SeDeUse: A Model for Service-Oriented Computing in Dynamic Environments

Hervé Paulino and Carlos Tavares

CITI - Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Portugal
herve@di.fct.unl.pt

Abstract. The current state-of-the-art in service-oriented computing targets mostly business-to-business interaction, as service directories store business specific instead of general, abstract, interfaces. Moreover, the established coordination models were designed to operate mainly over business processes with immutable, previously known, locations and tightly couple resource awareness and usage, inhibiting the programmer to separate the purpose of the program from its execution environment. In this paper we present SeDeUSe, a model that features novel programming abstractions sustained by a middleware layer that hides the idiosyncrasies of using service-oriented computing in highly dynamic environments.

Keywords: Service-oriented computing, Middleware for service-oriented computing, Middleware for mobile computing.

1 Introduction

The increase of networks composed of mobile and pervasive devices, and their interaction with the Internet, has established these as one of the main driving forces behind the research on distributed systems, and one of the top priorities of the service-based Internet business market. Nowadays, with the Internet available everywhere, the possibility of using a service deployed anywhere in the world is a reality. This fact has highly contributed to the current popularity of the service-oriented computing (SOC) paradigm.

Services, however, can be used not only to abstract businesses but also to abstract common publicly available resources, such as a network printer. Nonetheless, the current state-of-the-art in SOC targets mainly business-to-business interaction, as service directories store business specific instead of general, abstract, interfaces. Moreover, the established coordination models, such as service orchestration [1] and choreography [2], were designed to operate mostly over business processes with immutable, previously known, locations and tightly couple resource awareness and usage, inhibiting the programmer to separate the purpose of the program from its execution environment.

Several solutions have been proposed to bridge these limitations [3,4,5,6,7] but, in our opinion, it is time to think the problem from scratch and design programming abstractions custom made for this kind of environment.

In this paper we present SeDeUse, a model that hides the idiosyncrasies of using SOC in highly dynamic environments, such as the one composed of mobile devices. The model was designed with the service end-user in mind and, thus, focuses on the proposition of novel intuitive abstractions for service use. Resource (or service) awareness is completely abstracted from the functional logic by using the two-level application construction found in policy and aspect-based approaches [8,9,5,7]. This provides complete resource awareness isolation, which allows for the definition of a clean and simple coordination language for the functional components.

A middleware layer abstracts the user from the dynamic nature of the execution environment. It resides between the application and the services' technologies and provides transparent dynamic discovery, acquisition and management of services.

As discussed in [10], services and software mobility provide a good execution environment for today's networks. This reasoning inspired our middleware that also supports the migration of computations, providing suitable support for moving computation away from devices with low computational resources. The novelty is that the migration is associated to the nature of the services in use, being completely transparent to the functional logic. This means that the exactly same functional code can be used in both stationary and mobile settings.

The remainder of the paper is structured as follows: the next two sections present the state-of-the-art in services and software mobility in dynamic environments; section 4 presents the SeDeUse model; section 5 better illustrates the proposed concepts with a simple programming example; and, finally, section 6 presents some conclusions and future guiding lines.

2 Service-Oriented Computing in Dynamic Environments

Combining service-oriented with mobile and pervasive computing is a hot research topic that intends to port to dynamic environments the concepts of service-oriented architectures. However, as said before, services can be used not only to abstract business services but also to abstract common publicly available resources, such as a network resource (e.g. a printer) or a sensor (e.g. road condition information from sensors placed in a highway). This means that resource bindings may be ephemerous, thus requiring dynamic reconfiguration on the client side.

Although this execution model maps seamlessly in the loosely bound components of service-oriented architectures, the porting of SOC to these environments is not trivial. The state-of-the-art in SOC is mostly targeted at static environments and the coordination technologies operate over businesses, each with its own interface and logic. Service directories, such as UDDI [11], are used to store specific business interfaces instead of general, abstract, interfaces. For example, every insurance company registers its own specific interface (a WSDL [12] description in the usual Web Service technology), even if all of them provide the same kind of service, This is not the suitable for dynamic reconfigurations where bindings must be done towards abstract interfaces instead of specific instances.

