transition rules, but for the sake of simplicity we decided to omit them, what does not mean any loss of generality.

Rule $\textsc{Sum}$ is the standard rule for choice composition. Similarly, the customary rule $\textsc{Par}_1$ for the parallel operator can be applied to any label. The transition system is considered closed w.r.t. commutative and associative properties for sum ($+$) and parallel ($\|$) operators.

In order to illustrate how the local label synchronization works, let us consider again the smart air conditioning system presented in Section 2.2. The script of the air conditioning (AC) artifact is shown in Code 1, while a process AC with the actions being performed over the tuple space of the smartphones running the script (corresponding to lines 6–11 in Code 1) is shown in Code 2. Let us focus on the latter, where we find an rd action for reading the comfort temperature of the user from her virtual profile (line 2), followed by an in action (line 3) for removing the old value of the the room's temperature. We assume that this tuple was previously stored in the profile in a previous execution of the script. Next, the process updates the profile with the most recent value of the room temperature (line 4), and informs (line 5) the air conditioning system of the comfort temperature desired by the user (the behaviour of this last action for remote addition of tuples will be defined in Table 2).

Suppose now that the virtual profile, P, stored in the smartphone of one of the people present in the room contains tuples for representing the value of her comfort temperature (24°C), and the temperature of the room (27°C). That is, we assume that P includes the following tuples:

$$\begin{aligned} t_1 &= <\texttt{personal.comfortTemp}, 24> \\ t_2 &= <\texttt{room.currentTemp}, 27> \end{aligned}$$

and let us also assume that the required access rights for these tuples have been granted as explained in Section 2.2. Finally, suppose that the current temperature of the room has dropped to 26°C, a value that instantiates the `CurrentTemp` parameter of the `AC` process in Code 2 (line 1).

Then, the execution of the rd action in the instantiated script $AC(26)$ of Code 2 is a consequence of the application of the $\text{S}{\scriptsize\text{YNC}}_1$ transition rule:

$$\frac{\langle P : AC(26)\rangle \xrightarrow{t_1} \langle P : AC'\rangle \qquad \langle P : AC(26)\rangle \xrightarrow{\overline{t_1}} \langle P' : AC(26)\rangle}{\langle P : AC(26)\rangle \xrightarrow{\tau} \langle P : AC'\rangle}$$

Notice that the profile does not experiment any change (the tuple t_1 is still present in the profile, P), and the resulting process AC' will be:

$$AC' = in(<room.currentTemp, RoomTemp>).\ AC''$$

where

$$AC'' = out(<room.currentTemp, 26>).AC'''$$

In a similar way, AC' will progress to AC'' by applying now the $\text{S}{\scriptsize\text{YNC}}_2$ rule as follows:

$$\frac{\langle P : AC'\rangle \xrightarrow{t_2} \langle P : AC''\rangle \qquad \langle P : AC'\rangle \xrightarrow{\overline{t_2}} \langle P' : AC'\rangle}{\langle P : AC'\rangle \xrightarrow{\tau} \langle P' : AC''\rangle}$$

where in P' the tuple $t_2 = <\texttt{room.currentTemp}, 27>$ has been removed from the profile as a result of the execution of the in action.

If we consider now the next action to be executed (the first one in AC''), $out(<room.currentTemp, 26>)$, the application of rule $\text{O}{\scriptsize\text{UT}}_1$ will make the system to progress as follows:

$$\langle P' : AC''\rangle \xrightarrow{\tau} \langle P'' : AC'''\rangle$$

where P'' adds to P' the tuple `<room.currentTemp, 26>`, which indicates the current temperature of the room, as notified by the air conditioning artifact.

3.2. Bisimilarity and congruence

The scripts downloaded from an artifact may need to evolve under certain circumstances. For instance, a software upgrade, or the development of a new version of the script. In this kind of situations, a notion of script equivalence would be very relevant to reason about compatibility among different versions. To formalize this, we consider the usual notion of bisimilarity-based equivalence, taking into account the device in which the script has to be run.

Definition 3. Given a virtual profile P, two scripts S and T in $\mathcal{L}$ are locally bisimilar with respect to P, written $S \sim_P T$ if and only if for each $\alpha \in \Lambda$ and $d \in D$:

1. if $\langle P : S\rangle_d \xrightarrow{\alpha} \langle P' : S'\rangle_d$ then $\langle P : T\rangle_d \xrightarrow{\alpha} \langle P' : T'\rangle_d$ for some T' such that $S' \sim_{P'} T'$

2. if $\langle P : T\rangle_d \xrightarrow{\alpha} \langle P' : T'\rangle_d$ then $\langle P : S\rangle_d \xrightarrow{\alpha} \langle P' : S'\rangle_d$ for some S' such that $S' \sim_{P'} T'$

Local bisimilarity relation is, as usual, an equivalence relation as the following lemma states.

Lemma 1. *The bisimilarity relation $\sim_P$ is an equivalence relation.*

Proof. It is directly derived by reasoning on different rules in Table 2. □

In fact, the transition relation $\xrightarrow{\cdot}$ (restricted to devices) defines a notion of bisimilarity which permits to decide about script equivalence. In addition, it would be very useful that this bisimilarity relationship is a congruence with respect to the connectors $+$ and $\|$.

Theorem 1. *The bisimilarity relation $\sim_P$ is a congruence with respect to non-deterministic choice and parallel operators.*

Proof. Let P be a virtual profile, and let us assume $S_1 \sim_P S_2$. We will prove $S_1 \parallel T \sim_P S_2 \parallel T$ by structural induction. To do it, we will only analyze the first condition in Definition 3, since the second one is symmetric. That is,

$$\langle P : S_1 \parallel T\rangle_d \xrightarrow{\alpha} \langle P' : S'\rangle_d \tag{4}$$

First, we proceed by proving the proposition on the inductive base, processes 0 and $\alpha.0$ ($\alpha \in Act$). For these processes, the result is easily proved, by considering each case in rules (Out$_1$), (Out$_2$), (Read), (In) and (NRead$_1$).

In a general case, we have the following alternatives (depending on the rule in Table 1 triggering the transition):

1. If (Par$_1$) was the rule applied to get (4), then we have two possibilities: either
$$\langle P : S_1 \rangle_d \xrightarrow{\alpha} \langle P' : S_1' \rangle_d \quad (S' = S_1' \parallel T)$$
or
$$\langle P : T \rangle_d \xrightarrow{\alpha} \langle P' : T' \rangle_d \quad (S' = S_1 \parallel T')$$
In the first case, as $S_1 \sim_P S_2$, we have $\langle P : S_2 \rangle_d \xrightarrow{\alpha} \langle P' : S_2' \rangle_d$, with $S_1' \sim_{P'} S_2'$. Therefore, in both cases, by applying rule (Par$_1$):
$$\langle P : S_2 \parallel T \rangle_d \xrightarrow{\alpha} \langle P' : S'' \rangle_d$$
S'' being $S_2' \parallel T$ or $S_1 \parallel T'$, respectively. Then, by applying inductive hypothesis on the first case $S' = S_1' \parallel T \sim_{P'} S_2' \parallel T = S''$, and $S'' = S'$ (hence $S'' \sim_P S'$ by Lemma 1 in the second one.

