2010 International Conference on Complex, Intelligent and Software Intensive Systems

On a Journey from Message to Observable Pervasive Application

Geert Vanderhulst Kris Luyten Karin Coninx
Hasselt-University – transnationale Universiteit Limburg
Expertise Centre for Digital Media
Wetenschapspark 2, 3590 Diepenbeek, Belgium
Email: {geert.vanderhulst,kris.luyten,karin.coninx}@uhasselt.be

***Abstract*—Bringing together heterogeneous computing devices and appliances gives rise to a spontaneous environment where resources exchange messages, such as a mobile phone telling the car's stereo to mute. We also witness computer-augmented resources become physically simpler to use (e.g. less buttons) but become more complex to handle in their digital dimension (e.g. overloaded user interfaces). As a consequence, the behavior of the pervasive applications leveraging these resources gets even more complex to understand and configure. This demands for tools that help developers and end-users inspect and manipulate the current state of the pervasive computing environment during execution time. We present models and tools that support the development and deployment of applications that can be observed at runtime, by means of the messages they exchange, the properties they manipulate and the rules they define.**

I. Introduction

In the vision of pervasive computing users are assisted by applications that automatically adapt to the environment they are interacting with [15]. The development of context-aware applications is supported by many systems [6], [7], [12], but still remains a challenge due to the dynamic nature of a pervasive computing environment: the configuration of the environment is often not known in advance and can evolve at runtime, for instance when new resources become available. This advocates the need for scalable tools that provide real-time support for debugging context-aware applications and their execution environments at runtime.

Ko et al have shown that interactive debugging tools such as the Whyline help programmers better understand the state of an application [9]. However, integrating such tools in a pervasive environment is not evident because pervasive applications are no longer predeveloped software packages, but are composed dynamically by connecting a set of distributed services and interaction devices. The events that occur in one resource can have an impact on other resources and since events are hard to trace it is even more difficult to link them to certain behaviors in the environment, also for developers. Besides, system-driven behavior in a context-aware computing environment can cause unwanted side-effects, for example when gestures are recognized by accident. To recover from mistakes, an undo mechanism is required to enable users to correct unwanted behavior.

In this paper we present models and tools to support the development and use of *dynamically observable and configurable pervasive applications*. Our contribution lays in the combination of models that describe the context of use and the behavior of applications, and tools leveraging these models such as those in figure 1 to add live debugging capabilities to a pervasive computing environment. By adopting the proposed models in a pervasive software architecture, we show that pervasive applications can be inspected at different layers while they are executing. Tools leveraging these models allow developers to monitor the message flow (section III), inspect the environment configuration (section IV) and observe and configure the behavior of applications (section V). We discuss the architecture of our system and show a meta-user interface that integrates live debug tools (section VI) which we have applied in a case study (section VII).

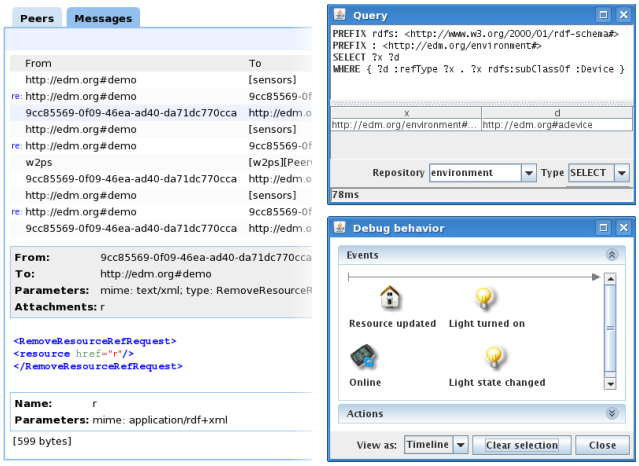


Fig. 1. Tools add live inspection capabilities to pervasive applications such as monitoring the messages that are exchanged between resources, querying the current state of the environment and observing the behavior of running applications.

