Programming Smart Object Federations for Simulating and Implementing Ambient Intelligence Scenarios

Yannis Georgalis[1], Yuzuru Tanaka[2], Nicolas Spyratos[3] and Constantine Stephanidis[1,4]
[1]*Institute of Computer Science, Foundation for Research and Technology - Hellas, Heraklion, Crete, Greece*
[2]*Meme Media Laboratory, Hokkaido University, Sapporo, Japan*
[3]*Laboratoire de Recherche en Informatique, Université Paris-Sud 11, Orsay, France*
[4]*Department of Computer Science, University of Crete, Heraklion, Crete, Greece*

Keywords: Programming Languages, Service Federation, Smart Objects, Ubiquitous Computing, Ambient Intelligence.

Abstract: This paper leverages previous work on the concept of smart object federations and proposes a new dynamic programming language for implementing and simulating smart objects and their interactions. Following their description in the proposed programming language, smart objects can be fully simulated and used for describing ambient intelligence scenarios. In this context, the contributions of the paper are two-fold: (a) the introduction of a new programming language whose runtime semantics allows for a simple and effective description of smart objects, and (b) the description of meaningful interaction strategies, that are implemented in the proposed language, through which executable smart object federations can be used for simulating and implementing ambient intelligence scenarios.

1 INTRODUCTION

Smart objects, at a basic conceptual level, are autonomous computing units that can operate in isolation, connect to other objects, thus creating complex structures in the form of object federations, and interact with each other within their environment (Tanaka, 2010). A smart object can be viewed in three different ways: as a container unit, as a structural unit (passive or active), and as an interaction unit. As a container unit, an object contains memory and an arbitrary number of external services. As a structural unit, it can initiate (active) or accept (passive) connections from other objects, and, lastly, as an interaction unit, a smart object is able to issue and to execute commands, to emit and capture events, and to use the functionality that is implemented by its contained services.

Smart objects are thus defined in terms of their characteristics, capabilities and behaviour. Provided that they follow a basic set of operational primitives (Tanaka, 2010); (Julia et al., 2012), they can be realized as either software or hardware entities. In this paper we will be focusing on software smart objects with the final goal being the seamless coexistence of software and hardware smart objects that federate and interact with each other in order to implement ambient intelligence (AmI) scenarios. Our primary motivation was to provide an effective and intuitive way to program software smart objects for (a) the implementation of scenarios whose architecture requires software smart objects and (b) the simulation of scenarios whose architecture requires a mixture of software and hardware smart objects.

Akkadian, the dynamic executable programming language proposed in this paper, allows for the full implementation of the structure of smart objects and all interactions among them. The decision to propose a completely new language is justified for three reasons. First of all, due to the asynchronous semantics of the software smart object model it is difficult to create syntactically intuitive and effective frameworks in existing popular general-purpose languages. Secondly, an implementation in a general-purpose language has to accommodate extraneous programming statements that increase the complexity of the text representation of a program without contributing anything to the core functionality of the objects themselves (boilerplate code). Thirdly, full control to the language's high-level constructs was necessary not only to simplify the description of smart objects and their interactions but to also accommodate for future extensions and common programming and usage patterns that will emerge from the application of the smart object paradigm to real-world scenarios. In this context, the proposed programming language for the description of smart objects falls into the category of Domain Specific Languages (DSL).

In the following sections we will describe Akkadian's semantics and usage. Specifically, in section 2 we will give an overview of related work, in section 3 we will present the language, its runtime semantics and the usage of its basic statements, in section 4 we will show two examples, one of which is a simulated AmI scenario, and finally, we will conclude with a short discussion and our future work in section 5.

2 RELATED WORK

The proposed programming language shares many core design goals with the programming language nesC (Gay et al., 2003), which is used for programming sensor nodes for Wireless Sensor Networks (WSNs). Both are component-based, event-driven programming languages where the execution happens concurrently as emitted events reach the relevant code units. Additionally, like nesC, the execution of a command in Akkadian does not block the flow of the program. However, they have four fundamental differences. First of all, the proposed language is dynamic and weakly-typed whereas nesC, being an extension of the C programming language, is static and strongly-typed. Secondly, in nesC, the bidirectional interfaces that are either "used" or "provided" by a WSN node are essentially static libraries that are linked with the final program at compile time. In the case of Akkadian, a port that "provides" or "requests" a service, resolves the service definition dynamically, at runtime, and the target service can be implemented in any of the supported service technologies (see 3.2). Thirdly, in nesC, event handlers are implemented for individual events, and trigger conditions that involve multiple events have to be explicitly identified by the programmer using shared variables. On the other hand, in Akkadian, multiple events can be considered in an event handler and connected with Boolean operators to create powerful and expressive trigger conditions. Moreover, concerning the fourth fundamental difference, changes to the memory of smart object ports, can be captured as events and participate as trigger conditions in event handlers.

Aside from the fundamental differences, the common design decisions between Akkadian and nesC are no coincidence, since WSNs share similar goals with smart objects. However, WSNs are primarily designed for implementing low-level application scenarios, realized by the redundant deployment of low reliability hardware devices. In this sense, WSNs are organized and acting autonomously while trying to achieve their goals requiring little or no user intervention (Akyildiz et al., 2002). On the other hand, the target of smart objects is the description and implementation of high-level user-centric application scenarios in which the actions of the user actively influence the operation of the smart objects in the environment (Tanaka, 2010). Additionally, smart objects as mentioned before, are designed to model both hardware devices and software entities and it is that potential for seamless interoperation that drives their effectiveness to describe high-level, user-centric scenarios.