Some work as already been done in the field of service description categorization and uniformity. The most known concept is the one of *semantic service-oriented architectures* that uses ontologies to upgrade service descriptions with semantic information. Popular examples of these description languages are OWL-S [13] and RDF [14]. Although some of these languages have been ported to Java APIs [15,16] and semantic service discovery can be incorporated in current UDDI directories [17], no standard APIs have yet aroused.

Regarding service coordination, this has been a research topic for quite sometime, and technologies such as BPEL [1] are widely used. These, however, were designed to coordinate business process that resort static bindings to services and the support for dynamic binding is very limited. The limitations of BPEL are discussed in detail in [3], a paper that proposes JOpera, a visual composition language that uses reflection to control, from within a composition, the binding and registry of the services available in the system.

Other approaches, such as WS-Binder [4] and the MASC middleware [5] provide dynamic binding by associating BPEL service bindings to locally generated proxies instead of particular service instances. These proxies are responsible for discovering services on-the-fly and relaying the invocations. The difference between both is that WS-Binder uses a graphical interface (the service integrator) for the user to specify the constrains to impose on a service (QoS or attribute values) while MASC resorts to policies by extending WS-Policy [18].

Constraints over services can be seen as a cross-cutting concern and thus the application of aspect-oriented programming (AOP) to this domain came naturally. Aspects have been proposed to solve several limitations of BPEL. AO4BPEL [6], for instance, concentrates in the ability to introduce cross-cutting concerns, such as logging or auditing, and alter the composition logic at runtime, i.e., use aspects to dynamically add/remove services to/from the process workflow. WSML (Web Services Management Layer) [7] provides dynamic selection and integration of services. Sequence diagrams are used to map abstract interfaces into concrete ones, which requires the definition of a diagram for each concrete service interface available. Based on this mappings, a middleware layer selects among the services currently available the ones to use. Actual interaction with services is done through a redirection aspect (a stub). Aspects are also used to define the non-functional properties of services.

3 Software Mobility in Dynamic Environments

Software mobility has been a research for quite some time now and many approaches have been proposed. Here we present the systems that are closer to our work, i.e., that focus on dynamic environments.

Lime (Linda in Mobile Environments) [19] brings the Linda model [20] to the world of software mobility in mobile environments. Mobile agents travel between (possibly) mobile hosts that are seen as *roaming boxes* which host the agents, providing them an execution environment. Lime introduces the concept of *transiently shared tuple space* to describe a shared tuple space between agents. The transient concept also applies to hosts and works the same manner. Hosts on the same network may constitute a federated tuple space.

Poema [8] uses policies to decouple mobility from the remainder functionalities. A computational component is divided in three parts: state, application behavior and mobility behavior. State is the data used or created by the component and the application behavior is the definition of how the application will use or produce the data. No mobility behavior goes on the application behavior part, there is complete separation of those concerns. Mobility is defined externally in a second phase through the use of policies: declarative event-condition-action rules used to reconfigure the application.

Mob [10] is a service-oriented scripting language for programming mobile agents in distributed systems. The main novelty of the language is the integration of both the service-oriented and mobile agent paradigms. Services may be provided transparently by several agents in the network which is especially important in networks with volatile resources. Agents may be simultaneously clients and servers, creating a more flexible framework for implementing distributed applications. The downside is that the language uses its own interfaces to specify services instead of standard technologies, such as Web services.

4 The SeDeUse Model

The SeDeUse model intends to hide the idiosyncrasies of using SOC in dynamic environments by proposing a set of intuitive programming abstractions sustained by a middleware layer that lives between the application and the standard service technologies.

