Coupling Contracts for Deployment on Alien Grids

Javier Bustos-Jiménez[3], Denis Caromel[1], Mario Leyton[1], and José Piquer[2]

[1] INRIA Sophia-Antipolis, CNRS-I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France
First.Last@sophia.inria.fr

[2] Departamento de Ciencias de la Computación, Universidad de Chile. Blanco Encalada 2120, Santiago, Chile
{jbustos,jpiquer}@dcc.uchile.cl

[3] Escuela de Ingeniería Informática. Universidad Diego Portales Av. Ejercito 441, Santiago, Chile
javier.bustos@udp.cl

Abstract. We propose coupling based on contracts as a mechanism to address the problem of exchanging information between parties that require information to work together. Specifically, we show how our approach can be used to couple the deployment of an application with a Grid infrastructure deployment descriptor using ProActive[11,2].

To achieve this, we identify the properties related with information exchange between parties, and we group the properties of interest into typed clauses. We then propose that interfaces can be built using shared typed clauses. If the interfaces between parties are compatible, the coupling of the interfaces can yield a coupling contract. The clauses belonging to the contract represent *what* information can be shared between the parties, and the type of the clause specifies *how* this information will be shared.

Finally, we show how the deployment of applications on the Grid can benefit from the proposed approach. Unfamiliar applications can couple with deployment descriptors to deploy on alien Grids, without modifying or inspecting neither of them.

1 Introduction

Originally, distributed resources were managed using a centralized approach. This has been shown to be unpractical in the Grid. The resources can be numerous, heterogeneous, with distributed ownership, and having different policies [8,14].

The problem of scheduling an application on distributed resources was addressed using different strategies. This generated a diversity of mechanism for resource acquisition protocols (LSF [16], PBS[10], SGE[9], Globus-gram[8], etc.). At that point in time, application developers were forced to choose and bind an application to a specific resource acquisition protocol. Migrating from one resource acquisition protocol to another required modifying the application.

W. Lehner et al. (Eds.): Euro-Par 2006 Workshops, LNCS 4375, pp. 61–73, 2007.

Later, new levels of abstractions were introduced which allowed the application developers to abstract the application, not only from the resource acquisition protocol used, but also from other Grid infrastructure details such as communication protocols, software location, etc.[3].

In the current scenario, we can now imagine having repositories of applications and repositories of Grid infrastructures. The problem of finding a suitable Grid infrastructure for an application can be seen as a problem of classified advertisements and matchmaking [12,13] or a problem of database search like UDDI web services [6].

We set sail from this point. Let us imagine two candidate parties (ex: application and Grid infrastructure) that have already been matched. To work together, each party requires and provides information from the other. We propose coupling based on contracts as a mechanism to address the problem of exchanging information in a generic way between unfamiliar parties. Specifically, we show how our approach can be used to couple the deployment of an unfamiliar application with an unfamiliar Grid infrastructure descriptor using ProActive[11]. Therefore, our objective is the deployment of an application on a Grid infrastructure without modifying or inspecting either.

This paper is organized as follows. In section 2 we review the related work. Then in section 3 we explain our coupling proposal, and in section 4 we show how this proposal is applied for deployment on the Grid using ProActive. Finally we conclude and present our future work in section 5.

2 Related Work

The problem of finding suitable resources for a given application have already been addressed by techniques such as matchmaking in Condor [12,13], collections in Legion [4], or using resource management architectures like Globus[5].

In the case of Condor, the resource acquisition is viewed as a three stage process composed of advertisement, matchmaking, and claiming. The requirements are advertised by the involved parties (jobs and resources), suitable matches are found, and finally the claiming of the resources takes place. To achieve the claiming, the advertised information from each party is exchanged.

While this approach has been acknowledged as suitable for finding matches, *how* the advertised information sharing is done has been overlooked. Up to now, techniques like the ones proposed by Condor allow finding suitable matches by specifying *what* information is exchanged, but no mechanism is provided for defining *how* the information exchange should take place.

For example, if an application is looking for n nodes, and a Grid infrastructure can provide m nodes ($n < m$), then these two parties will be matched. If no *how* semantics are provided for the claiming face, the following scenario could happen: the application could decide to take advantage of the m nodes provided by the infrastructure, while the infrastructure can decide to provide only the n

nodes advertised by the application. The result would be the application trying to use m nodes, while the infrastructure is only providing n nodes. Therefore, a mechanism is required to specify *how* the information exchange takes place.

