A RESTful and Decentralised Implementation of Open Objects

Paulo Ricca
Department of Computer Science
Royal Holloway, University of London
Egham, TW20-0EX Surrey
United Kingdom
paulo.ricca@cs.rhul.ac.uk

Kostas Stathis
Department of Computer Science
Royal Holloway, University of London
Egham, TW20-0EX Surrey
United Kingdom
kostas.stathis@cs.rhul.ac.uk

UbiComp'13 Adjunct, September 8–12, 2013, Zurich, Switzerland.

http://dx.doi.org/10.1145/2494091.2497585

Abstract
We show how to instantiate an existing framework for Open Objects to support a case-study for the Internet of Things. The resulting prototype illustrates the feasibility of the framework for a particular class of applications where physical objects with computational capabilities can collaborate in a decentralised manner. The framework described forms an initial step towards End-User Development on such complex distributed environments.

Author Keywords
Open Objects, Ambient Intelligence, Internet of Things, Web of Things, Ubiquitous Computing, Distributed Computing, Service-Oriented Computing, RESTful Services

ACM Classification Keywords
H.4.m [Information Systems Applications]: Miscellaneous

Introduction
As technological advances produce smaller and cheaper electronics, more everyday physical objects are becoming augmented with computation and communication capabilities, thus making us rethink the definition of a computer. At a low extra production cost, these added capabilities bring several advantages such as responding to the context of use, energy savings, collection of

anonymous usage information or malfunction reports, remote assistance, firmware updates, and the "killer applications" are probably yet to be thought of.

We envisage a near future where physical objects with a thin computational layer are able to (a) expose their capabilities to the outside, sharing them with other objects; (b) outsource capabilities from other objects or external services when needed; (c) allow their users to modify or attach new functionalities, effectively changing their behaviour.

We aim at giving back control to the users over their devices by promoting openness, transparency and innovation in the design and development of functionalities and behaviours of everyday objects. Furthermore, we argue that device owners should be released from unwanted design decisions that compromise the best usage of their equipment. We aim at making it possible to continue the design and development of an object, after its purchase: Object designers create objects in a way that allows these to be modified, expanded and connected to better target a user's personal needs. This concept is often referred to as Meta-Design [2].

An *Open Object* [4] is any physical object capable of capturing events from the environment in which it is situated, process them, and generate new events that allow it to interact with other objects and the user. The object is open because it can be augmented dynamically with new computational capabilities that allow the object to adapt its behaviour in order to become useful in situations that were not predicted at design time. As some objects may be designed with no particular use and behaviour in mind, a user may attach their own desired functionalities and behaviours to the objects or system of objects. Such open-ended, behaviour-less and incomplete Open Objects, we call *Hollow Objects*.

In order to support this paradigm, in the present work we further describe and define a lightweight, behaviour-based framework aimed at end-user Open Object choreography on Ambient Intelligence environments, and demonstrate an implementation with a real-world use-case scenario. With this holistic implementation we introduce how, based on the Open Objects framework, one can have a decentralised approach for collaboration on ad-hoc and/or network-less environments, which allows dynamic, context-aware and heterogenous interactions between users and objects.

We base our work on existing research in the field of Multi-Agents Systems [7] and the Web of Things [3, 1], which form both an inspiration and an underlying layer for the present work.

Scenario

Day 1: Between marking exams and responding to emails, John, a University lecturer is having a busy morning. Because in his online calendar John appears to be free, his electronic door sign still displays a green light and an inviting "come in!" message. John decides to change this status by turning the availability cube on his desk, turning the side with "busy" written on it, up, thus changing the door sign accordingly. By 4pm, the most pressing work has been done and John turns the availability cube back to the default "check calendar" position. Maria, who's been meaning to talk to John, saw John's green light from across the corridor and decided to come in for a chat.

