An Agent Construction Model for Ubiquitous Computing Devices

Ronald Ashri and Michael Luck

Dept of Electronics and Computer Science, Southampton University,
Southampton, UK
`ra,mml@ecs.soton.ac.uk`

Abstract. One of the main challenges for the successful application of agent-based systems in mobile and embedded devices is enabling application developers to reconcile the needs of the user to the capabilities and limitations of agents in the context of environments with changing and often limited resources. In this paper we present an attempt to move towards a solution through a framework for defining and reasoning about agents in a manner that is modular and reconfigurable at run-time. Departing from the theoretical basis afforded by the SMART framework, we extend it to enable the definition of fully re-configurable component-based agents. The guiding principles of this approach are an architecturally-neutral model that supports a separation between the description, behaviour and structure of an agent.

1 Introduction

The spread, and rise in influence of, embedded and mobile devices with limited computational power, which have found favour in many aspects of everyday life, from mobile phones to personal digital assistants (PDAs), provides a counterpoint to the tradition of desktop computing. In line with this profile, there is also an increasing demand for *integrating* the various different kinds of such devices in order to provide an environment where access to information and services is available in a seamless manner, while transcending physical location and computing platform.

Agent-based systems have a key role role to play in the effort to provide and support such integration, since agents embody several of the required characteristics for effective and robust operation in ubiquitous environments [4, 12]. A central area of concern for supporting agents operating in such heterogeneous environments, which place differing and varying demands and limitations on them, is the development of appropriate agent architectures for the device and environment within which they are operating. We identify below three specific challenges for the development of architectures in such environments.

- The heterogeneity of operating environments and devices makes it practically impossible to adopt a single optimal design for an agent architecture. For example, a purely BDI-based approach for agents on all types of devices may simply be overly complex for a number of limited devices that do not need to deal with complicated tasks and as such would not benefit from planning capabilities or a sophisticated

representation of beliefs. Instead, we should enable developers to create solutions that are tailored to specific devices and application domains, without constraining them to specific architectural approaches.

– The necessity to have multiple types of agent architectures, while providing the required level of flexibility, also introduces new challenges since it increases the overall complexity of the system design. A multiplicity of architectures also makes it more challenging to choose the best for a specific situation.

– Finally, the agent architecture should be able to deal with the unpredictable nature of computing devices and the environments they operate in. For example, devices may stop operating due to power failure but it is important that agents are able to keep some information about their state when the device is restarted.

In this paper, we address just these issues by presenting a model for agent construction that is *conceptually grounded* and *architecturally neutral*. It is conceptually grounded by the understanding of agent systems provided through SMART [10] and it is architecturally neutral because a number of different agent architectures can be expressed through the constructs provided. Through this agent construction model we advance the current state of the art for agent-oriented software engineering in three ways. Firstly, we provide an agent construction model that addresses the specific needs of agent construction on mobile devices. Secondly, through the implementation of the model we identify some specific techniques that can be used for adapting to changes in device capabilities and operating environments. Finally, as a result of this work we identify some more generalised features that can inform the construction of agents in other settings beyond ubiquitous computing.

We begin by introducing a series of *desiderata* that our agent construction model should fulfil, and then outline some of the key design decisions that guide its development as well as clarify its position with the context of application development. Subsequently, the agent construction model itself is presented followed by a discussion of its implementation for devices supporting the Java 2 Micro Edition. Finally, we conclude and compare our approach to others.

2 Design Principles

2.1 Desiderata for an agent construction model

In order to address the range of concerns raised above and provide some statement of requirements for the agent construction model, we identify four desiderata. Although the set is not exhaustive, we consider it to be the minimum necessary set of requirements.

Abstract Agent Model An agent construction model that addresses the issues raised above must be based on some understanding of how we can model agents in a manner that is independent of the agent architecture. This allows the comparison of alternative architectures at this more *abstract* level, ultimately providing application developers with more informed choices *before* they proceed to provide specific implementations for the domain in question. In our case, the SMART framework provides such an abstract agent model (as we discuss in Section 2.2).

Architecturally neutral The construction model should not lead to the construction of only a limited range of agent types, but should allow the widest possible range of architectures to be defined using the same basic concepts. In order to achieve this, there are two possible avenues to explore. One option is to define a generic agent architecture and describe other architectures in terms of this generic architecture, something that Bryson et al. suggest [6]. Apart from the inherent difficulty in constructing any general, all inclusive model, the drawback of this approach is that there may be features of other architectures that cannot directly be *translated* to the generic one. The second option is to provide an architecturally-neutral model, so as to avoid this translation problem. Here, the challenge is to provide a model that is specific enough so that it actually offers something to the construction of agents, but general enough to support the development of a wide range of architectures. Through an appropriate architecturally-*neutral* model we can view a range of architectures based on a common understanding of agents and without losing expressive capability.

