ReWiRe: Creating Interactive Pervasive Systems that cope with Changing Environments by Rewiring

Geert Vanderhulst Kris Luyten Karin Coninx

Hasselt University – transnationale Universiteit Limburg – IBBT
Expertise Centre for Digital Media
Wetenschapspark 2, 3590 Diepenbeek, Belgium
Email: {geert.vanderhulst,kris.luyten,karin.coninx}@uhasselt.be

Abstract

The increasing complexity of pervasive computing environments puts the current software development methods to the test. There is a large variation in different types of hardware that need to be addressed. Besides, there is no guarantee the environment does not evolve, making the software developed for the initial environment deprecated and in need for updates or reconfiguration. Software deployed in such an environment should be sufficiently dynamic to cope with new environment configurations, even while the system is in use. This goes beyond coping with new contexts of use and building context-aware systems: while most approaches are mainly focused on how the software behavior adapts according to the changing context in a fixed environment, our approach, *ReWiRe*, allows the environment configuration to change over time.

1 Introduction

Most software systems still expect a static environment where the devices and services that are used to complete a task and the number of users that interact with the software do not change in one usage session. When the environment is allowed to change while users are interacting with it, this also means the end-users will be able to interact in different ways with the system that can only be determined while the software system is in use.

The *ReWiRe* framework presented in this paper introduces a system that can adapt itself when new configurations arise such as the usage of a new mobile device. We refer to this dynamic (re)configuration process with the term '(re)wiring'. Achieving collaborations by bringing together several users and devices is part of a (re)wiring process.

Similar problems are tackled in [12], [3] and [8] where a semantic layer for describing the context of use is exploited to support context-aware pervasive systems. Most attention in this work goes into supporting changes in an existing architecture, although dynamically composed. We seek to support systems that have to cope with a changing configuration during usage and where the system as well as the end-user can coordinate the effect of these changes.

Our approach links the environment configuration with the software architecture. This link is continuously maintained during the lifetime of a pervasive application. When the environment configuration changes, the software architecture changes accordingly. For this purpose we encode a pervasive computing system as a graph structure that describes how different entitities in the environment are related with each other. A graph corresponds to an ontology instance and allows us to *query*, *update* and *rewire* the system while it is being used.

Various standardized techniques are combined to realize our system. *ReWiRe* is built on OSGi [1] technology which provides a dynamic platform that is well suited to manage the cycle of software services at runtime. By adding a unified semantic layer to a set of software services using standardized languages and technologies, we gain flexibility without loosing generality. The underlying techniques make use of semantic Web technologies such as the Resource Description Framework (RDF), the Web Ontology Language (OWL), the SPARQL query language for RDF and XML events over HTTP. An overview and sound motivation for the use of semantic Web technologies in pervasive environments can be found in [2].

The contributions in this paper are threefold. First, we introduce a dynamic semantic environment model that describes a pervasive environment in terms of its users, devices, services and supported tasks. Second, we present a framework incorporating this model that enables a developer to create software that runs in a pervasive computing environment and that can be rewired at runtime to take advantage of changes in the environment configuration. Third, we propose pervasive menus and interactive wizards as end-user tools that allow to discover features as well as to control the behavior of the pervasive system.

The remainder of this paper is organized as follows: section 2 discusses the runtime environment model which is the core of our approach. Next, section 3 delves deeper into the system behavior that is accomplished by using the *ReWiRe* approach. Section 4 contains an overview of the architecture of the system and in section 5 we present a pervasive application that was built using our framework. Finally, section 7 concludes the paper and states some directions for future work.

2 A Runtime Environment Model

The cornerstone of our approach is the runtime environment model. It describes the full context of use of the interactive software system during its lifetime. Instead of mangling

this information in the program code or provide this information at deploy time by means of configuration files, we use an *ontology instance* that is linked with the software architecture. An ontology instance adheres to an ontology, which on its turn describes classes of objects, relationships between (classes of) objects and domain constraints on their properties [7]. The ontology instance provides us with the semantics required to create a system that can rewire during usage.
Figure 1 illustrates the upper environment ontology of *ReWiRe* (note that 'a' labels correspond to 'isa' relationships). The upper ontology serves as a starting point for the environment and provides a structure upon which ontologies for specific domains can be constructed. Whenever new resources become available in the environment, the model definition can be extended with domain-specific knowledge *at runtime*. This is accomplished by merging the ontology associated with the new resource with the ontology that describes the current state of the system. We consider the environment as an autonomous part of the system that keeps track of all resources and entities that make use of the system. Note that an "environment" is not physically constrained (e.g. limited in space). We distinguish the