Moreover, languages like ESTEREL (Boussinot and de Simone, 1991) and LUSTRE (Halbwachs et al., 1991) target embedded hard real-time systems following an inherently synchronous message passing paradigm. PushLogic (Greaves and Gordon, 2006) is a compiled block-structured imperative language that provides the means to centrally manage the services offered by distributed or local devices. These devices are collectively referred to as pebbles. PushLogic allows the definition of rules that are evaluated concurrently ensuring that all the assertions over its variables hold during the execution of the program. Despite having similarities with Akkadian, since both consume diverse services and evaluate concurrently their rules, PushLogic targets safety-critical systems and thus provides only static memory allocation, does not implement arrays and provides lower-level constructs and operators in order to exercise very fine-grained control over the devices it manages and to keep the memory and processing requirements of the resulting programs low. ArchJava (Aldrich et al., 2002) is implemented as an extension of Java for allowing the latter to dynamically invoke external software components by binding them to bidirectional interfaces, called input and output ports. However, ArchJava is only applicable to components running under the same Java virtual machine, does not support the binding of distributed services, and provides no means for message-matching and concurrent execution.

Furthermore, Akkadian, can be viewed as a composition language (Nierstrasz and Meijler, 1995), since smart objects, through their ports, can offer or consume services that are either running on the same hardware or are distributed over the network. A smart object may incorporate an arbitrary number of services built on different technologies making an object described in Akkadian an effective container for composing functionalities from diverse, heterogeneous components. The composition language PICCOLA (Achermann et al., 1999) shares similar goals, while supporting both imperative and functional programming styles. It also supports concurrent component invocations but does neither provide any means for filtering invocation results nor is able to utilize distributed components.

Service composability, apart from the efficient and rapid implementation of high-level functionality, also allows for easier deployment and testing of the target application as it is being built. This is an important and useful property in the field of AmI, as also identified in the AMIGO project (Georgantas et al., 2005). In this project, composition is possible only for services implemented using the supplied libraries for the Java programming language. The Business Process Execution Language (BPEL) (Curbera et al., 2003) is a declarative, XML-based language that supports process-oriented service composition. BPEL describes a composition result (process), in terms of participating services (partners) and message exchanges or intermediate value transformations (activities). BPELJ (Blow et al.), extends BPL and enables the inclusion of standard Java code inside BPEL declarations. Aside from the fact that BPEL-based languages are available only for web services, their declarative XML-based syntax makes them verbose and difficult to read. Graphical development environments for BPEL, however, mitigate the readability problem at the cost of requiring the programmer to describe the composition with graphical elements representing process workflows.

3 LANGUAGE AND SEMANTICS

This section describes the structure and runtime semantics of Akkadian. At first, the structure of smart objects and the environment in which they can interact is presented, followed by the definition of services within the language's model. Subsequently, we present the basic syntactic constructs that allow an object to handle service and memory events, invoke service commands, and change memory elements. Lastly, we show how objects can sense the presence of other objects and establish connections in order to form smart object federations.

When appropriate, the Z notation (Spivey, 1992) will be used to describe more formally the structure and semantics of Akkadian's operations. We decided to use the Z notation over π-calculus (Milner et al., 1992), since the former allows for better modelling of the language's semantics through the changes operators perform on the system's data structures. Conversely, π-calculus would be ideal for modelling the execution of a specific scenario that utilizes many different objects. Hence, we will use Z's named schema notation where the *Name* of a schema appears on top, it is followed by the schema's variables, and the *Constraining-Predicates* appear last, below the short vertical line. Essentially, a schema is modelled as a set of tuples *Name* = $\{x_1:T_1;\ldots;\ x_n:T_n \mid \textit{Constraining-Predicates}\}$, where $x_i:T_i$ denotes that x_i has type T_i, i.e., $x_i \in T_i$.

3.1 Software Smart Objects

Software smart object federations are represented as labelled directed multi-graphs with no self-loops. These graphs are contained within the single smart object environment. In a smart object federation graph, the vertices represent the smart objects, and each directed edge a connection of a smart object to another one through a port. The head of an edge is the object that requests a service from another one (through a service-requesting port) and its tail is the object that offers a service (through a service-providing port). The label of the edge represents the common name of the ports through which the connection is established. Two objects may be connected to each other through multiple ports (multi-graph) but cannot establish a connection to a port contained in the same object (no self-loops). Figure 1 shows a smart object federation graph.

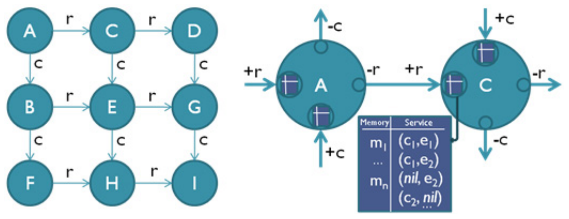


Figure 1: Example federation of 9 objects that are connected through ports named "r" (for row) and "c" (for column) (left), and the structure of smart objects and their connections, with service-requesting (-) and service-providing (+) ports (right).