Day 2: John has a meeting with a student at 9:30am. Unfortunately, his train is delayed and he won't be able to make it to the office in time. John then decides to display

a message on his door sign saying "running 10 mins late". In order to do that he sends himself an email from his phone with the title "Door Sign" and the message to be displayed as the email body. The student sees the message and decides to come back later. When John arrives at his office, he marks the email as read to signal the door sign that it should go back to its previous behaviour.

In this scenario, John makes use of Hollow Objects - a cube which senses orientation and door sign with a screen and a light of varying colour - and Internet services to build a web of objects that supports his needs in this context. His web calendar holds basic information on whether or not he is in the office and available to interact to peers. Therefore it's useful to use the calendar state as the default information to display at the door. When an exception happens, the user may change the message by turning the cube to an appropriate side or by sending an email to himself with the title "Door Sign". The cube rule takes precedence over the calendar's and the email rule takes precedence over the cube's.

The Open Objects Framework

Our approach is based on modular behaviour design. We see objects as physical entities that carry out different tasks according to the different behaviours that they display. We classify behaviours in three categories: (i) **Domain Independent Behaviours** offer generic support that the user may use to support his workflows (ii) **Domain Dependent Behaviours** are behaviours whose domain the user wants to control (e.g. his electrical affordances) and (iii) **System Behaviours** manage the system itself and include tasks such as coordination and service registration; they are in charge of the collaboration and form the minimum common denominator needed for the system to work independently. We will often refer to (i) and (ii) as **Application Behaviours**.

When a system of Open Objects manages itself, the System Behaviours coordinate the Application Behaviours in their user-defined tasks and themselves in an implicitly defined *System Workflow*. The System Workflow tells the System Behaviours how to work together and this is especially relevant as not all objects may implement all System Behaviours. Some objects implement none of these, due to their highly constrained capabilities. Very heterogeneous kinds of objects may, therefore, seamlessly coexist and collaborate in the same environment without the need for either implementing a compulsory set of system management instructions or for proxy objects to act on behalf of simpler ones.

As an example of behaviours we may think of a kitchen oven's main *Domain Specific Behaviour* to be *Cooking*, but the oven can also display *Domain Independent Behaviours* for *Time Keeping* and *Text Processing*. When introducing a new functionality to the oven, the user may combine the existing behaviours of the oven, but he can also use behaviours that exist on other objects nearby, or even behaviours of an online service, thus creating a collaborative activity between the selected objects.

Conceptual Framework

In Open Objects we define $Collaboration = \langle Objects, Behaviours, Rules, AccessPolicies \rangle$, where:

- $Objects$ is a (non-empty and finite) set of objects, $\{O_1, \ldots, O_n\}$, with $n >= 2$ and each object being equipped with a set of behaviours it can display;
- $Behaviours$ specify which behaviours objects are capable of displaying within an environment in

general and a collaboration in particular. We define $Behaviour =< Capabilities, Events, Behaviour\text{-}Definition >$.

- $Capabilities$ is a set of capabilities that objects possess and expose through their behaviours. $Capabilities$ take a set of inputs, $\{I1, ..., In\}$, and may produce an output. We define $Capabilities = \{Inputs, Output\}$.
- $Events$ is a set of events produced by the objects' behaviours, $\{E1, ..., En\}$; an event is defined as $< Origin, Time, Identifier, Arguments >$;
- $Behaviour\text{-}Definition$ is a definition or a pointer to a definition where the $Capabilities$ and $Events$ related to a concrete $Behaviour$ are defined;

- $Rules$ is a (finite) set of possible combinations of capabilities in $Capabilities$ available in a collaboration;
- $Access\ Policies$ is a (non-empty) set of possible combinations of objects (in $Objects$) and capabilities (in $Capabilities$) specifying which object can or cannot access which capability in a collaboration.