To address this issue, we propose the addition of a new stage called *coupling*, thus rendering four stages: advertisement, matchmaking, coupling, and claiming. Once the matchmaking has taken place, the semantics of *how* the information will be shared between the parties will be addressed in the coupling face, before the resources are successfully claimed.

Another related approach corresponds to the Web Services Agreement (WS-Agreement) Specification[1], which is about to become a draft recommendation of the Global Grid Forum[7]. The WS-Agreement is a two layer model: Agreement Layer and Service Layer. Many of the concepts introduced in this paper find their reflection in the Agreement Layer. According to the specification "an *agreement* defines a dynamically-established and dynamically-managed relationship between parties", much like the proposed coupling contracts. Also, the proposed coupling interfaces can be seen as *agreement templates* in WS-Agreement, since they are both used to perform advertisement. Additionally, in the same way that interfaces and contracts are composed of clauses, in WS-Agreement templates and agreements are composed of *terms*. Finally, the concept of constraints is present in both approaches.

The similarity of the proposed approach and WS-Agreement Specification is encouraging when we consider that both were conceived independently. On the other hand, the main difference in the approaches is that the definition of a protocol for negotiating agreements is outside of the WS-Agreement Specification scope. Therefore, we believe that WS-Agreement could benefit from the proposed automated coupling approach, built using typed clauses. From the WS-Agreement perspective, typed clauses can be seen as an automated negotiation approach because they provide an automated mechanism for accepting or rejecting an agreement.

3 Coupling Matches with Contracts

In this section we describe our approach for coupling parties (ex: application and descriptor) that require exchanging information to work together.

Figure 1i shows the problematic. Unfamiliar parties cannot exchange information with each other in a generic way. Our approach proposes to capture the properties of *how* the information exchange occurs into types (Figure 1ii). A group of typed clauses will then form an interface that will specify *what* information is required and provided by each party (Figure 1iii). The coupling of the interfaces will yield a contract, that will allow the parts to couple and work together on a common goal (Figure 1iv).

In the rest of this section we provide the details on how the parties can couple using the proposed approach. Later in section 4 we will show how this approach

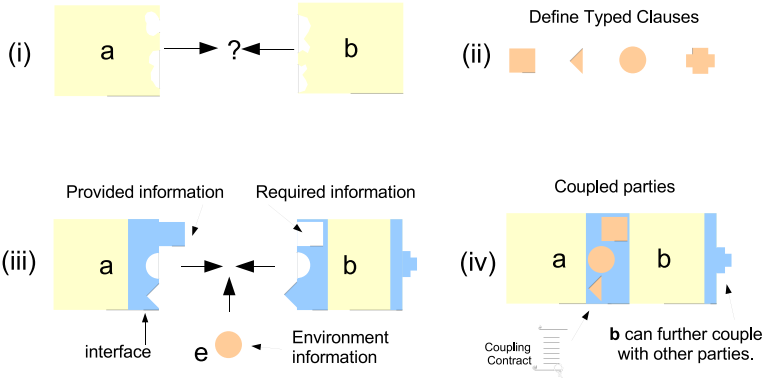

Fig. 1. Coupling Matches with Contracts

can be used to couple distributed application with Grid deployment descriptor using the ProActive[11] Grid middleware.

3.1 Clause Types

Let a and b be matched parties that require information from each other, or from an external source e like the environment to work together. We have identified that the information requirements can be exposed and fulfilled using typed clauses. The type of the clause represents a specific configuration of the following properties:

1. **Ability to set a value.** This defines which party has the ability to set a value for the clause. Possibilities are any permutation of a, b, e: $\{abe, ab, be, ae, a, b, e\}$.
2. **Ability to set empty values**. This defines which party can set this clause as empty. The possibilities are any permutation of a, b: $\{a, b, ab, -\}$.
3. **Ability to set constraints to the values**, thus narrowing the space of possible values. This can be done by providing an explicit list of alternatives, or using comparison operators ($<$, $>$, $=$, ...). The alternatives are permutations of: a, b: $\{ab, a, b, -\}$.
4. **Priority**. If more than one party can set a value, an empty value, or the constraints, this identifies which has the priority. The alternatives are combinations of a, b, e: $\{abe, aeb, bae, bea, \ldots\}$. The order in which they are expressed defines the priority.