Modularity The model should allow for modular construction of agents. This is necessary both in order to meet general software engineering concerns and to delineate clearly the different aspects of an architecture. As discussed in Section 2.3, our approach calls for a separation between describing agents in terms of their *characteristics*, their *structure* and their *behaviour*. Such a fine-grained approach can lead to a better understanding of the overall functioning of the agent as well as how it can be altered, since the different aspects of the architecture are clearly identified and the relationships between them made explicit.

Run-time reconfiguration The reality of current computing environments is that changes are often required as the system is running. With large systems that can contain dynamic, complex dependencies between various parts, it is crucial to be able to reconfigure agents at run-time. Reconfiguration may mean providing more functionality to an agent or changing the way it behaves in order to better meet application requirements.

2.2 SMART

The agent construction model is based on SMART [10] (Structured, Modular Agent Relationships and Types), which provides us with the foundational agent concepts that allow us to reason about different types of agents, and the relationships between them, through a single point of view. We chose SMART because it provides us with the appropriate agent concepts without restricting us to a specific agent architecture. Furthermore, SMART has already been successfully used to describe several existing agent architectures and systems (e.g. [8, 9]).

We avoid here a more complete presentation of SMART and focus on just those concepts that are used for the agent construction model. In essence, SMART provides a compositional approach to the description of agents that is based on two primitive concepts, *attributes* and *actions*. Attributes refer to describable features of the environment, while actions can change the environment by adding or removing attributes. Now, an agent is described by a set of attributes and a set of *capabilities*, where capabilities are actions an agent can perform. An agent has *goals*, where goals are sets of attributes that

represent desirable states of the environment for the agent. On top of this basic concept of an agent, SMART adds the concept of an *autonomous agent* as an agent that generates its own goals through *motivations*, which drive the generation of goals. Motivations can be preferences, desires, etc., of an autonomous agent that cause it to produce goals and execute actions in an attempt to achieve those goals.

This approach to agent description fits well with our requirement for architecture neutrality but does not sufficiently address our requirements for modularity and run-time reconfiguration. In Section 2.3 we discuss how the descriptive capabilities of SMART are enhanced to cope with these requirements.

2.3 Description, Structure and Behaviour

While SMART is suitable for *describing* agents, it lacks the necessary features for *constructing* agents. For the purposes of SMART, this was not a problem since the intended purpose was to provide a theoretical framework that would allow the description of a number of different agent systems. However, for our purposes it is crucial to be able to provide tools that facilitate the construction of agent architectures. Nevertheless, we do not want to *replace* the descriptive capabilities of SMART, since they offer some useful features as discussed above. Rather, we complement them with the additional aspects, which are identified below.

SMART allows systems to be specified from an observer's point of view. Agents are described in terms of their attributes, goals or actions, not in terms of how they are built or how they behave. In other words, the focus is on the *what* and not the *why* or *how*. We call this a *descriptive specification*, since this essentially describes a situation without analysing its causes nor the underlying structures that sustain that situation. However, these are just the issues we need to address within a construction model. Therefore, *along* with the descriptive specification we need to have the ability to specify systems based on their structure, i.e. the individual building blocks that make up agents, as well as their behaviour. We call these other views the *structural specification* and the *behavioural specification*, respectively.

The structural specification enables the identification of the relevant building blocks or components of an agent architecture. Different sets of building blocks and different ways of connecting them can enable the instantiation of different agent types. By contrast, the behavioural specification of an agent addresses the process through which the agent arrives at such decisions as what actions to perform. These specifications, along with the descriptive specification, provide a more complete picture of the system from different perspectives. It is interesting to note that it is possible to begin from any one of these views and derive the remaining two, but the correspondence is not one to one. Several behavioural and structural specifications could satisfy a single descriptive specification and *vice-versa*.