Framework Implementation

Based on the Open Objects framework briefly explained before, we propose one possible implementation. Firstly, we define a structure for behaviour and rule definitions and we proceed with a translation into a JSON[1] format, for each of them. In the diagram definitions, keywords between brackets represent data specific to each entity, that needs to be written for each behaviour/rule. The same keywords are represented in uppercase, on the JSON format.

[1] http://www.json.org/

Behaviour Definition

In our implementation, behaviours are defined as depicted in Figure 1.

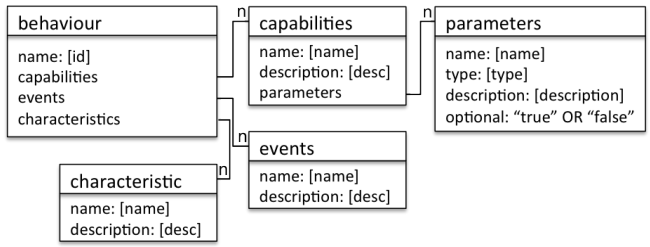

Figure 1: Generic structure of a Behaviour.

```
{
  'name': NAME,
  'description': DESCRIPTION,
  'capabilities':
  [
    {
      'name': NAME,
      'description': DESCRIPTION,
      'parameters':
      [
        {
          'name': NAME,
          'type': TYPE,
          'description': DESCRIPTION,
          'optional': 'True' | 'False'
        }
      ]
    }
  ],
```

```
  'events':
  [
    {
      'name': NAME,
      'description': DESCRIPTION
    }
  ],
  'characteristics':
  [
      {
        'name': NAME,
        'description': DESCRIPTION,
        'unit': UNIT_OF_MEASUREMENT
      }
  ]
}
```

During Discovery, objects advertise their behaviours by exposing a behaviour list in the following Behaviour Advertising format:

```
{
  'behaviours':
  [
    {
      'name': NAME,
      'description': DESCRIPTION,
      'behaviour_uri': URI,
      'definition': DEFINITION_URI
    }
  ]
}
```

Having the behaviour definition separated from the behaviour advertising brings several advantages: it minimises data transferred; reduces required memory space by allowing definitions to be stored externally; and encourages compliance to pre-existing behaviour definitions, making it easier to develop generic capabilities and rules.

An object may advertise other's behaviours, thus becoming a proxy for them. This is useful for having collaborations than span across several types of networks (Wi-Fi and Bluetooth, for instance).

Open Rules Definition

To coordinate Open Objects, we use Open Rules. Open Rules orchestrate behaviours by assigning them to *Purposes*. By default, a purpose has the same name as its behaviour but it is otherwise possible to assign several Purposes the same Behaviour by naming them differently, thus making different objects with the same Behaviour take on different roles in the collaboration. A generic rule definition is depicted in Figure 2.

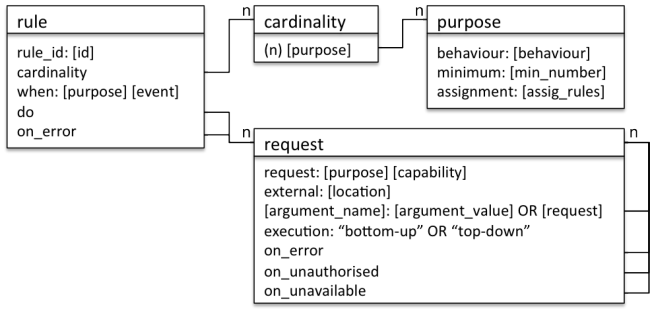

Figure 2: Generic structure of an Open Rule

An Open Rule describes the procedure that should follow the happening of an event from a certain behaviour. The procedure is defined by a tree of behaviours matched to concrete objects at run-time. This matching may be based on assignment rules, if specified in the Open Rule. Open Rules are purposely defined in this way (as opposed