For example, we have identified the types depicted in Table 1. Conceptually the types can be interpreted as:

A The value can only be set by a. Since b can set the value to empty, then b can force a to provide a value.
B Corresponds to the symmetrical of A.

Table 1. Types

Type Name — Set value — Set empty — Set constraints — Priority				
A	a	b	-	a
B	b	a	-	b
A-PRI	ab	-	b	ab
B-PRI	ab	-	a	ba
ENV	e	-	-	e

A-PRI The value can be set either by a or b, where b can provide a default value, and a can override the default.
B-PRI Corresponds to the symmetrical of A-PRI.
ENV The value can be set from the environment.

The flexibility of the approach allows defining the types of interest only, and extending the set of types as required. The definition of new typed clauses is possible using these or future imagined properties. For example, we could imagine handling the priorities at a finer grain, thus having to specify three priorities for setting the value, setting the empty value, and setting the constraints. In this work we will focus on the types depicted in Table 1, because these represent the types of interest in section 4.

3.2 Typed Clauses

We will define a typed clause (clause for short) as having the following fields:

1. **Type** Corresponds to one of the allowed clause types. These are: A, B, A-PRI, B-PRI, ENV.
2. **Name** Corresponds to the name of the clause.
3. **Value** The value that will be set, empty or not.
4. **Constraints** The restrictions imposed on the values that can be set, if allowed by the *type*.

We will say that a clause pair named cls_a and cls_b compose a **shared clause** cls if both clauses names match $cls_a = cls_b$. The shared clause cls is **type compatible** if $cls_a.type = cls_b.type$, and incompatible otherwise.

The fields of a type compatible shared clause are defined as:

- Name: $cls = cls_a = cls_b$,
- Type: $cls.type = cls_a.type = cls_b.type$,
- Value: $cls.value = cls.type.priority(cls_a.value, cls_b.value)$
- Constraint: $cls.constraints = cls.type.priority(cls_a.constraints, cls_b.constraints)$

We will say a clause, shared or not, is **valid** if and only if $cls.value \neq empty$ and $cls.value$ satisfies $cls.constraints$ such that: $cls.constraints(cls.value) = true$. Note that two invalid clauses can be separately invalid, but the shared clause composed using both of them can be a valid clause.

3.3 Coupling Interfaces

An coupling interface (interface for short) corresponds to a group of clauses. A party can expose more than one interface, thus allowing coupling with more than one party. An interface is defined by:

1. A name
2. Set of clauses identified by their names

Thus for a party a we can identify an interface by $a.int\_name$. And for identifying a clause belonging to an interface we write: $a.int\_name.cls\_name$.

We will say that two **interfaces can be coupled** ($a.int\_name$ and $b.int\_name$), if there are no type incompatible shared clauses between the interfaces. The result of the interface coupling corresponds to the set of all shared clauses, and will denote it as: $a.int\_name \diamond b.int\_name$.

3.4 Coupling Contracts

A coupling contract (contract for short) corresponds to the interaction between two interfaces of different parties. If there exists two interfaces $a.int$ and $b.int$, such that both interfaces can be coupled, then the contract is defined as a set of clauses:

$$Contract = a.int \diamond b.int \cup (a.int - (a.int \diamond b.int)) \cup (b.int - (a.int \diamond b.int))$$

This means that the clauses contract will contain the shared clauses between the interfaces, the unshared clauses of a, and the unshared clauses of b.

We will say that **two parties a and b can be coupled** if:

1. A contract can be built between them: two interfaces belonging to a and b can be coupled, and
2. the contract is valid: every clause in the contract is valid.

3.5 Matching Parties: Descriptors and Applications Example

Typed clauses can also be used to perform advertisement and matchmaking in the Condor style. Both parties can expose their interface (advertisement) to a matchmaker or broker. To determine if the two parties are a suitable match, the coupling contract can be generated and validated.

The clauses belonging to the interfaces will specify *what* information is shared (provided or required) for the matchmaking. And the type of the clauses will specify *how* the information is shared for the coupling.

4 Coupling Distributed Applications with Deployment on the Grid

In this section we show how the concepts defined in section 3 can be applied. Specifically, we aim at coupling a distributed application with Grid resources

using the Grid middleware ProActive. ProActive already provides a mechanism based on deployment descriptors for deploying on the Grid. We will show how this mechanism can benefit from the use of coupling contracts to couple applications with deployment descriptors.

