Available online at www.sciencedirect.com

ScienceDirect

Pervasive and Mobile Computing 4 (2008) 481–505

pervasive and mobile computing

www.elsevier.com/locate/pmc

Composing software services in the pervasive computing environment: Languages or APIs?[☆]

Jon Robinson[*], Ian Wakeman, Dan Chalmers

Department of Informatics, University of Sussex, Brighton, UK

Received 30 October 2006; received in revised form 7 December 2007; accepted 2 January 2008
Available online 6 January 2008

Abstract

The pervasive computing environment will be composed of heterogeneous services. In this work, we have explored how a domain specific language for service composition can be implemented to capture the common design patterns for service composition, yet still retain a comparable performance to other systems written in mainstream languages such as Java. In particular, we have proposed the use of the method delegation design pattern, the resolution of service bindings through the use of dynamically adjustable characteristics and the late binding of services as key features in simplifying the service composition task. These are realised through the Scooby language, and the approach is compared to the use of APIs to define adaptable services.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Pervasive computing; Programming languages; Performance evaluation

1. Introduction

In this paper we present the Scooby Service Composition system.[1] The main contribution of this research is a service composition language for pervasive computing

[☆] Funded under the EPSRC projects Nathab GR/S26408/01 and TrustUs GR/S69016/01.
[*] Corresponding author.
E-mail addresses: j.r.robinson@sussex.ac.uk (J. Robinson), ianw@sussex.ac.uk (I. Wakeman), d.chalmers@sussex.ac.uk (D. Chalmers).
URL: http://www.informatics.sussex.ac.uk/softsys/ (J. Robinson).
[1] Scooby is our acronym for *Service Composition Objects Ordered By You.*

1574-1192/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.pmcj.2008.01.001

environments. Past research in distributed systems suggests that separating service and configuration is a viable approach [1,2], but pervasive computing provides a new set of challenges because of connectivity, the need to take account of context in choosing services and the dynamicity of the services within any pervasive computing environment. Scooby attempts to provide programming abstractions and design patterns that make it easy for programmers to compose new services that are context-aware, and provides a platform to explore the following questions: *how do we combine services which have been produced by different developers?* and *How can we compose services to meet the demands of users?*.

1.1. Our research goals

One of the goals of our research has been to determine if the coupling of a domain specific language and middleware is an effective way to enable programmers to create services and compositions. In Fig. 1 we outline a basic Scooby service which provides the ability to route messages to different devices depending on their co-location with the user. The current user's location is used to dynamically rebind to the various output services. When the stock service provides a notification the event handling code routes this to the most appropriate output at that time. The example highlights the simplicity of the Scooby domain specific language, when considering the alternative of programming such a service in another high-level language, such as Java with the use of APIs to access middleware functionality. In Sections 4 and 5 we describe the Scooby language and constructs in more detail.

There are a number of technologies that can provide the building blocks towards such a system, including service discovery, remote invocation and messaging systems, such as CORBA and J2EE. However, we have opted to take the approach where we are not directly reliant on such technologies as we are targeting the system towards low-powered devices with limited processing power, such as PDAs and devices that can be embedded within home appliances. When taking the nature of the environment and the limited scope of the devices available into account, using heavy-weight technologies is not the most practical or viable way forward. Instead, we have chosen to utilise the publish/subscribe paradigm as the method for relaying event information between devices, using the content-based router Elvin [19]. This provides a light-weight communication medium on which we can then build our middleware. One of the characteristics of a pervasive computing environment is the ad hoc combination of devices and intermittent and unpredictable availability of devices. There is no guarantee of the device being available over the course of time due to a number of physical reasons such as network disruption, the device dropping outside of the influence of a wireless network and power loss/power saving modes. The adoption of a publish/subscribe mechanism therefore would not have an impact on the environment when taking into account the lack of message guarantee. Additionally, the utilisation of key-value pairs available within publish/subscribe is heavily relied upon when disseminating service characteristics.

The Scooby language acts as the medium in which a user can compose services. As the language and resultant compiler are lightweight, much of the complexity of implementation is pushed down to the middleware level instead of the language. If there were to be millions of services and thousands of events per second, then such an approach

```
service stockmonitor decorates cs: screen, sr : sms, em: email {
  // need to bind to an event which contains the current location of
  // the user. could have a ref id tag attached and associated service
  bind userlocation as event match { location: userloc }
  // we need to bind to relevant devices (screen, sms)
  // depending on the location of the user
  bind cs match { location: screenloc }
  bind stockservice as event match { price: stockprice }
  {
    boolean userpref_smsnotemail = true
  }
  {
  // only fire when we get an event from the stock service
  on notification( stockservice ) {
    // we need to call a relevant device depending on the location
    if( userloc == screenloc ) {
      cs.sendmessage( stockprice )
    } else {
      // depending on what the user wants, they will put code here to send
      // to email or sms, basic example here using a boolean variable
      if( userpref_smsnotemail ) { sr.sendmessage( stockprice ) }
      else { em.sendmessage( stockprice ) }
    }
  }
  } when bind exception() {
    reporterror("Error in binding.")
  }
}
```

Fig. 1. Follow-me service code.

would not be viable. Pervasive computing environments could potentially contain many thousands of devices where scalability is of paramount importance. However, the type of environment that we have positioned our system at is that of a home or office environment. This provides us with the opportunity to investigate an environment which is much more constrained than other larger scale systems thus reducing the issue of scalability. Much of our limitation in this respect is due to the use of Elvin below our system. The performance of this has improved since the work described here, and so Scooby would be less constrained than our results suggest. We have not targeted the smallest devices in pervasive computing, for example motes, at this stage, but consider PDAs and consumer appliances etc. We leave the investigation of the application of Scooby's principles on these devices to future work. This approach has also been adopted in similar intelligent environments such as EasyLiving by Microsoft [13] and Adaptive House by Colorado University [9], which are constrained by their physical presence within the real world.

Another aspect of the middleware that is not going to use existing technologies is that of method invocation. There are traditional methods for performing invocations by using Java RMI or CORBA, for example. However, these methods do not fit in with the overall model of the middleware as these types of communications fall under the category of method-based publish/subscribe [20] and are not in line with the view of utilising the content-based publish/subscribe paradigm.

A number of technologies already exist that have components which provide service discovery and identification such as CORBA,.net or JINI. As a result of the plethora of technologies, there is unlikely to be a monoculture in future pervasive computing due to the dominance of larger software companies. Indeed the use of different software technologies (CORBA [18], DCOM [21], RMI [16]) and different naming/service discovery paradigms (UPnP [22], JINI [23], SLP [24]) can hinder matters when exploring new ways of designing the home pervasive environment. Instead we have used our own minimal approach for implementing middleware based on the publish/subscribe system.

1.2. Paper structure

In Section 2, we provide an overview of available systems and technologies. Section 3 provides an overview of the Scooby system. Section 4 provides a more in-depth examination of a service along with examples. Section 5 focuses on the Scooby language and middleware aspects and discusses the main contributions. In Section 6, the evaluations used are discussed along with the results gained. Finally, in Section 7, we make our conclusions.