II. Related Work

Several frameworks have been developped that support the creation of applications in a pervasive computing environment [6], [7], [12]. In these frameworks, models are widely used to capture the context of use of the environment. Gu et al propose an ontology-based context model and an OSGi-based middleware infrastructure to share knowledge and to

978-0-7695-3967-6/10 $26.00 © 2010 IEEE
DOI 10.1109/CISIS.2010.52

IEEE computer society

dynamically discover the state of the environment [7]. Another approach, sTuples, uses Tuple Spaces and ontologies to provide a logically shared memory along with data persistence [8]. The ReWiRe framework which is applied as testbed in this paper incorporates a dynamic ontology-based runtime environment model [12] that allows both the semantics of the environment and its context data to change over time.

To assist developers with the design of a pervasive application, special-purpose tools have been proposed that range from programming languages and IDEs for prototyping pervasive applications [14] to runtime monitoring tools [13]. Most of these tools focus on a specific stage in the development process or are tailored to specific application domains. However, since the behavior of a pervasive application heavily depends on the context of use, pervasive applications must be inspectable and controllable at runtime as well.

Coutaz showed the importance of providing runtime control over end-user interfaces in [3]. She introduced the concept of a *meta-user interface* as the set of functions (along with their user interfaces) that are necessary and sufficient to control and evaluate the state of interactive ambient spaces. We mainly target a meta-user interface for developers that integrates tools to support truly observable applications whose state can be queried at runtime. To a certain extent, the Speakeasy framework [10] helps users to understand the computing system through low-level context properties that can be examined at runtime. The Whyline [9] goes one step further by allowing programmers to ask *why did* and *why didn't* questions about runtime failures in applications, and provides direct access to the runtime that they need to debug. It keeps track of a history of property values and expressions that modified these values. Pervasive applications, however, are driven by context changes and events that occur as a result of these changes. To trace the source and cause of an event at runtime, extra information is required that relates changes in the environment configuration with the resources that caused them to happen.

Weis et al introduce a specialized graphical programming language that provides constructs to master common programming tasks of pervasive computing applications and a tool to debug the application logic [14]. We seek to support the debugging of applications after they are released as well by feeding a runtime behavior model with Event-Condition-Action (ECA) rules. ECA rules have been proposed to build reactive pervasive software systems before [4], [11]. When an ECA rule is triggered because of an event that occurs, its condition is verified and its action is executed. However, current ECA-based systems are still difficult to understand and manage by end-users. This is largely due to the fact that users can get easily confused by things happening beyond their control [2]. The large number of rules typically required to achieve a useful behavior also increases the complexity to obtain a clear view on the overall system. We aim to reduce this complexity by linking the context of use with the behavior of an application and hence provide more accurate explanations about why things happen or do not happen. Moreover, by extending ECA rules with an inverse action (i.e. $ECAA^{-1}$ rules) users can rollback the state of the environment when needed.

III. Messages: a Vehicle for Data Exchange

The data flow between entities in a pervasive computing environment is an important measure to verify whether an application behaves as expected. Nevertheless, most pervasive frameworks treat the networking aspect as part of the underlying middleware and hence the applications designed using the framework are often difficult to debug at the network layer once deployed. In a message-oriented network, messages contain information about events that have occurred, requests that are made and responses that are sent back.

We use W2P[1] as a light-weight scalabale communication framework for pervasive environments. It supports live network inspection through a web interface along with different addressing and message orchestration schemes which are missing in popular communication protocols such as XMPP[2] or UPnP[3]. A W2P message gateway, designed as a web application with open REST architecture, routes messages to their destination, just like an IP router forwards IP packets. Through this gateway, W2P peers (i.e. communicating entities) can discover each-other and exchange messages in peer to peer style over the HTTP protocol. Since peers do not run a server and use default HTTP traffic to communicate, there is no need to reconfigure firewalls. Peers are addressed by a name they statically request or dynamically acquire upon registration with a W2P gateway. This allows entities to talk with each other using meaningful names instead of IP addresses. For example, a service that is referred to by a URI (e.g. http://edm.org#aservice) can use this URI as its network address, making communication as transparent as possible. Furthermore, peers can use group names to subscribe to a range of events (e.g. [http://edm.org#aservice][events]), an individual event (e.g. [http://edm.org\#aservice][events][OnlineEvent]) or to address a number of similar resources at once ((e.g. [http://edm.org][services]) even without knowing who is actually addressed, similar to sending a message to a mailing list. These are common needs for a pervasive application, just like asynchronous and synchronous message exchange patterns: although most of the communication in a pervasive computing environment will be asynchronous, there are also situations where an application must explicitly wait for a reply message before it can continue its execution, for example when it needs to execute a remote query and depends on its results. Therefore W2P offers asynchronous and synchronous message exchange by default together with a mechanism to dispatch incoming messages to dedicated message handlers. In promiscues mode, a W2P gateway captures all the messages exchanged over the network in a certain time span which are presented in a web interface along with the registered peers and the groups they are subscribed to. Figure 1 shows a W2P message with attachment in the web interface.