This section is organized as follows. We will first provide some background on ProActive. Then, we will show how coupling contracts have been incorporated into ProActive.

4.1 Background on ProActive Deployment Descriptors

Within the *ProActive Descriptor Deployment Model* [3], it is possible to deploy applications on sites that use heterogeneous protocols, without changing the application source code. All information related with the deployment of the application is described in an XML Deployment Descriptor. Thus, eliminating references inside the application code to: machine names, submission protocols (local, rsh, ssh, lsf, globus-gram, unicore, pbs, lsf, nordugrid-arc, etc.) and communication protocols (rmi, jini, http, etc.).

The Descriptor Deployment Model is shown in Figure 2.

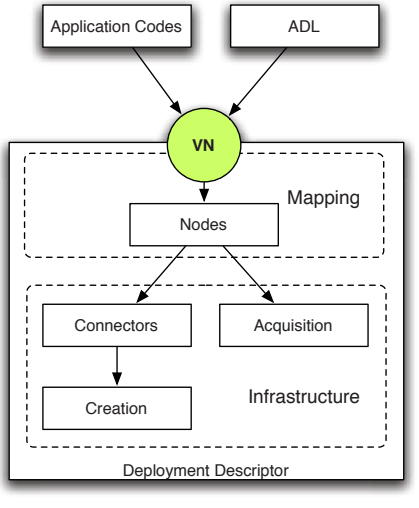

Fig. 2. Descriptor Deployment Model

The infrastructure section contains the information necessary for booking remote resources. Once booked, ProActive Nodes can be created (or acquired) on the resources. To link the Nodes with the application code, a Virtual Node (VN) abstractions is provided, which corresponds to the actual references in the application code. Virtual Nodes have a unique identifier which is hardcoded inside the application and the descriptor.

A deployer can change the mapping of the application → Virtual Node to deploy on a different Grid, without modifying a single line of code in the application.

4.2 The Problematic of Applications and Descriptors

In the traditional approach, the application developer and the descriptor developers need to have a previous agreement on the name of the Virtual Node. This means that the name of the Virtual Node is hardcoded inside the application and the descriptor. If the application wants to use a new descriptor, then either the descriptor or the application has to be modified to agree on the new Virtual Node name.

A possible solution to this problem is passing the Virtual Node name as a parameter to the application. Nevertheless, the problem of figuring out the proper Virtual Node name from the descriptor remains. To find out the name of the Virtual Node, inspection of the descriptor has to be performed, which can be a problem for someone alien with respect to the Grid infrastructure's descriptor.

Furthermore, the Virtual Node name is not the only information sharing problem that the application and descriptor have. For example, a descriptor might be configured to deploy on k nodes, but the application only requires j nodes ($j < k$). Without shared clauses, the descriptor has to be modified to comply with the requirements of the application.

Modifying the application or the descriptor can be a painfull task, specially if we consider that the person deploying the application (deployer) may not be the author of either. To complicate things further, the application source may not even be available for inspecting the requirements and performing modifications. Figure 3 illustrates the issue. The deployer is not aware of the application or descriptor requirements.

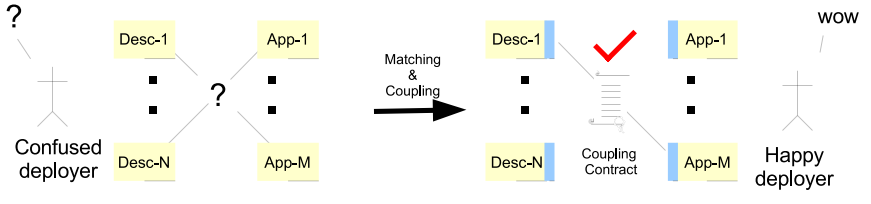

Fig. 3. Matching and Coupling Contracts

Nevertheless, using coupling contracts, the deployment can be further enhanced by enabling automated matchmaking and coupling of applications and descriptors.

4.3 Clause Types

The involved parties are the application (a) and the descriptor (b), and the environment information (e) (given through java properties). To improve the

Table 2. ProActive Deployment Clause Types

Type Name — ProActive Type Name	
A	Application
B	Descriptor
A-PRI	ApplicationPriority
B-PRI	DescriptorPriority
ENV	JavaProperty

clarity of the example, we have renamed the clause types identified in the Table 1 to the names shown in Table 2.