2. Other work

There has been a great deal of research in the pervasive computing domain that has attempted to provide an application framework based on APIs and middleware so that developers can create their own services. Examples of such systems are One.World [3], Gaia [5], Solar [6], SAHARA [7] and Ninja [8]. However, not all systems have taken this approach. The notable exceptions are Adaptive House [9], which uses a neural network-based approach to composition, Colomba [10] that utilises a high-level policy description language based on Ponder [11], and Olympus [12] that provides a high-level programming abstraction based on a set of operators. However, in the latter case, these operators are provided as a set of classes and can fall into the same category as an API style of producing services.

The middlewares that are used by these systems range from ones that are specifically written for them (One.World, SAHARA, Ninja, Solar, Easy Living [13], Aura [14]) to ones that build upon others (Olympus uses Gaia which itself uses Corba). In addition, different middleware systems are targeted to provide composition through other means. For instance, Adaptive House builds upon a house equipped with a variety of sensors that are linked to a neural network. This is used to provide a probability to base a decision on, that can affect the current state of the house. Another differing method through which communication is handled is an event-dispatch paradigm while others utilise more traditional client/server approaches. Both methods have their advantages and disadvantages. For instance, the event-dispatch paradigm can suffer from scalability and event delivery guarantee problems, while the client/server approach does not fully take into account flow, time and space decoupling [15].

Other more esoteric methods are employed for a number of key system components dealing with service discovery, invocation, service description and service management. The use of a standardised approach to service discovery that utilises a single service and/or client/server model is quite prevalent throughout the middlewares. In addition, remote invocation utilises existing methods and technologies, such as RMI [16]. Finally, service control can be provided by using Jini [17] or CORBA [18].

When viewing the systems as a whole, the majority of cases have taken the approach of tackling service composition by adding an API to an existing language. However, Scooby does this differently as it has a specifically designed compositional language in which users can write both the original services *and* composite services. This approach reduces the number of system concepts present within the language and so reduces the complexity of the programming. There are other systems available, for instance OASiS [30] which provides a light-weight middleware infrastructure for wireless sensor networks, in which services become subsumed within the middleware substrate. This approach is similar in concept to the way in which services within the Scooby system are absorbed within the middleware. Another vein in which there is overlap between Scooby and other work, is that of the work carried out in the RUNES [31] and SATIN [32] projects that have adopted the idea of reifying bindings amongst services. This supports the justification for the dynamic binding mechanism within the Scooby middleware.

The contribution of this research has been the investigation into the design space of language approaches to service composition, combining several approaches to language design that are particularly suited to this problem. In addition to the composition language, a middleware has been designed and implemented that uses late dynamic binding variables to allow services to identify others, which satisfy their needs most effectively. Finally, a methodology on how to compare systems has been formulated. The aforementioned binding variables, coupled with the use of service attributes, allow services to adapt their advertised characteristics over time. Service attributes are used to allow bindings to be flexible in their lifetime, during which they alter due to service or environment changes through a matching of service attributes. Building upon these system concepts and the results from a comparison with the *one.world* system, we claim that a domain specific language is the most effective service composition technique for programmers to use.

3. Scooby overview

The most practical way of introducing the Scooby service composition middleware is to split the infrastructure into multiple levels, where each offers a particular form of functionality. Fig. 2 outlines the general levels in the creation, compilation and execution of a scooby service.

Programmers would initially specify their service description (code) within the composition language (see Section 5). It is at this layer that the process of identifying which service or set of services is required. In order to form a new composition, service ontological information formed from the method signatures, characteristic interface definitions are used. The language composition compiler would translate the user's service into the relevant Scooby middleware programming language code which in turn would

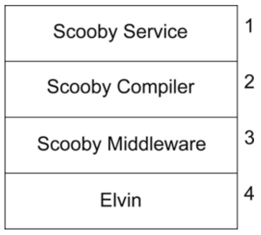

Fig. 2. Scooby hierarchy.

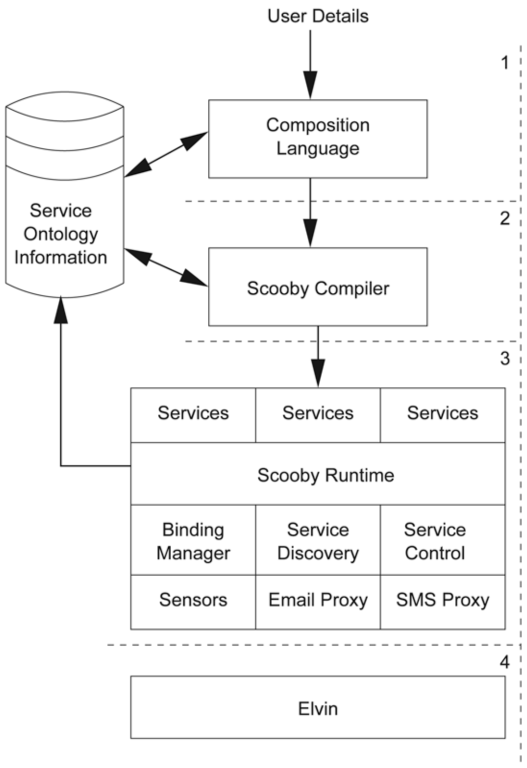

Fig. 3. Scooby middleware components.

be compiled into a new service that would then become part of the Scooby middleware infrastructure.

The infrastructure can be broken down, exposing more details of its composition. Fig. 3 expands the components within the layered model from Fig. 2.

The compilation process that occurs at layer two, takes the Scooby code, along with ontological information detailing the method signatures of referenced services and produces a Java program which is then compiled by the system. For more information regarding a comparison between the Java code produced and that of its Scooby counterpart, refer to Section 6.2.2 in which a more detailed comparison between both sets of code can be found.

It can be seen that the middleware layer (3) contains the most complexity. It is at this level in the hierarchy that all service and method publication, discovery and invocation, message notification handling and access to an external medium through a set of proxies (SMS, Email etc) take place. Additionally, once a new service composition becomes available it is placed within the middleware layer and inherits the core functionality that permeates throughout all services.

We have chosen Elvin [19] as the basis for handling all messaging within the system due to the event model that it employs. It fits well with the notion that a pervasive environment has a dynamic and fluid nature to it and, as such, an event model is the ideal form of communication between these services. We have identified that a pervasive computing environment needs to be able to cope with the dynamic availability of disparate services and devices within the middleware [25,26]. Due to the ad hoc dynamically changing topology of the availability of services, there is no guarantee of delivery. Such a topology requires the dimensions of time, space and flow decoupling [20] to be applied to the underlying messaging system. Other more classical technologies (CORBA, JINI) rely on a client server model, which does not satisfy these three dimensions. Information is required about both the sender and the receiver of the message and blocking occurs when invocations are performed. None of the three dimensions in this case are satisfiable due to the nature of these technologies (synchronous invocation). What is needed, however, is a way in which they can all be met. The only form of publish subscribe that adheres to these dimensions is that of content-based which has led us to adopt Elvin in order to provide the asynchronous message notification requirement of the middleware. Furthermore, the decoupling of the production and consumption of events using these three dimensions improves the scalability and reduces the overheads introduced through synchronisation and coordination, by removing any dependencies between interacting parties [27]. Another reason for adopting content-based publish/subscribe rather than one which is type-based is through the flexibility of the content message which can be built out of different key-value pairs. This allows flexibility of the characteristics variables (discussed in Section 5.1) which can dynamically change over time, rather than relying on a type-based system where registrations of subscriptions would have to change to take into account any alterations.