to a traditional programming language, for example) to bridge the gap between the definition of a system and the metaphors used in an End-User Development interface. In fact, it should be possible to implement such interface in a very simple way, but merely presenting the rule structure in a graphical structure, with a few suitable helper tools, to assist the user when connecting behaviours.

```
{
  rule:
  {
    rule_id: ID,
    cardinality:
    {
      PURPOSE:
      {
        behaviour: BEHAVIOUR,
        minimum: MIN_NUMBER,
        assignment: {}
      }
    },
    when: PURPOSE EVENT,
    do:
    [
      {
        request: PURPOSE CAPABILITY,
        external: URI,
        ARGUMENT_NAME: ARGUMENT_VALUE  OR
            REQUEST,
        execution: bottom-up  OR
            top-down ,
        on_error: {},
        on_unauthorised: {},
        on_unavailable: {}
      }
  ],
  on_error: {}
  }
}
```

System Behaviours

A number of behaviours need to exist in the environment for a collaboration to be possible. These manage the collaboration, keeping track of objects and their behaviours, passing and processing events and storing and executing rules. We call these System Behaviours. These behaviours do not need to be implemented on all objects and it is likely that many lighter-weight objects would not implement any of them. Next, we briefly describe each System Behaviour and its basic operations. Interaction steps between System Behaviours are depicted in Figure 3 and referenced in the text below.

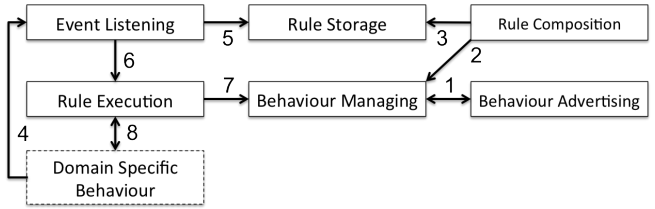

Figure 3: System Behaviours interaction - Meta-Workflow

Behaviour Advertising

Advertises each object's behaviours. The advertised behaviours are registered in the Behaviour Manager (step 1). Advertising is effectively the counterpart of a Discovery process, and this can happen in an active and passive ways. In the active mode, the Behaviour Manager searches for objects in the environment and requests advertising from them, whereas in the passive mode, objects have a reference to the Behaviour Manager and register themselves in it.

Rule Composition

A Rule Composition tool is one in which the user is able to develop their application rules. It uses the definitions of the behaviours registered in the Behaviour Managing

(step 2), or others supplied by the user, to present the different capabilities available for the user to combine and customise. Rules are then registered in the Rule Storage (step 3).

Event Listening
This behaviour triggers the execution of a rule by listening to events (step 4), querying the Rule Storage for suitable rules (step 5) and, in the existence of one, requesting its execution from the Rule Execution (step 6). Event Listeners may hold and share a registry (such as [5]) of past events to allow objects that arrive at the environment after an event has happened, to still react to it effectively.

Rule Execution
Carries out the execution of rules. When calling a capability (step 8), it retrieves the reference to the respectively assigned objects from the Behaviour Managing (step 7).

Rule Storage
Stores and provides rules. Each rule is mapped to the event(s) it related to.

Behaviour Managing
Assigns and keeps an up-to-date map between rule purposes and objects' behaviours.

System Life-Cycle

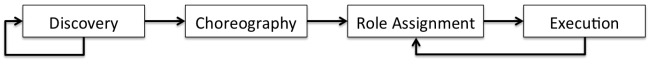

Figure 4: System Life-Cycle

A System Workflow regulates in the interactions between the system Behaviours, in the **System Life-Cycle**, depicted in Figure 4 and detailed below.

Discovery
Is the process of discovering objects in the environment and registering behaviours in the Behaviour Manager (step 1).

Choreography
The process of creating and registering a rule, carried out by the Rule Composing and the Rule Storage (step 3).

Purpose Assignment
The Behaviour Managing uses an internal logic when assigning purposes (step 7). It may, for example, keep a record of object performance or use a more complex reputation trust model such as [8], according to the assignment rules specified in the rule, if existent.