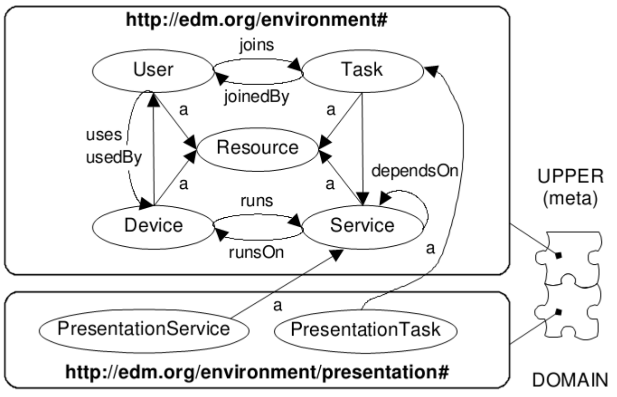


Figure 1: The environment is described in a set of domain-specific ontologies.

following generic concepts in the upper ontology:

- **User**: A user represents a human interactor. Each user is related with one or more devices that act as a tool to interact with the surroundings.
- **Device**: Devices offer computing power, storage, input and output modalities, etc.
- **Service**: A service offers a set of features which are exposed in a clean API and that can be (re)used by other services.
- **Task**: A *user* task represents a goal a user can accomplish and a *system* task specifies how other entities (users, devices, ...) can collaborate to reach a goal. Section 3.1 provides a more detailed discussion of the use of tasks in *ReWiRe*.

A change in an environment configuration can be one of the following:

$D+$: A mobile device enters the environment and can now be used to support the end-user tasks;

$D-$: A mobile device that was being used to support the end-users tasks leaves the environment;

$U+$: A new user enters and interacts with the system using the devices available in the environment;

$U-$: A user leaves and stops interacting with the software;

$S+$: ($T+$) A service and/or task becomes available;

$S-$: ($T-$) A service and/or task becomes unavailable.

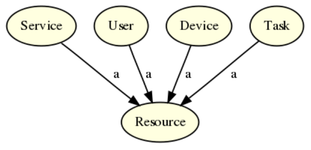
Figure 2 shows how a graph presentation of an ontology instance evolves over time. For example, between the graphs presented in figure 2(a) and figure 2(b), the following changes in the environment occurred: $D+$, $U+$, $S+$ and $T+$. The relations between the ontology instances show that the new device (*Device_PC*) is used by the new user (*User_Geert*) and that this device runs a software service (*PresentationService_I*) that is used in a task (*PresentationTask_I*); the latter is joined by the user.

3 Runtime Behavior

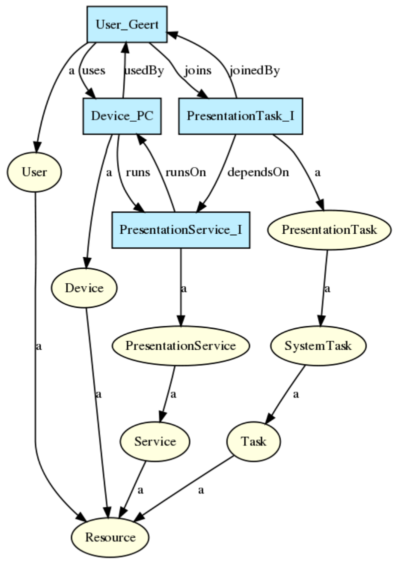
The relations that apply between resources in the environment are synchronized between digital and physical worlds. On the one hand, a user can create new relations by interacting with the surroundings using the available devices. On the other hand the system itself can wire resources together to better support users with their goals. Figure 3 presents an interaction diagram that illustrates how entities behave when *ReWiRe* initiates. The terms 'device agent' and 'environment agent' respectively refer to the *ReWiRe* software installed on end-user devices and the kernel of our system. Most of the initial actions are $D+$, $S+$, $T+$ and $U+$ actions, that populate the environment. Before a user can interact with the system using a personal device, a $D+$ and $U+$ action should have occurred and the user and device need to be related to each other. A device can discover the availability of a *ReWiRe* environment and register itself in this environment ($D+$) together with its services ($S+$) and related tasks ($T+$). By uploading her user profile on a registered device, a user is related with the device and also known in the environment ($U+$). Off course, users can also use shared equipment already present in an environment (e.g. whiteboard). In this case, often no user/device relations are established, unless a tracking mechanism or manual registration process is used. The *ReWiRe* system gracefully deals with the level of detail in the model: the more detail, the more intelligent the system can react on changes in the configuration.

3.1 Tasks and Goals

With these changes that can occur in an environment, we need to add information about which tasks can or cannot be done with the available resources. We distinguish between *user* and *system* tasks. A user task corresponds to a specific user goal for which a user interface is available (e.g.