2. If the applied rule to get (4) is (Sync$_1$), then $\alpha = \tau$, $P' = P$, and either S_1 or T is a parallel composition of processes T_1 and T_2 such that T_1 implies a transition labelled by t. If $T = T_1 \parallel T_2$, we would have in the previous alternative (1). So, let's suppose $S_1 = T_1 \parallel T_2$, and
$$\langle P : T_1 \rangle_d \xrightarrow{t} \langle P : T_1' \rangle_d \quad \langle P : T_2 \parallel T \rangle_d \xrightarrow{\bar{t}} \langle P' : T_2 \parallel T \rangle_d$$
with $S' = T_1' \parallel T_2 \parallel T$. By applying rule (Par$_1$) to the left transition above, we have $\langle P : S_1 \rangle_d \xrightarrow{t} \langle P : T_1' \parallel T_2 \rangle_d$. As $S_1 \sim_P S_2$, we have
$$\langle P : S_2 \rangle_d \xrightarrow{t} \langle P : S_2' \rangle_d$$
for some S_2' with $S_2' \sim_P T_1' \parallel T_2$. Taking into account that transition $\bar{t}$ only affects to the profile P, we also have $\langle P : T \rangle_d \xrightarrow{\bar{t}} \langle P' : T \rangle_d$. Therefore, rule (Sync$_1$) applied to $S_2 \parallel T$ gets
$$\langle P : S_2 \parallel T \rangle_d \xrightarrow{\tau} \langle P : S_2' \parallel T \rangle_d$$
At this point $S' = T_1' \parallel T_2 \parallel T$ and $S_2' \sim_P T_1' \parallel T_2$, which implies (again by inductive hypothesis) $S' \sim_P' S_2' \parallel T$.

(Remote)	$\dfrac{accept(out(e,t),Q)}{\langle P : out(e,t).S\rangle_d \mid \langle Q : T\rangle_e \xrightarrow{\tau} \langle P : S\rangle_d \mid \langle Q, t : T\rangle_e}$
(Sync$_3$)	$\dfrac{certify(e,P) \wedge links(P,e) = b \neq \bot}{\langle P : S\rangle_d \mid \langle Q : T\rangle_e \xrightarrow{\tau} \langle P, b : S \parallel S_e(Q)\rangle_d \mid \langle Q : T\rangle_e}$
(Par$_2$)	$\dfrac{D_1 \xrightarrow{\alpha} D_1'}{D_1 \mid D_2 \xrightarrow{\alpha} D_1' \mid D_2}$

Table 2: Transition system.

3. The other two alternatives to get transition (4) is applying rules (Sync$_2$) or (NRead$_2$). In both cases, the reasoning is similar to the previous one.

In a similar way, we could prove $S_1 + T \sim_P S_2 + T$ when $S_1 \sim_P S_2$. □

3.3. Remote transition system

In order to define how artifacts interact, we consider configurations composed of a parallel composition of artifacts as follows:

$$\langle P_1 : S_1\rangle_{d_1} \mid \langle P_2 : S_2\rangle_{d_2} \mid \cdots \mid \langle P_n : S_n\rangle_{d_n}$$

where P_i $(i = 1..n)$ are virtual profiles of artifacts —either smart devices or smart things—, S_i are scripts running in smart devices, and d_i represent the device identifiers. Notice that we denote in a different way the parallel composition of artifacts ($|$) and the parallel composition of processes inside a smart device ($\|$).

The transition system $\xrightarrow{\cdot}$ defined in Table 1 is extended to configurations by the inference rules given in Table 2.

Rule Remote models remote actions modifying the virtual profile which belongs to the smart thing from which the script being run was downloaded. We consider this transition as a silent step from an observational point of view. For this reason, we use the label τ. It is worth noting that although the syntax of primitive $out(d,t)$ may suggest that we allow adding tuples on any arbitrary artifact, our intention is to constrain this capability to add tuples either locally, or remotely only to the remote device whose script generated the out action, as it will be shown in Section 4. This limitation is consistent with the goal of guaranteeing a controlled access to virtual profiles.

Rule $\text{S}{\scriptstyle\text{YNC}}_3$ represents the interaction between two artifacts (typically, a smart device and a smart thing). In this case, the script associated with a smart thing e, previously certified, is downloaded through a link establishing a connection between the virtual profile P and e. The availability of this link is guaranteed because $\mathit{links}(P, e) \neq \perp$ (see equation 3 where *links* mapping was defined). Thus, the script to be executed in the context of the smart device d (in parallel with other possible pending processes in d) will be $S_e(Q)$, as it was defined in Definition 2. Notice that, in this case, the script is instantiated by the profile Q. This allows customizing the script to the artifact which provides access to it. In addition, the link tuple b is added to the profile P, recording this way that the smart thing has been already "visited".

In fact, this tuple b encodes information to prevent unlimited script downloads. Remember that the mapping *links* is defined in such a way that $\mathit{links}(P, e) = \perp$, when `<once,e>` $\in P$ or when `<wait,t,n,e>` $\in P$ and $t + n$ is greater than the current time. Thus, a tuple `once` would be added to the profile P the first time a connection is established, but no more connections will take place in the future. In a similar way, if the tuple `<wait,ts,n,e>` was added with a time stamp ts, and n being the time to wait before enabling running the script again, it will not be downloaded until n time units have passed. Which of those two tuples is added to the profile will depend on the metadata exhibited by the artifact e, as it was defined in equation 3.

Rule $\text{P}{\scriptstyle\text{AR}}_2$ describes the way in which the parallel composition of artifacts proceeds. Note that the parallel composition of processes inside a smart device is modelled by Rule $\text{P}{\scriptstyle\text{AR}}_1$ in Table 1. Actually, any interaction in the context of a smart device is governed by rules in that table.

We consider the transition system closed w.r.t. usual structural congruence (commutative and associative properties) of both parallel connectors.

The rules in Table 1 and Table 2 are used to define the set of derivations in an environment where smart devices and smart things are interacting with each other. Following [7], both reductions labelled τ and reductions labelled $\neg t$ are considered. Formally, this corresponds to introducing the following derivation relation:

$$D \longmapsto D' \quad \text{iff} \quad (D \xrightarrow{\tau} D' \text{ or } D \xrightarrow{\neg t} D').$$

```
<digitalavatars>
<header>
    <wait>120000</wait>
</header>
<script>
void AC(double currentTemp) {
    double comfortTemp = dac.read("Personal/comfortTemp");
    dac.remove("Room/currentTemp");
    dac.write("Room/currentTemp",currentTemp);
    dac.remoteWrite("ComfortTemperature",comfortTemp);
    }
</script>
</digitalavatars>
```

Code 1: Script for the smart air conditioning system.

```
AC(CurrentTemp) =
    rd(<personal.comfortTemp,ComfortTemp>).
    in(<room.currentTemp,RoomTemp>).
    out(<room.currentTemp,CurrentTemp>).
    rout(<desiredTemp,ComfortTemp>). 0
```

Code 2: Tuple actions for the air conditioning (AC) script.

4. Case study: a Treasure Hunt

Now that we have formally defined a framework for reasoning on Digital Avatars, we present a motivating example for showing how it works, and we discuss how the formalization provides useful tools for checking properties and inferring results of the systems built according to our proposal. The case study consists in a treasure hunt game, in which several players look for a set of five hidden treasures following clues. Each treasure found provides a clue for a new treasure, and the player that first finds all the treasures wins the game.

In the remainder of this section, we first present an implementation of the case study (Section 4.1) in which treasures are represented by beacons, scattered over the scenario of the hunt. We have developed a generic smartphone application for detecting beacons, downloading the scripts associated to them, and running these scripts in the context of the virtual profile stored in the smartphone. After that, Section 4.2 presents a formalization of the treasure hunt by means of the primitive actions defined in Section 2. Finally, in Section 4.3 we apply our formal framework to infer several properties of interest of the system.