The programming model follows the two-layer approach to software construction that can be found in most of the systems discussed in the previous section (WS-Binder, MASC, Poema, and the AOP based approaches). This allows to separate service usage (functionality) from awareness (non-functionality). The two layers of SeDeUse are:

– The **service awareness layer** (SAL) that defines the *kinds* of services to be used in the application, i.e., the services to be discovered in the network and the criterias that each of these must obey. We denote as kind the constraining of a service interface with a set of specific properties.
– The **service use layer** (SUL) that defines a simple coordination model, orthogonal relatively to the common existing programming languages (as is the Linda model [20]).

The model does not define a complete language and, thus, must rely on a hosting language to perform computation and to interact with the middleware

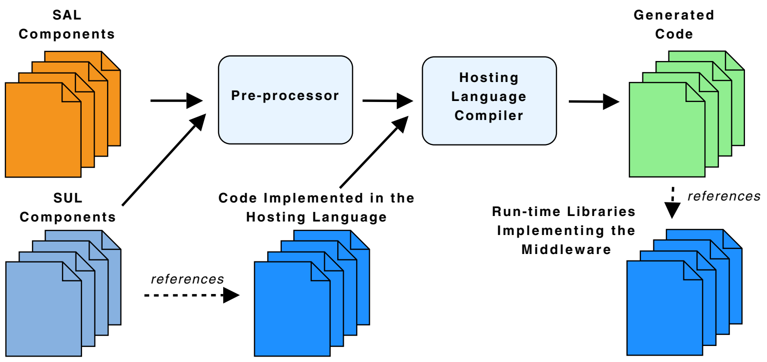
SAL Components
Pre-processor
Hosting Language Compiler
Generated Code
SUL Components
Code Implemented in the Hosting Language
references
Run-time Libraries Implementing the Middleware
references

Fig. 1. The compilation process

layer. As such, a pre-processing stage is required to generate code of the hosting language, which is then compiled by the language's own compiler, as illustrated in figure 1.

The generated code will naturally resort to the run-time middleware libraries that provide for service discovery based on structure and content, dynamic reconfiguration of the service bindings, transparent process mobility and failure recovery.

In the remainder of this section we will present both layers of the model in an informal manner.

4.1 Service Awareness Layer

The syntax for both SAL and SUL relies on the identifiers and values defined in table 1. We denote a sequence of zero or more elements of a given category γ by $\tilde{\gamma}$, an empty sequence by ϵ and an optional symbol or production by the usual [ ] notation.

Table 1. Values and identifiers

s, r	Service identifier
o	Service operation identifier
a	Variable identifier
t	Type identifiers of the hosting language
x	Exception identifiers of the hosting language
v	Values of the hosting language

As illustrated in table 2, the SAL is composed by a sequence of service kind declarations. A kind is defined by a service interface identifier and a set of attribute-value associations. The former defines the key for discovering the service in the available repositories, while the later narrow the search space by

Table 2. Syntax of declarative components

D	$::=$	$D\ D$	Sequence of declarations
	$\mid$	$s\ \{\ \tilde{A}\ \}$	Service kind declaration
	$\mid$	$s\ \{\ \tilde{A}\ \}$ **alias** r	Service kind declaration with alias
A	$::=$	[**pref**] $a\ =\ v$	Attribute constraint
	$\mid$	[**pref**] a **in** $\{\ v_1, v_2, \ldots, v_n\ \}$	Attribute soft constraint

imposing constraints on the service's attributes. These constraints may be hard, defined by a single value (the = operator), or soft, allowing the attribute to range over a set of values (the **in** operator). It is also possbile to declare them as simple preferences by using the **pref** keyword.

Aliases are introduced to avoid name clashing when requiring distinct kinds of the same service. These aliases are service identifiers and thus can be used both in the SAL and the SUL. To better illustrate these concepts we introduce a small example.

Consider a service interface Printer that defines a printing service that features attributes type, colors and paper, among others. We have, on the listing 1, the definition of a printer that prints in black and white on *letter* paper and, on listing 2, the definition of a second printer that prints in color on *A4* paper. The conjunction of a SUL with one of these printer declarations will produce different results, although the program is always the same. Recall that the declaration of the attributes of the Printer service does not define a concrete printer, but rather a printer kind.

