Supporting Tasks in a Programmable Smart Home

Anand Ranganathan, Roy H. Campbell
University of Illinois at Urbana-Champaign
{ranganat, rhc}@uiuc.edu

Abstract. One of the most promising domains of pervasive computing is in the area of healthcare. There has been a lot of interest in seeing how smart homes can foster independent living and an enhanced life style for elderly and disabled people, and how active, information-rich hospitals can support both patients and the medical staff in their daily activities. However, one of the key challenges in the area of smart homes is programmability – each smart home needs to support different kinds of tasks depending on the requirements of the inhabitants, and hence, needs to be configured and programmed appropriately. It is not scalable to build one-off, customized architectures and applications for different smart homes. Hence, we need a highly configurable architecture that allows the rapid development and customization of smart homes for different kinds of tasks. In this paper, we discuss the challenge and complexities of programming smart homes. We also propose a solution that allows developing high-level parameterized tasks that support users in their daily lives. The solution includes an autonomic task execution framework that automatically configures and repairs the execution of tasks depending on the current state of the environment, context-sensitive policies, and learned user preferences.

Keywords. Smart Home, Pervasive Computing, Autonomic Computing, Programmability

1. Introduction

Pervasive computing advocates the enhancement of physical spaces with computing and communication resources that help users perform various kinds of tasks. One particular area of interest has been in augmenting smart homes and hospitals with various kinds of devices, services and applications that enhance healthcare for the elderly and disabled. Various projects have looked at different aspects of this area such as computer-supported coordinated care[1], the use of aware technologies to help elders age in place[2] and the use of sensors to monitor and aid inhabitants[3,4].

The common theme behind many of the projects is the use of technology to support inhabitants while performing tasks. However, it is often difficult to predict, in advance, the set of tasks that must be supported by a smart home. It may also not be clear how exactly these tasks should be performed or supported in the smart home. The area of smart homes for the elderly is still in a nascent and growing phase, and researchers are still in the process of figuring out how best pervasive computing can enhance the lives of elders. Hence, smart homes must have a highly extensible and configurable underlying infrastructure that both aids researchers in understanding how technologies can be used to aid smart homes, and also supports the inhabitants and their

support networks in their daily lives. Since we often cannot anticipate the needs and requirements of users in a smart home, we must ensure that the infrastructure is flexible enough to be molded to their needs.

Thus, a key requirement of smart homes is programmability. Even though we may not know what are the killer applications for smart homes, we should be able to develop and deploy different applications and scenarios, and test their effectiveness. The infrastructure of smart homes must allow developers and administrators to specify the behavior of the smart home at a high-level. They should be able to rapidly specify how the smart home should support different kinds of tasks using different kinds of devices, services and applications. They should not be bogged down by the numerous low-level details that are inherent in any complex distributed system. This will allow them to configure the smart home to the specific requirements of the inhabitants quickly. This may also aid the discovery of killer smart home applications.

An important characteristic of many old-age homes, assisted living facilities and hospitals is that the tasks that are performed in these environments are often in the form of workflows or processes that require one or more people to perform a sequence of actions using various resources in order to achieve a goal. As an example, if a patient is getting a flu shot, there is normally a fixed set of actions that need to be performed before, during and after the flu shot. The patient may first get his temperature checked. If it is less than a certain minimum value (say 97 F) or more than a certain maximum value (say 101 F), then the patient may be directed to a doctor who decides whether it is all right for the patient to still get the flu shot or diagnose any other problem. If the temperature is in the acceptable range, the patient may be directed to a nurse who gives him the flu shot. After the flu shot is given, the patient may be placed under observation for a few minutes to make sure that there are no adverse reactions.

Hence, any pervasive computing infrastructure for smart homes must be able to support such tasks that involve a sequence of steps following a certain control flow. Each action in such a flow may be enhanced with the help of various devices and services. For example, if the temperature is found to be above the maximum value, the nurse measuring the temperature accesses her handheld to find an appropriate doctor to refer the patient to and the location of the doctor. The doctor who has been discovered and assigned the task of examining this patient gets notified on his handheld or cellphone about the patient. He is also able to pull up records about the patient and can decide the location or office where he wants to meet the patient. He sends this location to the nurse and the patient then makes his way to the doctor.

