The Environment – Application – Adaptation (EAA) Architecture: Introduction and Details of an Open Implementation

Rémi Emonet
Idiap Research Institute
Martigny, Switzerland
remi.emonet@idiap.ch

***Abstract*—This article considers the software problems of reuse and evolution in the context of Ambient Intelligence. The main contribution of the article is the *Environment, Application, Adaptation* (EAA) approach, evolved from state of the art methods used in software engineering and architecture. In the EAA approach, the *applications* are written such that they only reference some abstract functionalities. On the other side, the capabilities of the *environment* are exposed as an individual service. The power of EAA comes from its *adaptation* layer that bridges the gap between capabilities of the environment and functionalities required by the applications. The *adaptation* layer can be dynamically enriched and controlled, giving the end user an easy way to set up the system. The approach is shown to favor development of reusable services and to enable unmodified applications to use originally unknown services. Overall the contributions of the article are: a) the introduction of the EAA approach with an adaptation layer as first-class citizen, b) an illustration through different use cases, c) a feasibility evaluation with implementation details and complete source code available on-line.**

***Keywords*-Environment; Application; Adaptation; Open Source; Community Architecture; Ambient Intelligence; DCI; SOA; End-User Programming**

I. Introduction

With modern devices and technologies, and with sufficient engineering effort, it is relatively easy to implement smart office and smart home applications. Such applications are usually bound to the considered environment and hard to adapt to a new environment. In the context of Ambient Intelligence, such static application design fails because the user is mobile and the environment evolves continuously. Also, an Ambient Intelligence system is always running and is open: new services (of possibly unknown types) are introduced from time to time. The challenge of software architecture for Ambient Intelligence is to provide a way of maximizing reuse and limiting maintenance. For example, applications should not require any modification or redeployment to handle new service types. Our approach tackles this problem and others.

This papers provides additional details over [1], on various aspects of the work. Importantly, many implementation details had been omitted in [1] and were leaving the reader with unanswered interrogations. To improve on this, we provided both more details within the paper and an online release of all the source code necessary to run the experiments, in the form of a "git" repository (see [2]). Together with [1], this article brings the following contributions:

- we review two important software architectures: the Service Oriented Architectures (SOA), which are widely used in Ambient Intelligence and Data Context Interaction (DCI), which is a relatively recent innovation in the design of "traditional" systems and often ignored by the Ambient Intelligence community;
- we combine and adapt SOA and DCI, together with the factory and whiteboard patterns, and propose a new architectural approach that we name Environment, Application, Adaptation (EAA) and that favors reuse and runtime extensibility;
- we illustrate the EAA approach by detailing multiple use cases of applications and showing the advantages of the approach;
- we propose an implementation of the approach using an existing open source service oriented middleware;
- finally, we provide an open-access release of the source code of all the provided use cases to allow introspection, experimentation and reproducibility.

The article is structured as follows: relevant architectural approaches are presented in Section II and we introduce the new EAA architecture in Section III. Section IV introduces the implementation, which is fully detailed with complete examples in Section V. Finally, Section VI provides conclusions and future directions.

II. Related Work and Approach Foundations

Our approach can be seen in continuity with previous architectural concepts. In this section, we introduce the architectural concepts that motivate our approach and we provide discussions about related work.

A. Related Work in Service Oriented Architectures (SOA)

Service Oriented Architectures (SOA) are used in many different contexts ranging from business integration (within and between companies) to Ambient Intelligence. The principle of SOA is to expose software components as "services". Each service encapsulates a particular functionality and provides access to it through a clearly defined interface.

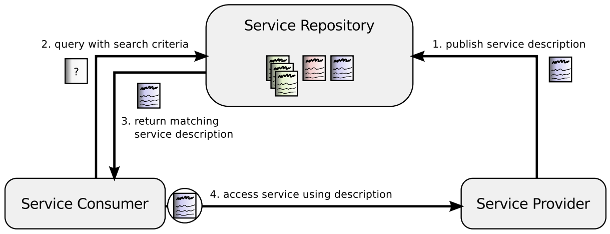

Service Repository
2. query with search criteria
1. publish service description
?
3. return matching service description
Service Consumer
4. access service using description
Service Provider

Figure 1. Service discovery, a fundamental aspect of Service Oriented Architectures (SOA). To find concrete providers of the functionality they are looking for, service consumers query a service repository to which all providers are registered. With service discovery, the consumer and the provider are properly decoupled.

One important characteristic of SOA is "service discovery": a service consumer first queries a service repository (or service resolver) to be able to access a matching provider. This discovery process is illustrated in Figure1. Most service oriented frameworks operate with networked services: services that can run on different machines and communicate through a network. A notable exception is OSGi that is broadly used as in [3]. With networked services, one effect of service discovery is to simplify configuration: service consumers only need to know where to find the service repository.

SOA encourages good encapsulation, loose coupling and abstraction. With little effort, it also helps service consumers in reacting to runtime events like the absence or disappearance of a particular service. With encapsulation and discovery, SOA makes it possible to replace a service by another equivalent one, providing the same interface.

As in many other domains, a variety of service oriented initiatives have been proposed but no single standard is clearly dominating. Also, even if service based approaches provide a good way of implementing some "dynamic distributed components", they fail at solving more advanced integration problems.

Consider the use case of having an application dynamically (and with no modification) start using services it was not originally designed to use. Such case is typical of Ambient Intelligence systems where applications and services evolve continuously. SOA allows this if the services have been properly abstracted out and if the integrators make the effort of writing adapter services to bridge the functionality gap. We consider that this integration use case is actually a common one, rather than an exception. Our approach is designed to encourage better abstractions and to make adapter writing a simpler task.

B. Semantic Web Services (SWS) and Service Composition

The convergence of "Semantic Web" and SOA have been trying to solve the integration problem by letting service designers use their own ontology to describe their services. Ontology alignment methods are then used to make correspondences between services from different providers. Using such correspondence, a service for a given provider can be consumed by a consumer that was designed in ignorance of this particular provider.

Multiple approaches mixes web services (WS) technologies with semantic web principles. These are called Semantic Web Services (SWS). Two major set of technologies are used for semantic web services: Web Ontology Language for Services (OWL-S) and the Web Service Modeling Ontology (WSMO) [4]. Both technologies have been very active. An analysis in [5] places WSMO as more promising but less mature than OWL-S; since this analysis was written, WSMO has evolved and matured.

One interesting element of WSMO is the concept of "mediators" that are used to do alignment, conversion or adaptation of different concepts, data and functionalities. From our point of view, this explicit role of mediators is important and close to our approach with an adaptation layer. Depending on the context of use, SWS technologies have some important drawbacks. First, SWS build upon on web service technologies which add complexity and overhead not suitable for certain platforms and developers. Second, service and functionality descriptions in SWS are very detailed, describing IOPE (inputs, outputs, preconditions, effects) of each operation. These details are used to make an automatic, sound and complete reasoning possible, but put an important modeling load on the service writers.

More recently, model driven approaches, such as UWE (UML-based Web Engineering) [6], have been proposed to try to do adaption. However, like other approaches such as [7], the focus is put on the adaptation of graphical user interface to different devices and contexts. The focus is put on proper engineering of web application while ours is to make ambient applications able to evolve and cope with the dynamic nature of the environment. Our proposed approach can actually be seen as orthogonal to such model driven approaches. As web applications are becoming pervasive, both approaches could be put in practice together, replacing our implementation layer by adapting the model driven web engineering technologies.

In Ambient Intelligence, many projects attempt to integrate different services by building upon both SOA and ideas from the semantic web. Fully automatic service composition and adaptation have been explored, e.g., using multi-agent reasoning as in [8]. Some interesting and well designed approaches are [9] and its evolutions. Also, the soft appliances from [10] envision a systematic decomposition of all existing appliances as independent services. In this vision, end-user programming is used to recreate new innovative appliances from services. One of the main difficulty (and limitation) of end-user programming is to make it both accessible to any end user and powerful enough.