In this sense, the environment is represented by the Z schema below, where $\mathbb{P}T$ denotes the power set of T, i.e., $A \in \mathbb{P}T \Leftrightarrow A \subseteq T$, and $X \rightarrowtail\!\!\!\!\!+\; Y$ denotes a partial function from a set, X, to another set, Y.

Environment

$objs : \mathbb{P}SO$
$conns : \mathbb{P}(SO \times SO \times NAME)$
$sport: SO \mapsto \mathbb{P}SP;\ rport: SO \mapsto \mathbb{P}RP$

$\forall x,y \in objs;\ l \in NAME \bullet (x, y, l) \in conns \Rightarrow x \neq y$
$\mathrm{dom}\ sport = \mathrm{dom}\ rport = objs$

Each smart object is essentially a pair of two sets: (*SP, RP*). The first set, *SP*, represents the service providing ports through which an external service can be used by the object or its peers, and the second set, *RP*, represents the service requesting ports through which an object can connect to another one and thus use the service that is provided by the corresponding service providing port.

An element *p* of *SP* is, in turn, associated with two partial functions, *mem* and *serv*, to an element of *M* and *S* respectively, with the set *M* representing the memory of the port and the relation $S \in I^* \leftrightarrow O^*$, (where $I^* = I \cup \{nil\}$ and $O^* = O \cup \{nil\}$), modeling Akkadian's runtime service representation (see also 3.2). The aforementioned types are described by the Z schemata below, where the symbol "$\leftrightarrow$" represents the set of relations between two sets, i.e., $S \leftrightarrow T = \mathbb{P}(S \times T)$. The types (sets) that are irrelevant for our modeling are referenced in the declaration header using Z's square-bracket notation.

$[NAME, SP, RP, M, I, O]$
$I^* \mathrel{\widehat{=}} I \cup \{nil\};\ O^* \mathrel{\widehat{=}} O \cup \{nil\}$

SO

$sp : \mathbb{P}SP;\ rp : \mathbb{P}RP$
$sname : SP \mapsto NAME;\ rname : RP \mapsto NAME$
$mem : SP \mapsto \mathbb{P}M$
$serv : SP \mapsto (I^* \leftrightarrow O^*)$

$\forall x, y \in sp \bullet sname(x) = sname(y) \Rightarrow x=y$
$\forall x, y \in rp \bullet rname(x) = rname(y) \Rightarrow x=y$
$\mathrm{dom}\ sname = sp$
$\mathrm{dom}\ rname = \mathrm{dom}\ mem = \mathrm{dom}\ serv = rp$

3.2 Services and Memory

From the programmer's viewpoint, a service-providing port has memory, and can export and make available to its containing object a set of commands and events. An object containing a service-requesting port can access and utilize all the available memory, commands and events as soon as it establishes a connection with a service-providing port. In this context, the functionality provided by a port, i.e., its commands and events, is referred to as a "service". A command invocation does not block the evaluation of the program and potential results are fed back to the invoking object as events.

Therefore, as mentioned in the previous section, services in Akkadian are modelled as a binary relation $S \in I^* \leftrightarrow O^*$. The domain of the relation represents the set of commands that can be issued through the service and the codomain represents the events that can be received through the service. The relation specifies that certain events (elements of the codomain) are generated as a result of the invocation of a command (elements of the domain). All the pairs of the form (*nil*, o), with $o \in O$, denote events that are generated by the service automatically without any previous command invocation (e.g., an asynchronous "door-opened" event). Similarly, all pairs of the form (i, *nil*), with $i \in I$, denote those commands that do not cause the service to produce any events (e.g., a "turn-off" command). Based on this, the semantics of the invocation of a command (*cmd*) that does not produce any event and the semantics of receiving an event (*evt*) that is not associated with a command are defined below. In the Z notation, the symbol "Δ" includes the referenced schema in the current one, allowing the use of the variables defined previously. The symbol "?" indicates the schema variables that work as the inputs of the operation. The symbol "$\lhd$" denotes the domain subtraction operation, which obtains all the pairs of the second set whose first member is not contained in the first set, i.e., $R \in S \leftrightarrow T$ and $A \subseteq S$, $A \lhd R = \{a{:}S;b{:}T \mid (a, b) \in R \wedge a \notin A\}$. Finally, variables decorated with an apostrophe (′) refer to their value in the post-state, after the evaluation of the schema.

InvokeCommand

ΔSO
$cmd?: I;\ p? : SP$

$p? \in sp$
$serv\ (p?)' = (\{cmd?\} \lhd serv\ (p?)) \cup \{(cmd?, nil)\}$

ReceiveEvent

ΔSO
$evt? : O;\ p? : SP$

$p? \in sp$
$serv\ (p?)' = (serv\ (p?) \rhd \{evt?\}) \cup \{(nil, evt?)\}$

As long as services adhere to this model, they can be fully used by smart objects through their service-providing ports. In the actual implementation of the language, different service technologies can be adapted to adhere to the model through the runtime environment's service-engine extension mechanism. Currently, we have implemented one service engine that is able to view any .NET (CLR) object as a service and another engine through which we can utilize all the services built on top of our in-house middleware, which is based on CORBA (DEC et al., 1992). It is our short-term goal to also implement a web service (Christensen et al., 2001) engine.