Listing 1. A printer

```
Printer {
  colors = "blackandwhite",
  paper = "letter"
}
```

Listing 2. Another printer

```
Printer {
  colors = "color",
  paper = "a4"
}
```

Many kinds of printers can be used in a single program. In order to distinguish between them we must resort to aliases, as in the listing 3. The ColorPrinter identifier stands for a service Printer with the specified attributes. ColorLaserPrinter further constrains the scope of the search by defining the type attribute of ColorPrinter as one of laser or ink jet.

Listing 3. More printer type definitions

```
Printer { colors = "color", paper = "a4" } ColorPrinter
ColorPrinter { type in {"laser", "ink jet"} } ColorLaserPrinter
```

4.2 Service Usage Layer

At this level the purpose is to provide good programming abstractions and a simple coordination model, in order to delegate computation in network services. The syntax for this layer is defined in table 3.

Table 3. Syntax of functional components

P	$::=$	**use** $\tilde{S}$ **in** $c(\tilde{a})$ P $\tilde{X}$	Service use abstraction
	$\mid$	$P \mid P$	Parallel composition
	$\mid$	P ; P	Sequential composition
	$\mid$	**{** P **}**	Grouping
	$\mid$	$[a =]$ E	Assignment
	$\mid$	**retry in** e	Restart a transaction
	$\mid$	$\tilde{i}$	Hosting language process
E	$::=$	**new** $c(\tilde{e})$	An instance of an use abstraction
	$\mid$	$s.o(\tilde{E})$ $\mid$ $s[e].o(\tilde{E})$	Method invocation
	$\mid$	e	Hosting language expression
S	$::=$	[**volatile**] E s $\mid$ [**volatile**] **all** s	Service allocation
X	$::=$	**catch** $(x\ a)$ **{** P **}**	Exception handling

Defining Computations: Actual computation is performed by processes (sequences of instructions) of the hosting language or by external services. Processes can be parallelly or sequentially composed[1]. For each parallel composition $P \mid P'$ a new thread is created to execute the rightmost process. The **;** operator defines a synchronization point, guaranteeing that all threads spawned from the current execution flow have terminated their execution. An example is illustrated in figure 2.

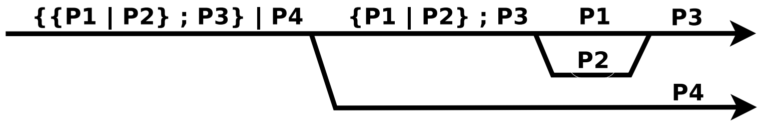

Fig. 2. Execution flow for $\{\{\ P1 \mid P2\ \}\ ;\ P3\ \} \mid P4$

Using services: Regarding process abstraction and service usage, the syntax borrows many constructions from the Object-Oriented languages: the **use** construct abstracts computation in a set of parameters much like a class; instances of such abstractions are created by the **new** construct, that binds the parameters of the abstraction to the values supplied as arguments, and; service operations are invoked as methods upon objects.

The novelty regarding computation abstraction is that the **use** construct allows for code to be also abstracted in service identifiers and that these are bound transparently, and on-the-fly, whenever an instance is created. Once bound, these identifiers can be target of service operation invocations within the abstracted code. For example, the value for the parameter doc in listing 4 is passed in the

[1] Regarding parallel composition we borrowed the syntax behind some process algebras, such as the π-calculus [21].

constructor, while the binding for service parameter Printer is obtained transparently by the middleware.