(a) Initial environment model without users, devices, services or tasks.



(b) A user has entered the environment, uses a device that runs a software service and participates in a task.

Figure 2: Two different graphs representing the state of the system. Each graph is the model of the environment at a particular time during the usage of the system. Graph (a) shows an environment model that has no user, devices or services at all. Graph (b) shows the environment model with one new user ($U+$), device ($D+$), service ($S+$) and task ($T+$) with their relationships. The service is used in a task that is joined by the user.

play music, toggle lights, navigate slides, etc). Note that a user task could be presented by different types of user interfaces (graphical, speech, . . .) suited for a specific context of use and/or target device. Hence user interfaces that are distributed to devices actually 'present' some user task in the model. A system task, on the other hand, orchestrates the collaboration of different resources (users, devices, services, . . .) and monitors the environment context to rewire when needed or useful. The system task logic will try to allocate devices for specific user tasks, either automatically or assisted by the user as discussed in section 3.2.

3.2 Pervasive Menu and Wizards

As denoted in [4] the shifting from the control of systems and applications confined to a single workstation to that of a dynamic interactive space brings up the need to control, mould and understand an interactive environment. The concept of meta-UI outlined in [4] suggests tools to control and evaluate the state of an environment by its users. Our approach incorporates a *pervasive menu* and *interactive wizards* that serve as such tools and that are focused on end-users who should have no technical expertise. A pervasive menu is composed of tasks that are deployed in the entire environment rather than on a single computing device. In other words, it provides a view on what the user can do in a pervasive environment. The purpose of such menu is twofold. First, it promotes collaboration and user-driven resource allocation through tasks and interactive wizards. Second, it grants access to services through migratable user interfaces.

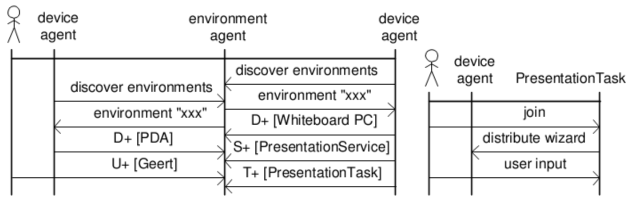


Figure 3: The interaction diagram shows how *ReWiRe* initiates. The most common actions during the initialization phase are adding new users, devices, services and tasks to a discovered environment. Next, wiring takes place to connect entities for a specific goal, e.g. a user joining a presentation task that has been deployed in the environment.

While system tasks can be considered as autonomous building blocks that orchestrate a pervasive computing environment, user input is still required to correctly deal with many non-trivial situations: it would not be realistic to expect the computer system to figure out what the user wants by itself. Hence *ReWiRe* is able to engage in a dialog with its users, i.e. ask questions, present choices, etc. This is achieved by means of *wizards* that are distributed to the end-user's personal device whenever user input is required. Previous research on mixed-initiave user interfaces already suggests agent entities that ask users about their goals and needs instead of guessing these goals [9]. [9] also presents a method to decide whether or not the system should act autonomously or ask for input. We distinguish different stages when input is required. First, when creating a new task instance, the user might manually indicate resources to allocate or walk through task-specific configuration steps. Second, when a task is joined by a user, the task might ask for the user's preferred role in the task (e.g. passive attendee or presenter in a meeting context) which will have an effect on the set of user interfaces that are to be distributed. A newly announced resource might also improve the performance of the task at hand. The input that is collected from the user on such occasions will result in new relationships that are created in the environment model or existing relations that are changed.

Figure 4 depicts the structure of a prototype menu which is generated at runtime from information in the environment model. The prototype is rendered as (X)HTML markup so almost any contemporary device can display it. The menu lists the set of tasks that are currently deployed in the environment and allows to create a new instance of a task or join/leave an existing one. These actions trigger a wizard by which the task is (re)configured. Note that the pervasive menu is dynamic as it evolves with the environment model and also personalized since the user context for instance determines if an actor can join or leave a task.