Notifications are directionless messages, which can contain any number of details that satisfy the space dimension. The purpose of this type of communication is to act as a broadcast detailing service information regarding the originating service. As Elvin handles notification messages utilising a tuple-based format, we have similarly adopted tuples as the atomic currency in which to describe information within the system. This is a fundamental criterion as it has been used extensively to add different sections to an event message.

A service description is sent to the discovery service, which details the interface of available functionality for that service. The structure of the service description notification contains a number of tuples that describe the service characteristics and method signatures offered by that service. Based on these forms of notifications, communication is achieved throughout the middleware in which service discovery, identification and invocation can

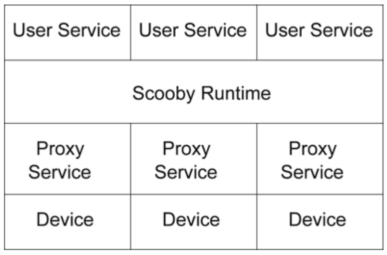

Fig. 4. Scooby services and middleware.

occur. The method and type definitions are determined by the middleware as soon as the service is instantiated, at which point this information is then inserted into the ontology.

The binding manager resolves requests from services for other services that match the required constraints. When a service wishes to invoke a method upon another service, it can request a binding from the binding manager for a service of the necessary type, and which has matching characteristics, such as location, performance or load levels. Our current matching algorithm simply provides the first application that provides a match, but more complex load balancing is possible.

The service control tracks the current state of each service, such as whether it has been initialised, whether it is accepting requests, and whether all necessary bindings have succeeded.

4. Service anatomy

Scooby provides a composition language for the management and organisation of services. Although the ultimate aim is for Scooby services to be compiled automatically, direct programming is designed to be easy. We have adopted a standard object-oriented approach to typing the interfaces describing services, using a form of method delegation. Interfaces are defined through the method and attribute signatures.

Scooby allows user defined services to be written and inserted into the middleware at run-time. User services are defined using the Scooby language and compiled into Java through the Scooby compiler. The middleware handles all service discovery, lookup, remote invocation and service control. Underlying the middleware, Elvin is used to propagate event information throughout the environment. Users' services are triggered when events are received. To allow Scooby services and the middleware to communicate and operate devices in the real world, an additional layer of services needed to be provided. These proxy services act as a point of control to the attached device and provide functionality to allow its operation. Fig. 4 outlines the layers and relationships of services in the Scooby middleware.

The Scooby middleware requires a service to go through a number of steps before it is made available to the rest of the infrastructure. The Scooby system first requires the resolution of any bindings within the service. The binding process locates matching components and provides a way for the service to use these components. Bindings can have a number of states, some of which generate exceptions and are discussed in more

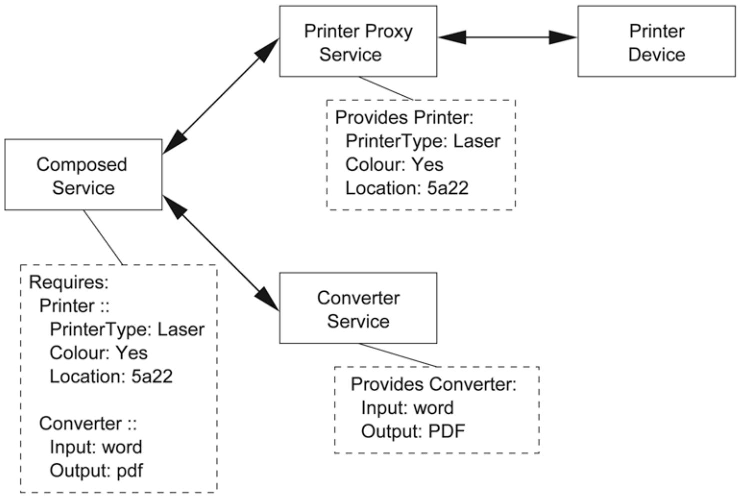

Fig. 5. Printing service characteristics.

detail in Section 5.2. Once a service has established its bindings, it sends a description which contains information regarding the types, methods and characteristics offered by that service. This description is cached by the middleware and can be used in the discovery process for other services trying to find services of that type.

To facilitate service composition, services are able to draw upon others that match a given criterion detailing their requirements. For example, a printer composition scenario where a document has to be in PDF format before it can be printed. The composition occurs when trying to print, the service will determine if the file is of the correct type. If it is, printing occurs, otherwise, the composition will lookup and bind to a conversion service to allow the document to be converted before it can be printed. In this instance, the service must resolve two other services on which to base the composition. The first is to resolve a printer, and the second, a PDF converter. Fig. 5 outlines the basic structure of the composition, shown by the solid-edged boxes. When a composition occurs, the discovery service will bind to services offering particular functionality through the use of service characteristics. These allow services to provide additional information to aid in the resolution process. In the previous example, the printer and converter services advertise a set of characteristics. The dashed-edged boxes in Fig. 5 show how the necessary bindings can be resolved with instances having particular characteristics. In this case, the printer proxy would offer a print method along with characteristics detailing its abilities and location, while the converter would detail what input and output it could accept and offer a convert method. The composed service would be constructed to bind to both services given its requirements.

The Scooby code required to perform composition is detailed in Fig. 6. One of the important features of the Scooby language is the ability to extend a service by utilising the *decorator* design pattern. In addition to this, another feature provided by the language is that it allows services to act as *proxies* to other services. For instance, in the previous example, the *printer* service which has been referenced, could in turn be referencing a

```
service composedService decorates p:printer, c: converter {
  bind p match { printertype: "laser" & colour:"yes" & location:"5a22" }
  bind c match { input:"word" &  output:"pdf" }
  {
    // put any characteristics that you want this service to offer. eg:
    // printertype: "laser"; location:"5a22"; input:"word"; output:"pdf"
  }
  {
    public void doprint( blob document ) {
      // convert the file if it is a PDF
      if( document is "PDF" ) {
        document = c.convert(document)
      }
      // print file
      p.print( document )
    }
  }
} when bindexception() {
  // This code-block is executed when a bound service fails.
  // For example, we can alter the binding attributes
  // defined within any binding variables so that we can
  // try to resolve something else. Following this, or
  // code, the service can be restarted, or terminated.
}
```

Fig. 6. Composed printer service code.

service provided by the original manufacturer. This enables a service to incorporate the functionality found within any referenced service.