[1] http://research.edm.uhasselt.be/w2p
[2] http://xmpp.org/
[3] http://www.upnp.org/

IV. Inspecting the Environment

Pervasive applications rely on the ability to query and manipulate the current context of use and thus must have easy access to context information and its semantics. In particular, pervasive applications must be able to (1) *discover available resources*, (2) *query the properties of a resource*, (3) *manipulate the properties of a resource* and (4) *get notified when properties undergo changes*. Besides, we argue developers must be able to inspect and manipulate the context of use to understand why applications behave in a certain way and to verify whether this behavior is correct. For this purpose, we designed a context model that can be encapsulated in a pervasive computing framework and queried by developers at runtime using debug tools.

A. Environment model

We present a decentralized model for storing and querying context information in a pervasive environment. In this model, context data is distributed amongst a set of heterogeneous computing nodes and the 'context store' (CS). The CS stores information about the *semantics of resources* such as properties of resources and relations between resources. The semantics are defined by OWL DL ontologies. An upper environment ontology shown in figure 2 defines common concepts such as users, devices, services, tasks and user interfaces and the relations that apply between these concepts. For example, the ontology defines a *Task* concept which is presented by a *UI* concept and hence acts as an abstraction for activities that are made available to end-users through e.g. a graphical or a speech-based user interface. The *Device* concept denotes a double role: a device can be used to interact with the environment by a user (*User* concept) as well as to host software services (*Service* concept). Finally, the *Sensor* concept acts as an endpoint through which context events that occur in a resource are published in the environment. In other words, a sensor publishes remote context events to interested entities in the environment. The structure of the sensor data is also described in the ontology so that services that are subscribed to the sensor can become aware of its semantics and hence consume its data.

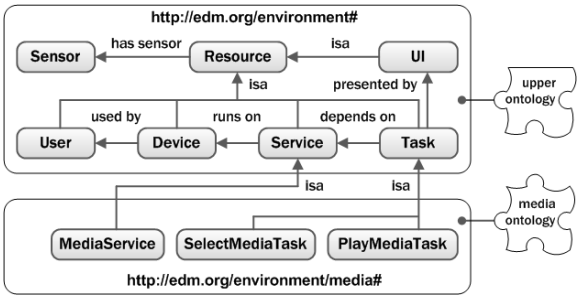


Fig. 2. The environment topology and current context of use are described by ontologies and instances of these ontologies.

The concepts defined in the upper ontology provide a base structure upon which ontologies for specific domains can be constructed. These domain ontologies are aggregated with the upper ontology at runtime and shared amongst applications that depend on their knowledge base. Besides the semantics of the environment and its applications, the CS also includes a registry of references to *instances of resources* whose execution context resides on distributed computing nodes. For example, a service running on device X is advertised in the CS as a service of type Y residing at location X. The CS facilitates the discovery of resources through references: software entities can interrogate the meta-information stored in references (e.g. type and location of a resource) and use this information to acquire the full context of the resource as RDF triples by sending a request message to the device the resource is published on. Hence, context is produced and updated locally and is transferred over the network *on demand*. The separation of semantic information and local context data demands for distributed query processing which is handled by the query engine integrated in the CS. The query engine processes a query in two steps:

1) *Data aggregation*: a temporary model is prepared prior to query evaluation. This model shares domain knowledge from the CS and includes context data fetched from distributed nodes. A subquery that is derived from a query first selects references to resources whose context should be resolved and included in the model.
2) *Query evaluation*: with all relevant context data aggregated in a temporary model the query can now be evaluated against this model.