In order for the smart home to support such processes, they need to be given a description of the flow of the process. Such descriptions are normally codified in various documents that describe the rules and procedures of the facility or the hospital. The pervasive computing system must be able to take the descriptions of the processes and decide the best way of supporting the processes depending on the people involved, the current context of the environment and the kind of process. In our research, we have tried to support such processes by describing them in the form of high-level, parameterized tasks.

1.1. High-level, Parameterized Tasks for Smart Homes

We have developed a framework that allows developers and administrators to program and configure pervasive computing environments in terms of high-level, parameterized tasks. A task is a set of actions performed collaboratively by humans and the pervasive

computing system to achieve a goal. The set of tasks that are deployed in a certain environment depends on the nature of the environment. For example, common tasks in a smart home may be preparing food, taking medications, performing leisure activities (like playing music or viewing movies) and locking or unlocking different doors n the house. Common tasks in a hospital include checking-in patients, monitoring the condition of patients, performing tests and supporting collaborations between doctors, nurses, lab technicians and other people. Common tasks in a smart conference room may include displaying slideshows, giving lectures, collaborating with others, and migrating applications between devices and across different rooms.

A task is made up of a number of smaller sub-tasks called activities. Developers first develop basic activities using a high-level programming model called Olympus[5]. These activities include operations such as starting, moving or stopping components, changing the state of devices, services or applications or interacting with the end-user in various ways. They then develop programs or workflows that compose a number of activities into a task that achieves a certain goal. These programs or workflows essentially mirror common processes in the home or hospital, augmented with instructions or recommendations on how the pervasive computing environment can enhance the performance of the individual activities in the process.

An important feature of tasks is that they are very flexible. The task program specifies the actions that must be performed by the pervasive computing environment and by the user at a very high level. The pervasive computing system decides exactly how the actions are to be performed – i.e. which resources (devices, services, applications, locations, etc.) to use or which algorithms to employ. Tasks are written in a high-level manner, in terms of abstract resources or parameters and are not strongly tied into the characteristics of any particular environment. This enhances portability, and the same task can run unchanged in different environments.

In our framework, flexibility of task execution is achieved by parameterizing tasks. Each task is associated with one or more parameters that influence how it is performed. The parameters may be devices, services or applications to employ while performing the task or they may be strategies or algorithms to use. For example, even the relatively simple task of displaying a slideshow has a number of parameters like the devices and applications to use for displaying and navigating the slides, the name of the file, etc. When the task is executed, the middleware obtains the values of the different parameters by either asking the end-user or by automatically deducing the best value of the parameter based on constraints specified by the developer, the current state of the environment, context-sensitive policies, and user preferences. It can also recover from failures of one or more actions by using alternative resources.

2. Task Model

We define a task as a set of actions performed collaboratively by humans and the pervasive computing system to achieve a goal. A task is made up of a number of sub-tasks, or activities, that have been composed together in a workflow.

As an example, Fig 1 shows the different steps that may be involved in the task of taking a medication, represented as a flowchart. In our framework, these tasks are currently, written either in C++ or in a scripting language called Lua[6], though we are also working on graphical interfaces for building these workflow charts.