The notion of late binding has been adopted when applied to binding variables such as *p* or *c* above. In this case, a binding variable is not initially associated with a service until as late into its lifetime as possible. Service descriptions are used to identify a matching service that satisfies the requirements laid out in the binding specification. Once a service has been identified, a link between the two will be maintained. However, during the lifetime of a service, the attributes that it advertises in its service description continue to evolve and as a result, any bindings to that service may become obsolete. Any referencing bindings within other services would change state at this point in order for a new service to be identified. In essence, bindings allow services to adapt to available services within the middleware automatically. The binding process is expanded in Section 5.2. When all bindings have been made, the associated code block is executed. Methods declared in the class body can then be invoked upon the instance. The exception handling block provides a way for the programmer to specify how the service should react to binding failure.

5. Language constructs

In this section we will introduce a number of constructs and concepts that are implemented within Scooby. We will discuss the motivation and use of the feature, describe the syntax for using the feature within the Scooby language, and highlight any issues concerning implementation.

5.1. Service characteristics

One of the primary goals within the middleware and language is to provide a service with the ability to dynamically alter its advertised capabilities over its lifetime. If a service's functionality is affected by the availability of other services, this allows it to reflect its current situation — so providing honest advertisements. This mechanism is accomplished by providing a set of *service characteristics*. These offer a means by which the service definition can be altered through the use of service variables, found within the service definition. Service characteristics expand upon the normal service description header that includes method definitions along with typing, return types and decorated services. Any characteristics to be linked to the service are defined within the characteristics stub which would be located after the initial binding definitions (see Fig. 6). Characteristics are defined by names, which are generally statically defined on service construction (the set of names can be varied, but this is not discussed here). The values associated with these names can be defined in one of two ways: statically or dynamically, providing flexibility depending on what is ultimately required.

If a characteristic of the service never changes throughout its lifetime, then it should be statically defined. For example, a printer service would be able to advertise that it is a "laser" printer through its service characteristics, but would not be expected to alter this characteristic during its lifetime. This would be defined in code by:

```
{
  printertype: "laser";
}
```

As seen, characteristics are name-value tuples. The left-hand side provides the key, which, in this case, is the characteristic name that is defined and advertised within the service description header. The right-hand side provides the value. As well as strings, as above, integer or boolean values can be used.

An example of a dynamic characteristic is the printer's state, which might vary between *busy*, *idle* or *attention*. Such changes clearly cannot require removal and re-definition of the service, so dynamic characteristics are used to represent this changing state. The following code fragment illustrates the dynamic form of a characteristic:

```
{
  printerstatus: status;
}
```

In this example, a characteristic variable has been defined on the right-hand side as a placeholder for a value that is automatically advertised once an *advertise* message is sent by the middleware. As the definition is a variable, it means that its visibility is global to all methods defined within the service and allows methods to alter the value stored within it, and hence advertised through the service description header, through the assignment operator.

These advertisements are used in order to allow the most appropriate service to be selected for a given situation, based on the context information supplied with the advertisement. The highly variable nature of pervasive computing environments means that the selection of services should be programmable; for instance, the location of printers is likely to be an important factor in service discovery.

5.2. Binding variables

Service characteristics allow services to describe themselves. We use *binding variables* to allow services to constrain which other services they bind to. Binding variables allow a service to discover, and via the middleware connect to, a remote service depending upon the type of service and its characteristics. Services are defined to provide a type, or alternatively to act as a proxy for another service, which describes the initial narrowing of the search space; the characteristics are used to identify services within the given type. Once a service has been identified at run-time, any remote invocation of methods is performed through the associated binding. The syntax for a binding is illustrated by the following example:

```
service s decorates s1:service1 {
bind s1
   match { aName1: aValue1 & aNamea: aValuea }
bind s2 as service2
   match { attributeName2: arbitraryValue2 }
...
```

It can be seen that there are two ways of writing the binding: in s1 the type of the service bound-to is defined in the first line, through the decoration; in s2 the type of the service is defined within the binding. As many matches as required can be defined.

Bindings are finite state automata that alternate between several different states during their lifetime depending on whether a remote service has been identified or not. Initially, bindings are placed into an *unresolved* state. Immediately after the instantiation of the service, a binding will change into a *discovery* state. This is the point at which service discovery takes place depending on the service type and attributes supplied within the service code. Moving on from this point, a binding is able to switch into either a *connected* state, where a service has successfully matched the search criteria and allows binding calls to be made, or it enters a *suspend* state indicating that the discovery of a relevant service has not succeeded at this time. If a binding is connected and a remote method is invoked, then depending on the method definition (i.e. if a return value is required) the binding will change to a *waiting* state, which only changes back to a *connected* state once a reply has been received. This process is repeated constantly throughout the lifetime of the binding. Due to the nature of a pervasive environment, the middleware automatically governs these state change operations transparently. However, if at some point, a service ceases to be available or ceases to match the required conditions, then a new discovery and binding process is started. If no service is available then the binding fails and the corresponding binding exception segment of code is executed. The use of bindings in practise would block when no remote service has been identified successfully. It is only when a successful state (*connected*) has been attained that any remote calls will be performed on that binding.

There is no guarantee that a service will be present in the middleware that successfully matches the search criteria specified for the binding. This is due to a number of factors such as different services becoming available at varying times, network loss and device availability. If a remote service is resolved there is no guarantee that the same service will be identified again when performing another discovery operation if multiple services match the search criteria. Additionally, bindings will automatically verify that the remote service is still available before initiating a remote invocation, as there is no way for the local service to know if the former is still accessible. This is performed automatically by the middleware, but would result in the binding performing an invocation, or changing into a *suspend* state which will perform a service rediscovery at some point.

When bindings fail before code blocks are completed, the system will raise an exception. If the conditions for execution of the composed service are still valid, then the system will attempt to bind other services matching the required context to the variables and re-execute the code block. However, this could introduce the problem of inconsistent state. Bindings could possibly become unsynchronised and potentially reference invalid services depending on when the exception was raised. The middleware does not specifically deal with this problem and relies on the programmer to provide a solution within the exception code block and how to recover from it. Providing more automated methods of recovery remains an issue to be addressed in future work.

The Scooby language provides two forms of binding constructs: a binding for performing a remote method invocation on another service and an event binding that allows complex events to be monitored and triggered when an event description is met. Event bindings will be discussed in more detail later in Section 5.3. However, both forms of bindings are similar in syntax and follow the same binding attribute rules.

One problem associated with discovering a service is that at compile-time, there is no guarantee that a service will be available that meets the search criteria at run-time. Similarly, we introduced the notion of a service characteristic that allowed change through its lifetime, which is either static or dynamic in nature. The requirement of a statically or dynamically defined matching ability of a binding is a pre-requisite. To revisit the syntax of a binding:

```
binding:= bind var1 as Type1 match { attribute* }
attribute:= key : data [ operator ]
operator:= & | == | > | < | != | >= | <=
```

A variable, visible to the service, *var*1 is defined, which identifies a binding to a service of type *Type*1, where that service matches the specified *attributes*. The *attributes*, as seen above, are *key* (or name) and *data* (or value) tuples, separated by a colon; an additional operator can be used to define to specify how the attribute is matched: the default is an exact match (==) of an attribute. The set of inequalities is limited to those which are defined for the type of the attribute value. The "&" operator represents the join operation that allows multiple attribute comparisons.