Listing 4. Using a service

```
use Printer in MyPrinter(doc) { Printer.print(doc) }
new MyPrinter(''myDocument'')
```

The pre-processor translates a **use** abstraction into a hosting language abstraction (e.g. a class in Java or C#). The constructor of this class will access the middleware to obtain instances of the required services. For a given service kind the procedure is as is illustrated in figure 3 and described below:

1. The program queries the middleware to obtain a service matching the Printer kind.
2. Check if a proxy for the service interface exists[2] (i.e. has already been generated)
   – If so, this proxy has access to the middleware's cache of previously discovered services. Look up this cache and check if any of the stored services is still available in the network.
     - If so, return this service.
     - If not, proceed to point 3.
   – If not, proceed to point 3.
3. Perform a search on the available service repositories. Remember that this discovery must obey the criterias defined in the SAL.
4. Analyse the outcome of the search:
   – If no service is found issue an exception.
   – If a service is found proceed to point 5.
5. Check if proxies for the retrieved services have already been generated. This step is required because proxies depend on the invocation technology supported by the server. For instance, the server may only support SOAP 1.1 and the middleware may only have generated a proxy that uses SOAP 2.0. In this case a proxy using SOAP 1.1 would be generated.
6. Store the retrieved services in the cache.

Another feature provided by **use** is the ability to easily bind to distinct services of a given kind. This is useful to load balance service requests or even to synchronize data between service providers. In the code these services are accessed as an array, as is illustrated in listing 5 that uses two services of kind SearchEngine.

Listing 5. Using several instances of a service

```
use 2 SearchEngine in Search(query) {
    SearchEngine[0].search(query) |
    SearchEngine[1].search(query)
}
```

[2] Proxies are bound to service interfaces in general, this means that distinct kinds of the same service interface use the same proxy.

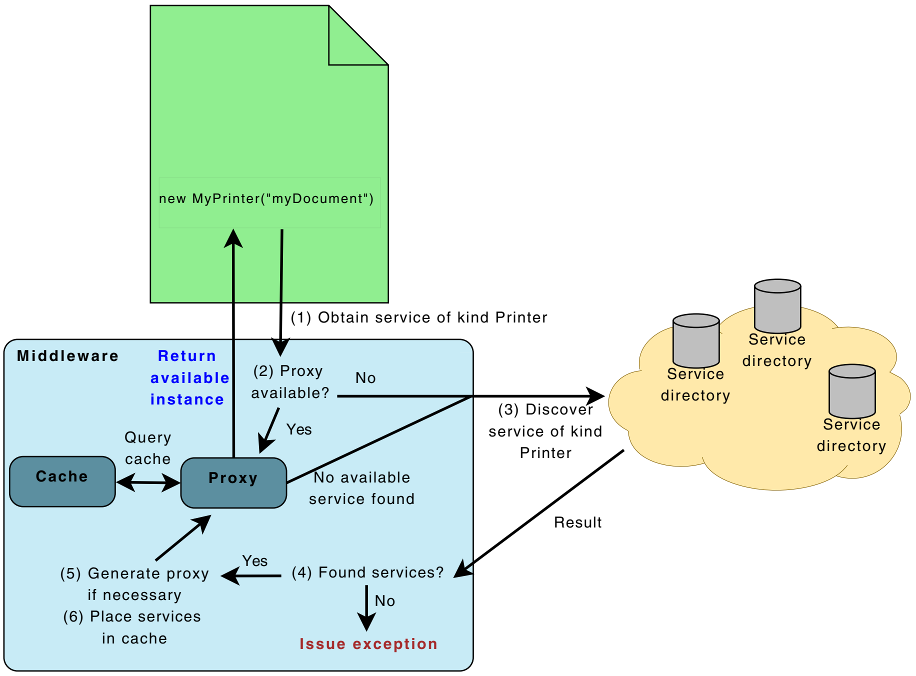
new MyPrinter("myDocument")
(1) Obtain service of kind Printer
Middleware
Return available instance
(2) Proxy available?
No
Yes
Query cache
Cache
Proxy
No available service found
(3) Discover service of kind Printer
Service directory
Service directory
Service directory
Result
(5) Generate proxy if necessary
(6) Place services in cache
Yes
(4) Found services?
No
Issue exception

Fig. 3. Obtaining instances of a service kind

This feature raises the problem of issuing an exception whenever the number of requested services is not available. To avoid this behavior, the value supplied does not define the exact number of services to retrieve, but rather the upper bound. This is a design choice that, in our opinion, provides a more flexible semantics desired for dynamic environments. To ensure that no out-of-bounds exceptions occur, indexes are always converted to values between 0 and the number of services available.