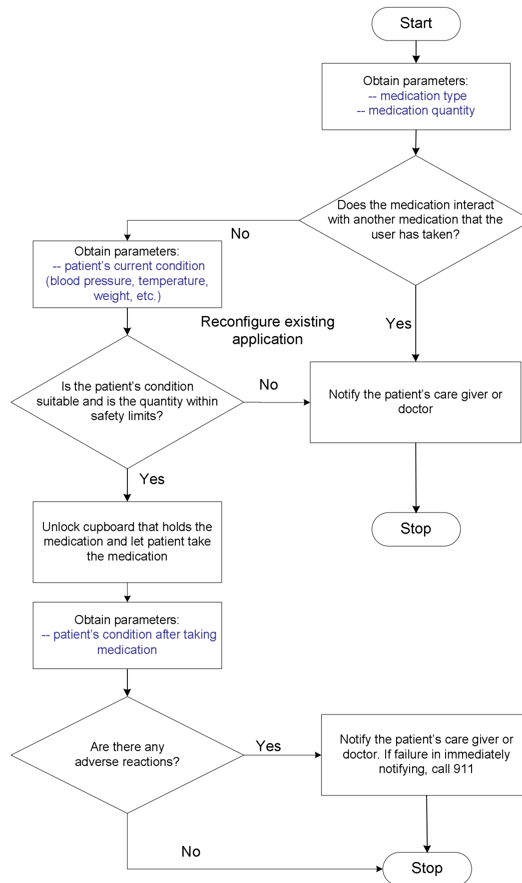

Fig 1. Flowchart for a medication task

The various parameters may be obtained by either asking the end-user directly (e.g. on a GUI in a touch screen or by speech) or by automatically obtaining the value of the parameter (by either querying a sensor or inferring the value based on various policies or the current context). The framework also decides the best way of performing the different actions in the task. For example, the actions of notifying a doctor or a caregiver is adapted depending on their current context – the framework chooses an appropriate device and notification mechanism depending on the location, activity and preferences of the notifiee. The framework can also support actions that have to be done by the patient. For example, the framework can monitor which medicines the patient takes out of the medicine cupboard (using RFID tags, for instance) and instruct him on how to take the medication using speech or by displaying the instructions on a GUI.

Tasks may be initiated either by an end-user or automatically by the framework in response to an event. The framework also decides how best to interact with the end-user automatically (e.g. using a handheld device or a touch-screen or by speech, etc.).

2.1. Activities in a Task

Tasks in our model are made up of reusable activities or sub-tasks, that can be recombined in different manners. Different tasks often have common or similar activities; hence it is easy to develop new tasks by reusing activities that have already been programmed.

There are three kinds of activities: parameter-obtaining, state-gathering and world-altering (Fig 2). Parameter-obtaining activities involve getting values of parameters by either asking the end-user or by automatically deducing the best value. State-gathering activities involve querying other services or databases for the current state of the environment. World-altering activities change the state of the environment by creating, re-configuring, moving or destroying other entities like applications and services.

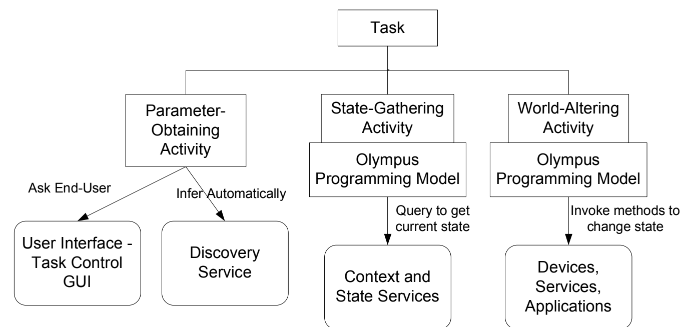

Figure 2. Task Structure

In parameter-obtaining activities, developers list various parameters that must be either obtained from the end-user or inferred by the middleware. For example, the quantity of medication to be taken by the patient may either be decided by the end-user or inferred by the framework based on the current state of the patient and rules that specify optimal amounts of medication. The descriptions of these parameters are in a task parameter XML file. This file lists the kinds of parameters and constraints that the value of the parameter must satisfy. For example, the developer can say that the patient must not take more than 5 tablets of a certain medicine. While it may not be possible to physically prevent the patient from taking more than 5 tablets, the framework can inform him of the maximum recommended number of tablets and automatically call 911 if it senses that the patient has taken an over-dosage (by say, measuring the weight of the medicine bottle)..

In the case of parameters whose values must be deduced automatically, a Discovery Service is queried to get the best value. The Discovery Service has access to different ontologies and policies that specify properties of different entities and have rules that constrain the values of different parameters.

World-altering and state-gathering activities are written in the form of C++ functions. These activities are developed using the high-level operands and operators provided by the Olympus programming model.