An attribute *key* value is statically defined at compile-time within the code to signify the value that will be present within the service description. The right-hand side *data* value can be defined as either static or dynamic. This does not reflect the static or dynamic nature of the characteristic values advertised by services (discussed above), but reflects the need for variation in the service searched-for and bound-to. Static values are quoted strings and require an exact match. In a fluid environment where services have the ability to alter their service descriptions dynamically, there needs to be a way in which bindings are able to be altered accordingly. Dynamic values are expressed by replacing a string-based item with an attribute variable. The language interprets the name as a variable and generates a code so that any value associated with the key component is automatically stored within the variable declared. This attribute variable is then globally available to all methods defined within the service. Additionally, the variable may be referenced in another nested attribute comparison in the same binding, or alternatively referenced with subsequent bindings.

The following code fragment highlights the differences between a simple static and a dynamic binding declaration:

```
Static: bind ptr as Printer match { location: "5a22" }
Dynamic: bind ptr as Printer match { location: whereitis }
```

In addition to specifying simple attribute conditions, a binding allows the use of more complex structures that incorporate an operator. The latter can provide different forms of equality expressions (conditions) as well as the "join" operation. Bindings can also be composed of a combination of statically and dynamically defined set of attributes. In the following code fragment, a binding searches for a printer in a static location, and an undetermined value stored in status:

```
bind ptr as Printer match { location: "5a22"
                          & status: printer_status }
```

In this fragment, the status that is advertised by a service implementing a printer would automatically store the value within the `printer_status` attribute variable. This example would provide a general catch-all of services matching these requirements when any processing of the value stored within the attribute variable would be performed by code calling the binding and would have to be specifically written beforehand. However, bindings can accommodate more complex processing by introducing different equality operations within the binding definition itself. For example:

```
bind ptr as Printer
  match { location: "5a22" &
          location: "5a23" &
          status: printer_status &
          printer_status != "attention"
  }
```

In this example, `location` is joined (&) and so can be *either* "5a22" or "5a23". The `printer_status` contains the status of the printer as advertised by the service, but the service matching does not succeed until the second component evaluating the value stored in `printer_status` does not equal "attention" (ie is "idle" or "busy"). Only when a condition for each key has been met will the associated binding change its state to connected. In this case, a static string has been used. However, if a value is not known at run-time, for instance if a comparison is required based on a characteristic from the local service, the reference can be added to the criterion match of the binding.

5.3. Event bindings

Event bindings allow the capture of an event based on a set of criteria much in the same way as a binding binds to a service. However, event bindings allow for complex composites of events as well as simple event handling. For instance, an event can be triggered once an event is received which matches a single tuple. Event criteria can also be formed over multiple event messages. This provides the ability to determine if a set of events has occurred before the event binding is satisfied. The syntax of an event binding is as follows:

```
bind X to event match { attributes * }
```

Attributes follow the same rules as those for service bindings and can be formed with predicate operations as well as single tuples. Any event information referenced within the set of attributes would map directly to a holder within the middleware that stores tuple information regarding events that are received. Only when all event matching attributes have been met does the event become triggered. A special event notification method construct is used within the Scooby language to capture the moment the triggering of the event becomes active. The following code demonstrates a notification handler for the above definition:

```
on notification( X ) {
// rest of code goes here
}
```

As the language does not constrain the use of multiple bindings, the same applies for notification handlers. The only criterion which governs the use of a notification handler is that the reference to the desired event binding must be used within the definition of the handler.

5.4. Method delegation

Scooby is an object-oriented language and follows the normal syntax for method invocation, where dot notation syntax is used to offset a method call within a binding. For example, `ptr.print(document)` would reference the print method passing in the document parameter. Whilst binding information is not known until run-time, return types as well as parameter types *are* known at compile-time. To facilitate type checking, interface definition files are used that specify typing and method information for a particular service. This can be linked in at compile-time to allow verification of any referenced service method.

The Scooby language does not provide single- or multiple-inheritance. Instead, as the language supports the decoration of other services within the definition of the service, method delegation becomes of paramount importance. The way in which the Scooby language deals with delegation is through the inclusion of public method definitions when the source code is generated by the compiler. As any referenced services have an associated interface file, methods that are not specifically defined within the service are automatically added as a wrapper by the compiler so that calls to that method would in turn call the implemented service. Therefore, interfaces act in a way to import external services into the

Scooby type system, thereby providing an alternative to inheritance by allowing transparent method delegation to services. The decorator design pattern is used within the language, as services can refer to others and add to their functionality. In addition, the proxy design pattern is used by bindings to provide indirect access to remote services. A service defining myprinter that decorates the printer service would be:

```
service myprinter decorates p:printer {
  {
    public void doprint( blob document ) {
      print( document )
    }
  }
} when bindexception() { }
```

The print method is not specifically defined within the myprinter service, but is instead defined within the printer service, as outlined by the following interface file:

```
interface printer {
  public void print( blob )
}
```

The compiler would include the print method within the generated code:

```
public void print( Blob val0 ) {
  p.callMember("print", new String[] { val0.encode() } )
}
```

The compiler automatically inserts additional method parameter conversion subroutines and all error handling is controlled within the middleware thereby greatly reducing the complexity of the generated code. Within this example, the printer is not bound to a specific service and therefore the middleware will automatically search for any generic printer and bind the first one it finds. However, if a specific printer is required, for example, located in a particular room, the system binding can be enhanced by providing a binding clause. In these circumstances the previous code would now become:

```
service myprinter decorates p:printer {
  bind p match { location : ‘‘5a22’’ }
  {
    public void doprint( blob document ) {
      print( document )
    }
  }
} when bindexception() { }
```

This allows the binding p defined in the service header to resolve a printer located in “5a22” thereby increasing the flexibility of automatically resolving which service to draw upon for the base functionality. Normally, delegation is sufficient for referencing methods defined within any services that are referred to in the decorates clause. However, there are

Table 1
Evaluation scenarios

Scenario	Description
1. Printer composition	A user would like to print a PDF document on a printer. The required printer must be able to print with the following characteristics: print in colour, double-sided and print on A4-sized paper.
2. Follow-Me service	A user has configured a stock monitoring service to inform them when a stock price reaches a certain point. The service is configured to display the stock price on the closest available smart device to the user. This could be in the form of a message on a screen, printout, sms or email message, depending on the location of the user within the smart environment. It is assumed that the user may walk around within this environment and alter their location.
3. Smart home	The user is in their car, returning home from work. A PDA is present within the car and is connected to a GPS system allowing a calculation of the time remaining before the user reaches home. Ten minutes before arriving, the PDA signals the home, telling it of the user's imminent arrival. Upon doing this, devices in the home activate. The heating device turns on so that the house is warm and hot water is available. The lights are turned on in the garage and entrance hall of the house. The curtains are automatically drawn and the coffee machine starts preparing some fresh coffee. As soon as the user reaches home and enters the house, the garage lights turn off as they leave the garage, motion detectors track the user's movements in the house, turn on the living room lights and start playing their chosen piece of music in the CD player.
4. Context-aware media players	The user is sitting in a dimly lit living room, listening to music. Suddenly the phone starts ringing. Connected to the phone is a device that detects an incoming call. This automatically causes the lighting levels to be returned to normal (if previously dimmed) and for the volume in the CD player to be reduced. The user picks up the phone and begins to talk. Once the conversation has finished, and the receiver is placed back on the phone, the lighting is returned to the previous levels, and the volume returns to what it was before the phone call.

instances where methods defined in more than one service are identical to each other. For instance, in the previous example, if another service is included, which also provides a print method with identical parameters, then on specifying a call to print, an ambiguity is introduced concerning which method is the correct one to use. A way to resolve this is to use the binding reference and offset the call against it. This directs the compiler to reference the appropriate method.