4.1. Implementation of the hunt

In order to implement our proposal we have proceeded in two steps. First, we have developed an implementation of the Digital Avatars framework for smartphones as one single and generic Android application which detects smart artifacts —in particular, Bluetooth Low Energy (BLE) beacons— in the surroundings, downloads the scripts associated to them, and executes these scripts on the user's terminal, for both interacting locally with the virtual profile in the phone, and remotely with that of the artifact from which a given script was downloaded. Then, we have written a script for specifying the behavior of the treasure hunt itself, and we have associated this script to a number of beacons that represent the treasures to be found.

The architecture of the system is shown in Fig. 1, which represents the core components of the framework. In particular, the Digital Avatar Controller (DAC) API provides a single point of access to the virtual profile of the smartphone's user. The Privacy Settings module allows the user to control which information is offered to external devices and third parties, and which scripts are trusted to be run on the smartphone.

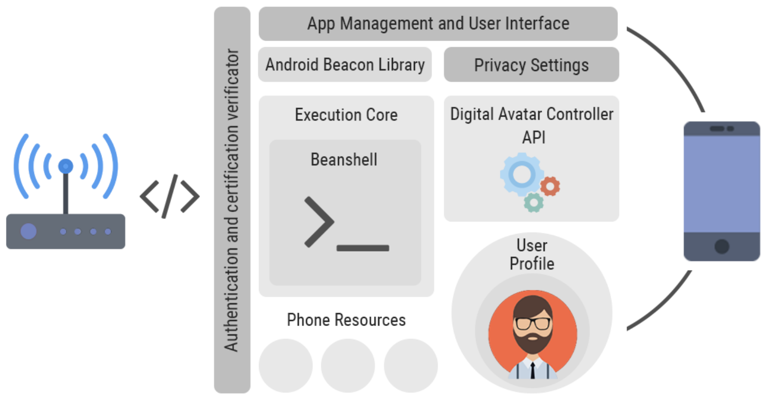
App Management and User Interface
Android Beacon Library
Privacy Settings
Execution Core
Beanshell
Digital Avatar Controller API
User Profile
Phone Resources
Authentication and certification verificator
</>

Figure 1: Architecture of the Digital Avatars framework (updated from [9]).

Scripts are encoded using operations of the DAC API, and they are always executed in a controlled way, first checking the certificate of the script provider, then being only able to access the virtual profile through the API. The execution of the scripts is handled with Beanshell (http://www.beanshell.org), a simple Java interpreter capable of uploading an executing code at runtime. Beanshell scripts do not need to be entire classes, just pieces of Java code. The DAC API offers a set of operations to the Beanshell execution core in order to perform simple actions of querying and updating the virtual profile, as well as accessing the Android operating system, for instance for displaying notifications and messages.

Apart from interacting with the virtual profile in the smartphone running it, a script may also remotely change the value of some entity in the script provider's virtual profile, as discussed in Section 2.2. In turn, these updated values may be used during future downloads for instantiating some of the parameters of the script. Thus, the framework provides a mechanism for changing the behavior of the smartphones running the script in these future interactions. This is how smart things are able to automatically adapt their behavior in a seamless way in order to suit the needs of their users.

Once we have presented the Digital Avatars smartphone app, for the implementation of the treasure hunt itself we need the following elements: (i) the Android application installed and running in the smartphones of the players, (ii) a cloud server for hosting the script that specifies the behaviour of the treasure hunt and equipped with its own digital avatar for holding the global state of the game, and (iii) a set of Eddystone BLE beacons, scattered over the scenario of the hunt, for representing the treasures. The beacons emit a BLE signal with a shortened URL where the script is hosted in the cloud server. These elements are represented in Fig. 2. The finding of a treasure is triggered by detecting the corresponding beacon. The URL contains a parameter which identifies the beacon issuing it, so we can pinpoint exactly which treasure has been found, and one single script works for all the treasures.

When a player finds one of the treasures (i.e. when the player gets close enough to a beacon), the Digital Avatars app in her smartphone detects it, accesses the corresponding URL encoded in the BLE signal, and downloads from the cloud server the Treasure Hunt script, which is shown in Code 3. Each time the script is downloaded, its parameters *status* and *clues* (line 6) are instantiated to their values in the virtual profile of the cloud server.

According to the definition of *links* in Section 2.2, the contents of the

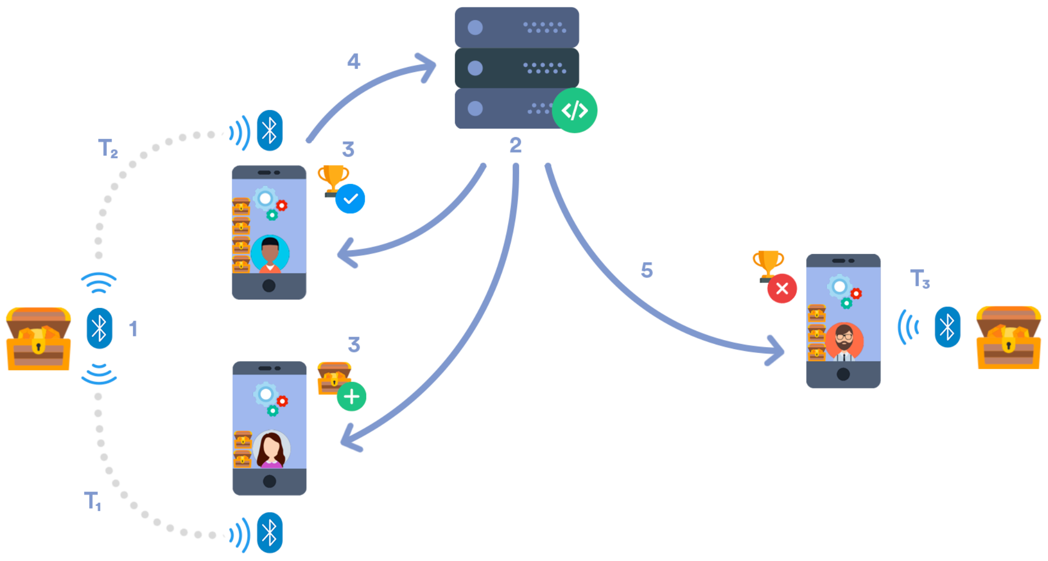

Figure 2: An example scenario of the Treasure Hunt.

`<header>` section in Code 3 (lines 2–4) are checked prior to the execution of the script. In this case, it contains a `wait` tag with a rate of one minute (in milliseconds). This is the time the smartphone has to wait between two successive executions of the script —giving the player the opportunity to get out of the range of the beacon already found, in her quest for a new treasure. Then, the presence of a previous `once`, or active `wait` is checked. If none of them are in the profile, the link is solved, adding the corresponding `wait` tuple to the virtual profile of the player. In any other case, *links* returns a void value, and the script is not executed.

```
1 <digitalavatars>
2 <header>
3       <wait>60000</wait>
4 </header>
5 <script>
6 void treasureHunt(String status, List<String> clues) {
7    if (dac.read("clues") == null) {
8       // The player has just started the hunt;
9       // entities are written to the profile
10      int i = 0;
11      for (String clue : clues) {
12        dac.write("clues/" + i++, clue);
13      }
14      dac.write("treasures", 0);
15    }
16    String c = dac.read("clues");
17    dac.remove("clues/", c);
18    if (status == "playing") {
19        // Checks no. of treasures already found
20      int treasures = dac.read("treasures");
21      if (treasures != 4) {
22        dac.write("treasures", treasures++);
23        dac.showToast("The new clue is " + c);
24      }else { // The player has won the game
25        dac.showToast("Congratulations. You Win!");
26        dac.write("treasures", 5);
27        String me = dac.read("Personal/name");
28        String now = dac.read("System/now");
29        dac.remoteWrite("winner/name", me);
30        dac.remoteWrite("winner/time", now);
31        dac.remoteWrite("gameover");
32      }
33    // Somebody else has won the game
34    }else { dac.showToast("You lose!"); }
35 }
36 </script>
37 </digitalavatars>
```

Code 3: Java Beanshell script for the Treasure Hunt.