Execution
The process triggered by the Event Listening, which, after listening to an event (step 4), queries the Rule Storage about appropriate rules (step 5), and passes it on to the Rule Execution (step 6). The rule contains references to purposes, which are assigned to objects' behaviours by the Behaviour Manager (step 7). This process happens throughout the execution, and a purpose may be re-assigned due to, for example, an object becoming offline. When assigned, an object is requested to perform the appropriate action, defined by the rule (step 8). Objects may produce other events (step 4) which may trigger the execution of other rules.

There are two basic ways of executing a rule tree in this context: top-down (default) and bottom-up, and this can be set differently for each capability request. Top-down implies that the parent node of a branch is executed first and the whole branch is passed on to the Capability. This comes with a data transfer and memory cost but allows for more complex workflow rules as each Capability

controls in which order, if, when and how many times the child branches are executed. This is useful in a "loop", a "conditional branching" or a "parallel split" domain independent capabilities, for instance. Bottom-up is lighter on the capabilities, as the child branches are executed first and only their outputs are passed on to the capabilities as the final values for each parameter.

Implementation-wise, each Capability (including System Behaviour Capabilities) are exposed as Rest [6] resources. By default, they can be accessed using the format "http://OBJECT_BASE_URL/BEHAVIOUR/CAPABILITY" but this can be specified differently by the Behaviour Advertiser. When requesting the execution of a Capability, a POST request is made to its url, and the rule branch (in the case of a top-down execution) or the capability request code (otherwise) form the message body. The Capability may respond with a HTTP 200[2] code and its output as the message body if the execution is successful or with 500 for execution errors, 401 for unauthorised access and 503 for an unavailable capability. In the latter three cases, the appropriate error execution branch, if specified, should be executed.

This implementation is used to produce an experimental setup described in the next section.

Case-Study

Ahead, we describe the implementation of the system described in the scenario using Open Objects.

[2]http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Figure 5: User interacting with Availability Cube

Figure 6: Message being displayed on the Door Sign

System Behaviours
(B1) Event Manager - listens, propagates and triggers execution of events; (B2) Rule Storage - stores rules; (B3) Rule Execution - executed rules; (B4) Behaviour Managing - keeps track and discovers behaviours in the environment; (B5) Behaviour Advertising - advertises the objects Behaviours.

Domain Independent Behaviours
(B6) Time Keeping - keeps track of time; (B7) Flow Control - controls the execution flow of rules by suspending, repeating or choosing branches; (B8) Parallel - creates a rule branches; (B9) Text Processing - performs operations on strings such as comparison or substitutions.

Domain Dependent Behaviours
(B10) Email Checker - checks an email account for unread emails; (B11) Calendar Checker - checks an online calendar for events; (B12) Orientation Sensor - communicates the orientation of an object, in relation to the gravity axis; (B13) Message Displayer - displays a message; (B14) Traffic Light - controls a coloured light.

Hardware
There are three different devices in the system: (D1) Computer, which implements B1, B2, B3, B4, B6, B7, B8, B9, B10 and B11; (D2) Desk Cube (Figure 5), which implements B5 and B12; (D3) Door Sign (Figure 6), which implements B5, B13 and B14. Both devices are based on the Arduino[3]-compatible Nanode[4], connected to an Ethernet network. A web client and a small web server were implemented and capabilities are exposed as Rest resources. These form a thin layer of connectivity over the existing capabilities, effectively opening the objects to the outside and making it available for collaborations. To ease development, all the System Behaviours are implemented on D1 only, but their implementation is simple enough to be embedded on a lighter-weight device.

Events and Rules
The events produced during the execution of the system are the following: (E1) startup; (E2) do_a_cycle; (E3) check_email; (E4) check_cube; (E5) display_availability_from_calendar; (E6) check_calendar.