In Akkadian programs, services can be attached to service-providing ports using Uniform Resource Identifiers (URIs). The schema indicates the service technology whereas the rest of the URI is evaluated by the indicated service-engine to identify and resolve the actual service. The syntax for attaching services to service-providing ports appears below.

```
+a("clr://mscorlib/System/IO/Console");
+b("ami://LightService/bedroom");
```

Apart from the set of commands and events, a service-providing port, as mentioned in the previous section, also contains memory independently from the attached service. The port's memory is defined only within the context of Akkadian's runtime and is not associated with the actual service implementation. This is the only way for an Akkadian program to keep its state, since the ad-hoc definition of variables is not allowed. The update operation of a port memory is defined below, where the \ symbol represents the set difference operation.

UpdateMem

$$\begin{array}{|l}
\Delta SO \\
n{:}M \mathrel{+\!\!\!\rightarrow} NAME \\
v? : M; p? : SP \\
\hline
p? \in sp \land \operatorname{dom} n = mem(p?) \\
mem(p?)' = (mem(p?) \setminus \\
\qquad \{m{:}M \mid n(m) = n(v?)\}) \cup \{v?\} \\
\hline
\end{array}$$

Given a service-providing port with the name "a", which is expressed in Akkadian as "+a", the following statements can be used to invoke a command (first statement) and to write to the service's memory (second and third statements):

```
+a.WriteLine("Hello World");
+a["last_msg"] = "Hello World";
+a.previous = 1821; //+a["previous"]=1821;
```

3.3 Handling Memory and Service Events

Memory and service events are emitted from a service-providing port in the following cases: (a) when the port's memory is modified by the container object, (b) when the attached service emits an event – either due to the execution of a command on the same service or because the attached service independently emitted the event to indicate a sudden change in its state, (c) when another object connected through a matching port modifies its memory, or (d) when another object connected through a service-requesting port invokes a command on the service associated with the port. On the other hand, memory and service events are emitted from a connected service-requesting port in each of the aforementioned cases except (d).

Regardless of the reason an event is emitted, in Akkadian, its presence is checked with the *when* statement. The statement's syntax is: ***when*** *capture-expr statement*, where *capture-expr* is a Boolean expression of either a *service-event-capture-expr* or a *memory-event-capture-expr*, and *statement* is any valid Akkadian statement. In EBNF-like syntax, this can be expressed as follows:

```
when-stmt := 'when' capture-expr statement;
capture-expr := capture-expr
  ('and' | 'or') capture-expr
  | 'not' capture-expr
  | service-event-capture-expr
  | memory-event-capture-expr;
```

Essentially, a service-event capture expression and a memory-event capture expression are satisfied when the respective predicates below are true. In Z, the symbol "Ξ" adds the referenced schema in the current one, like "Δ", but also indicates that its post-state is not affected. We also assume three functions, *memmatch*, *eventmatch*, and *payloadmatch* that check whether a memory element, an event and its payload are, respectively, satisfied by the relevant capture expression. The operator "ran" obtains the range of the relation.

$$\begin{array}{|l}
\Xi SO \\
memmatch{:}\ M \longrightarrow \{true,\ false\} \\
eventmatch{:}\ O \longrightarrow \{true, false\} \\
payloadmatch : O \longrightarrow \{true,\ false\} \\
v? : M; e? : O; p? : SP \\
\hline
p? \in sp \land v? \in mem(p?) \land memmatch(v?) \\
p? \in sp \land e? \in \operatorname{ran} serv(p?) \ \land eventmatch(e?) \\
\qquad \land payloadmatch(e?)
\end{array}$$

When the capture expression is satisfied by all the conjunct event-capture expressions, then the statement is said to be "matched" and its associated actions are evaluated by Akkadian's runtime. Consider, e.g., the capture statement below.

```
when +t.TemperatureChanged(*sensorId,
    temp < +t["low_temp_thr"])
  and not +t.disabled {actions...}
```

This statement captures an event called "TemperatureChanged" (checked by *eventmatch*) that is emitted by the service attached to the service-providing port with name *t*, which is contained in the object described by the Akkadian program. Also, this statement captures two memory locations (checked by *memmatch*) that are part of the aforementioned service-providing port (*t*). Those memory locations are named “low_temp_thr” and “disabled” respectively. Interpreting the inequality and the Boolean operators, this specific statement is triggered when the “TemperatureChanged” event is emitted, its second parameter is less than the number written in memory position “low_temp_thr”, and the value of the memory position “disabled” is evaluated to *false*. The event in the above statement has two named parameters in its payload. The second one is called “temp” (for temperature) and is compared (checked by *payloadmatch*) with the value of the memory location “low_temp_thr”. The first parameter of the payload is the id of the specific sensor that reported the temperature, and since it is not needed in this specific instance, we could have omitted it completely from the event-capture expression. Alternatively, we can use the parameter alias syntax as above, an identifier prefixed with ‘*’ (in which case *payloadmatch* evaluates to *true*), in order to be able to refer to that parameter with the given identifier in the capture statement’s actions.

The actions of a “when” event-capture statement are evaluated every time the capture expression is matched. Testing of whether the capture expression is matched by the emitted events happens every time an event that participates in a specific event-capture expression is emitted to the containing object.