In the script, several DAC API operations are invoked. While *showToast* shows a notification through the smartphone of the player, operations *read*, *write*, and *remove* access the information stored in its virtual profile as key/value pairs. Finally, *remoteWrite* updates the profile of the cloud server that stores the global status of the game, from which the script has been downloaded.

The script begins by initializing the game if this is the first time the player downloads it. For that, it writes to the profile the clues for finding the treasures, and it sets the number of treasures found by the player to zero (lines 7–15). Then, it randomly takes from the profile one of the clues (lines 16-17). If the status of the game, downloaded as a script parameter from the cloud server, is "playing" (line 18), it checks the number of treasures already found by the player (line 20). If they are less than four (i.e. the player still needs to find some of the five treasures), the script increments by one the number of treasures found, and shows the new clue (lines 21–23). Otherwise, if the player has already found four treasures, the treasure currently found is the only one missing, so the script declares the player as winner of the hunt (line 25) and it finalizes the game, remotely updating the status of the game in the server's virtual profile by adding a *gameover* tuple, together with the name of the winner and the current time (lines 29–31). Alternatively to lines 18–32, if the game status is *gameover* —i.e. if someone else has already found all the treasures—, the script just informs the player that he has lost (line 34).

An example of the global flow of the game is shown in Fig. 2. A treasure, represented by the beacon in the left of the figure, broadcasts a BLE signal (1) with the URL of the script. When any of the users in the left approaches the treasure and detects this signal, their smartphone's Digital Avatars app uses this URL for connecting to the server in the top, downloading the script for running it on their terminal (2). For the player in the bottom, arriving at time T_1, the script updates the virtual profile in her phone with the treasure found and shows a new clue (3). Later on, the same script downloaded at T_2 by another player, makes him win the game as he has already found all the treasures (3). In that case, the script also executes operations for updating the state of the game in the server's virtual profile (4), which changes the behaviour of the script in future detections of this or any other beacon, as the script will be instantiated with the new status. Thus, the player in the right, downloading the script at time T_3 is informed that the game is over (5).

In our case study, the server represents a smart artifact with computing and storage capabilities, endowed with a digital avatar holding the status of the game, and a script template which will be instantiated with this status each time the script is requested for downloading. This way we keep all the players synchronised, so they know whether the game is still running or it has been already won by another gamer. We have implemented the behavior of the server in Node.js with Express while its virtual profile is represented by means of a small MongoDB database which contains information about the existing treasures and the available hints. A preliminary version of this case study was presented in [10] where its technical aspects are described in more detail. The full implementation of both the server and the Android applications as well as the script are available in Github[2].

[2]https://github.com/apvereda/TreasureHuntBeacons

In our Treasure Hunt scenario we have employed Bluetooth beacons as they are simple and non-expensive devices. However, they lack computing capabilities and are only able to store and broadcast a short URL string. Thus, they cannot be properly considered as smart devices. For this reason, we needed a back-end server artifact for hosting the global status of the game. In a more general scenario, the Bluetooth smart device itself would be in charge of holding and serving the script, not only a URL to it, and also to offer a more sophisticated behaviour. Consider for instance the smart air conditioning system previously presented, which adjusts the temperature of a room to the preferences of the people staying in it, or a home music system that selects the songs to be played based on the preferences or even the mood of the listeners. In both cases these preferences would be kept in the virtual profiles of the users, from which they would be accessed by means of the appropriate scripts.

4.2. Formalizing the game

For formalizing the game, we only need to define one script template *TreasureHunt* for all the beacons. This script is stored in a cloud server which also hosts a virtual profile containing tuples with the clues for the treasures, and one extra tuple for indicating when the game is over, as it will be explained below.

```
1  rd(<clue1,_Clue1>). rd(<clue2,_Clue2>).
2  rd(<clue3,_Clue3>). rd(<clue4,_Clue4>).
3  rd(<clue5,_Clue5>). (
4      rd(<gameover>).
5      TreasureHunt(<gameover>,_Clue1,_Clue2,_Clue3,_Clue4,_Clue5)
6      +
7      nrd(<gameover>).
8      TreasureHunt(<playing>,_Clue1,_Clue2,_Clue3,_Clue4,_Clue5))
```
Code 4: Script instantiation for the Treasure Hunt.

Prior to each download, the parameters of the script must be instantiated. The instantiation process for the TreasureHunt script (shown in Code 4) sets the clues for finding the five treasures (binding `_Clue1`, . . . , `_Clue5`). They are fetched from the server's virtual profile by the read actions in lines 1–3. Then, the current status of the game depends on whether there exists a `<gameover>` tuple (lines 4–5) or not (lines 7–8).

The execution of the script interacts with the virtual profile in the smartphone, checking and updating which treasures have been already found by the player, and showing the clue for a new treasure. It also informs the game when a player has found all the treasures. The rest of the players will be notified the next time they find a beacon, downloading again the script. The full script of the Treasure Hunt is shown in Code 5. There is not a strict correspondence with the Java encoding in Code 3, although we have tried to keep both versions of the script aligned, for a better understanding.

```
TreasureHunt(_Status,_Clue1,_Clue2,_Clue3,_Clue4,_Clue5) =
    nrd(<clue,C>).
        out(<clue,_Clue1>). out(<clue,_Clue2>).
        out(<clue,_Clue3>). out(<clue,_Clue4>).
        out(<clue,_Clue5>). out(<treasures,0>). 0
    +
    in(<clue,C>). out(<_Status>).(
        in(<playing>).(
            nrd(<treasures,4>).
                in(<treasures,X>). out(<treasures,X+1>).
                out(<notify,C>). 0
            +
            GameOver)
        +
        in(<gameover>).
            out(<notify,"You lose!">). 0 )

GameOver =
    in(<treasures,4>). out(<treasures,5>).
        out(<notify,"Congratulations. You Win!">).
        rd(<personal.name,Me>).rd(<system.now,Now>).
        rout(<winner,Me,Now>). rout(<gameover>). 0
```

Code 5: Script for the Treasure Hunt.

Before explaining the behaviour of the code, we need to make some clarifications. First, concerning the use of the *out* primitive, we simplify its syntax, considering only one argument (instead of two, as defined in *Act* in Section 2, in such a way that the first argument is missing when the *out* action is applied locally, while *rout* is used to explicitly refer to a remote out. In other words, a script code including an action $out(t)$ has to be interpreted as $out(l, t)$, l being the identifier of the (local) artifact where the script is being executed; and $rout(t)$ denotes $out(r, t)$, where r is the (remote) artifact fro which the script has been downloaded. Second, we abuse of notation by considering variables and arithmetic expressions in tuples: variables in *rd* actions are conveniently instantiated as a consequence of the matching produced when reading a tuple, and arithmetic expressions are conveniently evaluated when occurring on *out* actions.