3.3 Rewiring

We use rewiring as an indication of changes that occur in the environment model. When edges are added or removed from the environment graph, a notification is sent to interested parties. In particular, system task entities will subscribe to these events since they coordinate the runtime behavior and changes in the environment configuration might affect this behavior. For example, the disappearance of a device from the environment model ($D-$) without a user logging out of the system, means the user interface that was presented on the device has to be migrated to another device. When the model changes, a task will verify whether it can still support its goal or if reconfiguration is needed. The combination of push (sending a notification) and pull (querying the model for changes) helps to smoothen the rewring process.
In practice, rewiring at runtime can increase the cognitive load of using the pervasive computing environment and result in disorientation of the user. A rewire operation causes *task interruption*, an event that causes interruption of a process that is meant to be a continuous flow of interaction [11]. *ReWiRe* uses *rewiring strategies*: patterns of behavior that have proved to be useful to guide the user from one situation into another situation while minimizing the extra cognitive load of a task interruption. Based on the layered reference model proposed by Massink and Faconti in [11], rewiring patterns can be defined on several levels: the (5) group level, which specifies cooperation in cooperative applications, the (4) task/conceptual level, (3) Propositional level, (2) Perceptual level, that deal with information processing, and the (1) Physical level, where the physical interaction takes place. *ReWiRe* currently operates on different levels: it combines patterns on the task/conceptual level with manual control to smoothen transitions: an intelligent wizard is used to guide the user through a rewiring operation. Section 3.2 already mentioned the use of wizards in the context of creating and joining task instances. Further research is still needed to create a mature set of rewiring strategies to deal with the complexity of the target environments.

4 Architecture and Design

4.1 Basic Architecture

The main architecture of *ReWiRe* uses the OSGi framework as a modular platform to deploy components. OSGi components, also referred to as bundles, are software components implemented in Java that can be remotely installed, started, stopped and updated, even when the system is active. The suitability of OSGi as a foundation to create pervasive computing environments has been discussed in several other papers, e.g. [10, 14]. The OSGi framework already offers several services useful for dynamic computing environments such as life cycle management and loosely-coupled service-oriented components. We currently use the Apache Felix OSGi framework[1] which is a very stable implementation of the latest OSGi specification.
Figure 5 describes the architecture of *ReWiRe* in terms of execution environment and basic services. We distinguish

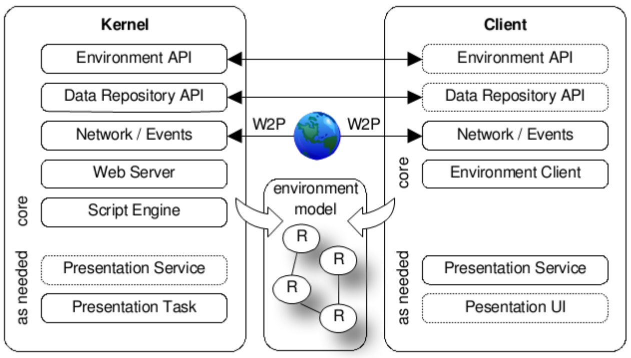


Figure 5: Overall architecture of *ReWiRe*. The kernel and a client are interconnected by a W2P network and can access a shared environment model.

a kernel and a client side that differ in terms of deployed core OSGi components. The kernel serves an environment that is discovered by *ReWiRe* clients that are installed on all devices that are used with our system. Clients and kernel are interconnected through a shared network layer and can transparently access the pervasive environment through higher level APIs that are built on the network layer. A data layer provides access to a semantic data repository (Jena[2] in our implementation), whereas the environment layer exposes an API to interact with the environment model and deploy wizards, services, user interfaces, etc.
The model is queried using the SPARQL query language; queries are processed in the data layer. The binding with an XML-based results format[3] renders SPARQL a good candidate for language-neutral processing of query results. Each concept in the environment ontology is mapped on a Java class equivalent and each resource is identified by a URL by which its context can be acquired and updated. For example, an existing user context can easily be retrieved as follows:

```
User u = new User('http://edm.org/environment#User_Geert');
u.update();
```

Note that the User class is derived from Resource and thus matches the ontology structure. Since the context of a resource is stored in the model and the API provides synchronized access to the model, resources can be accessed from anywhere a connection with the model can be established.
Although kernel and client expose the same API in the data and environment layer, the kernel actually hosts the layer logic while the clients have a proxy component installed that interoperates with this logic (proxy components are indicated by a dotted line in figure 5). An embedded Web

[1]http://felix.apache.org/
[2]http://jena.sourceforge.net/
[3]http://www.w3.org/TR/rdf-sparql-XMLres/