For example, let us consider an agent operating on a user's mobile device, whose purpose is to determine what physical devices (e.g. projector, fax machine, laser printer) and information services (e.g. local weather information, maps of the building) are available for use in a conference room, and, based on the current goals of its user, identify the devices that are relevant to the user or query the information services for the required information. A *descriptive specification* of such an agent may state that the

Descriptive Specification	Behavioural Specification	Structural Specification
Attributes: Agent Owner = Ronald Ashri Allowed to Use Devices = True Capabilites: Search for devices Search for infromation services Goals: Find laser printer Get directions to closest restaurant	Step 1: Discover available devices Step 2: Discover available info services Step 3: Evaluate suitability against goals Step 4: Notify user about device Step 5: Get results from info service if one discovered	Active Devices Discover Active Information Services Discovery Devices Evaluation Component Information Services Evaluation Component Information Services Query Component User Notification Component

Fig. 1. Distinguishing between description, structure and behaviour

agent belongs to a user, has certain right with regards to accessing devices and services in the conference room, has the goals of printing an overview of the user's presentation and finding out where the closest restaurant is, and so forth. A *behavioural specification* may state that this agent begins its operation by collecting information about available devices and services, then filters for those that are best suited to the user's goals, and decides whether to inform the user about them or interact directly with them to achieve the goal. A *structural specification* may state that the agent has different components, each able to handle specific functionalities such as searching for Bluetooth-enabled devices, communicating with information services or directory agents, and so forth. The different aspects are illustrated in Figure 1. This separation between the different views allows us to change some aspects without necessarily impacting on the others. For example, the behaviour could be altered so that the agent reports on the first device that fits the required profile, rather than attempting to find the best one. The structural specification could also be based on a tightly integrated control loop rather than a component-based approach.

The agent construction model reflects these levels by allowing direct access to these different aspects of agents, based on a clear decoupling at the architectural level.

2.4 Component-Based Construction

In order to support the division of an architecture's different aspects as described above, and satisfy the requirement for modularity and re-configurability, we take a component-based view of agent architectures.

Component-based software engineering is a relatively new trend in software engineering [7, 11]. Separate developments within the fields object-oriented computing, re-usable software code, formal methods, and modeling languages have all steered research towards a component-based approach [16]. Components are understood as units of composition that can be deployed independently from each other, while a *third-party* coordinates their interactions [18]. Interaction with a component takes place through a well-defined interface, which allows the implementation of the component to vary independently of other aspects of the system.

There are three main benefits to architecture design following a component-based approach, that are in line with our aims. Firstly, having to describe an agent architecture through the composition of components promotes a clearer identification of the

different functionalities and allows for their re-use in alternative contexts. Secondly, different types of components can be composed in a variety of ways to achieve the best results for the architecture at hand. Finally, by connecting the abstract agent model of SMART to component-based software engineering we bring it much closer to practical development concerns within a paradigm that is not foreign to developers.
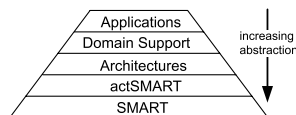
2.5 From SMART+ to Applications

Fig. 2. From SMART+ to applications

In this section we clarify the relationships between the agent construction model which, from now on, we will refer to as *act*SMART, the abstract agent model SMART, and the application level. These clarifications serve to indicate how the work presented here can be used within the context of the agent development process.

The relationships are illustrated in Figure 2. At the most abstract level lies SMART. Then, *act*SMART represents an extension of SMART to deal with the *construction* of agents. Architectures for agents, which can range from application-independent architectures, such as BDI, to application-specific architectures, can thus be designed using the framework provided by *act*SMART, and based on the concepts provided by SMART. We should note that generic architectures are not always required and may not always be advisable. For example, an agent dedicated to dealing with requests for quotes on fast-changing financial information, where performance optimisation is crucial, would benefit from a basic architecture tailored to that situation. Conversely, agents expected to deal with a variety of changing tasks and complex interactions with other agents, such as sophisticated negotiations, might benefit from a more generic and sophisticated deliberative architecture. One of the benefits of our approach is that while it distinguishes between the different cases, it can still consider them within the same conceptual and practical framework.

The next level is domain-specific support, which involves choosing appropriate middleware to support agent discovery and interactions between agents in the specific distributed environments in which the applications will operate, as well as other components that could supplement agent capabilities. Finally, specific applications can be built, making use of all the layers below.