The branch starting from line 2 in the script is performed when a player begins the treasure hunt (we assume an additional beacon located in the starting place of the hunt), as no clues are present in his profile yet (line 2). In that case, all the five clues are added to the virtual profile of the player (lines 3–5). The last tuple added in line 6 is `<treasures,0>`, indicating that no treasures have been found yet.

Alternatively (line 7), a random clue is read (and consumed) from the profile, and the current status of the game (`<gameover>` or `<playing>`) is written in the player's profile. Then, we have again two alternatives. Either the game is over (lines 15–16) and the player is notified of this fact, or the hunt is still being played (lines 8–13). In the latter case, the tuple `<treasures,N>` stores the number of beacons that the player has already found. If they are less than four (i.e. `<treasures,4>` is not in the profile, line 9), then the script increases the number of treasures found (line 10), and it shows a new clue to the player (line 11).

On the contrary (line 13, linking to process `GameOver` in line 18), if the player had already found four treasures (line 19), she wins the game (please recall that the script is executed whenever the player finds a beacon, which makes it the fifth one). In this case, both the name of the player and the current time are got from the player's profile (line 21), and they are used to update the global state of the game. Indeed, two tuples are remotely added to the server from which the script has been downloaded: one with information on the winner, and the other one stating that the game is over (line 22). Again, it is worth noting that the remote out actions in line 22 only have one argument, instead of two arguments as defined in Section 2, because the first argument implicitly corresponds to the server providing the script. That is, although our formal model allows adding tuples to other arbitrary artifacts (through the *rout* primitive), we restrict this capability only to the artifact that generated the script including the *rout* primitive.

4.3. Reasoning on the case study

The formalization of Digital Avatars presented in this paper allows us to reason on the behavior of the treasure hunt game. One of the properties that we may want to analyze is whether it is ensured that eventually someone wins the game. Indeed, this property can be proved with the Linda-based semantics presented in Section 3, just making a couple of basic assumptions. First, let us consider an initial configuration composed of a non-empty set of smart devices (players) and at least one beacon pointing to a smart artifact (a cloud server) which contains the script. This configuration is represented by a parallel composition of all those elements:

$$C_0 = \Pi_{i=1}^{n} \langle P_i : 0 \rangle_{d_i} \mid \langle P : 0 \rangle_b \tag{5}$$

being d_i the smart devices of the players, each with a profile P_i, and b the beacon associated with a server with a profile P and script template S_b as specified in Code 5.

Second, let us assume that the cloud server is certified (i.e. $\mathit{certify}(b, P_i)$) in all the profiles P_i, which means that a tuple `<certified,b>` $\in P_i$, and that all the devices d_i can always get a link to the beacon b; that is, there exists a link l_i such that `<link,b,`l_i`>` will be eventually included in P_i. In addition, we will assume that the script includes the heading information `<wait>60000</wait>` for leaving one minute before running the script again. In other words, we will assume that for every instant of time t we will have that $\mathit{links}(P_i, b)$ will include the tuple $<\!\mathit{wait}, t, 60000, b\!>$ for some later time t and for every $i = 1..n$. This means that any player will eventually have access to a beacon connected to the server. These two assumptions are formalized as hypothesis of the next proposition, which ensures the eventual end of game.

Proposition 1. *Let us consider a smart thing b referring the script template $S_b = TreasureHunt$ as defined in Code 5, and the initial configuration C_0 specified in equation (5), such that $\mathit{accept}(a, P_i)$ for every action a in any profile P_i, and $\mathit{certify}(b, P_i)$. If for every sequence of transitions $C_0 \longmapsto^* C$, some of the smart devices d_k in C has a virtual profile Q_k such that $\mathit{links}(Q_k, b) = <\!\mathit{wait}, t, 60000, b\!>$ for some t such that $t + 60000$ be less than the current tick of the global clock, then there exists a trace:*

$$C_0 \longmapsto^* C' \mid \langle t, P' : 0 \rangle_b$$

with t=`<gameover>`.

Proof. Applying the assumption to the empty sequence of transitions $C_0 \longmapsto^* C_0$, we can find a device d_{k_1} such that rule $\textsc{Sync}_3$ can be eventually applied, as both conditions of that rule are fulfilled: $\mathit{links}(P_{k_1}, b) = <\!\mathit{wait}, t, 0, b\!> \neq \perp$, and *certify*$(b, P_{k_1})$. Therefore, running the script on that device, after applying several times the rules of Table 2, we obtain a trace:

$$C \longmapsto^* C_1$$

where a new configuration C_1 is achieved containing a device d_{k_1} whose profile includes the tuple `<treasures,1>`. If we apply again the hypothesis to C_1, we find a device d_{k_2} such that rule $\textsc{Sync}_3$ may be applied once more, and a new instance of the script S_b will make C_1 to progress to C_2 ($C_1 \longmapsto C_2$). If $d_{k_2} = d_{k_1}$, its profile will include the tuple `<treasures,2>`. If this is not the case, then we will have a new device with a profile also including `<treasures,1>`. Taking into account that we can always repeat this procedure (the *links* mapping will eventually allow the application of the $\textsc{Sync}_3$ rule), and that we have a finite number (n) of smart devices, after at most $4n$ iterations, some of the devices will exhibit a profile with the tuple `<treasures,4>`. Hence, the branch represented by lines 13–15 of S_b will be eventually triggered, adding the tuple `<gameover>` to b's profile. $\square$

The proposition above shows that, under some basic assumptions, a proper initial configuration will eventually progress to a gameover status.

Additionally, some unexpected scenarios can be detected if we analyze the script in Code 5 more in depth. An exhaustive exploration of all possible traces generated from a configuration C, like in equation (5), would provide interesting information about the soundness of the script. In fact, a model checker capable of exhaustively exploring all possible traces achievable from C would detect some target configurations such that $C \longmapsto C' \mid \langle P' : 0 \rangle_b$, where the profile P' includes two or more copies of a tuple `<winner,Me,Now>`. This means that two or more players could postulate themselves as winners of the treasure hunt. Indeed, it is easy to imagine how C can progress to a configuration D, such that:

$$D = \ldots \mid \langle P_d : S_d \rangle_d \mid \langle P_e : S_e \rangle_e \mid \ldots \mid \langle P : 0 \rangle_b \qquad (6)$$

where `<treasures,4>` is both in the profiles P_d and P_e, and `<gameover>` is not yet in P. In other words, two devices would have found four treasures each, and the game is still being played. In this situation, if we consider a scenario where both $links(P_d, b)$ and $links(P_e, b)$ are not undefined, then two consecutive transitions can occur:

$$\begin{array}{lll} D & \xrightarrow{\tau} & \ldots \mid \langle P_d, w_d : S_d \parallel S_b(P)\rangle_d \mid \langle P_e : S_e\rangle_e \mid \ldots \mid \langle P : 0\rangle_b \\ & \xrightarrow{\tau} & \ldots \mid \langle P_d, w_d : S_d \parallel S_b(P)\rangle_d \mid \langle P_e, w_e : S_e \parallel S_b(P)\rangle_e \mid \ldots \mid \langle P : 0\rangle_b \end{array}$$

Both transitions are a consequence of applying rule $\textsc{Sync}_3$, where $w_d = \mathit{links}(P_d, b)$ and $w_e = \mathit{links}(P_e, b)$. The scripts downloaded from b, $S_b(P)$ and $S_b(P)$ are conveniently instantiated with information on profile P. As at the time of script download, the tuple `<gameover>` had not yet been written to P by any of the devices, both instances of S_b will add a `<playing>` tuple to the local profiles of d and e (line 7 in Code 5). And then, because `<treasures,4>` is present in both profiles P_d and P_e, the actions in lines 19–22 of Code 5 are executed:

$$\begin{array}{lll} C & \longmapsto^* & D \\ & \longmapsto^* & \ldots \mid \langle P_d, w_d : S_d \parallel S_b(P)\rangle_d \mid \langle P_e, w_e : S_e \parallel S_b(P)\rangle_e \mid \ldots \mid \langle P : 0\rangle_b \\ & \longmapsto^* & \ldots \mid \langle P'_d : S'_d\rangle_d \mid \langle P'_e : S'_e\rangle_e \mid \ldots \mid \langle P' : 0\rangle_b \end{array}$$

where both P'_d and P'_e include the tuple `<treasures,5>`, both players will be notified as winners of the hunt, and the game will be stopped by remotely adding duplicated `<winner,M,T>` and `<gameover>` tuples to P', which of course is not what we would desire as the result of the treasure hunt.