Often one wants the code to be agnostic regarding the number of services currently in use, i.e., omit the index and simply invoke operation regardless of the target. This is possible in SeDeUse because $s.o(\tilde{e})$ in fact stands for $s[\mathsf{i}{+}{+}].o(\tilde{e})$, where i is an integer initialized with 0. Therefore, when more than one instance of a required kind is available, the service identifier ranges the instances using a round-robin strategy.

The abstractions presented until now allow for the simple definition parallel flows of execution that may access distinct services of the same kind in the network, and thus increase resource usage efficiency. for example in the listing 6 two search operations are done in parallel both in the client, since two flows of execution are defined, and also (possibly) in the server side, since probably more then one instance of SearchEngine is being transparently used.

Listing 6. Using several instances of a service transparently

```
use 2 SearchEngine in Search(query) {
   SearchEngine.search(query) |
   SearchEngine.search(query)
}
```

Failure recovery: SeDeUse features an exception handling mechanism that is syntactically similar to the one found in Java [22]. This mechanism is used to protect the code inside a **use** against broken service bindings. If the discovery or the invocation of one of the required kinds fails the exception is raised. The difference from the usual exception handling mechanisms is that here the scope of the protection is not the delimited code, but rather the use of the services required by the instance of **use**.

Consider the code in listing 7, once the setPrinter method invocation is executed the code in the **use** terminates. The service, however, is probably still being used by vpc and while this is true the handler will be active, thus avoiding duplicating exception handling code.

Listing 7. Handling exceptions

```
use Printer in VirtualPrinter(VirtualPC vpc) {
   vpc.setPrinter(Printer);
}
catch (ServiceException e) {
  vpc.unsetPrinter();
}
```

The code delimited by a **use** can be seen as a transaction. If an exception is raised the transaction can be restarted with **retry**. The instruction restarts the discovery procedure, eliminating the existing instance from the middleware's cache.

When the invocation of a service operation does not depend on any of the previous there is no notion of state. In fact, the code could be decomposed in several **use**s as in listing 9. In this case, **use** does not define a transaction but simply the scope of the service kinds required, and so, the search for a new service can be done without forcing the transaction to restart. This feature is supported in SeDeUse by the qualifying the required kinds with the **volatile** keywork, as shown in listing 9.

Listing 8. Decomposing a use

```
use Service in Abs() {
  Service.op1()
}
use Service in Abs() {
    Service.op2()
}
```

Listing 9. Using the volatile keyword

```
use volatile Service in Abs() {
    Service.op1();
    Service.op2()
}
```

Handling Software Mobility: It is not our intention to explicitly refer to mobility. The program is not explicitly ordered to visit a certain host, as is common in mobile agent systems [10,23,24]. It is the nature of the services it requires that will define its location. A special attribute (@) allows the programmer to state if a given resource must be, or should be, either local or remote to the computation and to the device itself. Of course that this migration can only happen if the target host is willing to accept the incoming code. The odd identifier prevents name clashes with existing attribute identifiers.

The values of the attribute may range from: local, to ensure that the resource is local to the device, remote, to ensure the opposite; coupled, to ensure that the resource is local to the computation (not the device), closest, to state that the resource and the computation must as close as possible, and; performance, to delegate in the system the choice of the best instance regarding performance.

When a new instance is created and the service bindings solved, the instance is passed to the middleware that, according to the criterias chosen and the willingness of the servers, decides where the execution takes place. Remote execution may require proxies to be created on the server side and, more important, some services may not be reachable from the new location. This will trigger a new discovery procedure and possibly an exception.

5 Programming Example

We now present a simple programming example to better illustrate the concepts and the capabilities of our model. We first need to choose a hosting language, in order to have completeness. Our choice falls on Java, since it is a widely known language.

Our example is an operating system shell that enables a small portable device to use existing resources in a local network to perform computation, display its desktop environment and print documents. With this application the device can simply serve as an interface between the user and the actual computational resources it is using. In listing 10 we show how SeDeUse can be used to manage the bindings of application.