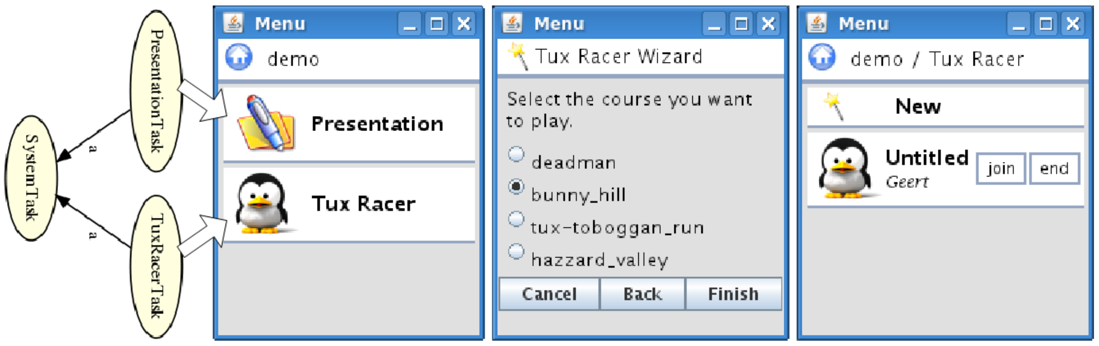


Figure 4: A pervasive menu generated from the environment model is rendered as a Web interface. Users can create, join and leave task instances and (re)configure them at runtime using interactive wizards.

server in the kernel is used by task entities to share resources and serve wizards when user input is required. To alter the behavior and configuration of the system at runtime, we integrated a script engine in the kernel that can execute scripts when the system is in use. We elaborate on this in section 4.4. Apart from some core services, kernel and clients exploit the OSGi framework to dynamically install components on an as needed basis. While service components are distributed amongst clients, task components are injected in the kernel.

4.2 Communication

While simple ad-hoc communication protocols might simplify the implementation of pervasive applications, the typical non-standard protocols used in these applications make it harder to integrate a pervasive system with other distributed systems (e.g. Web applications) [6]. Most of these communication protocols are specific for a platform or even for an application domain. To keep the development of applications simple yet avoid ad-hoc protocols, our architecture makes use of a light-weight cross-platform communication library called W2P [16] that works on top of the HTTP protocol; it is integrated in the network layer shown in figure 5.
W2P facilitates both synchronous and asynchronous message exchange between loosely coupled distributed components and with its built-in publish and subscribe mechanism, the framework also accommodates a solution to propagate events. In our implementation, events are for instance used to notify a subscriber of a sensor that has been triggered. Some specific applications could require other types of communication protocols, e.g. for media streaming, which can be provided in an OSGi bundle and dynamically deployed in our system when needed (see figure 5).

4.3 Sensors and Groundings

Since the set of services deployed in the environment is not static, a mechanism is needed to bind to services at runtime, i.e. invoke their functions, exchange context data or present a user interface to interact with service logic. Dynamic service binding is commonly used to interact with Web services; various description languages (OWL-S, WSDL, . . .) have been defined to automate Web services interoperation. Inspired by these languages, we extended the upper environment ontology with *sensors* and *groundings* that provide the semantic glue to bind to services at runtime. Figure 6 shows that the meta concepts *Grounding* and *Sensor* are integrated with the upper ontology and further concretizised in specific ontologies. The *Grounding* concept serves as a base for concrete grounding ontologies that hook up services with invocation logic. One such concrete grounding, an OSGi grounding ontology, provides the semantic glue to relate services with OSGi bundles. This OSGi grounding relates a service with a *proxy* bundle which exports a software interface to remotely interact with it. Besides, a *UI* bundle (which depends on one or more *proxy* bundles) provides *ReWiRe* clients with a user interface that presents a user task. When installed, these bundles export factories that are used to load a proxy object or user interface from a service or user task URL respectively; the service or task context is retrieved through this URL from the environment model. Note that to dynamically bind to a Web service, an interactor should be able to generate invocation code from a description document, whereas the OSGi grounding assumes an OSGi platform where the service invocation logic (packed in an OSGi bundle) is directly injected. This plug and play approach greatly simplifies the service binding process in our architecture. Nevertheless, alternative groundings can be defined to integrate OSGi-less platforms. For example, an AJAX grounding might be useful for pervasive Web applications that run in a Web browser. To interact with a service, a software component has to lookup a compatible grounding in the model and install it. Likewise, to execute a user task, a *ReWiRe* client will lookup a suitable user interface in the model and render it on the target device.

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 provides remote context events to interested entities in the environment. For example, an *RFIDScanSensor* describes a sensor that is triggered when an RFID tag is scanned by an RFID service. The structure of the sensor data is described in the ontology (not shown in the figure), so services that are subscribed to the sensor can easily consume it.