3 Components

The first step for our agent construction model, as discussed above, is to introduce and define *components* as the basic buildings blocks for an agent. These can be thought of

as the structural representations of one or more related agent functionalities, which are considered at two different levels. At an *abstract level*, the functionality is described in generic terms, which we will present shortly. At the *implementation level*, the abstract functionality is instantiated through the actual mechanisms that support it. The reason for distinguishing between these different levels is so that we can use *generic* component types to specify an agent architecture at a high level of abstraction without making direct reference to the detailed behaviour of each component. This allows us to move between the different levels while retaining a good understanding of the overall architecture, and identifying which specific components best suit each of the generic functionalities.
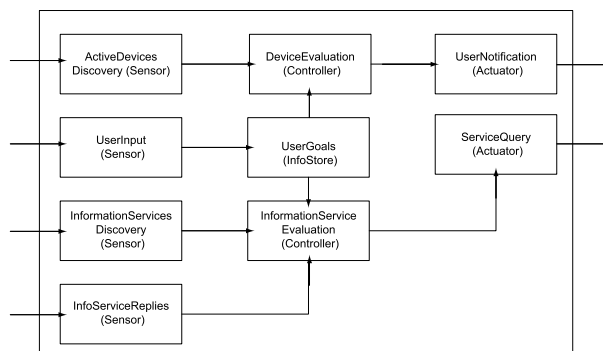

Fig. 3. Example Agent Architecture

Generic Component Types From here on, we set out the terms that can be used to describe components at an *abstract* level. We begin by dividing components into four generic types, each one representing a class of functionality for the agent.

We use the example architecture illustrated in Figure 3 to explain each generic component type. The diagram presents an architecture for an agent, based on the example discussed in Section 2.3.[1] The domain specific functionality of components is as follows. Information about available devices and information services is collected by the *ActiveDevicesDiscovery* and *InformationServicesDiscovery* components. The user provides information through the *UserInput* component, which defines the goals of the user. These goals are stored in the *UserGoals* component. Based on these goals the *DeviceEvaluation* and the *InformationServiceEvaluation* components choose which of the devices and services are relevant to the user. Information services are queried through the *ServiceQuery* component and replies are received by the *InfoServiceReplies* component. Finally, the user is notified about relevant devices and replies from information services through the *UserNotification* component.

---

[1] Note that while this architecture is sufficient for illustrating the agent construction model and how it can benefit agent design we do not claim that it is a complete design for such a type of agent

The *generic* functionality of the components can be divided into information collection (sensors), information storage (infostores), decision-making (controllers) and finally those directly able to effect change in the environment (actuators). These four generic types of components, described in more detail below, can be used to describe a wide range of agent architectures and fit particular well to the context of ubiquitous devices, where there is clear distinction between the external environment to the device and the agent itself.

- *Controllers* are the main decision-making components in an agent. They analyse information, reach decisions as to what action an agent should perform, and delegate those actions to other components.
- *Sensors* are able to sense environmental attributes, such as signals from the user or messages received from other agents. They provide the means through which the agent gains information from the environment.
- *Actuators* cause changes in environmental attributes by performing actions.
- *Infostores* are components whose main task is that of storing information. Such information could be anything from the beliefs of an agent about the world, to plans, to simply a history of the actions an agent has performed or a representation of its current relationships.

Component Statements The *internal* operation and structuring of components, irrespective of their type, is divided into a functionally-specific part and a generic part. In this subsection, we describe the generic part that is common to all components, and outline how and what types of information components can exchange, before describing how that information is processed by a component.

Each component accepts a predefined set of inputs and produces a predefined set of outputs. A component generates an output either as a direct response to an input from another component, a signal from the environment or an internal event. For example, a a sensor component attached to a thermometer may produce an output every five minutes (based on an internal clock), or when the temperature exceeds a certain level (an external signal) or when requested from another component (response to other component).

In *act*SMART, inputs and outputs share a common structure; they are *statements*, which have a *type* and a *body*. The body carries the main information (e.g., an update from a sensor), while the type indicates how the information in the main body should be treated. We make use of three types of statements, described below.

- INFORM-type statements are used when one component simply passes information to another component. In order for one component to inform another of something, it must be able to produce the INFORM-type statement as an output and the other must be able to accept it as an input.
- REQUEST-type statements are used when one component requires a reply from another component. In this case, the receiving component processes the request and produces an INFORM argument that is sent to the requesting component. The mechanisms through which statements are transmitted from one component to an other are introduced in Section 4.

– EXECUTE-type statements are used to instruct another component to execute a specific action. Typically, controller components send such statements to actuators so that changes can be effected in the environment.