As a conclusion, SOA provides a good basis for Ambient Intelligence but it does not ensure good integration capabil-

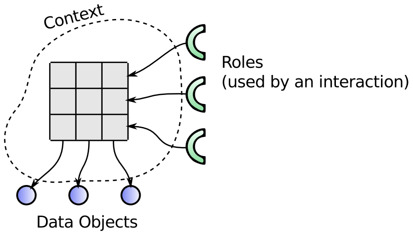


Figure 2. Common representation of the Data, Context, Interaction (DCI) architecture, which complements the Model, View, Controller. Each use case or interaction (I) is implemented using only roles that are fully abstract. The Data (D) are plain objects holding only data and no business specific logic. The Context (C), assembled automatically or through user interaction, is responsible for making some objects play a certain role in the interaction.

ities. Semantic web services are well designed solutions to some of these integration problems but go somewhat to technical and fail at being usable and focused on the adaptation problem. We also think that fully automatic approaches are not desired by the end user: these are not optimal and thus can create frustration, and they prevent end users to express their creativity. Classical end-user programming is also too limited to exhibit, at the same time, these two important aspects: enabling anyone to customize and innovate with applications, and enabling some users to help in integrating new devices. The approach we propose has an explicit adaptation layer and focuses on it, removing the need to describe every possible element in the computational world and making it accessible to most developers.

C. Data Context Interaction: DCI

In our opinion, the most interesting and relevant evolution in recent software architecture and design is the Data Context Interaction (DCI) [11] approach. DCI can be seen as a second attempt to make object orientation (OO) right. The original goal of object oriented programming (and design) was to align the program data model with the user's mental model. This feature is the key to a good human computer interaction: you cannot hide a bad design behind any interface. This becomes more and more important in Ambient Intelligence where user interaction is augmented.

The main principles of DCI are illustrated in Figure2 and can be explained as follows. The *data* objects have the only responsibility to access data (e.g., from a database or memory). In DCI, any use case of the software is a piece of code that manipulates some *roles*, which are fully abstract. A use case is actually an interaction between roles and can be pictured as the scenario involving different roles. A use case uses only a set of roles and never manipulates directly data objects. The concept of *role* together with the *context* are the cornerstone of DCI. A *context* is responsible for doing the mapping of some roles onto some concrete data objects. The context is populated in response to user interaction (e.g., selecting things then clicking on a submit button) and then the use case is executed using this context.

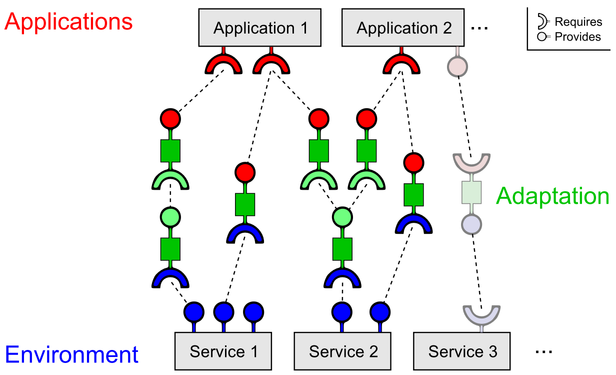


Figure 3. Proposed EAA architecture – *Environment* provides low-level services. *Applications* manipulate only high-level abstract services. *Adaptation* bridges the two and is dynamically extensible and user-controlled. Lighter chain on the right: inversion due to the whiteboard pattern.

As an example, we can consider a banking application with the use case of making a money transfer between two accounts. More precisely, consider the MoneyTransfer use case: it involves three roles that are the SourceAccount role, the DestinationAccount role and the MoneyAmountProvider role. The MoneyTransfer code will start a transaction, then query the amount to transfer from the MoneyAmountProvider, then call *withdraw* on the SourceAccount and call *credit* on the DestinationAccount. The context is created and populated by the application when the user is asked to select a source account (e.g., his CheckingAccount data object) and a destination account (e.g., one of his SavingsAccount) and an amount (e.g., could be just a plain "int" value).

D. Other Related Work

A mobile agent is an autonomous program that can migrate between computers over a network. Even if this is an interesting feature for Ambient Intelligence, it can be seen as orthogonal to the subjects discussed in this article and can complement the proposed approach. An example of using mobile agents as an infrastructure is presented in [12].

The domain of human computer interaction tends to evolve from desktop-like applications to Ambient Intelligence. In this context, an emphasis is put on how to dynamically split and distribute user interfaces based on the available devices. The concept of meta-User Interfaces (meta-UI) has been introduced in [7] and consists in having an interface to control and introspect an Ambient Intelligence environment. A deep and interesting analysis related to our problems is conducted in [7], however, their application is limited to the migration and adaptation of graphical user interfaces between devices.

III. Proposed Approach

In this section, we introduce our Environment, Application, Adaptation (EAA) approach and how it can interact with a community built around it. In the same way as DCI

Table I
CORRESPONDENCE BETWEEN DCI AND EAA TERMS, TOGETHER WITH TYPICAL IMPLEMENTATION (OUTSIDE THE WHITEBOARD PATTERN).

DCI Term	EAA Term	Implementation
Data	**E**nvironment	service providers
Context	**A**daptation	adapter factories
Interaction	**A**pplication	service consumers

is an attempt to make OO right (see Section II-C), EAA is an attempt to make SOA right.

A. Environment, Application, Adaptation

The Environment, Application, Adaptation (EAA) approach builds on top of Service Oriented Architectures (SOA) and takes similar inspiration as Data, Context, Interaction (DCI). In EAA, most of the elements are services: in some sense, services act as objects (with interfaces) that can be distributed and dynamically discovered. As in SOA, the capabilities of the *environment* are exposed as plain services in EAA. In a parallel with DCI, these environment services are corresponding to the data part from DCI.

Most importantly, EAA has the equivalent of roles in DCI. Any *application* only manipulates some abstract services (roles) that correspond to its exact requirements. The design of the application is done without bothering about what concrete service can or will be used to fulfill the role. With this choice, the environment will never directly provide any service that an application needs.

In DCI, the context is responsible for the casting: concrete data objects are recruited to play some roles. In EAA, the *adaptation* layer is responsible for the equivalent, which consists in using services from the environment to create services required by the applications. The adaptation layer is populated through implicit or explicit interaction with the end user (same as in DCI).

In Figure 3 (ignoring the lighter rightmost elements), a set of applications, environment services and adapters are shown. Colors are used to distinguish service types coming from the environment (in blue), the applications (in red) or the adaptation (in green). Table I provides a mapping between EAA and DCI terms, and indication on how EAA elements are implemented.

B. Using Service Factories for Adapters

To populate the adaptation layer, some adapter factories are used. Each factory is actually a service that exposes which kind of adapters it can create and that creates them on demand. The concept of service factory is taken from [13] and restricted to adapters: we do not consider the case of "open factories" that can create services without requiring any other service. With our restriction, the number of instantiable adaptation paths becomes finite and it is thus possible to filter and display them to the user (see Section IV).

C. Refinement using the Whiteboard pattern

A useful pattern in service oriented design is the "whiteboard" pattern [14]. The goal of this pattern is to simplify the design of clients of a particular service. Let's consider a Text2Speech service that is designed to receive some text sentence and that outputs it as speech through loud speakers. In a classical approach, any client of the Text2Speech service would first look for the service, then connect to it and then send the message to it. Eventually, the search-and-connect code is here duplicated in all clients.