In order to coordinate the behaviours in the system, different rules were written, and each attached to an event: (R1) start_availability_system, on E1 - produces E2 to start the systems main rule cycle; (R2) cycle_availability, on E2 - every 5 seconds, produces two events: E4 and E3, causing R2 and R3 to execute; (R3) email_checking, on E3 - checks if there is an email targeting the door sign and changes the message if appropriate, if not, it produces E4, causing R4 to execute; (R4) cube_checking, on E4 - displays a message and changes the colour of the traffic light according to the side of which the cube is turned. If the cube is unavailable or the side is the default one, the rule produces E5; (R5) display_availability_from_calendar, on E5 - changes the message and colour of the traffic light according to the state of the calendar. uses a secondary rule, R6, to check the calendar; (R6) check_calendar, on E6 - checks one particular calendar to see if there is an event at the current time.

As an example, we include R6:

```
{ rule:
  { when: check_calendar,
    do:
    [ { request: calendarchecker get_state,
        calendar:
            'https://www.google.com/??',
        date_time:
        { request: timekeeping now }
    } ] } }
```

This rule says that when "check_calendar" event happens, "get_state" on the "calendarchecker" behaviour should be requested and, as it is the only, and last request of this

[3]http://www.arduino.cc
[4]http://www.nanode.eu

sequence, the outcome value returned by this capability will also becomes the outcome of the rule itself. "get_state" accepts two parameters: "calendar" is the url of the calendar to check (not presented fully here); and "date_time" which is connected to another capability request that returns the current date and time. The output of the rule is the current state of the user's calendar, which is then used in a condition on R5.

Execution

After rule registration and behaviour discovery, behaviours produce events that are picked up by the rules. If the cube is turned to the default side, the door sign displays a message given by the current state of the user's web calendar (R5). The cube has different labels on the sides and when the user wants to display an exception message, they turn the cube to the appropriate side. This causes R4 to execute, which changes the message and the light colour on the door sign according to cube orientation. In the existence of an unread email with the title "Door Sign", the message is set to the email message body instead, as specified by R3, which takes precedence over the cube and calendar-related rules.

Conclusion

We have shown how to instantiate the Open Objects framework [4] to support a case-study for the Internet of Things, thus illustrating the feasibility of the framework on a scenario where physical computational objects can collaborate in a decentralised manner. However, some challenges arise in the case of extreme decentralisation and network instability, such as how to prevent a certain rule from being executed more than once when this behaviour is not desirable and the object ecosystem has been split into more than one isolated group with similar capabilities and stored rules. The usefulness of the presented framework is yet to be demonstrated in these cases. Future work include Behaviour Assignment rules and an End-User Development interface definition.

References

[1] Dillon, T. S., Talevski, A., Potdar, V., and Chang, E. Web of things as a framework for ubiquitous intelligence and computing. In *Ubiquitous Intelligence and Computing*. Springer, 2009, 2–13.

[2] Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. Meta-design: a manifesto for end-user development. *Communications of the ACM 47*, 9 (2004), 33–37.

[3] Guinard, D., and Trifa, V. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain* (2009).

[4] Ricca, P., and Stathis, K. Open objects for ambient intelligence. In *Ambient Intelligence*. Springer, 2012, 320–327.

[5] Ricca, P., Stathis, K., and Peach, N. A lightweight service registry for unstable ad-hoc networks. In *Ambient Intelligence*. Springer, 2011, 136–140.

[6] Richardson, L., and Ruby, S. *RESTful web services*. O'Reilly Media, 2008.

[7] Schaeffer-Filho, A., Lupu, E., Sloman, M., Keoh, S.-L., Lobo, J., and Calo, S. A role-based infrastructure for the management of dynamic communities. In *Resilient Networks and Services*. Springer, 2008, 1–14.

[8] Wang, Y., and Vassileva, J. Trust and reputation model in peer-to-peer networks. In *Peer-to-Peer Computing, 2003.(P2P 2003). Proceedings. Third International Conference on*, IEEE (2003), 150–157.