This unwanted situation might be addressed in several ways. The first idea that may come up to us would be adding to the game a final podium stage: once the game is over, the cloud server announces the actual (photo finish) winner depending on the time of each of the two (or more) of the `<winner,M,T>` tuples. Although this solution would solve the problem, it looks somehow artificial and *ad hoc*, and it cannot be easily generalised to other scenarios were a similar situation may appear.

A different alternative for solving this and similar issues would be implementing a mechanism of locks or mutexes for ensuring the atomic execution of the scripts, blocking them for download until their execution in some other device ends. Again, our formal framework would allow us to analyze the behavior of the system for the uniqueness of a tuple, in order to detect problems and check whether we are able to solve them by means of locks. However, the use of mutexes would make scripts more complex, obfuscating their behavior, and they are error-prone when writing the scripts.

A more elegant approach to deal with situations like those mentioned above is extending the model with a new primitive, similar to the remote out action, but blocking when the tuple is already in the corresponding tuple space. We have called this primitive *nrout*, and the next section presents a extension to the formal model presented in Sections 2 and 3.

5. Extending the expressiveness of the framework

One of the problems illustrated in the Treasure Hunt example is the lack of expressiveness of the model to prevent adding a tuple to a virtual profile when it already includes it. This is because both the local and remote *out* primitives always success adding a tuple, without considering the status of the tuple space. In fact, only the *rd* and *nrd* primitives suspend when a given tuple does not exist or exists (respectively) in the shared tuple space. Therefore, we propose to extend the language $\mathcal{L}$ by considering a new action in *Act*.

Hence, we redefine the syntax of an extended version of the language $\mathcal{L}$' as follows:

$$\begin{array}{rcl} S \in \mathcal{L}' & ::= & 0 \mid \alpha.S \mid \mathit{nrout}(d,t).S > S \mid S+S \mid S \parallel S \mid S(\tilde{t}) \\ \alpha & \in & \mathit{Act} \end{array}$$

where $\mathit{nrout}(d,t)$ represents a version of *out*, blocking when the tuple t is already present in d, and the connective $>$ provides an alternative to proceed when tuple t is in the profile of d.

Although the newly added primitive *nrout* may also be used locally (i. e. affecting the profile of the device where the action is being executed), by now we will consider that it is only applied remotely. That is, we will define its behaviour by extending only Table 2, without modifying Table 1.

Thus, Table 3 presents two additional transition rules, NREMOTE$_1$ and NREMOTE$_2$, to define how $\mathit{nrout}(e,t).S_1 > S_2$ works. The first rule models how a tuple t is added to the profile Q of an artifact e when the tuple was not previously present in Q. Of course, adding the tuple has to be an acceptable action in the profile Q. Note that the intended semantics of $\mathit{nrout}(e,t)$ is similar to the sequence of two actions: first, an $\mathit{nrd}(t)$ action on e's profile, and a second action $\mathit{out}(e,t)$. The main difference is that $\mathit{nrout}(e,t)$ is atomic, and no interference can occur between checking whether the tuple is in the profile or not, and adding it.

Rule NREMOTE$_2$ models how the connective $>$ progresses when the tuple t is already contained in the profile of the artifact e. In such a situation, the process $\mathit{nrout}(e,t).S_1 > S_2$ proceeds as S_2.

$$(\text{NR}\textsc{emote}_1)\quad \frac{accept(out(e,t),Q)\ \wedge\ t \notin Q}{\langle P : nrout(e,t).S_1 > S_2\rangle_d \mid \langle Q : T\rangle_e \xrightarrow{\tau} \langle P : S_1\rangle_d \mid \langle Q, t : T\rangle_e}$$

$$(\text{NR}\textsc{emote}_2)\quad \frac{t \in Q}{\langle P : nrout(e,t).S_1 > S_2\rangle_d \mid \langle Q : T\rangle_e \xrightarrow{\tau} \langle P : S_2\rangle_d \mid \langle Q : T\rangle_e}$$

Table 3: Extended transition system for Digital Avatars.

We still denote by $\longmapsto$ the transition relation derived from the transition rules in Tables 1, 2, and 3.

With this new primitive, we can solve the problem presented in the preceding section, in which two players postulate as winners of the game. Indeed, the atomic character of *nrout* allows an alternative definition for the Treasure Hunt script, by substituting the process `GameOver` of Code 5 by the new version in Code 6.

```
1 GameOver =
2     in(<treasures,4>). out(<treasures,5>).
3     out(<notify,"Congratulations. You Win!">).
4         rd(<personal.name,Me>).rd(<system.now,Now>). (
5             nrout(<gameover>). rout(<winner,Me,Now>). 0
6             >
7             out(<notify,"You lose!">). 0 )
```

Code 6: A new version of GameOver for the Treasure Hunt.

As it is shown in Code 6, when the first player finds the last treasure, her smartphone executes the *nrout* primitive (line 5 in Code 6) to add the `<gameover>` tuple indicating the end of game. If another player downloads and executes the script at the same time, and he or she is in the same situation (i.e. with all treasures already found), the execution of the *nrout* action in line 5 will be blocked, because a `<gameover>` tuple already exists in the remote tuple space, and then the process following the > connective is executed. Hence, the first player who is able to add the tuple can postulate as winner of the game remotely writing the tuple `<winner,Me,Now>` (line 5), whereas subsequent players will locally write the tuple `<notify,"You lose!">` (line 7). The new `GameOver` process is illustrated in Fig. 3. Suppose that the two players in the figure download the script almost at the same time (1 and ∼1). However, when putting the *gameover* tuple in the server with the *nrout* primitive, the player in the left performs this action at time 2, blocking the corresponding action of the player in the right at time ∼2 (assuming $2 < \sim 2$) and he wins the game.