2.2. The Olympus Programming Model

The Olympus programming model allows developers to write programs that consist of high-level operators and operands. The underlying framework resolves the high-level operators and operands into suitable low-level implementations and entities.

High-level operands in Olympus are Active Space entities including services, applications, devices, physical objects, locations, users and Active Spaces. Each of these basic types of operands is associated with a hierarchy in the ontology and developers can use any sub-concept of these basic types while programming. The Olympus framework takes care of resolving these abstract, high-level operands into actual Active Space entities based on constraints specified by the developer, the resources available in the current space, policies and the current context of the space. High-level operands may be either passed as parameters into the activity or may be declared locally within the activity. For example, the following world-altering activity migrates a suspended slideshow application belonging to a certain user to a given device:

```
void migrateSlideshowApp(User user1, Device device1) {
    App app1;
    app1.hasClass("SlideShowApp");
    app1.hasProp("status","suspended");
    app1.instantiate();

    app1.resume(device1);
}
```

In the above example, `app1` is a local high-level operand while `user1` and`device1` are high-level operand parameters. The `resume` function is a high-level \operator that starts a suspended application on a certain device. Such parameterized activities form the basic elements of a task. In the above example, the `migrateSlideshowApp` activity may be part of a larger task, for example a task that enables a slideshow application to follow a user as he moves around a building. The parameters `user1` and `device1` are parameters of the task and may be discovered in a previous parameter-obtaining step. In this case, the class and properties of the parameter are specified externally in the task parameter XML file. In this XML file the developer can specify constraints on the choice of the parameter value –e.g. he can specify that `user1` should have the role of "student" or "presenter" and that that `device1` should be of class *VisualOutput* (which is a class in the device ontology), and have a resolution of 800*600. He can also declare a metric for ranking different candidate devices – e.g. he can give location as a metric so that the one that is closest to `user1` is picked.

High-level operators in Olympus operate on the high level operands (i.e. the different Active Space entities) to do one of the following:

1. Manage the lifecycle of Active Space entities. These include operators for starting, stopping, suspending applications and other entities

2. Query or change the state of Active Space entities. For example, the state of a light may be on, off or dim. High-level operators allow querying or changing the state of the light. Another such high-level operator exists for notifying a user of some information (and hence changing his state by adding that information to his knowledge).

3. Query or change the relationships between Active Space entities. For example, operators exist to query the relationship between a user and a location (e.g. where is a

user?), or for changing the relationship between a device and an application (e.g. move an application to a different device), etc.

High-level operators help abstract away low-level implementation details from developers. Hence, developers do not have to worry about how operations are performed in a specific environment and the same program can run in different environments, as long as they both support the same high-level operators. The basic high-level operators provided by Olympus form a set of primitive activities of our task model. In addition, developers can develop other activities by reusing these operators along with other high-level operands.

2.3. Composing Activities to form Tasks

The different activities are composed together to create a task. The control flow piecing together the different activities is specified either in C++ or in a scripting language called Lua[6]. Hence, a task looks like (and has the same expressive power as) any C++ program or Lua script, except that the variables in the program correspond to classes in the ontology and the functions are activities that have been defined using the Olympus high-level programming model. The parameters of the task are declared in the external parameter XML file.

The composite model of a task consisting of several activities borrows from models of workflows like BPEL[7]. BPEL has different kinds of primitive activities like invoking web services or throwing exceptions. These primitive activities can also be combined and executed in sequence, in a loop, or in parallel. However, the key difference between our task model and other workflow models is that our tasks are more dynamic. Our middleware can configure how the task is executed by picking appropriate vales of parameters depending on the current context and the end-user. In BPEL, however, the web services used and the pattern of interaction are pre-specified statically and are difficult to adapt to different situations.

Our task model, like other workflow models, has the advantage that developers and administrators can change the control flow or pattern of interaction easily without having to change the underlying components involved (the different activities, services and applications). It also improves the scalability of task development and prototyping. This is because activities and tasks can be composed; portions of tasks and user interactions can be developed and tested independently.