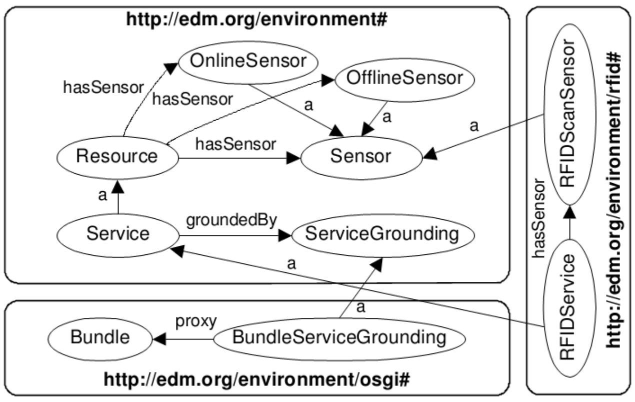


Figure 6: The semantic glue to link model and software is described in an extended upper environment ontology.

4.4 Behavior scripts

With a platform for software services and a protocol to allow these services to communicate in an heterogeneous environment a complete functioning interactive distributed system can be created. When the system configuration would change over time, for example because of a device entry ($D+$) that causes several $S+$ actions because the device carries several services, tasks can start a (re)wiring process. Nevertheless, more flexibility might be required to adapt the behavior of the system to specific contexts of use, e.g. to dynamically attach a behavior to RFID tags or to play a sound when a new resource becomes available. We support these dynamic behaviors by means of scripts, i.e. small pieces of code that are interpreted at runtime. A script is linked with a sensor and executed when a context event is fired through this sensor. We integrated the Rhino scripting engine[4] in *ReWiRe* which allows developers to define behavior using JavaScript code.

5 Implementation Case

5.1 Development Process

Before looking at a specific implementation case, we give a global overview of the steps a designer has to traverse to create a pervasive application using the *ReWiRe* framework. We distinguish between modeling and engineering steps:

Modeling Design an ontology for the application domain. This can be done with the Protégé [5] ontology editor in which the upper environment ontology and some other domain-specific ontologies can be imported and extended. In this step the developer should identify services, tasks, relations between resources, the sensors and groundings that are needed for the application, etc. The ontology is the application's skeleton and foundation for its design.

Engineering Implement the services, (system) tasks and user interfaces that have been identified during the modeling phase. To adhere to the OSGi grounding (see section 4.3), each service implementation (i.e. OSGi bundle) should have its own proxy bundle that allows to access the service logic from any device in the environment. Software interfaces and common classes can be shared between logic and proxy components and W2P can be used to handle the communication. When engineering a system task, the environment API can be used to subscribe to sensors, query resources, distribute user interfaces, deploy wizards, etc, in order to coordinate services and devices to support the user(s) in reaching the envisioned goal(s). Implementing a system task can also involve creating wizards to collect user input. The embedded Web server component in the *ReWiRe* kernel can be used to serve wizards in plain HTML or using servlets. Finally, user interfaces that present a user task are grounded with an implementation which could also be a (distributable) OSGi bundle.

5.2 Ubiquitous Tux

Tux Racer is an open source arcade game. In the game, the player controls Tux as he slides down a course of snow and ice collecting herring. The player can choose to control Tux either by keyboard or joystick. Various forks of the game exist and we bundled one of them, PlanetPenguinRacer[5], with our framework to create a ubiquitous game experience. The goal of this demo setup is to allow gamers to play the game against each other and compete using the equipment that is available. For our experiment we used a room that is equipped with a tabletop display, a PC with projector and each gamer also has a PDA that has the *ReWiRe* initialization client software pre-installed. The gamer can register herself in the environment using her PDA. Unlike the original game, our pervasive variant scales to an heterogeneous environment and encourages collaboration. The game is configured at runtime by a task that has been deployed on the *ReWiRe* platform and that monitors changes in the environment model and (re)wires whenever necessary to continuously guarantee an optimal game experience. By querying the model the system can easily discover the devices on which the native game software, encapsulated in a Tux Racer service, is installed. The SPARQL query in listing 1 illustrates this and will return the digital table device and PC with projector in our setup. Notice how the upper ontology and a domain-specific ontology are seamlessly accessed in the query.

```
PREFIX e: <http://edm.org/environment#>
PREFIX tux: <http://edm.org/environment/tux#>
SELECT ?d
WHERE ?d a e:Device . ?d e:runs ?s . ?s a tux:TuxRacerService
```

Listing 1: SPARQL query that asks for those devices that run a Tux Racer service.

Once the system is initialized, users can walk towards the projector screen or tabletop display and scan an RFID tag using their PDA device (equipped with an RFID reader) to participate in a game at that location. The RFID scan triggers a sensor which results in the execution of a script that connects the user with a game task. A wizard, that allows

[4]http://www.mozilla.org/rhino/

[5]http://planetpenguinracer.com

the gamer to select a course and start a new game or join one and optionally indicate her preferred input device to control Tux, is deployed on the PDA. An input task for the game service provides a user interface that is distributed to a selected input device. The rendering of the game is automatically adapted to the target device, e.g. on the digital table the game is rendered in a mirrored split-screen mode, thus orientend towards gamers sitting on both sides of the table as shown in figure 7. The graph in figure 8 illustrates how connections between users, devices, the game service and game tasks are reflected in the environment model. The player's name is automatically derived from the model and displayed during a race.