The information within a statement's body is, in its most general form, is described through *attributes*, as per the definitions given in Section 2.2. Attributes can be divided along the lines of *architecture-specific* attributes and *domain-specific* attributes. Architecture-specific attributes are attributes that are only relevant within the internal scope of an agent architecture. For example, a BDI-based architecture could define attributes such as *plans*, *beliefs*, *intentions* and so forth. [2] Architecture-specific attributes can be though of as defining the *internal* environment of an agent. Domain-specific attributes define features that are relevant to the environment within which the agent is operating. So in the case of the example above it may include features such as *device name*, *location*, and so forth. Generic agent architectures, such as BDI-based will typically make use of both types of attributes, including domain specific attributes *within* the architecture-specific attributes. So a *plan* may prescribe an action to contact a service, as identified by its *service name*. The components of an AgentSpeak(L) architecture could then manipulate *plans* and *beliefs*, and have some generic mechanisms to manipulate the *domain-specific* attributes. However, a developer may also choose to develop an agent that has no architecture-specific attributes, creating components that can directly manipulate domain-specific attributes.
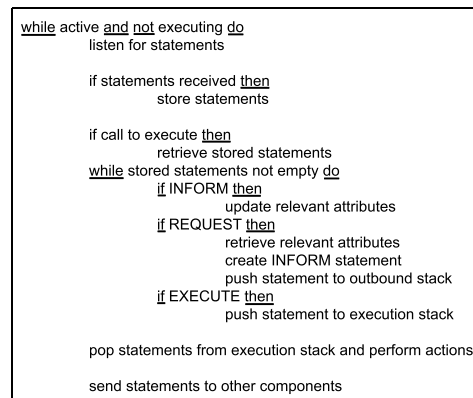
```
while active and not executing do
        listen for statements

    if statements received then
            store statements

    if call to execute then
            retrieve stored statements
    while stored statements not empty do
            if INFORM then
                    update relevant attributes
            if REQUEST then
                    retrieve relevant attributes
                    create INFORM statement
                    push statement to outbound stack
            if EXECUTE then
                    push statement to execution stack

    pop statements from execution stack and perform actions

    send statements to other components
```

Fig. 4. Component Lifecycle

Component Operation An outline of the component operation is shown in Figure 4. Components begin their operation in an inactive state, but once activated they perform any relevant initialisation procedures and wait for receipt of statements or for the command to execute. When a statement is received, it is stored within the component until

---

[2] This approach was followed by d'Inverno and Luck when formalising AgentSpeak(L) [9].

the component enters its component execution phase. At this point, all statements received by a component are processed. The processing of statements may result in the component performing a set of actions or firing statements itself. This process stops when the component is de-activated.

The reason that components store statements instead of dealing with them immediately is that components can be made to react immediately after each statement is received through external control, as we discuss in the next section. In other words, the behaviour of a component within the context of the entire architecture (i.e., when it acts or sends messages) depends on an external controller.

The state of a component, in terms of the information to be manipulated, at any given time is given by the set of statements that have not been processed yet, the set of statements in the execution stack, the set of statements in the outbound stack and any attributes that the component manipulates. Depending on the specific implementation of a component it may be possible to interrogate components for their individual states. We discuss this issue further in Section 5.

With components, we are able to differentiate between the different tasks an agent architecture needs to perform, from a structural perspective. In contrast, the composition of components and the control of the flow of information between them provides the behavioural specification. In the next section we see how this is managed.
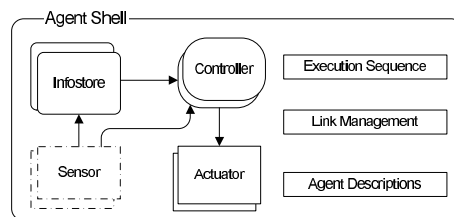
4 Shell

Fig. 5. Agent shell

As discussed in Section 2.4, the basic principles of a component-based approach is that components should be independent of each other and the coordination between them should be handled by a third-party. The design of the components presented above achieves independence by defining the interface of the components so that interaction is reduced to the exchange of statements, with no consideration as to where a statement arrives from. The third-party coordination is achieved by placing components within a *shell*, which acts as the *third party* that manages the sequence in which components execute and the flow of information between components. This management takes place by defining *links* between components and the *execution sequence* of components. The basic aspects of a shell are illustrated in Figure 5. We use different representations for the

different types of components in order to aid the illustration of agent architectures. *Sensors* are dashed rectangles, *infostores* are rounded corner rectangles, *actuators* are continuous line rectangles, and *controllers* are accented rounded corner rectangles. Components are placed within a shell, links are created between components to allow the flow of statements, and an execution sequence is defined. In addition, the shell can be used to maintain *descriptions* of agents in terms of attributes, capabilities and goals. We consider each of these aspects in more detail below.

Links Information flows through *links* that the shell establishes between components. Each link contains *paths* from a *statement-producing* component to the *statement-receiving* components. Each component that produces statements has a link associated to it that defines the components that should receive those statements. Links also ensure that in the case of a REQUEST statement the reply is sent to the component that produced the request. Thus, links manage paths which are one-to-one relationships between components. They are usually unidirectional, except in the case of a REQUEST statement for which an INFORM may be returned in the opposite direction.