Capture statements can be nested in other capture statements. Consider, for example, the following statements:

```
when +t.TemperatureChanged(temp < 1)
  { when –g.Humidity(v > 0.2){} }
```

If the outer statement’s actions do not contain any other invocations, the above statements are semantically equivalent to the following:

```
when +t.TemperatureChanged(temp < 1)
  and –g.Humidity(v > 0.2){}
```

When a capture statement is matched, then all the encapsulated capture statements become eligible for being matched, i.e., they become active. Conversely, when a matched capture statement becomes unmatched, all the encapsulated capture statements cannot be matched even if their capture expression can be satisfied by the emitted events, i.e., they become inactive.

3.4 Scope, Linking and Path

A smart object *A* can connect to another smart object *B* through a port named *p*, when: (a) object *A* becomes “aware” of object *B*, (b) object *A* has a service-requesting port with name *p* (-*p*), which is unconnected, (c) object *B* has a service-providing port with the same name *p* (+*p*), (d) the command “*Link(-p, B)*” is invoked in the context of object *A*.

An object can become aware of another object through the *InScope* event. The event’s payload includes a reference to the object that enters the scope of the first object. Currently, Akkadian defines scope only in terms of proximity among smart objects. Namely, when the Euclidean distance between two smart objects in a two-dimensional space becomes less than the predefined value that represents the first object’s range, then, an *InScope* event is sent to that object with a reference to the second object. If the first and second objects have equal range, the *InScope* event is emitted to both objects. Among objects that are in the same federation (graph), *InScope* is not emitted, as those objects can be accessed through a path expression (see 3.4). Conversely, the *OutScope* event is emitted when an object exits the scope of another object. It is our short-term goal to define the scope in a broader, more systematic manner, in order to incorporate security primitives in the smart object model.

A connection between two smart objects can only be established through their ports. A service-requesting port that is not already connected can be connected to a service-providing port of another object as long as the latter has the same name as the former, i.e. the ports are “matching ports”. If these conditions are met, the *Link* command, successfully establishes the connection.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Link}} \\
\hline
\Delta \mathit{Environment} \\
\Xi \mathit{SO} \\
o1?, o2? : \mathit{SO}; p1? : \mathit{RP}, p2?\ \mathit{SP} \\
\hline
p1? \in \mathit{rport}(o1?) \wedge p2? \in \mathit{sport}(o2?) \\
\mathit{rname}(p1?) = \mathit{sname}(p2?) \\
\neg(\exists\, x \in \mathit{objs}; l \in \mathit{NAME} \bullet (o1?, x, l) \in \mathit{conns}) \\
\mathit{conns}' = \mathit{conns} \cup \{(o1?, o2?, \mathit{rname}(p1?))\} \\
\hline
\end{array}
$$

The *Link* command’s syntax in an Akkadian program, given a reference to an object (*objref*) and a service-requesting port (-*p*) is the following:

```
self.Link(-p, objRef);
```

The identifier “self” above, refers to a standard port that is part of all objects. Its attached service and memory enables an Akkadian program to invoke object-wide commands (e.g., Link/Unlink), to access object-wide state (e.g., object’s name, position), and to sense object-wide events (e.g., InScope). The *self* port cannot be used for establishing a connection.

Each smart object can refer to another object in the same federation through Akkadian's path expression. In a federation a port sequence defined in the context of a specific object can describe a path from that object to another one by means of the established connections through the ports that appear in the path sequence. Therefore for the federation depicted in Figure 1, object *A* can refer to objects *C* and *H* respectively through the expressions below, which are evaluated in the context of object *A*.

```
-r(self);     // object C
-c-c-r(self); // object H
```

Path expressions can participate in capture statements through which an object can capture the events of other objects. E.g., in Figure 1, object *A* can capture the connection of object *H* to *I* through port *r* with the following statement.

```
when –c-c-r-r(self).Linked(*obj) {}
```

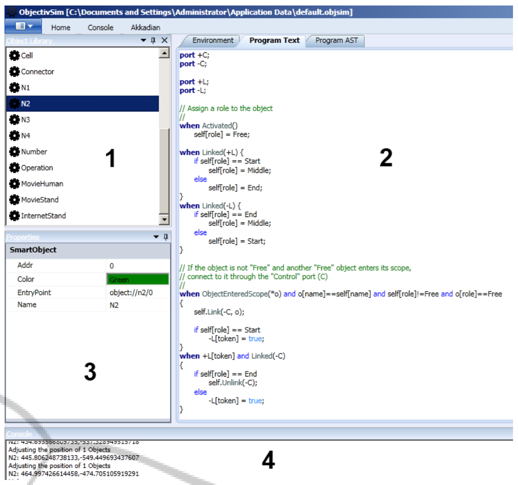
If any of the ports referenced by a path expression is not connected, then the path is invalid, i.e., it is dangling. Capture expressions that contain dangling paths cannot be matched. However, as soon as the path becomes valid, the affected expressions are evaluated and are subsequently tested against the emitted events on the target object.

The above examples, lastly, showcase another trait of Akkadian, which is weak typing. Whenever possible, the type of a value changes implicitly to match the one expected by the operation in which the value participates. Therefore, in the above examples, a port reference is implicitly converted to an object reference (to the object that contains it).