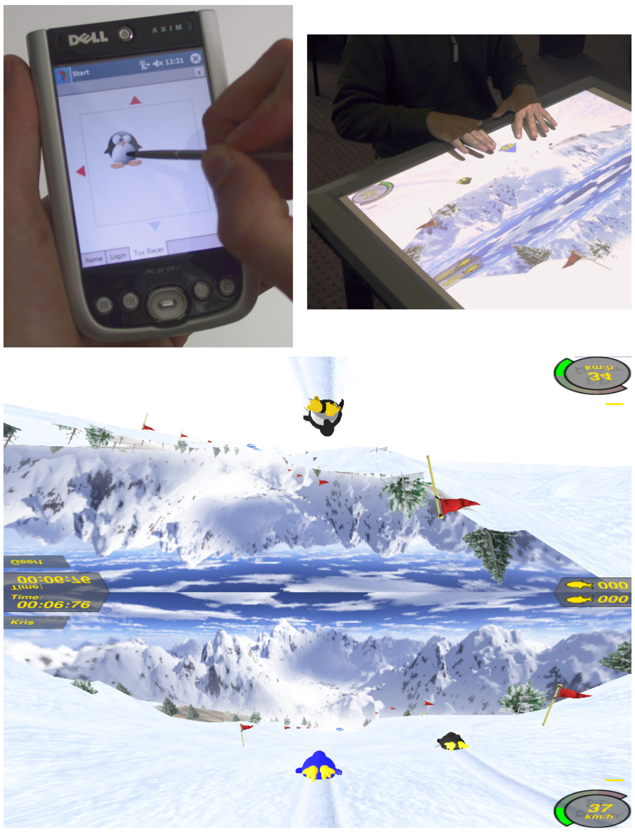


Figure 7: Screenshot of a pervasive Tux Racer game when running in mirrored split screen mode on a tabletop display. One user uses a PDA device to control Tux, while the other one uses the table itself as input device.

The experiment shows the *ReWiRe* architecture offers a solid foundation to integrate existing applications in a pervasive computing environment. Users particularly appreciated the initiative of the system: most of the configuration steps are handled in the background, but when needed the system engages in a dialogue with the user, e.g. to ask for the course that she wants to play.

6 Related Work

The use of an ontology to describe the context of use has been researched before. For example, in [8] and [13] ontologies are proposed for ambient intelligent environments.

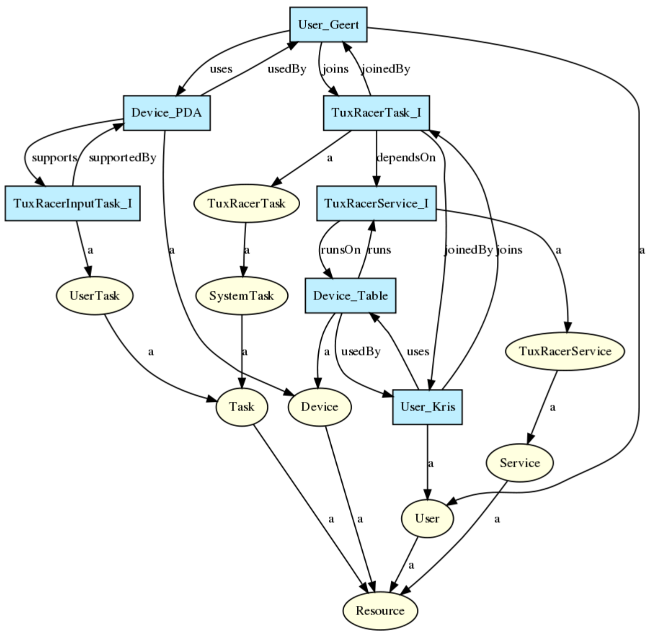


Figure 8: The graph that gives rise to the game setup depicted in figure 7.