3. Architecture

Fig 3 shows the overall architecture of our framework for developing and and executing tasks. Developers program tasks with the help of the Olympus programming model. The task programs are sent to a Task Execution Service, which executes the tasks by invoking the appropriate services and applications. The Task Execution Service may interact with end-users through a Task Control GUI to fetch parameter choices, give instructions and provide feedback regarding the execution of the task. It also fetches possible values of parameters from the Discovery Service, which has information about the current state of the environment and policies in Prolog that help inferring appropriate and best ways of performing different kinds of actions. The Ontology Service maintains ontologies defining different kinds of task parameters, contexts and other entities.

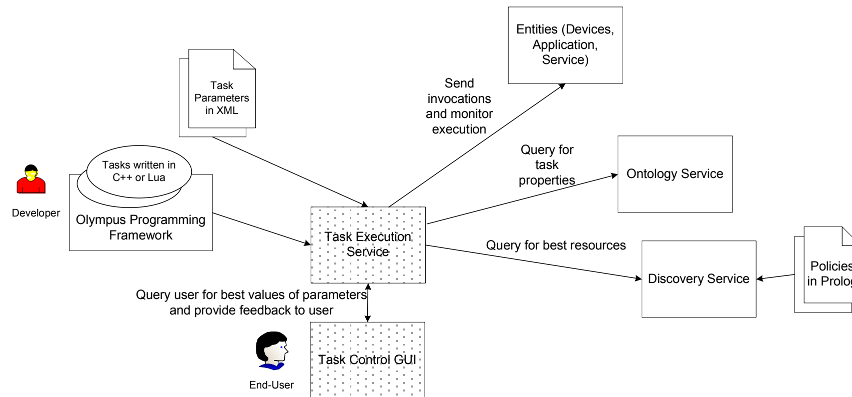

Figure 3. Architecture of the Middleware

4. Implementation and Conclusions

We have implemented the middleware as part of Gaia[8], our infrastructure for pervasive computing. The various services such as the Task Execution Service, the Discovery Service and the Ontology Service are implemented as CORBA services. Ontologies in OWL were developed using Protégé. Reasoning over the ontologies was done using Jena and the Prolog policies were evaluated using XSB Prolog[21]. We have built a number of tasks on top of this middleware. We have built various sample tasks include displaying slideshows, playing music, notifying and communicating with users, and collaboratively working with others on a document, a spreadsheet or a drawing. These tasks are primarily meant for smart conference rooms and offices. However, we believe that the framework can also be used to support tasks in smart homes and hospitals.

5. References

[1] S. Consolvo, P. Roessler, B.E. Shelton, A. LaMarca, A., B. Schilit, S. Bly, “Technology for care networks of elders”, in IEEE Pervasive Computing, pp 22- 29, Apr-Jun 2004, Vol 3, No.2

[2] E.D. Mynatt,, A.S. Melenhorst, A.D. Fisk, W.A. Rogers, “Aware technologies for aging in place: understanding user needs and attitudes”, in IEEE Pervasive Computing, pp 36- 41, Apr-Jun 2004, Vol 3, No.2

[3] D. A. Ross, “Cyber crumbs for successful aging with vision loss”, in IEEE Pervasive Computing, pp 30- 35, Apr-Jun 2004, Vol 3, No.2

[4] N. Johnson, A. Sixmith, “A smart sensor to detect the falls of the elderly”, in IEEE Pervasive Computing, pp 42-47, Apr-Jun 2004, Vol 3, No.2

[5] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R.H.Campbell, D. Mickunas, “Olympus: A High-Level Programming Model for Pervasive Computing Environments,” in IEEE PerCom 2005, Kauai Island, Hawaii, 2005

[6] R. Ierusalimschy et al, “Lua: An Extensible Extension Language,” Software: Practice and Experience Journal., Vol 26, No. 6, 1996, pp 635-652.

[7] "Business Process Execution Language for Web Services Version 1.0," BEA, IBM and Microsoft, August 2002: http://www-106.ibm.com/developerworks/library/ws-bpel/

[8] M. Roman, C.K.Hess, R.Cerqueira, A. Ranganathan, R.H. Campbell, K.Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces," IEEE Pervasive Computing Magazine, vol. 1, pp. 74-83, 2002