We choose to associate state to the CPU service to simulate a remote shell session. Thus, lines 29 to 47 are seen as a transaction. If no CPU service is found or the connectivity is lost, the exception handling code is trigerred and the user may decide to search for a new CPU (we assume the existence of classes Question and Info that, respectively query and inform the user).

The actual code of the transaction creates an object instance of a local class that manages the availability of the printer and display resources, instances of VirtualPrinter and VirtualDisplay, respectively.

VirtualPrinter (lines 1 to 7) is responsible for discovering printers in the network, making one available to the user whenever it is possible. Note that the service is volatile, thus as long as printers are available no exception is raised. The user is notified every time printers are made available or not.

VirtualDisplay (lines 9 to 27) discovers the displays available in the network prompting the user if it must continue the search of keep the last service found. If the connectivity to the display in use is lost, the application resorts only the local display and restarts the discovery procedure.

Listing 10. A virtual PC - SUL layer

```
use volatile Printer in VirtualPrinter(VirtualPC vpc, int minutes) {
   vpc.setPrinter(Printer);
}
catch (ServiceException e) {
  vpc.unsetPrinter() ;
  retry in minutes;
}

use Display in VirtualDisplay(VirtualPC vpc, int minutes, boolean search) {
   Question q = new Question("Found new display: " + Display.getInfo() + ". Keep?");
   if (q.getBool())
      vpc.setDisplay(Display);
}
catch (ServiceException e) {
   if (search) {
       new Info("Switching to local display for now. Performing searches periodically");
       retry in minutes;
   }
   else {
      Question q = new Question("Remote display not available. Want to search for a new one?");
      if (search = q.getBool()) {
         new Info("Switching to local display while performing search") |
         retry in 0;
      }
   }
}

use CPU in VirtualPC() {
   class _VirtualPC(CPU c) {
      CPU c;
      Printer p;
      Display d;

      setPrinter(Printer p) {
         this.p = p;
         new Info("Printer " + p.getId() + " available");
      }
      unsetPrinter() {
         this.p = null;
         new Info("Printer not available");
      }
      setDisplay(Display d, boolean local) { this.d = d; }
      ...
   }
   _VirtualPC vpc = new _VirtualPC(CPU) |
   new VirtualDisplay(vpc, 10, false) |
   new VirtualPrinter(vpc, 10)
}
catch (ServiceException e) {
   Question q = new Question("CPU service not available. Want to search for a new one?");
   if (q.getBool()) {
      retry in 0;
   }
}
new VirtualPC();
```

Listing 11. A virtual PC - SAL layer

```
 Printer { @ = "closest" }
 Display { type = "TFT" }
CPU { processor in {Intel, AMD}, @ = "coupled" }
```

In listing 11 we define a simple SAL to define the CPU, Display and Printer kinds.

6 Conclusions and Future Work

The service-oriented computing paradigm provides the ideal framework for resource abstraction, since resources can be modelled as services. However, the current state-of-the-art does not handle adequately the porting of these software architectures to dynamic environments.

Some approaches have been proposed to cope with these limitations, but they feel more like patches than real solutions. Our approach is to back to the foundations and design a model that is tailored to use services in this context. The

SeDeUse model features novel abstractions that are simple and orthogonal relatively to the common existing programming languages. It features two layers that separate service usage from awareness and that must be combined to generate the final code to execute.

In order to move computation away from devices with low computational resources, SeDeUse uses a special service attribute that provides complete transparent software mobility to the functional components. Thus, mobility is no longer coupled with computation. The exactly same functional code can be used in both stationary and mobile settings. It all depends on the restrictions applied to the services (resources).

The expressiveness and capabilities of the model were illustrated in a simple example, by resorting to Java, as the hosting language. In our opinion, the model provides a good framework for the development of distributed and mobile software. In turn, the loosely-bound properties of service-oriented computing, plus the ability to migrate computation provides a good support for the deployment of applications in highly dynamic environments, such as the ones composed of mobile devices.