Listing 1 shows a SPARQL query that asks for the names of the services running on a device. We extended the SPARQL syntax with a 'RESOLVE' keyword that denotes a subquery used for context aggregation. Since the CS only contains references to resources, we first need to acquire the context data of resources included in the query in order to evaluate it. The resource references returned by the RESOLVE subquery are selected based on their type. For each reference, the RESOLVE query looks up its location in the CS, fetches its context and merges it in a temporary model. Finally, the names of the services are selected from the aggregated context data.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://edm.org/environment#>
SELECT ?n
WHERE {?s :name ?n . ?s :runsOn <:adevice>}
RESOLVE {
    SELECT ?r WHERE {?r :refType ?t . ?t rfds:subClassOf :Service}
}
```

Listing 1. SPARQL query with subquery for aggregating context data.

B. Developing context-aware applications

We integrated the environment model discussed previously in an OSGi-based middleware platform for pervasive computing environments, called ReWiRe [12]. Application logic and user interfaces are embedded in OSGi components which are distributed amongst heterogeneous computing devices running the ReWiRe client software and are dynamically deployed into

the ReWiRe runtime. The runtime leverages the environment model in several ways to support the development and deployment of context-aware applications:

1) *Import and query ontologies:* When an application is deployed, it imports the domain ontologies it depends on in the CS. Instances of these (aggregated) ontologies define the current context of use of the application. Hence both the semantics of the environment topology and the context of use can change over time.
2) *Integrate and share resources:* Sharing resources is considered a default strategy in a pervasive computing system [6]. The properties of a shared resource adhere to an OWL concept and are stored in an object on the local computing device. Only a reference to the resource is published in the CS through which the resource's properties can be resolved on an as-needed basis.
3) *Subscribe to sensors and trigger sensors:* Sensors include information about context changes and are described in domain ontologies. They are transported over the network in a special type of message to which applications can subscribe. Since sensors are related to resources, pervasive applications actually subscribe to $\{resource, sensor\}$ pairs, denoted as sensor events, such as for example $\{*, OnlineSensor\}$ or $\{MediaService_1, *\}$ where '$*$' matches any resource or sensor respectively.

V. Observable Pervasive Applications

We call a pervasive application 'observable' if its current state and behavior can be monitored and optionally (re)configured while the application is in use. Application-specific user interfaces can help to make an application observable, but this will not be sufficient as a developer can not take into account all kinds of resources present in a pervasive environment that might interact with an application in useful ways. Besides, the behavior of an application is influenced both by end-users and the computing system:

- *User-driven behavior:* interacting with an application's user interface (e.g. pressing a button) triggers several actions that can result in context changes which on their turn give rise to new actions and so on.
- *System-driven behavior:* when the pervasive computing system is programmed to react on events, it will automatically invoke actions which also might trigger new events. Since events can occur without the user even interacting with the computing environment (e.g. when a sensor value is updated), it can be very confusing to understand why the system behaves in a certain way [2].

A typical pervasive environment is characterized by a combintation of user-driven and system-driven behavior: users interact with the environment while they are assisted by the computing system. By modeling an application's behavior, we can reuse tools for analyzing and configuring behavior and hence observe *any* pervasive application.

A. Behavior Model