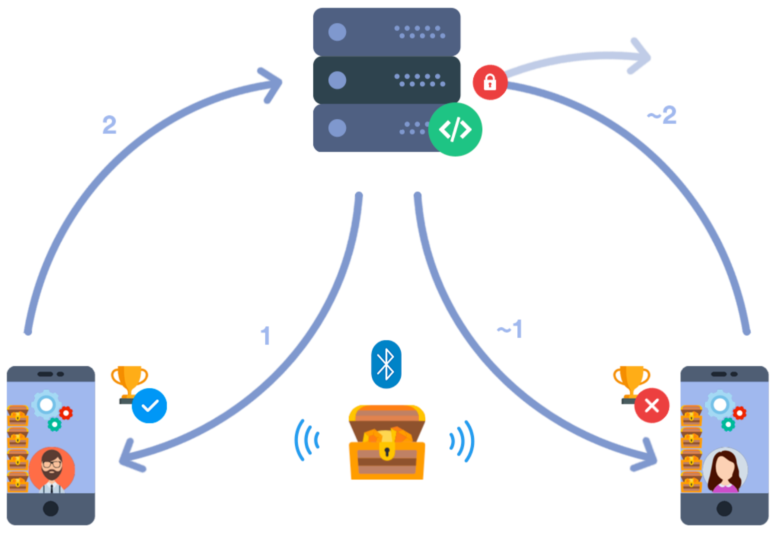

Figure 3: Scenario of game ending with the *nrout* primitive.

Therefore, we can prove that this new version of the script `TreasureHunt` guarantees that only one player will win the game, as it is stated in the following theorem.

Theorem 2. *Let us consider a smart thing b referring the script template $S_b =$ $TreasureHunt$ as defined in Code 5, with process GameOver as redefined in Code 6, and the initial configuration C_0 specified in equation (5), satisfying the same conditions as in Proposition 1. Then, there exists a trace*

$$C_0 \longmapsto^* C' \mid \langle t, P : 0\rangle_b$$

with t=`<gameover>` and for every progression from C'

$$C' \mid \langle t, P : 0\rangle_b \longmapsto^* C'' \mid \langle t, P' : 0\rangle_b$$

P' has no any other occurrence of t.

Proof. Reasoning as in the proof of Proposition 1, we can conclude that eventually (at most after $4n$ iterations) some of the devices will trigger the process GameOver of line 13 in Code 5, as defined in Code 6. If the tuple `<gameover>` is not in the cloud server's b profile, then it will be added to it as a consequence of applying rule $\textsc{NRemote}_1$. On the contrary, if this tuple is already in the profile of b, the rule $\textsc{NRemote}_2$ would be applied, and never again a new tuple `<gameover>` would be added to that profile. □

6. Discussion

The development of smart things is transforming people's lives, as we increasingly interact with them everyday. Our approach for handling all these interactions was inspired by the Programmable World Roadmap [11], a vision paper which foresees the evolution from today's IoT, merely based on data recollection, to truly programmable devices. This way, both smart things and smartphones would be able to learn from each other, and to evolve through each interaction in a transparent and dynamic way, promoting a technology that works for the people without requiring tedious manual configurations.

Architectures based on P2P models are gradually acquiring a greater presence in fields such as social networks [12] or recommendation systems [13]. One of the advantages of these architectures is that they offer natural place for storing virtual profiles of the users, containing contextual data (e.g. activities, relations with other users, etc.) [14]. Our proposal shares the goal of having single and coherent virtual profiles which are made available to third parties interested on giving a personalized service to the users. This way we are able to develop a seamless IoT environment which dynamically adapts to the needs and context of their users.

Social Computing (SC) [15] is the area of computer science that deals with the interaction between social behavior and computer systems. SC encompasses all those systems that collect, process and disseminate information related to individuals and groups of people. The goal is learning about people and their preferences and providing an easy adaptation of their IoT environment, reducing manual configuration of devices to a minimum. Indeed, a number of recent research works agree on giving support to the IoT by means of a paradigm focused on the people [16].

Nowadays, very few companies are able to access and process this enormous quantity of social information, and to exploit and make a profit from it. In practice, this reduces the SC marketplace to a small number of big stakeholders. As Tim Berners-Lee declared recently [17], SC systems should empower people, making them the fair owners of their information, and deciding who has access to it. Moreover, this information must be stored in a unique and accessible place which lets third parties use it in a controlled way, following the privacy preferences of the users.

A number of research groups are currently working in the development of techniques for collecting and processing contextual information with the purpose of building virtual profiles covering many different aspects, habits, and processes of the users. See for instance [18] for a review of the literature in this field.

In particular, Social Devices [19] is a mobile computing model which puts its focus in the wide functionality of current smartphones, some of them approaching them to humans, for instance translating a spoken text to written and vice versa. The model has been implemented as a JavaScript middleware platform called Orchestrator, which allows to proactively start interactions between IoT devices and the people around them. The platform helps in developing personalized smart devices using Arduino, Raspberry Pi or .NET Gadgeteer. Our proposal shares with Social Devices the goal of adapting the behaviour of IoT artifacts to the needs of their users, and also the use of small pieces of code —like our scripts— for specifying the interactions between artifacts and people. However, Social Devices does not elaborate a concept of virtual profile of the users, nor it makes it evolve through the interactions with the IoT. Furthermore the model is devoid of any formal foundation which might allow to reason about the behaviour of the systems built.

The authors of this paper have previously participated in the design of a mobile computing architecture called People as a Service (PeaaS) [20]. This reference architecture promotes inferring and building virtual profiles with the preferences and context information of people. Differently to what currently happens in social networks and other similar SC systems, these profiles are stored locally in the smartphones of their owners, instead of the servers of the social network enterprise. In this same sense, we advocate for developing collaborative architectures based on smartphones. Their pervasive presence in people's everyday lives and their increasing sensoring and computing capabilities, together with their communication skills, make smartphones key elements for obtaining, processing, and sharing information about their users [21]. Smartphones are also the most appropriate devices to be in charge of negotiating the interactions of their users with smart things in their environment.

Hence, Digital Avatars can be considered as a concrete implementation of the PeaaS reference architecture, which does not assume implicitly any underlying technology, nor there is any formal foundation associated to PeaaS. Moreover, the PeeaS model mainly focused on using the smartphone for inferring information for the virtual profiles, and then offering this information as a service to third parties in a secure manner. Instead, in Digital Avatars the interactions involved are not just simple transfers of data, but they consist in executing scripts with elaborate behaviour, which are run in the smartphone and access the virtual profile stored in it. Hence, Digital Avatars extends the PeaaS model with a mechanism that combines the data stored in the virtual profiles with the behaviour expressed in the scripts, allowing a general mechanism both for configuring smart things, and to complete the virtual profiles themselves with context information obtained from these interactions.

With the aim of proving the correctness and strength of our solution, in this work we have validated our framework using an approach based on a Linda-like model. Linda [6] is a coordination language where synchronization is achieved by means of a shared tuple space, and through a set of simple but enough expressive primitives [22]. However, a single shared tuple space would not be adequate for describing and analysing complex, pair-wise interactions among several participants, nor for defining restrictions on which tuples can be accessed, when, and by whom, as required in the Digital Avatars model.

A number of research proposals derived or related to Linda have been made by different authors. Several of them introduce some kind of mobility, mainly based on adding capabilities for remotely modifying a given tuple space. In particular, Lime [23] (Linda in a Mobile Environment) was proposed as a Linda extension to support mobile computing, considering both physical mobility of computing resources, and logical mobility of software agents. Lime defines transiently shared distributed tuple spaces by establishing P2P communications. Many of the goals of Lime are common with ours, but our framework also takes into account privacy issues, which are crucial for addressing security in virtual profiles.

Another well-known proposal in this field is KLAIM [24], a language which extends Linda by considering the possibility of remotely adding tuples to an *accessible* tuple space. In KLAIM both processes and data can migrate through different computing environments, explicitly supporting localities as first class data. KLAIM allows to describe read/write/execute intentions of processes with respect to localities, and it is endowed with a type system that checks whether processes actually comply with their intentions, preventing access right violations. Our proposal shares both concerns of addressing process mobility and data access rights.