Using a whiteboard pattern, the situation is reversed and the Text2Speech service is actually doing the search-and-connect. Each client just declares itself as Text2SpeechSource and the Text2Speech will connect to it as soon as it finds it. With the whiteboard pattern, some code is moved from the client to the "server", which limits redundant code writing and makes backward compatible evolutions easier (the server handles the various versions of clients). From a service point of view, now the "server" looks for its clients, which causes an inversion of the provides/requires dependency as shown in Figure 3 (on the right) and in Figure 4 (on the right).

In EAA, the whiteboard pattern is typically used on the view side, i.e., when the application state needs to be brought back to the user (through the environment). The above example of voicing the output of an application using a Text2Speech service is a typical example of this.

D. Community Architecture and Sharing

The structure of the proposed EAA makes it a "community architecture" [15] in a double sense. First, the approach encourages the creation of a community around it and provides a structure for it, and second, it is the community itself that is creating the actual, live, evolving architecture.

We distinguish four entry points in EAA for innovation and extension, each requiring different skills. Compared to some end-user programming approach where there is trade-off to make between the expressive power of the programming and the required skills to use it, EAA has multiple values for this trade-off. It would be interesting to investigate how EAA can be combined with an end-user approach targeting more ease of use than power of expression (higher expression power being provided by EAA).

The first two entry points are for a relatively large audience. First, most end users will be able to innovate at the adaptation level by doing a smart and original choice of adapters for a particular application in their environment. Also, any end user can take part in the community by suggesting new ideas for services, applications or adapters. With proper documentations and examples, we can expect a reasonable part of the users (surely less than 10%) to be able to create new adapters by copying an existing one or using a wizard tool (in current implementation, an adapter is an XML file that can be easily copied and tuned as shown in following sections).

More advanced extension points concern the contribution of new applications or new environment services. Both require more advanced computer skills but really different ones. Application developers will probably write their application and maybe a couple of adapters to integrate it into the existing ecosystem: the skills required here are mostly classical application development skills. The contributors of new environment services will probably be people that like hacking with new devices or new signal processing methods (image or audio processing, accelerometers, etc.): their goal would be to innovate by providing innovative input or output medium to transform existing applications.

The EAA does not define by itself what kinds of services are used by the people. It is the community itself, by creating new environment services, applications and adapters that decides on what is the actual architecture. We cannot rely on any user to make the best architectural choices. However, if the community is sufficiently large and open, we can expect to find a small proportion of "architects/moderators" as in other open community projects: their role could be for example to avoid proliferation of totally similar concepts and avoid fragmentation of the community.

IV. General Implementation Aspects

To experiment with the proposed approach, we implemented different test cases. The source code of all use cases can be found on-line [2] for additional details and reference. For easier understanding and to allow for reproducibility of the approach, we detail the main aspects of the test case implementation.

Throughout this section and the following, we may reference projects or files from the code available on-line [2]. It is interesting that most of the tasks presented, from coding applications or services to deciding which adapters to use can each be executed by different actors (each with their own skills). This illustrates that most tasks are actually independent and that the resulting system is thus highly extensible.

We implemented the whole presented EAA approach. Most of the tools are written in Java but some commands, usually available under Linux, are used for special functionalities (e.g., text to speech). The "community architecture" aspect, that was totally left out in [1], is now provided. A simple script now allows for an easy download of adapters shared on a central web server. Write access to the server is currently restricted: interested users need to contact us to upload new adapters, or another custom repository can easily be used.

A. The OMiSCID Service Oriented Middleware

Our implementation is based on the open-source OMiSCID [16] service-oriented middleware. A service in OMiSCID can be written in almost any programming language (Java, C++, Python, and most languages running on the JVM) and is discoverable on the network. Each service has a name and may have state variables (also used as properties). In our implementation, we use service variables to expose the information related to the EAA architecture.

Each service can also have connectors. The term "connector" refers to a communication port that can be used to receive messages, broadcast messages or do both. Each message is of arbitrary type but most often either plain text, JSON, or XML. Connectors are the normal mean of passing information between services and we will use it as such.

OMiSCID comes with a graphical user interface (GUI) that can be used to list, monitor and control the services running on a network. The GUI is designed to be highly extensible and allows the user to install plugins in an easy way. As illustrated in the examples and in Figure 4 (detailed later), we designed an interface for the user to decide which adapters should be instantiated among the possible ones. This interface is actually implemented as a plugin for the OMiSCID GUI.

B. Environment Implementation

To compose the environment, we created a set of small reusable functionalities, all exposed as services. Each functionality is actually exposed as a service and an OMiSCID variable "provides" is used to specify which functionalities are provided by the service. In the case of inversion due to the whiteboard pattern, the services from the environment might instead have a "requires" variable.

The developed services that are available online [2] include the following ones: exporting a display area (on a screen or video projector), exporting a mouse pointer, and exporting a "chat" service to allow to open pop-up messages on a computer. Also, under Linux operating systems, we provide additional features like a text-to-speech service based on "espeak", a volume controller and a service to generate synthetic keyboard events on a computer (this one is used for example to control presentations, slide-shows or games). We also provide a computer vision based button to allow user interaction via the real world (e.g., the program detects when the user "clicks" a post-it that he put on a board).

C. Applications Implementation

The applications are also implemented as OMiSCID services that explicitly require some functionalities. The needed functionality is expressed using the "requires" variable. Symmetrically to the environment, when the whiteboard pattern is applied, the "provides" OMiSCID variable is used instead.

The applications that we provide at [2] are a TicTacToe game and a MagicSnake game. It is important to be noted that, thanks to the whiteboard pattern, it is possible to combine multiple services from the environment with only adapters, without having any application. This can also be seen as a logic-less application. An example of this is to use a button (e.g., any event or a computer vision based button as in Figure 8) to step to the next slide in a presentation.

D. Adaptation Implementation: adapter factories

For the adapters, we designed a generic program that takes an XML description of a family of adapters and starts the corresponding adapter factory (that can start an adapter instance on demand). By convention the service name for adapter factories is AdapterFactory. The XML description contains information about the adapter such as which functionality it takes as "input" and to which one it converts it. The adaptation code, that is usually simple, can be provided within the XML file using languages such as JavaScript, XSLT or dedicated custom languages. Examples of XML descriptions are provided in Section V.

The factory description contains information about what adapter the factory can create (see Figure 7 for an example fully detailed in Section V). A "from" variable contains the name of the functionality that the adapter will take as input. The "to" variable is used for the target functionality that the adapter produces. Some parameters can be used in the "to" variable and are defined in the "parameters" variable. In the "parameters" variable, a special construct can be used to specify that the parameter must take a value that corresponds to an existing requirement (a value present in the "requires" variable of a running service).

E. Adaptation Implementation: user interface

As mentioned in previous section and illustrated in Figure 4, we implemented the control of the adaptation layer as an OMiSCID GUI plugin. This plugin mainly involves Netbeans Platform programming (source to be found in `projects/AdapterFramework`) and won't be detailed here, only its behavior will be described.

Using service discovery, the plugin lists of all relevant services:

- any service having a "provides" variable,
- any service having a "requires" variable,
- any service named AdapterFactory.

Then the plugin displays all provided functionality, together with all required ones and all possible adaptation paths that can get constructed using the running adapter factories. The adapters on the paths are initially not instantiated and displayed using a shaded style. When the user double clicks on an adapter, the GUI plugin automatically formats a message and sends it to the appropriate AdapterFactory that in turn will create the necessary adapter. Once the adapter is started, it changes from shaded to solid in the GUI panel. The user can also easily stop any started adapter, by using a dedicated action provided in the OMiSCID GUI service tree.

V. Detailed Test Cases

To showcase our approach, we detail the case of a simple tic-tac-toe game we developed, starting by a global architecture, which is then detailed.