4 EXAMPLE

Akkadian's programming and runtime environment is implemented as a graphical tool in which developers can program and deploy smart objects. This tool, named ObjectivSim, can be seen in Figure 2 and Figure 3 and contains all the necessary components for (a) editing Akkadian programs and smart objects, (b) executing, simulating and visualizing their federations and interactions.

In ObjectivSim, when an object is defined in terms of its visual characteristics and its program, it can be dropped inside the environment (Figure 3) where it is evaluated and starts interacting with the other objects. The environment in our implementation is limited neither with respect to the number of objects it can host nor its size and can be zoomed in and out to allow a better view of all objects that operate in it.



```
port +C;
port -C;

port +L;
port -L;

// Assign a role to the object
//
when Activated()
    self[role] = Free;

when Linked(+L) {
    if self[role] == Start
        self[role] = Middle;
    else
        self[role] = End;
}
when Linked(-L) {
    if self[role] == End
        self[role] = Middle;
    else
        self[role] = Start;
}

// If the object is not "Free" and another "Free" object enters its scope,
// connect to it through the "Control" port (C)
//
when ObjectEnteredScope(*o) and o[name]==self[name] and self[role]!=Free and o[role]==Free
{
    self.Link(-C, o);

    if self[role] == Start
        -L[token] = true;
}
when +L[token] and Linked(-C)
{
    if self[role] == End
        self.Unlink(-C);
    else
        -L[token] = true;
}
```

SmartObject

Addr	0
Color	Green
EntryPoint	object://n2/0
Name	N2

```
Adjusting the position of 1 Objects
N2: 445.806248738133,-549.449693437607
Adjusting the position of 1 Objects
N2: 464.997426614458,-474.705105919291
```

Figure 2: The design environment (ObjectivSim) with the object library (1), Akkadian text editor (2), object property editor (3), and output console (4).

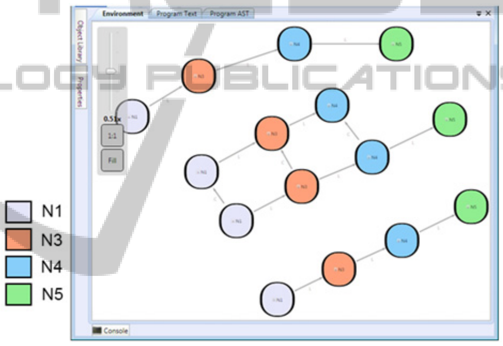


Figure 3: ObjectivSim's smart object environment.

4.1 Replicating a Simple Chain

As the first example of using Akkadian to describe smart objects we will use a simplification of the smart object connection algorithm, presented in (Julia et al., 2012). Given a sequence of an arbitrary number of smart objects connected through port *L* towards the same direction, i.e., a connected chain of n objects, the path *–L–L–L... –L* = $-L^n$, evaluated in the context of the first object, will reference the last (nth) one, the Akkadian program presented in this section, enables them to make an exact replica of their chain – provided that (a) the environment contains enough instances of the required objects, and (b) those objects can enter in the scope of the objects that participate in the initial chain.

In the first part of the Akkadian program, objects are assigned a role depending on their relative position in the chain:

```
port +L; port -L;
when Activated()self.role = Free;
when +L.Linked() {
  if self.role == Start
    self.role = Middle;
  else self.role = End; }
when –L.Linked() {
  if self.role == End
    self.role = Middle;
  else self.role = Start; }
```

The *Activated* event is emitted the first time an object is placed in the environment and its program is evaluated – i.e., the object is activated. At first, the object is not connected through any other object, and thus its "role" in the chain is marked as "Free". In Akkadian, when an identifier is not an alias for an event payload (see 3.3), it is equivalent to a string with the same name. Subsequently, when an object connects to the current object through port *+L*, then, provided that *–L* is unconnected, the current object is surely at the end of the chain, and so it is assigned the role "End". Conversely, when an object connects through port *–L* and *+L* is unconnected, then the current object is at the beginning of the chain and is assigned the role "Start". If both *+L* and *–L* are connected, then the current object is assigned the role "Middle". This can also be seen in Figure 3, where object *N1* has the role "Start", *N5* the role "End" and all others "Middle".

After assigning roles, the key point in the algorithm is for objects participating in the chain to connect to other "Free" objects with the same name through a port *–C*. This happens as soon as those free objects enter the scope of the object in the chain. Following that, when the next object connected through *-L* (the one on the right in the case of Figure 3) connects with another object through port *–C* and the current object has an object connected through its own *–C* port, then those child objects are connected to each other through their *L* ports, thus gradually forming a replica of that part of the chain.

```
port +C; port -C;
when InScope(*o) and
  o[name]==self.name and
  self.role!=Free and o.role==Free {
  self.Link(-C, o);
  if self.role == Start
    -L.token = true; }
when -L-C(self).Linked(*nextObj)
  and –C.Linked()
    -C+self(self).Link(-L,nextObj);

when +L.token and –C.Linked() {
  if self[role] == End
    self.Unlink(-C);
  else -L.token = true; }
when -L-C(self).Unlinked() self.Unlink(-C);
```

Finally, the last two *when* statements describe the terminal conditions and proceed in breaking all the connections through the *C* ports in order to finalize the replication of the chain (Figure 3). The key part of the Akkadian program in this case is the transmission of a "token" from the start-object of the chain to the end-object, which is gradually propagated to the next object as soon as the current object connects with another one through port *–C*. When the token reaches the end-object, then it means that the replication is finished and so end-object breaks the connection through port *–C* and, recursively, every previous object that observes that disconnection, also disconnects its own *–C*.