4.4 Clauses in ProActive Descriptors

Clauses can be specified using XML tags as shown in the example of Figure 4 for the descriptor. To define the clauses a new section labeled `clauses` has been added at the beginning of the descriptor to hold the `interfaces`. The clauses shown in the example correspond to:

`PROACTIVE_HOME & MAX_NODES` Correspond to descriptor set clauses. The value is set directly in the descriptor, and can be used later on, inside the descriptor or the application.

`VIRTUAL_NODE_NAME` Corresponds to a clause that the descriptor enforces the application to set. If the application doest not set this value, the clause inside the coupling contract will not be valid, and the application will not be allowed to couple with the descriptor. In the example, we force the application to set the name of the Virtual Node.

`LOAD_BALANCING` Corresponds to a clause that the application has set, but the descriptor can override. In the example, we imagine that an application is capable of handling, or not, the load balancing. By default the application will assume that no load balancing is provided by the Grid infrastructure (Figure 5), and thus handle the load balancing at the application level. Nevertheless, the descriptor is aware if load balancing can be done at the Grid infrastructure level and activate it. The application can then access the contract's clauses to learn if the infrastructure is using the load balancing and disable the application load balancing mechanism.

`NUMBER_OF_NODES` Corresponds to a clause that the descriptor has set a value, but the application may override. Additionally, the descriptor has set constraints indicating that the value must be an integer between 1 and `MAX_NODES`.

`USER_NAME` Corresponds to a clause that is set from the environment. In this case, the username can be specified from the environment as a java property.

Figure 4 also shows an example of how the clauses can be used inside descriptors. Note that the value of the clause `VIRTUAL_NODE_NAME` has not been set in the descriptor, since it is of type Application. This means that the value used inside the descriptor will be the one set from the application. Also note, that clauses obtained from the environment can also be used, like the `USER_NAME` clause.

```
<clauses>
  <interface name="descriptor-example-interface">
    <Descriptor name="PROACTIVE_HOME" value="ProActive/"/>
    <Descriptor name="MAX_NODES" value="100/"/>
    <Application name="VIRTUAL_NODE_NAME" value=""/>
    <DescriptorPriority name="LOAD_BALANCING" value="on"/>
    <ApplicationPriority name="NUMBER_OF_NODES" value="1">
      <!--// (NUMBER_OF_NODES>0) && NUMBER_OF_NODES<=MAX_NODES -->
      <integerConstraint>
        <and>
          <biggerThan>0</biggerThan>
          <smallerOrEqualThan>${MAX_NODES}</smallerOrEqualThan>
        </and>
      </integerConstraint>
    </ApplicationPriority>
    <JavaProperty name="USER_NAME" value="user.name"/>
  <interface>
</clauses>
...
  <virtualNodesDefinition>
   <virtualNode name="${VIRTUAL_NODE_NAME}"/>
  </virtualNodesDefinition>
...
<sshProcess class="org.objectweb.proactive.core.process.SSHProcess"
      hostname="example.host" username="${USER_NAME}"/>
```

Fig. 4. Example of clauses in descriptor

4.5 Clauses in ProActive Applications

We have also provided a mechanism for specifying clauses and interfaces from the application. This can be done through an API, or loading the clauses from an

```
//Create a new interface
ClausesInterface ci= new ClausesInterface("application-example-interface");

//Set the clauses in this interface
  //set(<type>, <clause name>, <value>, [<constraint>])
ci.set(Application, "VIRTUAL_NODE_NAME", "testnode",);
ci.set(ApplicationPriority, "NUMBER_OF_VIRTUAL_NODES", "16");

// LOADBALANCE="on" || LOADBALANCE="off"
OrConstraint oc = new OrConstraint();
oc.add(new EqualsConstraint("on"));
oc.add(new EqualsConstraint("off"));
ci.set(DescriptorPriority, "LOAD_BALANCING", "off", new StringConstraint(oc));

//Parse and load the descriptor using the coupling interface. If the application and
     descriptor can not be coupled an exception will be thrown
ProActiveDescriptor pad = ProActive.getProactiveDescriptor("descriptor.xml", ci);

//Clauses from the coupling contract can be used in the application
CouplingContract cc = pad.getCouplingContract();
String loadBalancing = cc.getValue("LOAD_BALANCING");

//The application can take decisions based on the clauses
if(loadBalancing.equals("on")){...}
else{...}
```

Fig. 5. Example of clauses in application

external XML file. Since the XML approach has already been shown for the descriptor, Figure 5 shows an example using the API. First an *interface* is created, and then the clauses are added to the interface. The interface is then passed as a parameter when parsing the descriptor. The parsing will try to generate a coupling contract using the application's and the descriptor's interfaces.