A. Tic Tac Toe Architecture and Benefits

For now, we consider that the environment contains only two computers, and from each one we export some services: a Display, a Mouse3 (mouse pointer with 3 buttons) and a Text2Speech. In total we thus get six environment services running, three on each computer. Each exported Display service has a unique identifier and follows a whiteboard pattern to connect to any matching DisplaySource it finds. A DisplaySource is expected to send drawing commands to the Display.

The game logic is implemented as a service that exposes a TicTacToeModel functionality. The TicTacToeModel encapsulates the state (current board, current player) and the rules of the game (only the current player can play, who wins, etc.). In addition it also requires some functionalities for the input of the players, more precisely, it needs two Grid3x3Clicker with two different unique identifiers. Following a whiteboard pattern, the game logic automatically connects to the matching Grid3x3Clicker it finds.

To bridge the gap between the environment (Display, Mouse3) and the application (Grid3x3Clicker, TicTacToeModel), we introduced a set of simple adapters. The first ones are for input and can be heavily reused in other context: one adapter converts a three button mouse Mouse3 to a single button mouse Mouse1, the second adapter converts a Mouse1 to a Grid3x3Clicker by converting clicked x, y position to some grid index from 0 to 8. We could have skipped the distinction between Mouse3 and Mouse1 but we kept it as it is useful in some other contexts. On the display side, a specific adapter was written to convert TicTacToeModel to a DisplaySource: the tic-tac-toe state change events are converted to drawing commands such as drawing circles.

By letting the user control the adaptation layer, EAA makes the tic-tac-toe become ambient. The use of properly decoupled services ("SOA done right") makes it possible for the user to dynamically select where and how to display the game and how to control it. EAA, with its explicit adaptation layer, makes it also possible to easily create variations of the game that integrates into an Ambient Intelligence vision. To this end, different adapters can be used. A first adapter, which is simple but specific, transforms the game state (TicTacToeModel) to some short textual output to be processed by a Text2Speech service. A reusable adapter, used for input of the game, could use a SpeechRecognizer and converts voice commands such as "play in three" to a Grid3x3Clicker. In addition to the audio modality, computer vision is also used as a possible input: by sticking post-its

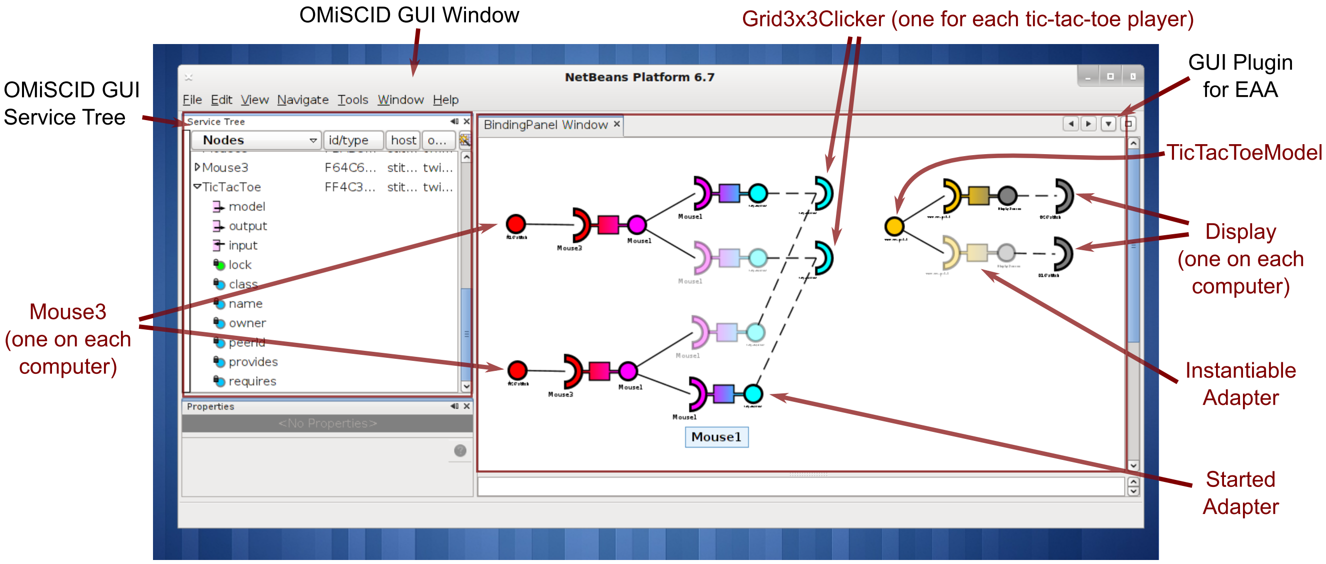


Figure 4. Screen capture of the OMiSCID Graphical User Interface (GUI) for service monitoring and control. On the left side, the default service tree provided by the GUI. On the right, the panel provided by the plugin for the EAA architecture. All provided and required functionalities are shown, using a color for each functionality type. The plugin also considers information from all running adapter factories and proposes all possible adaptation paths to the user. The user can click on a instantiable adapter (shaded), then the plugin queries the corresponding factory for the creation of the adapter. Once it is started, the adapter becomes opaque. Note that due to the whiteboard pattern, the provides/requires relation is reversed for the display side (right part of the panel) where the environment (Display) requires services from the applications (TicTacToeModel).

on a surface, the user can transform it to a Grid3x3Clicker thanks to a dedicated adapter.

B. Tic Tac Toe Implementation Details

The source code for all elements mentioned in this section is provided at [2]. As suggested by the feedback received about [1], we provide a detailed view of how different parts of the system are implemented and articulated. To help in following the explanations of this section, Figure 6 provides a sequence diagram of the interactions between different elements of the system.

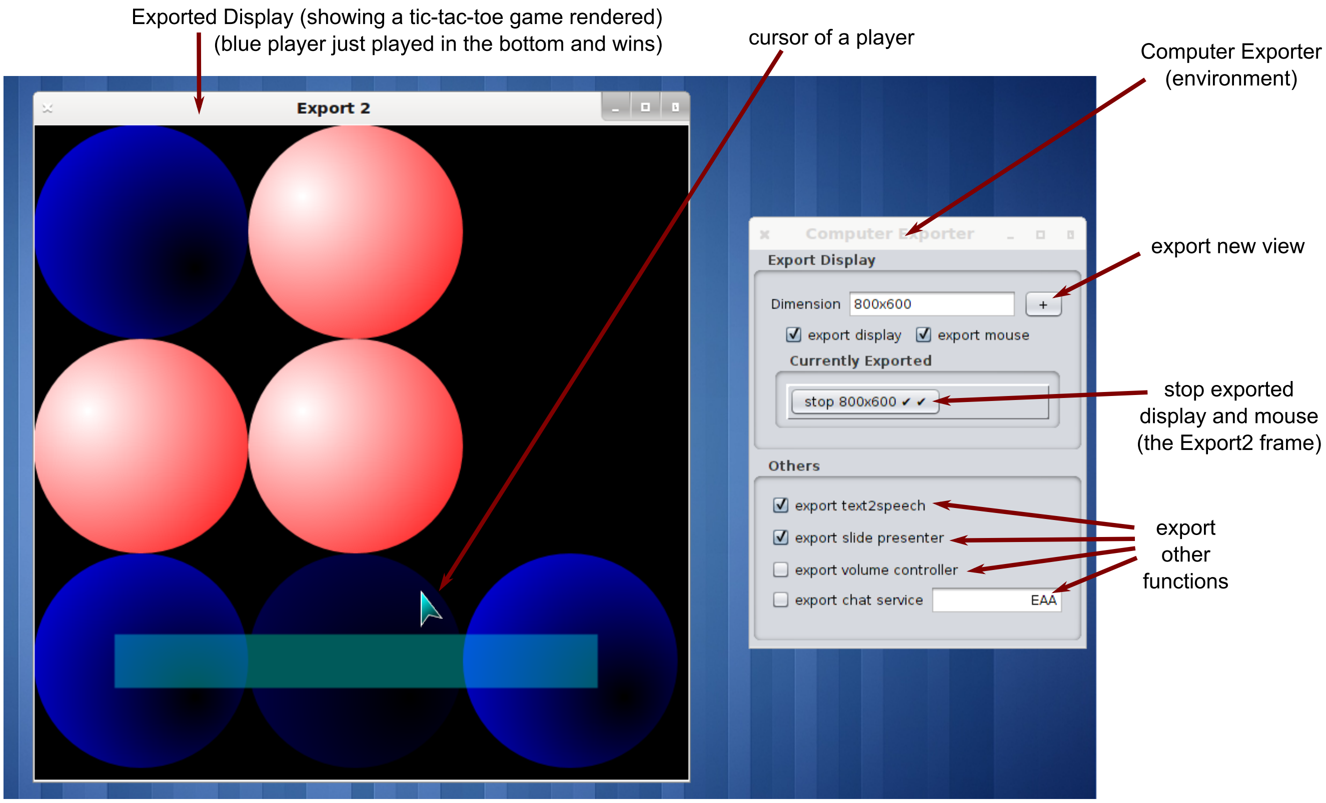
Environment – In the provided use case, the environment is populated using a single Java program, the code of which can be found in `projects/ComputerExporter`. The user interface for exporting environment capabilities is shown on the right of Figure 5. This interface can be used to export any number of views, each being a frame, only one being shown on the left of Figure 5. Using multiple views is useful for example when multiple screens or video projectors are plugged to a single computer: in such case, exporting one view per physical display makes more sense.