6. Evaluation and results

Our central claim is that the use of a domain specific language makes the composition of services easier. To support this claim, we have devised a number of scenarios showing the use of service composition in the domestic environment. We have then implemented these scenarios both within the Scooby language and within the *one.world* [3] programming environment for pervasive computing. We then compare the service compositions from the two environments across a number of metrics. The scenarios are presented in Table 1 and their use is described below.

The types of scenarios highlighted in the aforementioned table are for exploring simple interactions between the user and the environment rather than something which is more in-depth and complex. Simplicity, in terms of the definition of the scenario and resulting interactions would be more useful in showing the applicability of a purpose built composition language. However, the language itself is expressive enough to handle more complex scenarios.

6.1. Introduction to One.World

One.World [3,4] is a pervasive computing environment developed at the University of Washington. We have chosen this system to act as a control since it is well-documented and has been used in a number of projects. One.world is implemented as a set of Java packages that add functionality to the language. The system itself shares a number of common notions with Scooby (albeit implemented and manifesting themselves in different ways), such as events and bindings. However, as One.World has adopted an API approach to service development, we can use this as a basis for comparison as we advocate that providing a domain specific language is the better route to take. What follows is a more in-depth description of the key components to the One.World system and a general guide on how it works.

Each device runs a single instance (node) of One.World that acts independently. One.World allows applications to run within the environment sharing the same node and the same instance of the architecture. However, there is a separation of concerns by sorting abstractions into application data and functionality. Applications store and communicate data in tuples, which include type information and can be nested. An application is allocated a unique identifier to support symbolic references and a meta-data field to support application specific annotations that are composed of components.

Components provide functionality and interact by importing and exporting event handlers, which consume asynchronous events. Event handlers are dynamically linked and unlinked, but statically declared. The type of event handlers that components use are defined in the constructor when instantiated. These can change throughout the component's lifetime, where it can be linked or unlinked to other event handlers.

The purpose of an environment is to provide structure and control, and act as containers for tuples, components and other environments. Each application has at least one environment which can also span several other environments. Within One.World, the latter are used for dynamic composition. Leases allow access to local and remote resources, for instance, tuples within the environment. Migration allows the environment to be copied or moved to another node.

Remote event passing allows events to be sent to remote receivers that support point-to-point communication and service discovery. In addition three abilities are provided: export, send and resolve. The export functionality allows an event handler to be accessible from remote nodes through the use of a symbolic descriptor. This results in a binding between an event handler and a descriptor that is leased. When exporting an event handler for service discovery, the binding is propagated to the discovery server of the local network. The send functionality sends an event to the previously exported event handler by using an exported descriptor or discovery query as the remote address. When using late binding, the event is routed to the discovery server where the discovery query is resolved. However, if this is not the case, the event is dispatched to the node exporting the targeted event handler instead.

Finally, the resolve functionality resolves event handlers contained within the discovery server.

6.2. Comparison metrics and analysis

Devising metrics to determine the usability of a programming language is a difficult challenge, which can only be truly undertaken through the comparison of the outputs of experienced programmers working within the language. In the absence of such experience, we have attempted to use standard software engineering approaches to show that Scooby is more expressive than using one.world, and produces comparable code sizes in the final Java output. Our metrics have been designed to highlight:

Service composition language: where comparisons between Scooby and the control system can be made based on the inputted code, constructs and syntax. We are primarily interested in how each approach compares through the use of finding common concepts and notions on which to base the evaluation.

Size of code generation: compared to service definition in both Scooby and the control system. This would give an indication on how many lines of code are required to approximate similar services in either system and provide an initial indication on the ease of use of each one of them.

Expressiveness of service: comparing the inputted code of both systems. We are interested in viewing comparisons based on common concepts and notions present within the languages that closely approximate each other. This can give rise to information regarding the number of constructs used, as well as other comparisons.

We built implementations for each of the four scenarios described in Table 1, and examined each against the metrics above.

6.2.1. Service composition language

In Table 2 we analyse the service compositional languages in order to determine the similarities between the two with respect to the common traits and constructs that they share. The purpose of the criteria is to determine the expressiveness of the Scooby language when viewed as a whole. We are mainly interested in how the languages compare when it comes to common concepts, notions, constructs and language features. In Table 2, we show that Scooby provides more features needed for composing services than one.world, but if we use the number of keywords as a measure for complexity, then scooby is less complex. Since the motivation for Scooby was to design a language containing the necessary abstractions for composing services, it is perhaps not surprising to discover that Scooby is feature rich when compared to one.world, yet appears to have a lower level of complexity. This is borne out by the line counting metric described below in Section 6.2.2.

Both languages were object-oriented but only One.World (through Java) allowed the use of inheritance. Inheritance was mostly used for defining event handlers and in subclassing the main class for any applications. Our experience in composing services has shown that inheritance is often used as a overly restrictive way of building the delegation pattern.

Table 2
Language comparisons

Metric	Scooby	One. World
Binding Variables	✓	✓[a]
Service Discovery	✓	✓
Service Alteration	✓	×
Service Descriptions	✓	×
Remote invocation	✓	×
Object-oriented	✓	✓
Keyword count	74	20 Packages, 353 classes, rest of Java language
Event handling	✓	✓
Java Extension (API)	×	✓
Method delegation	✓	×
Inheritance	×	Single
Typing	Static	Static
Exception handling	Bindings	✓
Arithmetic handling	✓	✓
Event driven	✓	✓[b]
Global variables	✓	✓[c]
Procedural	✓	✓

[a] Binding variables found in One.World differ considerably from those found in Scooby. However, the terminology used is the same.
[b] One.World provides an API that augments Java with a way to interact with the environment. Used within this context, One.World is able to achieve similar functionality to that found in Scooby, but requires additional programming to accomplish this.
[c] Global variables can be achieved by using the Java Language. However, these do not strictly adhere to the same definition as found within Scooby.

We therefore decided against providing inheritance and instead used method delegation to access any referenced services and automatically build proxies.

6.2.2. *Size of code generated*

When comparing languages, one of the metrics that can provide insight is through the examination of the physical size and number of lines required to produce a service, as detailed in Table 3. At this point we were not interested in the final object code size, but by the amount of work needed to be done by the user to construct the service they wanted. We were able to observe that the number of lines of Scooby code (and for that matter, the size of code) is significantly smaller than that found in the comparable One.World service. However, when viewing the resulting Java code produced by the Scooby compiler we found it to be somewhat similar in size to the comparable One.World code that was generated. From this, we can have confidence that the Scooby compiler produces code that is at least as efficient as that hand-crafted in Java.