The shell then uses the information within links to coordinate the flow of statements between components. Ultimately, this coordination depends on the choices that a developer makes, since it requires knowledge of each component and how they can be composed.

By decoupling the handling of statements between components from the components themselves, we gain considerable flexibility. We can manage the composition of components and the flow of information without the components themselves needing to be aware of each other. It is the architecture developer's task to ensure that the appropriate links are in place. At the same time, we have flexibility in altering links, and it becomes easier to introduce new components. Furthermore, basic transformations can be performed on a statement from one component to the other to ensure compatibility if the output of one component does not exactly match the required input for another. For example, if a sensor component provides information from a thermometer based on the Celsius scale while a controller that uses that information makes use of the Fahrenheit scale, the link can be programmed to perform the necessary transformation. These features satisfy our requirement for facilitating the re-configuration of architectures.

Execution Sequence Apart from the management of the flow of information, we also need to consider the execution of components for a complete view of agent behaviour. This is defined via an *execution sequence* that is managed by the shell. Execution of a component includes the processing of statements received, the dispatch of statements and the performance of any other actions that are required. The execution sequence is an essential part of most agent architectures and, by placing the responsibility of managing the sequence within the shell, we can easily reconfigure it at any point during the operation of the agent. For many architectures this may be purely sequential, but there are cases in which concurrent execution of components is desired (e.g., the DECAF architecture is based on a fully concurrent execution of *all* components [13]). In general, the issue of supporting complex execution sequence constructs, such as conditional paths and loops, is considered to be an issue that goes beyond the scope of this research, and

there is a wealth of existing research that can be accessed to address this need. For example, recent developments within the field of Semantic Web Services provide a process model language for describing the operation of a web service [1]. Through our proposed mechanisms we facilitate the necessary separation of concerns to enable the integration of such work within the scope of agent architecture development. In Section 5, where we present an implementation of *act*SMART we only implement a sequential execution sequence, which is sufficient for our current purposes.

Agent Description The description of the agent as a whole is maintained by the shell. The shell can store a number of attributes that describe the agent owner, its location, user preferences, etc. The level of detail covered by this description is mostly an application-specific issue, and this information can either be provided directly to the shell by the developer, or collected from the various components. The shell could query a component that is able to provide information about current location for example, and add that to the profile of the whole agent. Likewise, it may keep a record of the current goal an agent is trying to satisfy or the plan it is pursuing. The capabilities collecting and providing attributes describing the agent within the shell may be particularly useful in a situation in which a developer wants to export a view of the agent for debugging purposes, or when some information needs to be advertised, such as the agent's capabilities.

Agent Design With the main aspects of the agent construction model in place we now briefly describe how the agent design process can develop. Agent design begins with an empty shell. One could envisage implementations of shells being provided by environment owners, which would ensure compatibility with their environment, while allowing the agent developer relative freedom as to the structure and behaviour of the agent within the confines of the shell. Subsequently, based on the purpose the agent is meant to fulfill, the necessary components for sensing, acting, decision-making (controllers) and information storage can be identified. If such components already exist the main task of the developer is to decide what is the appropriate behaviour, in terms of execution sequence and flow of information, and whether any of the output of components need to be transformed in order to be aligned with the input needs of other components.

    The components are then loaded into the shell, and links, as well as an execution sequence, can be defined. With the execution sequence in place, the operational cycle of the agent can begin. Agent operation can be suspended or stopped by stopping the execution sequence. This operational cycle can be modified by altering the execution sequence and modifying links between components.

5 Implemenation of *act*SMART

In order to evaluate the viability of the agent construction model, we have developed an implementation of the ideas described above in Java. The resulting toolkit consists of a core set of applications programming interfaces (APIs) that represent the basic code required for defining a shell, components and links between components. This core has

been programmed using solely classes supported within the Mobile Information Device Profile (MIDP) of the Java 2 Micro Edition [14]. As such, the core ideas can be used by wide range of devices, from workstations to limited capability mobile phones.

We then provide two extensions to the core, one for desktop computers and one for mobile devices, respectively. The *desk* part provides enhancements to the core, such as a graphical user interface for building entities and run-time loading of components, that are not possible for limited capability devices. Furthermore, in order to speed up the development process, one can define the required components, attributes, links and execution sequence within an XML file and use that to create an agent in the desktop environment. The *mobile* part provides functionality which is specific to mobile devices such as permanent record stores and user interfaces for mobile devices. In both cases we can manage the information flow between components at run-time as well as change the execution sequence within the limits of the types of execution sequence that we currently support.