Each view can be used as input, exporting a “Mouse3” functionality, which is backed by an OMiSCID service, which sends XML messages for each mouse event such as motion events and click events. Each view is also a “Display”, also backed by an OMiSCID service that expects to receive messages containing some rendering code fragments. The display service handles multiple simultaneous clients and merges their rendering code fragments. A typical example of a display having multiple clients is the case where we want to display the tic-tac-toe game and also add the rendering of a mouse cursor on top of it (e.g., the cursor of a remote player or a cursor controlled using some hand gestures) as it is the case in Figure 5.

Technically, the rendering code fragments are expressed in JavaScript. The display service sets up a script engine for JavaScript interpretation and fills some context variables so that the snippet can access the rendering context of the frame. Then, the received code fragment is interpreted in context, and this results in graphical elements being drawn in the frame.

Overall, all the services exported by the ComputerExporter are very generic and can be reused over and over for different applications. They are indeed reused in the other use cases presented in Section V-C.

Application – The tic-tac-toe application code can be found in `projects/TicTacToe` and is written in Java. The code simply implements the tic-tac-toe game logic and starts an OMiSCID service. In addition to the variables “provides” and “requires” (presented in previous section), the service has one input connector to receive commands such as “player 1 plays in bin 0”, encoded as two digits “10”. The service also broadcasts on two output connectors. The connector “model” is used to send the complete game model (expressed in XML) each time a change is made to it. For convenience for the clients, a connector “output” is also used to broadcast events of model changes, i.e., the fact that



Dimension 800x600

☑ export display ☑ export mouse

Currently Exported

stop 800x600 ✓ ✓

☑ export text2speech

☑ export slide presenter

☐ export volume controller

☐ export chat service EAA

Figure 5. The “ComputerExporter” user interface is shown on the right. It is used to start environment services. On the left is shown an exported display that is currently displaying a tic-tac-toe game. See Section V-B for details.

a given player played at some position, or the fact that one player won. Overall the tic-tac-toe application is minimal, containing only the necessary elements: the game model, the game logic and some interfaces for control and view using an OMiSCID service.

Adaptation – By convention, any OMiSCID service named “AdapterFactory” is considered as a factory of adapters, provided it has the necessary variables introduced in Section IV. Such a factory service has a “create” input connector and when it receives a message on it, it parses the parameters provided inside and starts the corresponding adapter.

Even, if we could write each adapter factory from scratch, we simplified the writing of these. The code, which is generic and shared by all factories, is encapsulated in a java program that can be found in `projects/AdapterTools`. All the information specific to a given adapter factory is included in an XML service description file using conventions that we will illustrate below. For convenience, a script located at `tools/xml-service.sh`, can be called with multiple XML descriptions as parameter and it invokes appropriately the Java program to start all the adapter factories described by the XML files.

Figure 7 illustrates how adapters are written, it provides a complete example of the cursor renderer. The goal might be to create a visual feedback in the same way it is done on classical desktop interfaces to show the mouse pointer position (but here many modalities can be used, e.g., a pointer controlled by hand gestures). As detailed hereafter, this adapters uses the Java2D API through JavaScript and also uses XSLT (Extensible Stylesheet Language Transformations): this adapter can be seen as one of the most complex adapters involved.

The XML service description provided in Figure 7 has a root “service” element and the “name” attribute is used to provide the service name, here “AdapterFactor” on line 1. Then the description contains a succession of OMiSCID variable descriptions: in the example, all these service properties are defined as “constant”, meaning they don’t change during the lifetime of the service.

Lines 2 to 7 (Figure 7) define the functionality adaptation that this factory can perform. The factory can transform any “Mouse3” functionality, as expressed in the “from” variable. The “to” variable on line 6 is a little more evolved: the factory can create any “Display” source with properties “for” and “z” set according to some parameters. The two variable references “${id}” and “${z}” are references to instantiation parameters, declared in the “parameters” variable detailed