When comparing the effectiveness of a dedicated service composition language to an API-based approach, it can be seen that the Scooby language fulfills the goal of making services and compositions much easier than its counterpart. This can be justified when comparing the initial number of lines used to write the Scooby code and its resulting size in Bytes, with its One.World equivalent. The reduced code and size, coupled with the

Table 3
Code generation comparisons, A is Scooby, B is One.World

	Scenario 1		Scenario 2		Scenario 3		Scenario 4	
Metric	A	B	A	B	A	B	A	B
Initial size (Bytes)	832	12 231	1 635	23 073	2 123	24 476	2 034	18 611
Initial lines	44	366	62	712	88	724	91	549
Final size (Bytes)	14 535	12 231	15 980	23 073	27 550	24 476	18 295	18 611
Final lines	525	366	575	712	974	724	640	549

less complicated syntax of Scooby, results in a language that is compact and lightweight. However, due to the reduced set of semantics found in Scooby it does not allow any additional functionality to be described. This is not the case when looking at One.World, as Java can be used to provide any other functionality that may be required, and as a result provides more flexibility. However, this is at the expense of complexity and ease which are the metrics that we are interested in.

6.2.3. Expressiveness

We have found it difficult to measure the expressiveness of a language by providing any objective metrics. Therefore, a set of criteria on which to base our results had to be formulated (as shown in Table 4). The approach adopted was to view common notions and concepts present within both languages. This included amongst others, data types, method declarations, characteristics, event handlers and bindings. One of the overlapping concepts was that of bindings, which provided a way in which we were able to compare them and subsequent notions that were introduced into the language. Similarly, remote invocations were relevant in the design and implementation of a language, and therefore were used as a way of seeing how the languages compared. Finally, standard language constructs, for instance, different types of variables, were used on which to base a comparison of the respective languages. One of the first and most obvious differences we found when comparing services between the two was that there was considerably less coding involved when writing a Scooby program. This fact is borne out when comparing the number of code lines as in Table 3.

Even though One.World shares similar terminology with Scooby, in this instance, the use of binding variables, the constructs in both languages were considerably different in concept and execution. In One.World, event handlers are used to describe a way in which a service could offer its functionality to others, as well as leases and bindings that maintain a connection between local and remote services. However, Scooby provides a more streamlined approach in which services provide service descriptions and attributes to resolve a service based on a set of criteria, instead of a typed interface which defines an event handler as found in One.World. In addition, Scooby provides both static and dynamic service attributes that allow services to alter their descriptions during their lifetimes, whilst One.World does not offer similar capabilities. It is due to this very reason that some of the values in the One.World columns are represented by zeros. This is mainly because a number of the metrics used for comparisons were available within the Scooby system but not so much in its counterpart. This does not necessarily mean that it would not be possible

Table 4
Expressiveness comparisons

	Scenario 1		Scenario 2		Scenario 4		Scenario 5	
Metric	A	B	A	B	A	B	A	B
Number of services	3	2	3	3	4	4	3	3
Lines	44	366	62	712	88	724	91	549
Blank lines/Comments	10	57	18	96	14	96	15	96
Keywords	86	1139	83	2122	199	2171	248	1642
Variable declarations	3	22	0	41	1	44	5	72
Binding constructs	2	2	5	7	5	8	4	7
Binding attributes	2	0	4	0	3	0	4	0
Binding variables	1	0	4	0	0	0	0	0
Method declarations	3	20	3	4	9	40	10	30
Remote invocations	2	0	2	0	10	0	11	0
Dynamic characteristics	3	0	2	0	0	0	0	0
Static characteristics	3	0	3	0	2	0	3	0
Event handlers	0	4	1	8	1	8	4	6

Code A is written in Scooby, whilst B is written in One.World.

to extend One.World to incorporate this additional functionality, but it would however, incur a high programming cost to add these features.

Remote invocations between services are easily accommodated in Scooby, as binding constructs provide a conduit in which remotely executing a method is able to take place. However, in One.World, this is not possible due to the event model used by the middleware. It can only be performed where events are sent to a service event handler, which then processes the event to determine what operation needs to be performed. When compared to Scooby, this is a less elegant way for remote invocations to take place and as a result introduces a considerable amount of complexity. In addition, to aid in remote invocations, Scooby automatically offers any defined methods within the service to others through its service description.

As well as the high-level metrics described above, we have also run extensive tests on the low-level characteristics of the systems, such as measurements on the various latencies in the propagation of events, and the scalability of the system across events and services. Space precludes a full discussion of these tests, since they are mostly concerned with the design of the middleware and the use of the Elvin router, but performance was comparable to the one.world system in all cases. Full details can be found in [28].

6.3. Summary of results

The principle objective of our research was to determine whether using a specifically designed composition language was a better alternative to using an API-based approach. One.World is a stable and mature platform for developing services. Its developers, University of Washington, are experienced and well-respected researchers in the area of Pervasive Computing. We believe One.World is therefore representative of all API-based approaches providing service composition and our conclusions can therefore be generalised across all API-based systems.

We can summarise the results of the high-level language comparisons into the following observations and key points which support our initial supposition.

- The expressiveness of both languages was considerably different as Scooby consisted of 74 keywords whilst One.World comprised 353 classes, which additionally required and relied upon the Java language.
- The number of lines required to program a Scooby service was significantly smaller than its counterpart in One.World.
- Constructing a Scooby service was much easier than programming one using One.World.
- Scooby provided a rich set of features for handling binding variables and service description adaptation. Binding attributes provided a clean way in which to specify the criteria on how to resolve required services. In addition, the use of static and dynamic binding variables provided a powerful feature for the language.
- Scooby was exclusively designed and implemented for service composition, whilst One.World basically just extended Java.
- Both languages were object-oriented but One.World allowed the use of inheritance through Java whilst Scooby provided a form of method delegation to access any referenced services.
- Due to the reduced set of semantics found within Scooby, the language did not allow any deviation from the purpose of service composition.
- Even though Scooby shared similar terminology with One.World, the constructs in both languages were considerably different in concept and execution.
- Remote invocations between services were easily accommodated in Scooby. The binding constructs provided a conduit through which the remote execution of a method could take place. Conversely, One.World relied on event handlers where processing of events determined what action needed to be done.

From the results gained from the evaluations, we found that service composition was indeed enhanced for the user. This led to an easier domain specific language for the user to learn and use that resulted in less time and work required on their part.

7. Conclusions

We have presented the Scooby Service Composition system. We believe that this provides an effective method for combining services to meet the needs of users in a variety of pervasive computing scenarios. Scooby uses a specifically designed composition language, which separates coding of services as building blocks from their composition into useful applications which respond to available resources.

The Scooby language and middleware provide a rich system for matching advertised services with requirements, while using many familiar techniques from languages such as Java. Both the advertised and required characteristics of those services can be dynamic, so that the inevitable changes in capability and requirements that flow from pervasive computing scenarios can be accommodated.