With a similar philosophy, SCEL [25] (Service Component Ensemble Language) was designed to provide a parametric language to capture various programming abstractions for autonomic components and their interaction. SCEL explicitly represents aggregations, behaviors and knowledge according to specific policies, and supports formal reasoning on component behavior, from which properties of the overall systems can be inferred.

In both KLAIM and SCEL, Linda-like primitives were added to allow remote interactions with shared tuple spaces, similarly to our approach. Hence, digital avatars and virtual profiles could have been represented using any of these languages. However, our proposal makes a number of specific assumptions and constraints. Among them, that accessing a virtual profile has to be made only by its owner, with the only exception of remote writing under well-defined and restricted conditions. For that reason, we have chosen to define a new Linda-like language, with specific interaction primitives and semantics, and endowed with a particular access control mechanism built in the language, while in SCEL one would need to specify it via a specific acccess control policy.

7. Conclusions

In this paper, we have shown how the formalization of Digital Avatars by means of a multiple tuple space approach provides interesting tools to verify different properties of the system. Properties of bisimilarity and congruence allow to validate the compatibility of different versions of a given script in a Digital Avatars system. Moreover, we have shown that it is feasible to check formally the interactions in a Digital Avatars system, which gives the opportunity of studying its correctness, proving desired (or undesired) properties, and studying what happens in different situations. An example of it has been shown in the Treasure Hunt case study, ensuring that the game always ends. In order to avoid some issues on expressiveness, we have extended the framework by considering a remote blocking write action, which allows to ensure expected properties, for instance that only one player wins the hunt.

As previously discussed, our proposal presents shared concerns with other languages derived from Linda, such as KLAIM or SCEL. In particular, the definition of multiple data spaces, data access rights, and process location and mobility. In this sense, whether the specific restrictions of the operational semantics of Digital Avatars could be encoded in terms of any of these or other languages derived from Linda deserves some exploration, as it would give us the chance to reuse their results and tools. We leave this for future work.

Also as future work, we plan to model our framework in the executable specification language Maude [26, 27], which offers model checking capabilities to allow the analysis of desirable or undesirable properties exhibited by the scripts. The intention is specifying the transition rules defined for the Digital Avatars framework as rewriting rules, and then using Maude's model checker to prove properties of interest of the scripts.

References

[1] D. Guinard, V. Trifa, F. Mattern, E. Wilde, From the Internet of Things to the Web of Things: Resource-oriented architecture and best practices, in: Architecting the Internet of Things, Springer, 2011, pp. 97–129.

[2] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions, Future generation computer systems 29 (7) (2013) 1645–1660.

[3] A. Pérez-Vereda, D. Flores-Martín, C. Canal, J. M. Murillo, Towards dynamically programmable devices using beacons, in: Current Trends in Web Engineering – ICWE 2018 International Workshops (Revised Selected Papers), Vol. 11153 of LNCS, Springer, 2018, pp. 49–58.

[4] A. Pérez-Vereda, C. Canal, E. Pimentel, A formal programming framework for Digital Avatars, in: SEFM 2019 Collocated Workshops – Revised Selected Papers, LNCS, Springer, 2020 (in press).

[5] N. Carriero, D. Gelernter, Linda in context, Commun. ACM 32 (4) (1989) 444–458. doi:10.1145/63334.63337.

[6] D. Gelernter, N. Carriero, Coordination languages and their significance, Commun. ACM 35 (2) (1992) 96–. doi:10.1145/129630.376083.

[7] N. Busi, R. Gorrieri, G. Zavattaro, On the Turing equivalence of Linda coordination primitives, Electr. Notes Theor. Comput. Sci. 7 (1997) 75.

[8] R. Menezes, A. Omicini, M. Viroli, On the semantics of coordination models for distributed systems: The LogOp case study, in: Foundations of Coordination Languages and Software Architecture (FOCLASA 2003), Vol. 97 of Electronic Notes in Theoretical Computer Science, Elsevier, 2004, pp. 97–124.

[9] A. Perez-Vereda, J. Murillo, C. Canal, Dynamically programmable virtual profiles as a service, in: 5th IEEE International Conference on Internet of People (IoP 2019), IEEE, 2019, pp. 1789–1794. `doi:10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00317`.

[10] D. Bandera, A. Pérez-Vereda, C. Canal, E. Pimentel, One step towards dynamically programmable things: an implementation using beacons, in: 2019 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2019, pp. 1171–1176.

[11] A. Taivalsaari, T. Mikkonen, A roadmap to the programmable world: software challenges in the IoT era, IEEE Software 34 (1) (2017) 72–80.

[12] Y. Wang, A. V. Vasilakos, Q. Jin, J. Ma, Survey on mobile social networking in proximity (MSNP): approaches, challenges and architecture, Wireless networks 20 (6) (2014) 1295–1311.

[13] W.-S. Yang, S.-Y. Hwang, iTravel: A recommender system in mobile peer-to-peer environment, Journal of Systems and Software 86 (1) (2013) 12–20.

[14] T.-M. Grønli, G. Ghinea, M. Younas, Context-aware and automatic configuration of mobile devices in cloud-enabled ubiquitous computing, Personal and ubiquitous computing 18 (4) (2014) 883–894.

[15] F.-Y. Wang, K. M. Carley, D. Zeng, W. Mao, Social computing: From social informatics to social intelligence, IEEE Intelligent systems 22 (2) (2007).

[16] J. S. Silva, P. Zhang, T. Pering, F. Boavida, T. Hara, N. C. Liebau, People-centric Internet of Things, IEEE Communications Magazine 55 (2) (2017) 18–19.

[17] T. Berners-Lee, Solid, `https://solid.inrupt.com/`, accessed: 2019-01-21.

[18] P. Makris, D. Skoutas, C. Skianis, A survey on context-aware mobile and wireless networking: On networking and computing environments' integration, IEEE Communications Surveys and Tutorials 15 (1) (2013) 362–386.

[19] N. Mäkitalo, et al., Social devices: Collaborative co-located interactions in a mobile cloud, in: Proc. 11th Int'l Conf. Mobile and Ubiquitous Multimedia, 2012, p. article no. 10.

[20] J. Guillen, J. Miranda, J. Berrocal, J. Garcia-Alonso, J. M. Murillo, C. Canal, People as a Service: a mobile-centric model for providing collective sociological profiles, IEEE software 31 (2) (2014) 48–53.

[21] M. Raento, A. Oulasvirta, N. Eagle, Smartphones: An emerging tool for social scientists, Sociological methods & research 37 (3) (2009) 426–454.

[22] A. Brogi, J.-M. Jacquet, On the expressiveness of coordination via shared dataspaces, Sci. Comput. Program. 46 (1-2) (2003) 71–98. `doi:10.1016/S0167-6423(02)00087-4`.

[23] G. P. Picco, A. L. Murphy, G.-C. Roman, Lime: Linda meets mobility, in: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, ACM, 1999, pp. 368–377. `doi:10.1145/302405.302659`.

[24] R. De Nicola, G. L. Ferrari, R. Pugliese, Klaim: a kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330. `doi:10.1109/32.685256`.

[25] R. De Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, A. Vandin, The SCEL Language: Design, Implementation, Verification, Springer, 2015, pp. 3–71. `doi:10.1007/978-3-319-16310-9_1`.

[26] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic, Springer, 2007.

[27] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C. Talcott, Programming and symbolic computation in maude, Journal of Logical and Algebraic Methods in Programming 110 (2020) 100497.