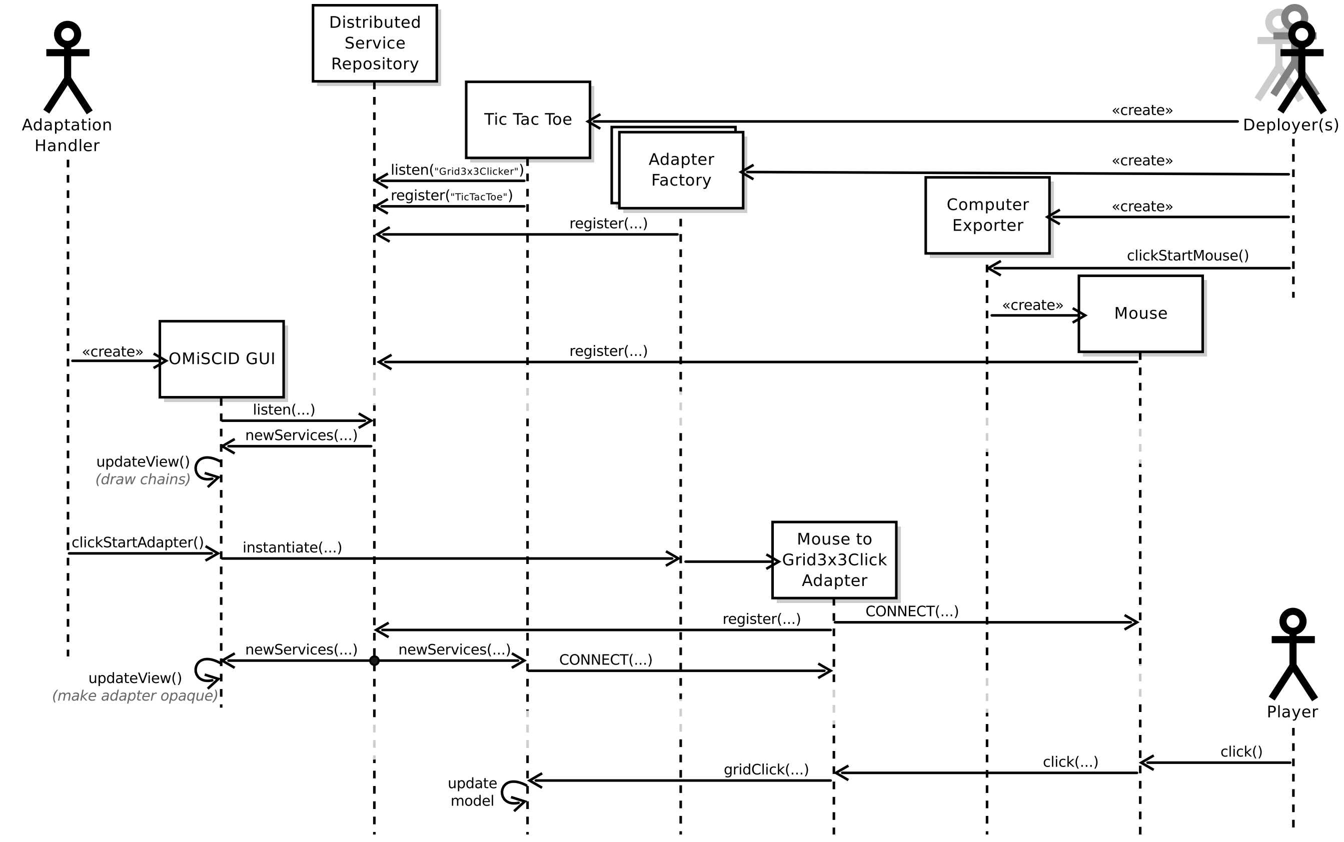


Figure 6. Sequence diagram showing the interactions between different parts of the system. First, the application and some base services are deployed, then the adaptation layer is populated and finally the player plays a turn. Only one input is provided, no display/output for the game is shown in this diagram. See Section V-B for details.

below.

Lines 10 to 14 (Figure 7) define what we call the instantiation parameters. When a client (the main client being the GUI plugin) asks a factory to instantiate an adapter, it sends a message containing the OMiSCID service identifier of the service to adapt (here, the identifier of the Mouse3 to adapt) plus a set of custom instantiation parameters. In the example, there are 5 instantiation parameters. The four last ones are just arbitrary customization parameters, each having a name, a type and a default value. The first parameter in the example, named “id” uses a special construct in place of the default value.

To understand the “#(someRequirement DisplaySource for)” from line 10, we have to remember that the display services from the environment, are actually using a whiteboard pattern. Each Display has a unique identifier. It explicitly “requires” and automatically connects to any service with a “DisplaySource” functionality having a “for” property equal to this unique identifier. The “someRequirement” construct just expresses that the value of the instantiation parameter should correspond to the value of the “for” property of a currently required “DisplaySource” functionality.

Lines 17 to 48 define two other variables that have a special purpose: instead of being solely OMiSCID variables, they also describe the behavior of the adapters. Their respective names “start” and “code” are conventions and these are treated specially by the program that interprets the XML description. The “start” variable describes some (additional) code that will be executed each time an adapter is instantiated by the factory. Starting with “js:”, it tells the interpreter that the behavior is expressed in JavaScript. The two lines 19 and 20 respectively add a “display” output connector to the adapter (that it will use to send messages) and create a local input connector that it plugs to the “events” connector of the source service (the Mouse3 to be adapted) on which a listener is registered. With the “listenTo(...)” command, all messages received on the corresponding OMiSCID connector will be processed by the code described in the “code” variable, starting at line 23.

Lines 24 to 44 (Figure 7) define the message handler that each adapter will use to process messages. The code is protected inside an XML CDATA section to avoid escaping all brackets. The code starts with “xslt:” and thus is expressed using the XSLT language which is a standard explicitly

```
<service name="AdapterFactory" ...>
   <variable name="from"> <access>constant</access>
      <value>Mouse3</value>
   </variable>
   <variable name="to"> <access>constant</access>
      <value>DisplaySource for=${id} z=${z}</value>
   </variable>
   <variable name="parameters"> <access>constant</access>
      <value>
         id : string ... = #(someRequirement DisplaySource for)
         size : float ... = 32
         z : float ... = 90
         color1 : Color ... = 0x00FFFF
         color2 : Color ... = 0x000000
      </value>
   </variable>
   <variable name="start"> <access>constant</access>
      <value>js:
         addOutput("display");
         listenTo("events");
      </value>
   </variable>
   <variable name="code"> <access>constant</access>
      <value><![CDATA[xslt:
         <xsl:template match="events/move">
            <message on="display" type="text">
               var x = <xsl:value-of select="@x"/>;
               var y = <xsl:value-of select="@y"/>;
               var s = <xsl:value-of select="$size"/> / 50.0;
               var C = java.awt.Color;
               var c1 = C.decode("<xsl:value-of select="$color1"/>");
               var c2 = C.decode("<xsl:value-of select="$color2"/>");
               g.translate(x, y);
               var p = java.awt.geom.GeneralPath();
               p.moveTo(0,0);
               p.lineTo(s*31.12, s*(50-10.83));
               p.lineTo(s*12.13, s*(50-15.28));
               p.lineTo(0, s*50);
               p.closePath();
               var grad = new java.awt.GradientPaint(s*5, s*5, c1, s*30, s*40, c2);
               g.setPaint(grad);
               g.fill(p);
               g.setColor(C.WHITE);
               g.draw(p);
            </message>
         </xsl:template>
         ]]></value>
      </variable>
</service>
```

Figure 7. XML description of a adapter factory transforming a “Mouse3” functionality into a “DisplaySource” functionality, the goal being to render a cursor on the display at the position of the Mouse3 cursor. See Section V-B for details. For reproduction in this paper, file is reformatted and irrelevant parts are replaced with “...”. In this case the “...” are placeholders for namespace declarations and some type information that is unused in this context.

designed to transform XML documents. The adapters expect to receive XML messages and line 25 defines how the “<move>” messages received on the “events” connector should be processed. Line 26 expresses that (for each move message that is received) the adapter should broadcast a new text message on the “display” OMiSCID connector. Remember that these output messages are eventually reaching a Display service, which will expect some drawing code written in JavaScript. That is exactly what lines 27 to 44 produce: some JavaScript code using an implicit graphic context “g”. The JavaScript code is generated by resolving some XSLT variable (using “<xsl:value-of...”). The “@x” and “@y” make use of the XSLT notation to access attributes, here the attributes of the “<move>” element. The “$size”, “$color1” and “$color2” use the XSLT notation for accessing variables. These variables actually correspond to the instantiation parameters for the considered adapter and they are brought into the XSLT context by the adapter factory program. In the example, we see that the position from the move message is used to translate the rendered cursor (lines 27, 28, 33), while the parameters (that can be tuned by the client of the factory or by the GUI plugin) control the size and the colors of the cursor.

Figure 9 provides another adapter factory description, transforming a Mouse3 into a Mouse1 by just filtering click events and forwarding move ones. All the concepts involved in this adapter factory description have been covered in previous paragraphs. The main novelty in this adapter lies in the “code” variable at lines 12 to 18. Again the input messages are expected to be XML messages but this time two different possible root elements are handled: the “<click>” and the “<move>” elements on lines 13 and 16. Another difference is that the output messages sent at lines 14 and 17 are not textual messages but rather XML messages (when omitted, the “type” attribute defaults to “xml”). In the case of XML output, the XSLT language is again very well suited as it has been designed for XML transformation.