This approach lends itself to the development of end-user configuration tools, as explored in the NatHab project [29], where programming language APIs and exposure to source code present a significant obstacle to most users. The (re-)binding of services is an important part of the middleware, and the Scooby system provides for bindings which take account of both types and characteristics of the service in order to dynamically respond to the requirements of the application and the visible facilities. A content-based event communications model was employed, to support this approach.

Since embarking on this research, a number of systems have started to materialise that support our initial supposition in which a dedicated composition language and light-weight middleware are of a viable nature, for example, the work carried out in OASiS [30]. The use of reifying bindings amongst services is another area in which there is overlap with other research projects (for example, RUNES [31] and SATIN [32]) and provides a justification for taking this route initially.

There still remain aspects of our research that require further exploration with regards to the issues of standardisation between developers and the sharing of information regarding method signatures and the semantic meanings of parameters. Currently, our ideas focus on using an ontological database in which causal relations between parameters and methods can be produced. However, this is still in an initial state at present.

In summary, one of the most important benefits of the Scooby language is that service creation and composition can be achieved in very few lines of code while others require significantly more. This simplicity, in comparison to other middleware systems exemplified by one.world, is explored through the development of various example services in our evaluation.

References

[1] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Proceedings of the 5th European Software Engineering Conference, Springer-Verlag, London, UK, 1995, pp. 137–153.

[2] E. Lupu, M. Sloman, Conflicts in policy-based distributed systems management, IEEE Transactions on Software Engineering 25 (6) (1999) 852–869.

[3] R. Grimm, One.world: Experiences with a pervasive computing architecture, IEEE Pervasive Computing 3 (3) (2004) 22–30.

[4] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall, System support for pervasive applications, ACM Transactions on Computer Systems 22 (4) (2004) 421–486.

[5] M. Romn, C.K. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, K. Nahrstedt, Gaia: A middleware infrastructure to enable active spaces, IEEE Pervasive Computing (2002) 74–83. URL http://choices.cs.uiuc.edu/gaia/html/publications.htm.

[6] G. Chen, D. Kotz, Solar: A pervasive computing infrastructure for context-aware mobile applications, Tech. Rep. TR2002-421, Dept. of Computer Science, Dartmouth College, February 2002.

[7] B. Raman, S. Agarwal, Y. Chen, M. Caesar, W. Cui, P. Johansson, K. Lai, T. Lavian, S. Machiraju, Z.M. Mao, G. Porter, T. Roscoe, M. Seshadri, J.S. Shih, K. Sklower, L. Subramanian, T. Suzuki, S. Zhuang, A.D. Joseph, R.H. Katz, I. Stoica, The sahara model for service composition across multiple providers, in: Pervasive '02: Proceedings of the First International Conference on Pervasive Computing, Springer-Verlag, London, UK, 2002, pp. 1–14.

[8] S.D. Gribble, M. Welsh, J.R. von Behren, E.A. Brewer, D.E. Culler, N. Borisov, S.E. Czerwinski, R. Gummadi, J.R. Hill, A.D. Joseph, R.H. Katz, Z.M. Mao, S. Ross, B.Y. Zhao, The ninja architecture for robust internet-scale systems and services, Computer Networks 35 (4) (2001) 473–497.

[9] M.C. Mozer, The adaptive house. http://www.cs.colorado.edu/mozer/nnh/index.html, 2004.

[10] P. Bellavista, A. Corradi, R. Montanari, C. Stefanelli, Dynamic binding in mobile applications: A middleware approach, IEEE Internet Computing 07 (2) (2003) 34–42.

[11] E.L.N. Damianou, N. Dulay, M. Sloman, The Ponder policy specification language, in: M. Sloman, J. Lobo, E.C. Lupu (Eds.), Policies for Distributed Systems and Networks, in: Lecture Notes in Computer Science, vol. 1995, Springer Verlag, 2001.

[12] A. Ranganathan, C. Shankar, J. Al-Muhtadi, R. Campbell, M. Mickunas, Olympus: A high-level programming model for pervasive computing environments, in: PerCom 2005: Third IEEE International Conference on Pervasive Computing and Communications, Kauai Island, Hawaii, USA, 2005.

[13] Easy living, http://research.microsoft.com/easyliving/, 2004.

[14] D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste, Project Aura: Toward distraction-free pervasive computing, in: IEEE Pervasive Computing. http://www-2.cs.cmu.edu/aura/publications.html.

[15] P.T. Eugster, P.A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys 35 (2) (2003) 114–131.

[16] Java rmi specification. ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf, 2002.

[17] S. Microsystems, Jini architecture specification, 2004. URL http://www.sun.com/software/jini/specs/jini1.2html/jini-spec.html.

[18] Corba notification service specification. http://www.omg.org/docs/formal/02-08-04.pdf, 2002.

[19] Elvin. http://www.elvin.org/, 2003.

[20] P. Eugster, R. Guerraoui, J. Sventek, Type-based publish/subscribe, Tech. Rep. IC200029, School of Computer and Communication Sciences, EPFL, 2002.

[21] Distributed component object model technologies, http://www.microsoft.com/com/default.mspx, 2005.

[22] Upnp device architecture. http://www.upnp.org/, 2000.

[23] Jini architecture specification. http://www.sun.com/software/jini/specs/jini1.2html/jini-spec.html, 2004.

[24] E. Guttman, C. Perkins, J. Veizades, M. Day, Service location protocol (slp) v2, Tech. Rep. RFC 2608, IETF, June 1999.

[25] O. Nierstrasz, T.D. Meijler, Requirements for a composition language, in: ECOOP '94: Selected Papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-based Models and Languages for Concurrent Systems, Springer-Verlag, London, UK, 1995, pp. 147–161.

[26] M. Satyanarayanan, Pervasive computing: Vision and challenges, IEEE Personal Communications (2001) 10–17.

[27] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, T.D. Chandra, Matching events in a content-based subscription system, in: PODC '99: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 1999, pp. 53–61.

[28] J. Robinson, The exploration and design of a language and middleware architecture dedicated to service composition in a pervasive computing environment, Ph.D. Thesis, Dept. of Informatics, University of Susse, June 2006.

[29] T. Owen, I. Wakeman, B. Keller, J. Weeds, D. Weir, Managing the policies of non-technical users in a dynamic world, in: IEEE 6th International Workshop on Policies for Distributed Systems and Networks, Stockholm, Sweden, 2005.

[30] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, J. Sztipanovits, OASiS: A Programming Framework for Service-Oriented Sensor Networks.

[31] P. Costa, G. Coulson, C. Mascolo, G.P. Picco, S. Zachariadis, The RUNES middleware: A reconfigurable component-based approach to networked embedded systems, in: Proc. of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications, PIMRC'05, Berlin, Germany, 11–14 Sept. 2005.

[32] S. Zachariadis, C. Mascolo, W. Emmerich, SATIN: A component model for mobile self-organisation, International Symposium on Distributed Objects and Applications (DOA), Agia Napa, Cyprus, October, 2004.