Ongoing work focuses on the actual implementation of the model, using Java as the hosting language. The middleware will resort to the language's native code mobility support and the APIs available for service discovery and usage, namely for UDDI interaction and SOAP based communication.

References

1. Chen, L., Wassermann, B., Emmerich, W., Foster, H.: Web service orchestration with bpel. In: ICSE 2006: Proceeding of the 28th international conference on Software engineering, pp. 1071–1072. ACM, New York (2006)
2. Yang, H., Zhao, X., Qiu, Z., Pu, G., Wang, S.: A formal model for web service choreography description language (ws-cdl). In: IEEE International Conference on Web Services 2006, pp. 893–894. IEEE Computer Society, Los Alamitos (2006)
3. Pautasso, C., Heinis, T., Alonso, G.: Jopera: Autonomic service orchestration. IEEE Data Engineering Bulletin 29 (2006)
4. Penta, M.D., Esposito, R., Villani, M.L., Codato, R., Colombo, M., Nitto, E.D.: Ws binder: a framework to enable dynamic binding of composite web services. In: SOSE 2006: Proceedings of the 2006 international workshop on Service-oriented software engineering, pp. 74–80. ACM, New York (2006)
5. Erradi, A., Maheshwari, P.: Dynamic binding framework for adaptive web services. In: ICIW 2008: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services, pp. 162–167. IEEE Computer Society, Washington (2008)
6. Charfi, A., Mezini, M.: Ao4bpel: An aspect-oriented extension to bpel. World Wide Web 10(3), 309–344 (2007)
7. Verheecke, B., Cibrán, M.A., Vanderperren, W., Suvée, D., Jonckers, V.: Aop for dynamic configuration and management of web services. Int. J. Web Service Res. 1(3), 25–41 (2004)
8. Montanari, R., Lupu, E., Stefanelli, C.: Policy-based dynamic reconfiguration of mobile-code applications. Computer 37(7), 73–80 (2004)

9. Talcott, C.L.: Policy-based coordination in pagoda: A case study. Electronic Notes Theoretical Computer Science 181, 97–112 (2007)
10. Paulino, H., Lopes, L.: A programming language for service-oriented computing with mobile agents. Software Practice and Experience 38(7), 705–734 (2008)
11. Bellwood, T., et al.: Uddi version 3.0.2, http://uddi.org/pubs/uddi_v3.htm
12. Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl
13. OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/
14. Resource Description Framework (RDF), http://www.w3.org/RDF/
15. McBride, B.: Jena: Implementing the rdf model and syntax specification. In: SemWeb (2001)
16. JRDF (Java RDF), http://jrdf.sourceforge.net/
17. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Importing the semantic web in UDDI. In: Bussler, C.J., McIlraith, S.A., Orlowska, M.E., Pernici, B., Yang, J. (eds.) CAISE 2002/WES 2002. LNCS, vol. 2512, pp. 225–236. Springer, Heidelberg (2002)
18. Bajaj, S., Box, D., Chappell, D., Curbera, F., Daniels, G., Hallam-Baker, P., Hondo, M., Kaler, C., Langworthy, D., Nadalin, A., Nagaratnam, N., Prafullchandra, H., von Riegen, C., Roth, D., Schlimmer, J., Sharp, C., Shewchuk, J., Vedamuthu, A., Yalçinalp, U., Orchard, D.: Web services policy 1.2 - framework (ws-policy). Technical report, W3C (2006)
19. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15(3), 279–328 (2006)
20. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
21. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes (parts I and II). Information and Computation 100(1), 1–77 (1992)
22. Sun Microsystems, Inc.: Java tutorial, http://java.sun.com/docs/books/tutorial/
23. Lange, D.B., Oshima, M.: Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley, Reading (1998)
24. Glass, G.: Objectspace voyager - the agent orb for java. In: Masunaga, Y., Tsukamoto, M. (eds.) WWCA 1998. LNCS, vol. 1368, pp. 38–55. Springer, Heidelberg (1998)