Figure 10 shows the use of a dedicated language in the “code” variable. This example is actually the one of an adapter from Android key events to some slide controller commands. The language has been designed to make it easy to express a mapping between arbitrary string messages to string messages. XSLT could be used for this purpose but would be unnecessarily verbose. The “code” variable from Figure 10, lines 12 to 15, starts with “map:” indicating the use of the custom mapping language. This language is a domain specific language designed for mapping string to strings. The mappings are given, one by line, each string on the left of the arrow “->” is mapped to the string on the right. In this case, for example, when a message containing “KEY25UP” is received from the “events” connector of the source service, the adapter will broadcast a message containing “next” on its “events” connector. What is not illustrated in this example is the fact that the left part of the arrow is actually a regular expression and the right part a replacement expression. With the “map:” language, most people can create new mappings by copying and updating an existing adapter, without any knowledge of XSLT, JavaScript or even XML.

All adapters are present in a `adapters/` folder in the git repository. The adapters are written using exactly the principles explained previously in this section. Some knowledge of XSLT is required to understand advanced constructs, e.g., as in Figure 7 and Figure 9.

C. Other Test Cases

Apart from the tic-tac-toe game, we also implemented other environment services, applications and adapters.

Games With Analogous Controls – For example we created a MagicSnake game that consists in guiding a snake in a 2D maze to reach a target as fast as possible while avoiding walls. The game can be found in `projects/MagicSnake` and, in the same way as the tic-tac-toe game, it requires a specific controller and exposes its model (but no event-based output). The game is rendered thanks to a dedicated adapter (output illustrated in Figure 8) and reuses the exact same Display service as the tic-tac-toe. As an experiment, we also modified a game called “Nuncabola” where the player controls a ball rolling in a 3D environment. Both games use a two dimensional analog input: we implemented this input with different combinations of environment services and adapters. Eventually, we control these games using:

- obvious device such as a mouse or a keyboard,
- more exotic devices such a accelerometer-based devices (e.g., smart phone, WiiMote) or WiiFit-like devices,
- computer vision and human tracking (e.g., the player moves in the room to control the ball acceleration, or the player moves his hands, arms, etc.)

Various Use of Simple Events – The main use of the “map:” dedicated language that we proposed (see Section V-B) is to easily write adapters for services that exchange only very simple events. We considered various input modalities for such events and various applications and logic-less applications where an adapter links directly an input service from the environment to an actuator service of the environment.

Using simple generation of keyboard events, we implemented a slide presentation controller. We used various methods to skip to the next/previous slide including for example computer vision, e.g., gestures; sound recognition (clapping hands); and voice recognition, e.g., saying “next slide”.

We provide an example of computer vision based push button: the code is available in `projects/AdditionalModules` and is actually implemented using a multi-language component framework (see [2]), the assembly of components being defined in

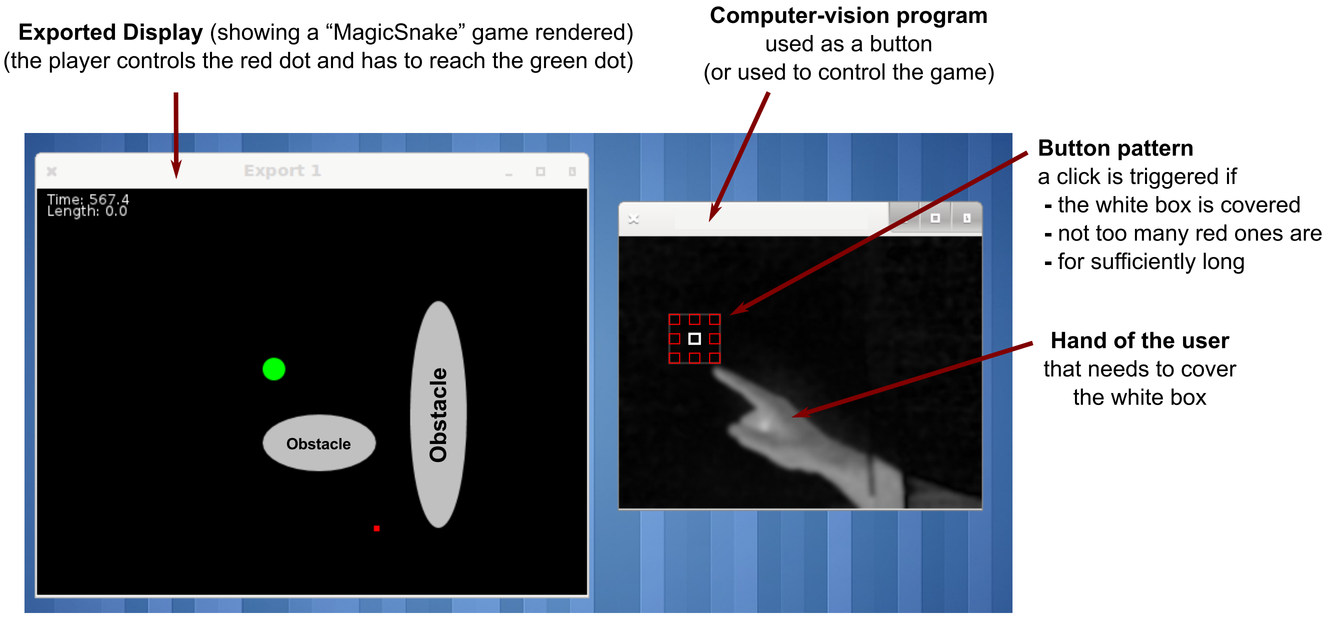

Exported Display (showing a "MagicSnake" game rendered)
(the player controls the red dot and has to reach the green dot)
Computer-vision program
used as a button
(or used to control the game)
Button pattern
a click is triggered if
- the white box is covered
- not too many red ones are
- for sufficiently long
Hand of the user
that needs to cover
the white box
Export 1
Time: 567.4
Length: 0.0
Obstacle
Obstacle

Figure 8. Showing a rendered view from the MagicSnake game together with a debugging window explaining computer vision based environment services. See Section V-C for details. Detecting and tracking the finger tip, its 2D position can be used as an analogue input for the MagicSnake game. We also show a computer-vision based button: we want the button to be triggered if the white box in the center is covered by the user's hand. We also want to avoid unwanted clicks, for example, when the user's hand is fully covering the patterns (including the red boxes) we don't want a click to be triggered.

the file `pipelines/clicklet-omiscid.xml`. The push button is exposed as an OMiSCID service and sends simple "ON" or "OFF" events when it changes state. The overall principle of the vision-based button is to extract the foreground image of the scene, e.g., containing the user or his hand.

Imagining we want a small rectangle (e.g., a post-it) to act as a button, then, we define a region of the size of this rectangle and we detect when it is covered by the foreground pixels. An example of such region is the white box shown in Figure 8. However, in the example of Figure 8, we want a click to be generated if the user covers the white box with his finger. However, if the full hand covers it, then it probably means that the intention of the user is different (reaching another button or just passing between the camera and the post-it). To be able to filter out these wrong clicks, we reason about whether the red boxes from Figure 8 are covered or not (how many of them and which ones). To avoid unwanted repetitive clicks, some delay is added: a button click is validated only if the click detection stays stable for a few frames.

We also implemented a minimal application for Android devices: it exposes an OMiSCID service that sends events each time a physical key is pressed (e.g., volume keys, camera button, etc). The source code is available in `projects/AndroidDeviceExporter`. We could also export a "Display" for the Android device but the Android platform does not implement Java2D and thus it would require to express the rendering in a different way (compared to what we currently used). The Scalable Vector Graphics (SVG) format is a good candidate for an evolution of the display, to have some cross-platform rendering primitives.

Events produced from the Android application can be transformed, for example to a controller for a slide presentation. The corresponding adapter uses the "map:" language and is provided in Figure 10. Similarly, using the computer vision based push button to change slide can be done with a simple mapping like "ON -> next", considering only the press of the button and ignoring when it gets released.