4.2 Ambient Theatre

In this example we consider a simple AmI scenario. When the user approaches a movie screen and performs a hand wave gesture with both hands, the lights turn off and a movie starts playing. While a movie is playing, the waving of the user's right or left hand instructs the movie screen to start playing the next or previous movie in the playlist, respectively. When the user moves away from the screen, the movie stops playing and the lights turn back on. Subsequently, when the user approaches an internet stand, the display shows information about the movies previously watched. Had the user not used the movie screen prior to approaching the internet stand, no information is displayed.

Breaking down this scenario, first of all, we assume that the following services are implemented and can be utilized by smart objects: (i) a Human tracking service that can track the position of the user inside a room, (ii) a Video playing service that controls a screen, maintains a playlist and is able to play movie files on the screen, (iii) a Human-gesture recognition service that is able to tell apart different hand gestures performed by a human, (iv) an Internet service that is able to present information on a screen about a topic, and (v) a Lights service that is able to control the room's lights.

Having the aforementioned services available, we can fully implement the scenario with software smart objects. Specifically, we can model the scenario using three smart objects: (i) a "Movie Human" smart object whose role is to move around the room, control the video screen and the internet stand, (ii) a static (unmovable) "Movie Stand" object that offers the video service and can control the lights, and (iii) an "Internet Stand" object that just offers the Internet service mentioned before. Implementing the scenario with these smart objects, apart from being able to utilize the services through smart objects (see 3.2), requires the “Movie Human” to be able to “move” in the environment, following the user’s physical movement into the room. This can be achieved by using the events of the Human tracking service:

```
port +Track("ami://room1/HTracking");
when +Track.PosChanged(*dx, *dy)
  self.Move(dx, dy);
```

In this example, however, we will show the implementation of a simulation of this scenario. In this case, the movement of the object representing the user can be simulated by dragging (with the mouse) the visual representation of the object in ObjectivSim. Additionally, the Human gesture service can be simulated by providing buttons on the visualized objects, and lastly, we can show the actions performed on the video and internet services by altering the colour and label of the participating objects. The buttons that trigger the simulated behaviour of the gesture service are declared in Akkadian as ports with the “button” scheme:

```
port +GestWave("button:///WaveHands");
port +GestLeft("button:///LeftHand");
port +GestRight("button:///RightHand");
```

The object representing the user can, therefore, determine when it should access or disconnect from the video service and the internet service through the *InScope* and *OutScope* events:

```
port -Video; port -Internet;
when InScope(*obj){
  if obj.name == "MovieStand"
    self.Link(-Video, obj);
  else if obj.name=="InternetStand"
    and self[movies] != null
      self.Link(-Internet, obj);
}
when OutScope(*obj){
  if obj.name == MovieStand
    self.Unlink(-Video);
  else if obj.name == InternetStand
    self.Unlink(-Internet); }
```

Finishing the description of the object representing the user, the following statements describe how the object interacts with the other ones:

```
when -Video.Linked() and
  +GestWave.Pressed() -Video.PlayMovie();
when -Video.Linked() and
  +GestLeft.Pressed() -Video.PrevMovie();
// Similarly for GestRight
when -Video.FinishedMovie(*m)
  self[movies] = self[movies] + m;
when -Internet.Linked()
  -Internet.Info(self[movies]);
```

On the other hand we simulate the behaviour of the “Movie Stand” smart object as follows:

```
port +Video;
when +Video.Linked() self.Color("Black");
when +Video.Unlinked() {
 self.Color("Gray");
 self.Display("MovieStand"); }
when +Video.PlayMovie(){
 self.Color("Blue");
 self.Display("Playing"); }
when +Video.PrevMovie(){
  +Video.FinishedMovie("Blue");
 self.Display("Prev"); }
// Similarly for NextMovie
```

Lastly, for simulating the “Internet Stand” smart object, in this example, we just change the colour of the object to a value that reflects the name of the first movie watched.

```
port +Internet;
when +Internet.Info(*m) self.Color(m[0]);
when +Internet.Unlinked()self.Color("Gray");
```

The visualization and execution of this scenario in ObjectivSim can be seen in Figure 4.