As an extension to the environment model discussed in section IV-A, we use a separate model that captures the programmed behavior of an application. The behavior model is built up from behavior rules which are described in a behavior ontology that is linked with the upper environment ontology, as depicted in figure 3. A behavior event corresponds to a $\{resource, sensor\}$ pair, and conditions and actions refer to functions defined in a script on a remote device. In order to guarantee that the system can recover from mistakes, we have extended the concept of Event-Condition-Action (ECA) rules with inverse actions, denoted as $ECAA^{-1}$ rules. An inverse action allows users to return to a former state by undoing the rule's action that caused the unwanted state. This requires rules to cache information about the current context of use. For example, an action that dims a light resource to 30% of its default intensity should store the previous state of the light in order to allow the intensity to be reversed to its previous state. Therefore $ECAA^{-1}$ rules store context information about an executed action which is interpreted by an inverse action when a user chooses to undo the action.

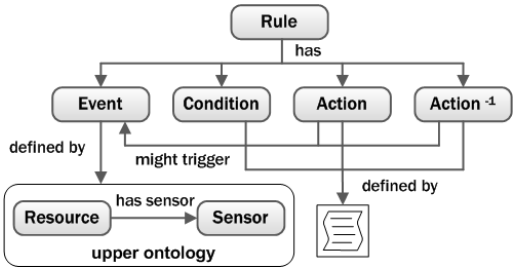


Fig. 3. The behavior of an application is described using $ECAA^{-1}$ rules.

The advantage of storing behavior rules in a semantic model particularly lays in its querying facilities: semantic queries are a powerful instrument to observe the runtime behavior of applications. A query allows to:

- *predict what will happen:* a query can ask which rules will be executed when a sensor is triggered. By evaluating the conditions of these rules, the pervasive software system can estimate which actions will be executed. Although the state of the environment can change and the rule's condition can evaluate differently when the event is actually triggered, the estimation will give the user more insight in what might happen. To make more accurate assumptions, we annotate behavior rules with extra information about the events they might trigger.
- *understand why something happened and undo it:* if a rule is executed, this is because an event was triggered. The sensor and the resource that caused the event can be traced; a description of the rule's condition and action helps to understand why the rule was executed and what it did exactly. If an event happens directly after an action is executed that is marked to trigger this type event, the event

is probably related with the action. Moreover, the rule's action can be reversed by executing its inverse action.

B. Configuring the behavior of applications

Configuring a pervasive application is a complex task, especially because the execution of an application depends on the context of use. Configuration goes beyond editing the properties of an application: developers also need to specify how an application must react on changes in the environment configuration, such as a new service coming online or an interaction device going offline. To keep end-users in control of their environment, it is important they can adapt the behavior of applications to their own preferences, also denoted as end-user programming [5]. We classify approaches towards end-user programming of pervasive environments in three major categories, ranging from a low-level approach to higher-level approaches:

- *Behavior scripts:* Scripts can be edited and executed at runtime which makes them very suitable for programming the runtime behavior of applications and thus to add rules to the behavior model. Although scripts are targeted at (amateur) developers, end-users can still enable or disable them at runtime, giving them limited control over what will (not) happen when the state of the environment changes.
- *Visual programming languages (VPLs):* VPLs mask programming code with visual constructs and help non- or less technical users to program an application. Although VPLs have a broader target public than scripts, experience learns that it is sometimes more complex to visually model a rule using a generic VPL than to write a few lines of code in a script. Yet, VPLs targeted at a specific user group and domain can enhance the experience of configuring the behavior of applications. Consider for example the Kodu project, where children can create a game using an icon-based programming interface [1].
- *Application-specific configuration interfaces:* Dedicated user interfaces for configuring an application can add or remove behavior rules in the background while the user interacts with the interface. Consider for example a configuration interface for a thermostat that allows its users to specify temperature offsets and timings to (de)activate the central heating. An advantage of modeling the thermostat's configuration by means of behavior rules is the ability to debug the thermostat's behavior using generic tools. Moreover, rules provide the option to undo actions or can be disabled. However, high-level user interfaces for configuring an application are limited to pre-defined behavior scenario's that were considered at design time and thus are less powerful than scripts or VPLs.