These ontologies are mainly geared towards the creation and deployment of context-aware systems. The Context Ontology Language (CoOl, [15]) is another approach that defines the context of an application by using the aspect-scale context model. This means each aspect has one or more scales to express context information. SOUPA [3] provides a richer ontology created for ubiquitous and pervasive applications. It is expressed in OWL and combines useful existing vocabularies in a shared ontology that can be used by a broker-centric agent architecture. In *ReWiRe* light-weight ontologies that describe an application's domain are merged and instantiated at runtime. This ensures we can keep queries simple for each context of use.
Previous work discusses the use of OSGi technology in a pervasive environment [10, 8, 14]. Gu et al. make use of the OSGi platform to support service delivery and provisioning, and deal with security and privacy issues that are crucial for a pervasive deployment of context-aware services and applications. In contrast with [14] we do not extend the OSGi framework itself to add distribution capabilities, but add semantics separate from the framework that describe how to interoperate with OSGi components or other software services. This way, we gain flexibility to integrate our design with all kinds of distributed systems.

7 Conclusions

We presented *ReWiRe*, a framework that supports networked heterogeneous environments by design: distribution of system components over the available devices is an integrated part of our approach, as is the support for multiple users and thus cooperative applications. The cornerstone of our approach, a semantic graph that describes the environment and that is linked to the pervasive computing system, provides us with a solid foundation to reason over the system properties and the relations between different entities that occur in the environment. We believe a main contribution of *ReWiRe* is its ability to deal with changes in the environment configuration while the system is in use. The runtime environment model, migratable services and user interfaces, orchestrating tasks, scripts, a pervasive menu and interactive wizards are all tools that help accomplish this. Since our framework is built on standardized technologies such as OSGi, semantic Web technologies and a uniform communication framework (W2P), *ReWiRe* enables developers to create pervasive applications using familiar technologies. We illustrated that *ReWiRe* can be used to integrate existing software in a pervasive environment.

We acknowledge various improvements can still be made. For example, further research is needed to improve rewiring strategies to mimize the cognitive load when the end-user has to cope with a changing environment. We also plan to (partially) automate the development of tasks, wizards (using some formal description language) and behavior scripts. Hence we are currently experimenting with a configuration tool to graphically edit the behavior of a pervasive system in a more user friendly way.

Additional information on *ReWiRe* can be found at `http://research.edm.uhasselt.be/rewire/`

Acknowledgments

Part of the research at EDM is funded by EFRO (European Fund for Regional Development) and the Flemish Government. Funding for this research was also provided by the Research Foundation – Flanders (F.W.O. Vlaanderen, project number G.0461.05).

The authors would like to thank Alexandre Demeure and Luc Claesen for their valuable input.

References

[1] OSGi Alliance. *OSGi Service Platform*. 2003.

[2] Harry Chen, Tim Finin, and Anupam Joshi. Semantic Web in the Context Broker Architecture. In *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PERCOM'04)*, pages 277–286, 2004.

[3] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *Proceedings of the 1st International Conference on Mobile and Ubiquitous Systems (MobiQuitous'04)*, pages 258–267, 2004.

[4] Joëlle Coutaz. Meta-User Interfaces for Ambient Spaces. In *Proceedings of the 5th International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA'06)*, pages 1–15, 2006.

[5] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W. Fergerson, and Mark A. Musen. Creating Semantic Web Contents with Protege-2000. *IEEE Intelligent Systems*, 16(2):60–71, 2001.

[6] Robert Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 03(3):22–30, 2004.

[7] Thomas Gruber. A Translation Approach to Portable Ontology Specification. *Knowledge Acquisition*, 5(2):199–220, 1993.

[8] 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.

[9] Eric Horvitz. Principles of Mixed-Initiative User Interfaces. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'99)*, pages 159–166, 1999.

[10] Choonhwa Lee, David Nordstedt, and Sumi Helal. Enabling Smart Spaces with OSGi. *IEEE Pervasive Computing*, 2(3):89–94, 2003.

[11] Mieke Massink and Giorgio P. Faconti. A Reference Framework for Continuous Interaction. *Universal Access in the Information Society*, 1(4):237–251, 2002.

[12] Stephen Peters and Howard E. Shrobe. Using Semantic Networks for Knowledge Representation in an Intelligent Environment. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PERCOM'03)*, page 323, 2003.

[13] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an Extensible Context Ontology for Ambient Intelligence. In *Proceedings of the 2nd European Symposium on Ambient Intelligence (EUSAI'04)*, volume 3295 of *LNCS*, pages 148–159, 2004.

[14] Jan S. Rellermeyer and Gustavo Alonso. Services Everywhere: OSGi in Distributed Environments. In *EclipseCon 2007*, 2007.

[15] Thomas Strang, Claudia Linnhoff-Popien, and Korbinian Frank. CoOL: A Context Ontology Language to Enable Contextual Interoperability. In *Proceedings of the 4th International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, pages 236–247, 2003.

[16] Geert Vanderhulst, Kris Luyten, and Karin Coninx. Middleware for Ubiquitous Service-Oriented Spaces on the Web. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, pages 1001–1006, 2007.