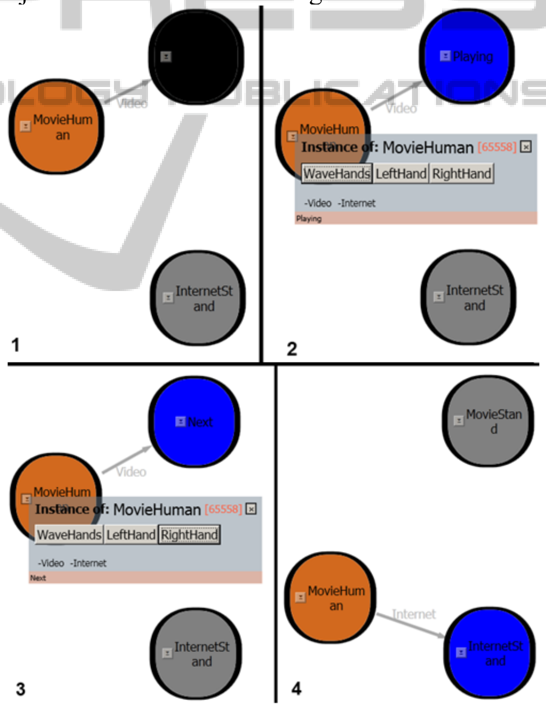


Figure 4: The Ambient Theatre scenario running under ObjectivSim.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented a dynamic, weakly-typed programming language for describing the behaviour and interactions of smart object federations within their environment. Using the proposed programming language, Akkadian, we can fully implement ambient intelligence scenarios that are described in terms of software smart objects. Those software smart objects can consume, and offer to other objects an arbitrary number of networked services through which they can sense and change the physical environment in the way the scenario defines. Additionally, for scenarios described in terms of hardware smart objects, Akkadian can be used for simulating their executions. Finally, we demonstrated the language's effectiveness through two examples in which smart objects are implemented as Akkadian programs.

As mentioned in the previous sections, one important short-term goal for Akkadian runtime is the implementation of additional service engines. Specifically, the implementation of a Web Service engine will allow software smart objects to fully utilize web services through their ports in addition to the already available service technologies. Provided that there is a plethora of publicly available web services on the Internet, this addition will greatly expand the utility of software smart objects.

A big section missing from Akkadian, and the smart object model in general, is the support for security primitives. Currently, as mentioned before, an object can connect to another object and use the services it offers as soon as it senses its presence (see 3.4). For achieving an effective level of security, the notion of the scope and the process through which objects can connect to each other, needs to be refined. Towards this direction, we are working on introducing scope-specific statements in Akkadian, through which an object will be able to define its own scope and control if, and under which conditions, it can appear in the scope of other objects.

Ultimately, the most important reason for proposing a new language – and not developing a framework in a general-purpose language – for programming smart objects is to systematically extract and support, in a syntactically intuitive way, common usage patterns as smart objects are used in ambient intelligence and ubiquitous computing scenarios. It is an important goal of Akkadian, its runtime environment, and its design environment to be robust and easily extensible for incorporating new, high-level behaviours.

ACKNOWLEDGEMENTS

This work has been supported by the FORTH-ICS internal RTD program "Ambient Intelligence Environments".

REFERENCES

Achermann, Franz, Markus Lumpe, Jean-guy Schneider, and Oscar Nierstrasz. 1999. *PICCOLA - a Small Composition Language*.

Akyildiz, I. F., W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. "Wireless Sensor Networks: a Survey." *Computer Networks* 38 (4) (March 15): 393–422.

Aldrich, J., C. Chambers, and D. Notkin. 2002. "ArchJava: Connecting Software Architecture to Implementation." In *Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002*, 187 –197.

Blow, M, Y Goland, M Kloppmann, F Leymann, G Pfau, D Roller, and M Rowley. "BPELJ: BPEL for Java Technology".

Boussinot, F., and R. de Simone. 1991. "The ESTEREL Language." *Proceedings of the IEEE* 79 (9) (September): 1293 –1304.

Christensen, Erik, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. 2001. *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl.

Curbera, F, Y Goland, J Klein, F Leymann, Thatte, and S Weerawarana. 2003. *Business Process Execution Language for Web Services, Version 1.1*.

Digital Equipment Corporation, Object Management Group, and X/Open Company. 1992. *The Common object request broker : architecture and specification, revision 1.1*. New York, NY: John Wiley.

Gay, David, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. "The nesC Language: A Holistic Approach to Networked Embedded Systems." *SIGPLAN Not.* 38 (5): 1–11.

Georgantas, N., S. B. Mokhtar, Y. Bromberg, V. Issarny, J. Kalaoja, J. Kantarovitch, A. Gerodolle, and R. Mevissen. 2005. "The Amigo Service Architecture for the Open Networked Home Environment." In *5th Working IEEE/IFIP Conference on Software Architecture, 2005. WICSA 2005*, 295 –296.

Greaves, D., and D. Gordon. 2006. "Using Simple Pushlogic." In *WEBIST 06: Proceedings of the Second International Conference on Web Information Systems and Technologies*. Citeseer.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud. 1991. "The Synchronous Data Flow Programming Language LUSTRE." *Proceedings of the IEEE* 79 (9).

Julia, Jeremie, Yuzuru Tanaka, and Nicolas Spyratos. 2012. "Formalization of an RNA-inspired Middleware for Complex Smart Object Federation Scenarios." In PECCS 2012, 96–105.

Milner, R., J. Parrow, and D. Walker. 1992. "A Calculus of Mobile Processes, i." *Information and Computation* 100 (1): 1–40.

Nierstrasz, Oscar, and Theo Meijler. 1995. "Requirements for a Composition Language." In *Object-Based Models and Languages for Concurrent Systems*, 924:147–161. LNCS. Springer.

Spivey, J. M. 1992. *The Z Notation: a Reference Manual*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd.

Tanaka, Yuzuru. 2010. “Proximity-Based Federation of Smart Objects: Liberating Ubiquitous Computing from Stereotyped Application Scenarios.” In *Knowledge-Based and Intelligent Information and Engineering Systems*, 6276:14–30. LNCS. Springer.