To support end-user programming and debugging of the environment's behavior at runtime, we have integrated the suggested behavior model in ReWiRe. This model is also dispersed over heterogeneous computing nodes, just like the environment model. In a Rule Store (RS) that shares information with the Context Store (CS), rule instances are stored that rely on conditions and actions defined in scripts on computing devices. ReWiRe clients – computing devices that run the ReWiRe client software – support behavior scripts through the Rhino JavaScript engine[4]. VPLs and application-specific user interfaces that add rules to the RS could be integrated through additional OSGi components. The integrated behavior model supports user-driven and system-driven behavior as follows:

- *Log user actions:* When application user interfaces are programmed to log the actions behind widgets that have an impact on the environment context, user-driven behavior becomes traceable and even reversible. To accomplish this, user actions are advertised as behavior rule instances with a special type of event (i.e. a $\{User_x, UserActionSensor\}$ pair) and optionally include an inverse action.
- *Evaluate conditions and execute actions:* Conditions and actions are implemented as functions in JavaScript code. Via the Rhino framework, scripts can access the environment model and application logic. While a condition returns true or false, an action returns a context record, i.e. a number of properties which reflect the state of the resource before it is altered by the action.
- *Execute rules:* A rule engine integrated in the RS subscribes to $\{resource, sensor\}$ pairs (events) just like any other application. Rules that match the event are executed and give rise to system-driven behaviors. If the rule engine receives an undo request for an action, it looks up the related rule, fetches the context of the action and passes it to the rule's inverse action (if any). Note that when an action is reversed, the computing environment will return to an old state and changes that have been performed after the action was executed, will be lost.

VI. Architecture and meta-user interface

The architecture of our system consists of a single host platform and several client platforms, both developped in Java using the Felix OSGi framework[5] and Jena[6] for processing context data. The host platform 'serves' the pervasive computing environment and is discovered by the client platforms. It acts as an access point for advertising new resources, querying the environment context and storing behavior rules. The client platform runs on heterogeneous computing devices and provides a runtime for pervasive applications. Applications are integrated as OSGi components and are composed of services, user interfaces and other resources. They exchange messages with remote applications through a message gateway integrated in the host platform. The environment and behavior models discussed in sections IV-A and V-A are distributed amongst host and client platforms. Ontologies describing the semantics of the environment and references to resources are stored in the context store at the host platform while the actual state of a resource is stored in data objects maintained by an application at the client platform. Likewise, behavior rules are

[4]http://www.mozilla.org/rhino/
[5]http://felix.apache.org/
[6]http://jena.sourceforge.net/

stacked in the rule store at the host platform and refer to scripts that are executed at a client platform, from where they can access an application's data objects.

To interact with a pervasive computing environment, users first need to sign in to the environment. This advertises the presence of the user in the environment and relates one with an interaction device. Once signed in, users are presented with a *meta-user interface* that provides an overview of the resources available in the environment and the tasks these resources support as defined by the available applications. Besides, the meta-user interface features several tools for debugging and configuring the behavior of applications (see figure 4). We call this interface 'meta' because it serves as a generic user interface from where end-users can both interact with the environment and configure its behavior.

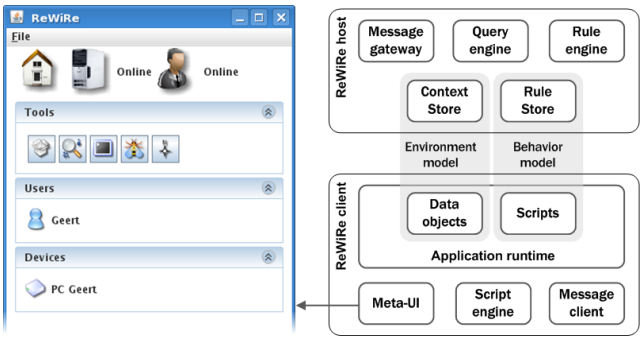
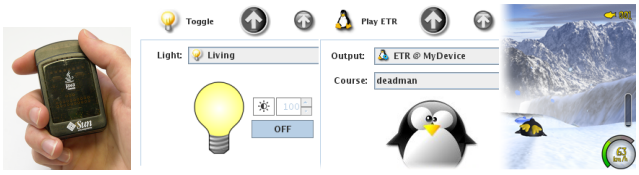


Fig. 4. Environment and behavior models span a host platform and client platforms installed on heterogenous devices. A meta-user interface integrates debug tools for pervasive applications such as those listed in figure 1.

VII. Case Study