The computer exporter presented in previous sections (source to be found in `projects/ComputerExporter`) can export various services that expect simple string events. The slide presenter accepts four commands: "next" and "previous" for stepping within slides (actually sending left and right arrow key events to the system) and "next+" and "previous+" for skipping directly to the following slide, skipping animations (actually sending up and down arrow key events to the system). There is also a volume controller that accepts "volumeup", "volumedown" and "mute", which impact the system volume in the expected way.

Two services from the computer exporter actually accept any simple string message: the chat service and the text to speech (TTS) service. When it receives a message, the chat service displays the message in a frame on the computer where it is running (the frame is opened if it was closed before). This can be used for popup notifications, application reporting or the textual rendering of models such as the tic-tac-toe model. Similarly the TTS service will use "espeak" to voice any text message it receives. The TTS can be used for mostly the same purposes as the chat service, the main differences being the throughput (speaking is slower than writing) and the fact that the TTS works even if the user is not facing a screen.

VI. Conclusion and Future Work

This article presented the Environment, Application, Adaptation (EAA) architectural approach. It reuses service oriented principles (SOA) and takes inspiration from the Data, Context, Interaction (DCI) approach. Within our EAA, the environment and the applications are fully independent of each other. This both encourages the design of more generic environment services and eases the deployment of an unmodified application in a new environment: this deployment is possible even if, eventually, the application ends up using only originally unknown services. The glue between what a particular environment offers and what a particular application requires is done by a dedicated adaptation layer. This layer makes the overall system easier to adapt and open to user control and innovation.

An implementation of this approach was showcased: this implementation is fully operational and allows dynamic run-time extension with new services, applications and adapters. To incorporate informal feedback received on [1], we provided very detailed explanations of the mechanisms involved in the implementation. We also cleaned up and made available the source code for all presented use cases on a dedicated page [2].

The foreseen future directions involve the improvement of the user interface (icons for service types, quick filtering, etc), and the exploration of a dedicated interface to create simple adapters based on the mapping language we used. More structured variations of the proposed approach, with different implementation choices, should also be explored: indeed, the approach matches recommendations made by the European IST Advisory Group (ISTAG) in a recent report [17] mentioning that "we might expect to see new programming or modeling languages which include adaptation mechanisms as first-class citizens".

References

[1] R. Emonet, "Environment - Application - Adaptation: a Community Architecture for Ambient Intelligence," in *2011 1st International Conference on Ambient Computing, Applications, Services and Technologies (AMBIENT)*, Oct. 2011.

[2] "Webpage for the source code for the EAA demonstration (this article)." accessed 12-July-2012. [Online]. Available: http://eaa.heeere.com/

[3] C. Escoffier and R. Hall, "Dynamically adaptable applications with iPOJO service components," in *Software Composition*, 2007, pp. 113–128.

[4] ESSI WSMO working group: research and development efforts in the areas of Semantic Web Services, 2007, website and working draft: http://www.wsmo.org/ and http://www.wsmo.org/TR/.

[5] R. Lara, D. Roman, A. Polleres, and D. Fensel, "A conceptual comparison of wsmo and owl-s," in *ECOWS 2004*, ser. LNCS, vol. 3250. Springer, 2004, pp. 254–269. [Online]. Available: http://www.springerlink.com/content/p8358uyre5kw3h7h

[6] J. Preciado, M. Linaje, R. Morales-Chaparro, F. Sanchez-Figueroa, G. Zhang, C. Kroiß, and N. Koch, "Designing rich internet applications combining uwe and rux-method," in *Web Engineering, 2008. ICWE'08. Eighth International Conference on*. IEEE, 2008, pp. 148–154.

[7] J. Coutaz, "Meta-user interfaces for ambient spaces," *Task Models and Diagrams for Users Interface Design*, 2007.

[8] M. Vallée, F. Ramparany, and L. Vercouter, "Dynamic service composition in ambient intelligence environments: a multi-agent approach," in *Proceeding of the First European Young Researcher Workshop on Service-Oriented Computing*, Leicester, UK, April 2005.

[9] M. Assad, D. Carmichael, J. Kay, and B. Kummerfeld, "PersonisAD: distributed, active, scrutable model framework for context-aware services," *Pervasive Computing*, 2007.

[10] J. Chin, V. Callaghan, and G. Clarke, "Soft-appliances: A vision for user created networked appliances in digital homes," *Journal of Ambient Intelligence and Smart Environments*, pp. 69–75, 2009.

[11] J. O. Coplien and G. Bjørnvig, *Lean Architecture: for Agile Software Development*. Wiley, 2010.

[12] R. Razavi, K. Mechitov, G. Agha, and J. Perrot, "Ambiance: a mobile agent platform for end-user programmable ambient systems," in *Proceeding of the 2007 conference on Advances in Ambient Intelligence*. IOS Press, 2007, pp. 81–106.

[13] R. Emonet and D. Vaufreydaz, "Usable developer-oriented functionality composition language (ufcl): a proposal for semantic description and dynamic composition of services and service factories," in *Intelligent Environments, 2008 IET 4th International Conference on*. IET, 2008, pp. 1–8.

[14] O. Alliance, "Listener Pattern Considered Harmful: The Whiteboard Pattern, 2nd rev." http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf, 2004, [Online; accessed 15-December-2012].

[15] F. Moatasim, "Practice of community architecture: A case study of zone of opportunity housing co-operative," Ph.D. dissertation, McGill University, 2005.

[16] R. Emonet, D. Vaufreydaz, P. Reignier, and J. Letessier, "O3miscid: an object oriented opensource middleware for service connection, introspection and discovery," in *International Workshop on Services Integration in Pervasive Environments*, 2006.

[17] ISTAG, "Software Technologies: The Missing Key Enabling Technology," http://cordis.europa.eu/fp7/ict/docs/istag-soft-tech-wgreport2012.pdf, 2012, [Online; accessed 15-December-2012].

```
<service xmlns="..." name="AdapterFactory">
 <variable name="from">...Mouse3...
 <variable name="to"> ... Mouse1...

 <variable name="parameters"> <access>constant</access>
    <value>button1 : int  = 3</value>
 </variable>

 <variable name="start">... js: addOutput("events"); listenTo("events"); ...

 <variable name="code"> <access>constant</access>
    <value><![CDATA[xslt:
      <xsl:template match="events/click[ @button = $button1 ]">
         <message on="events" type="xml"><click button="1" x="{@x}" y="{@y}"/></message>
      </xsl:template>
      <xsl:template match="events/move">
         <message on="events"><move x="{@x}" y="{@y}"/></message>
      </xsl:template>
    ]]></value>
 </variable>
</service>
```

Figure 9. XML description of an adapter factory transforming a Mouse3 functionality into a Mouse1 functionality by simply filtering click messages and forwarding move messages. The “code” of the service uses an “xslt:” transformation, see Section V for details. For reproduction in this paper, file is reformatted and irrelevant parts are replaced with “...”.

```
<service xmlns="..." name="AdapterFactory">
 <variable name="from"> ... AndroidKeys ...
 <variable name="to"> ... RemoteControl for=${id} ...

 <variable name="parameters"> <access>constant</access>
    <value>id: float ... = #(someRequirement RemoteControl for)</value>
 </variable>

 <variable name="start"> ... js: addOutput("events"); listenTo("events"); ...

 <variable name="code"> <access>constant</access>
    <value>map:
       KEY24UP -> next
       KEY25UP -> previous
       KEY80UP -> next
    </value>
 </variable>
</service>
```

Figure 10. XML description of an adapter factory transforming an AndroidKeys functionality into a RemoteControl functionality. The “code” of the service uses a custom “map:” type, see Section V for details. For reproduction in this paper, file is reformatted and irrelevant parts are replaced with “...”.