The user interfaces provided with *act*SMART, for both the mobile and desktop devices, allow direct access to all the relevant information on the state and operation of an agent as well as the capability to manipulate each agent as a whole or individual components. As such, they can serve as the basis for effective debugging tool for agents as well as a means to manipulate and change agent configurations during run-time.

5.1 Implementing an architecture

In order to illustrate some of the benefits of this approach for mobile devices we have implemented the architecture and scenario described in Section 3 for a MIDP 2.0 compliant device using the J2ME Wireless Toolkit, which provides an environment that can simulate various mobile devices and the operation of MIDP applications within them. [3]. The discovery of available services and devices is simulated, with the mobile device using the APIs provided by MIDP 2.0 (Generic Connection Framework) for network communication and communicating via TCP/IP with independent processes that provide the relevant information. We avoid here a detailed descriptions of the exact statements exchanged between components, due to a lack of space, and focus on some of the specific implementation issues for mobile devices.

Adapting to changes in the environment Mobile devices are inevitably limited in their capabilities. However, pervasive environments present a constantly changing set of devices and services to interact with as well as modes of interactions. For example, a device may be able to communicate with other devices through a variety of low-level protocols such as 802.11b wireless, Bluetooth or even SMS messages as well as higher level agent language communication protocols. By abstracting interaction using different methods to dedicated sensor components we can change the methods used at runtime based on device capabilities. For example, upon initialisation a shell could determine if a device supports Bluetooth communication and accordingly activate and link the Bluetooth-enabled sensor component. Similarly a device can determine that a certain protocol, although supported by the device, is not supported by anything else in the

---

[3] http://www.java.sun.com/j2me

environment, and by consequence the relevant component is unlinked and deactivated by the shell, thus minimising the load the agent places on the device.

Suspending operation A particularly useful feature of *act*SMART was the easy access it provides to the state of individual components and the agent as a whole. This allowed us to *suspend* the operation of the agent either through a user command or when the device was interrupted (e.g. by a phone call). The MIDP application management model allowed us to write the relevant information to the persistent record store of a device, thus allowing us to resume operation from where we left off.
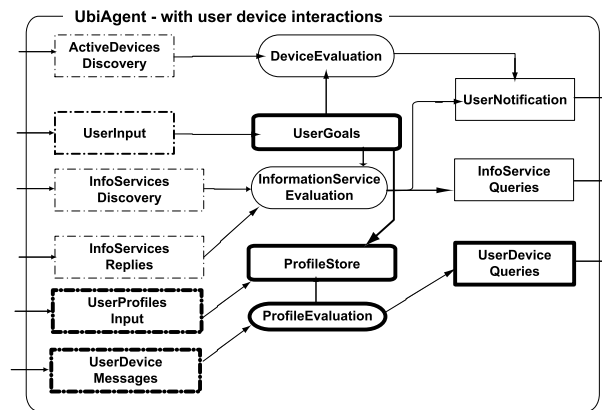
Modifying the architecture

Fig. 6. Extended Ubiquitous Agent Design

Finally, through the mechanisms provided by Java mobile device technologies, and particularly,*over-the-air* provisioning of MIDP applications we can take advantage of the flexibility afforded by *act*SMART to replace existing architectures with modified versions that they can support more functionality. For example, in Figure 6 the basic architecture is extended to support interaction with other user devices that can provide profiles of their owners. The bold components are the additional or modified components over the existing architecture. Essentially, the user is able to input another goal, which is handled by the new components to store, evaluate and sent profiles to other devices.

5.2 Discussion

The implementation of the architecture in *act*SMART has provided useful experience as to the suitability of the model for agent construction in a ubiquitous computing environment setting. Although the implementation of interactions with other sources was

based on a simulation of the environment, the APIs used are those directly supported by the majority of high-end mobile phone devices and we also tested the application on low-end PDAs (Palm m100).

The fine-grained control over every aspect of the agent aids significantly in testing and debugging, since components can be tested individually and, more importantly, they can be tested in connection with other components without requiring an instantiation of the entire architecture. Moreover, the state of each component, and the agent as a whole, is clearly defined, and changes to individual components and to the overall architecture are easy to achieve.