In order to evaluate the presented models and tools, we developed two 'SPOT applications', shown in figure 5, that connect pervasive services with SunSPOT devices[7] on the fly. A SunSPOT is an embedded computing device with a number of built-in sensors such as temperature and light sensors and 3D accelerometers. SunSPOTs communicate wirelessly with a base station that is connected over USB with a desktop computer. We attached the base station to a notebook on which we installed the ReWiRe client platform. A Java MIDLet running on a SunSPOT collects sensor readings on this device and sends them to a SunSPOT service encapsulated in an OSGi component on the client platform. The SunSPOT service then transforms the received information into ontology-compliant sensor events such as $\{SunSPOT_1, TiltSensor\}$. From this point, we can leverage the environment and behavior models to connect the events produced by a SunSPOT with actions that control a pervasive application. We designed a prototype application for automating the lights in an environment and used a SunSPOT as input device to play a game. Both applications import ontologies describing their application domain, integrate the necessary resources in the environment and subscribe to SunSPOT-specific sensors. We show that these applications are observable at runtime, through the messages they exchange and the models they adopt.



(a) SunSPOT. (b) SPOT Lights and SPOT Racer applications.

Fig. 5. A SunSPOT device and a number of services are assembled at runtime into pervasive applications whose behavior can be observed and configured.

A. SPOT Lights

The SPOT Lights application integrates a light service that embeds application logic to turn on/off lights in the pervasive computing environment, a number of virtual light resources and a user interface for operating lights. In the application's user interface, a light can be selected and turned on/off. The user interface logs user actions that alter the state of a light and links them with events that might be triggered. Furthermore, a small piece of JavaScript code injects two $ECAA^{-1}$ rules in the behavior model that connect a SunSPOT device with a light resource. These rules automatically turn on/off the light if the light sensor readings of the SunSPOT drop below or rise above a predefined value. The script also defines the conditions and actions that are executed by the rule engine when a $\{SunSPOT_1, LightSensor\}$ event is triggered as illustrated in figure 6. Since the SPOT light application is dynamically composed of various resources (a service, a user interface, a behavior script) traditional debug tools based on code inspection are unable to explain the application's behavior. However, using the tools integrated in the meta-user interface discussed before, the application can be debugged in several ways at different layers:

- Query the environment model for light resources and services to verify that the application initiated correctly.
- Query the behavior model to check which rules can have an impact on light resources and which events can cause these rule to execute.
- Monitor the events that are triggered by the SunSPOT device when the light sensor value changes and inspect the network messages that caused these events. The network message provides additional information about the properties (e.g. sensor values) of an event.
- Request explanations about why an event occurred, e.g. did the user manually switch on the light or was it turned on automatically because a SunSPOT device sensed darkness?
- Request a user interface to reconfigure a light that was set to a faulty state or to undo an action that caused this effect.
- Step into the rule engine's flow of execution and pause it at runtime. This allows one to play with the SunSPOT and learn which actions will be executed when SunSPOT events are fired.

[7]http://www.sunspotworld.com/

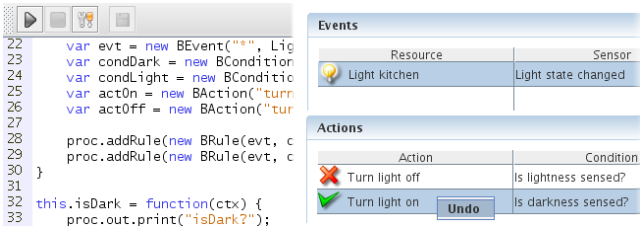


Fig. 6. A script connects a SunSPOT device with a light resource through behavior rules. The events that are fired and the actions that are executed by the rule engine as a reaction to these events can be monitored and undone.

Even though SPOT Lights is not a complex application, it illustrates that runtime debug tools are useful instruments to deal with the complexities introduced by the pervasive computing environment it operates in.

B. SPOT Racer

The SPOT Racer application illustrates that legacy applications can be integrated in a pervasive setting as well using our framework and that they can be controlled by input devices they were not designed for at first. We created a Tux Racer service that passes input events it receives over the network to a game process. Tilting the SunSPOT steers and (de)accelerates Tux while he slides down a ramp full of snow and herring. The tilt events generated by the SunSPOT are connected to behavior rules that map tilt information, via a script, to navigation commands defined in a Tux Racer service. Similar to the SPOT Lights application, the execution of the game can be monitored at runtime. Although it is not the most efficient approach to encode real-time interaction using behavior scripts/rules, it can be used to prototype the game. As expected, some lag was noticeable on a dual core 1.8GHz notebook with 2GB of memory, but it did not prevent us from playing Tux Racer smoothly using a SunSPOT device.