6 Conclusions

A component-based approach to agent design is, of course, not unique. The majority of current agent toolkits (e.g. [19, 15, 3, 17]) support some sort of component-based approach. However, they do not explicitly support the definition of a *range* of architectures, with the exception of JADE [3] that only provides basic support for agent architectures and as such does not constrain the developer. However, at the same time it does not aid the developer by providing a conceptually grounded construction model, such as *act*SMART. In fact, a future goal is to attempt to incorporate our agent construction model within the wider development environment of JADE, which also provides support for mobile devices.

Methodologies for agent development have also introduced similar notions. The most relevant example is the DESIRE design method [5], where compositional design is not simply a low-level software engineering issue but a methodological approach. Components, other that at the lowest level are seen as encapsulating processes and composition of components is, therefore, a composition of processes. Our approach should not be considered as an alternative to DESIRE, but rather complimentary since it can be seen as *restricting* many of the concepts already supported within DESIRE. We provide a more lightweight approach to component-based approach that is far more specific route in the nature of components and the ways that they can communicate. Our approach has been developed with ubiquitous computing devices in mind, so the concepts map particularly well to them, and ease of application in practical development environments so that the agent construction model is easily understood and maps directly to implementation. Using this more *lightweight* approach we are able to describe very simple architectures, such as just a reactive agent [4], to more complex architectures [5], while retaining a clear separation between an abstract specification level and practical implementation.

The approach offers several real contributions to the state-of-the-art as follows: it addresses development concerns directly by providing clear links between conceptual models and implementation; it builds on existing theoretical work while adopting the most valuable practical developments; and it tackles some of the implementation concerns of mobile devices and is supported by a sound foundational approach that is easily realisable in practice.

---

[4] In our case a reactive architecture would consist of just sensors linked directly to actuators.

[5] We have used the same approach to describe complex negotiating agent architectures [2].

References

1. A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, S. A. M. Drew McDermott, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In I. F. Cruz, S. Decker, J. Euzenat, and D. L. McGuinness, editors, *The First Semantic Web Working Symposium*, pages 411–430. Stanford University, California, 2001.

2. R. Ashri, I. Rahwan, and M. Luck. Architectures for Negotiating Agents. In V. Marik, J. Muller, and M. Pechoucek, editors, *Mutli-Agent Systems and Applications III*, volume 2691 of *LNAI*, pages 136–146. Springer, 2003.

3. F. Bellifemine, A. Poggi, and G. Rimassa. Developing Multi-agent Systems with JADE. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories Architectures and Languages*, volume 1986 of *LNCS*, pages 89–103. Springer, 2001.

4. F. Bergenti and A. Poggi. Ubiquitous Information Agents. *International Journal of Cooperative Information Systems*, 11(3–4):231–244, 2002.

5. F. T. Brazier, C. Jonker, and J. Treur. Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, 41:1–28, 2002.

6. J. Bryson and L. A. Stein. Architectures and Idioms: Making Progress in Agent Design. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories Architectures and Languages*, volume 1986, pages 73–88. Springer, 2001.

7. J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying COmponent-Based Software*. Addison-Wesley, 2000.

8. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *Intelligent Agenrs IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, volume 1365 of *LNCS*, pages 155–176. Springer, 1996.

9. M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A Formal Computational Model. *Journal of Logic and Computation*, 8(3):233–260, 1998.

10. M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer, 2nd edition, 2004.

11. D. D'Souza and A. Wills. *Objects Components and Frameowrks with UML*. Addison-Wesley, 1998.

12. T. Finin, A. Joshi, L. kagal, O. V. Patsimor, S. Avancha, V. Korolev, H. Chen, F. Perich, and R. S. Cost. Intelligent Agents for Mobile and Embedded Devices. *International Journal of Cooperative Information Systems*, 11(3–4):205–230, 2002.

13. J. Graham and K. Decker. Towards a Distributed Environment-Centered Agent Framework. In N. Jennings and Y. Lesperance, editors, *Intelligent Agents VI Agent Theories, Architectures, and Languages*, volume 1757 of *LNCS*. Springer, 1999.

14. J. . E. Group. Mobile Information Device Profile for the Java 2 Micro Edition - Version 2.0. Technical report, Java Community Press, 2002.

15. H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. *Applied Artifical Intelligence*, 13(1):129–186, 1999.

16. I. Srnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Specification, Implementation and Deployment of Components. *Communications of the ACM*, 45(10):35–40, 2002.

17. V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.

18. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

19. T. Wagner and B. Horling. The Struggle for Reuse and Domain Independence: Research with TAEMS, TDTC and JAF. In T. Wagner and O. Rana, editors, *Infrastructure for Agents, MAS and scalable MAS, Workshop in Autonomous Agents 2001*, pages 17–23, 2001.