VIII. Conclusions

We have presented a framework for developing and deploying *observable pervasive applications*. The major contribution of this framework is a combination of decentralized models that capture the context of use and describe the behavior of applications. Using this framework we have successfully designed two prototype applications that can be observed at runtime by means of the messages they exchange, the properties they manipulate and the rules they define. Moreover, we have illustrated that the pervasive computing environment can recover from mistakes caused by system-driven behavior. For example, if a light was turned off without the user wanting this to happen, the user can still intervene and manually correct unwanted behavior through a meta-user interface for the pervasive environment.

There is still room for various improvements. For example, we currently do not provide an approriate solution to avoid cycles that can exist between behavior rules: when a rule's action triggers an event that causes the rule to execute again, rules are executed in an infinite loop. Though we partially address this issue by annotating actions with information about the events they *might* trigger, we can only predict cycles without certainty. Shankar et al illustrate how pre- and post-conditions can help to resolve this problem [11]. Furthermore, we have focussed on tools for developers for now, but non-programmers can benefit from runtime tools as well to get insight in the state of the system and to alter it. We plan to conduct a user-study to point out the strengths and weaknesses of our approach for a less technical audience. We believe an improvement for end-users would be an interface that allows to ask more natural questions about the environment such as 'Why are the lights on?'.

References

[1] *Kodu*. http://research.microsoft.com/en-us/projects/kodu/.

[2] Victoria Bellotti and Keith Edwards. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human-Computer Interaction*, 16(2):193–212, 2001.

[3] Joëlle Coutaz. Meta-User Interfaces for Ambient Spaces. In *Task Models and Diagrams for Users Interface Design (TAMODIA'07)*, pages 1–15, 2007.

[4] Anind K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[5] Krzysztof Gajos, Harold Fox, and Howard Shrobe. End User Empowerment in Human Centered Pervasive Computing. In *Proceedings of the 1st International Conference on Pervasive Computing (Pervasive'02)*, pages 1–7, 2002.

[6] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[7] Tao Gu, Hung Keng Pung, and Da Qing Zhang. Toward an OSGi-Based Infrastructure for Context-Aware Applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004.

[8] Deepali Khushraj, Ora Lassila, and Tim Finin. sTuples: Semantic Tuple Spaces. In *Proceedings of the 1st International Conference on Mobile and Ubiquitous Systems (MobiQuitous'04)*, pages 267–277, 2004.

[9] Brad A. Myers, David A. Weitzman, Andrew J. Ko, and Duen H. Chau. Answering Why and Why Not Questions in User Interfaces. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI'06)*, pages 397–406. ACM, 2006.

[10] Mark W. Newman, Shahram Izadi, W. Keith Edwards, Jana Z. Sedivy, and Trevor F. Smith. User interfaces when and where they are needed: an infrastructure for recombinant computing. In *Proceedings of the 15th annual ACM Symposium on User Interface Software and Technology (UIST'02)*, pages 171–180. ACM, 2002.

[11] Chetan Shankar and Roy Campbell. A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule-Actions. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA'05)*, pages 255–258. IEEE Computer Society, 2005.

[12] Geert Vanderhulst, Kris Luyten, and Karin Coninx. ReWiRe: Creating Interactive Pervasive Systems that cope with Changing Environments by Rewiring. In *Proceedings of the 4th IET International Conference on Intelligent Environments (IE'08)*, pages 1–8, 2008.

[13] Upkar Varshney. Pervasive Healthcare and Wireless Health Monitoring. *Mob. Netw. Appl.*, 12(2-3):113–127, 2007.

[14] Torben Weis, Mirko Knoll, Andreas Ulbrich, Gero Muhl, and Alexander Brandle. Rapid Prototyping for Pervasive Applications. *IEEE Pervasive Computing*, 6(2):76–84, 2007.

[15] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.