If the application can be coupled with the descriptor, then the application can retrieve the coupling contract and consult the contract's clauses. For example, using this strategy the application can know if the descriptor activated the infrastructure load balancing, and avoid using the application load balancing.

4.6 Constraints

Constraints are boolean expressions that will be evaluated for each clause when the contract is built. The constraints can be of two types: *integer* or *string*. For each constraint the logical operators: *and*, *or*, *xor* are allowed. Also, boolean operators are provided for each type of constraint. The integer operators are: `biggerThan`, `biggerOrEqualThan`, `smallerThan`, `smallerOrEqualThan`, `equals`. The string case sensitive operators are: `subString`, `superString`, `equals`. Figure 6 shows the constraint grammar specified using XML Schema[15] for the integer type constraints.

```
<xs:element name="integerConstraint">
  <xs:complexType>
    <xs:choice>
      <xs:element name="and" type="intConst"/>
      <xs:element name="or" type="intConst"/>
      <xs:element name="xor" type="intConst"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
</xs:complexType>
  <xs:complexType name="intConst">
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="and" type="intConst"/>
    <xs:element name="or" type="intConst"/>
    <xs:element name="xor" type="intConst"/>
    <xs:element name="biggerThan" type="xs:string"/>
    <xs:element name="biggerOrEqualThan" type="xs:string"/>
    <xs:element name="smallerThan" type="xs:string"/>
    <xs:element name="smallerOrEqualThan" type="xs:string"/>
    <xs:element name="equals" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

Fig. 6. Integer Constraint Schema Grammar

Figure 4 shows an example where the clause `NUMBER_OF_NODES` is constrained to be: $0 <$ `NUMBER_OF_NODES` $<=$ `MAX_NODES`. Note that `MAX_NODES` is defined as a `Descriptor` type clause. Figure 5 shows an example using string constraints. The clause `LOAD_BALANCING` is constrained to be either `on` or `off`.

5 Conclusions and Future Work

We have shown an approach for coupling parties that require exchanging information to work together. To achieve this, we have identified the properties related with information exchange between parties, and we have grouped the properties of interest into typed clauses. We have then proposed that interfaces can be built using shared typed clauses.

If two interfaces between parties are compatible, the coupling of the interfaces can yield a coupling contract. The clauses belonging to the contract represent *what* information can be shared between the parties, and the type of the clauses specify *how* this information will be shared.

Using the proposed coupling approach, we have shown how coupling contracts can be applied for automated deployment of unfamiliar applications on alien Grids. For this, we have provided a mechanisms to specify clauses in the application and the deployment descriptor using the Grid middleware ProActive. As a result, the approach can now be used to couple applications with descriptors, without having to modify or inspect either.

Nevertheless, it can be argued that the proposed approach requires each party to know beforehand the names of the clauses used in the coupling. In reality, only a subset of the clauses belonging to the coupling contract have to be known: the ones that must be provided with a value to make the contract valid. Furthere more, if two different interfaces couple with a third generating two valid coupling contracts, the clauses contained in these contracts can be different. While this seems strange, it is a direct result of the proposed approach being boolean: either the contract is valid or not. In the future, we would like to extend this concept by introducing Conformance Levels in coupling contracts. Thus, a minimum conformance level (i.e. minimum set of known clauses) could be provided for basic applications, and higher conformance levels (i.e. a superset of the lower conformance levels) could be used for more advanced features that require more specific clauses.

From the Grid infrastructure side, in the future we would like to identify standard interfaces for coupling applications with different types of Grids. The idea is to be able to release applications packaged with interfaces that certify the deployment of an application with a Grid interface. On the other hand, from the application point of view, we would like to identify interfaces for common structured parallel programming patterns. For example, if an application uses the master-slave pattern, then it can benefit by coupling with a Grid interface optimized by deploying the master on a more powerfull or better connected resource than the regular slaves. Thus, a Grid could provide an optimized interface for applications exploiting different patterns such as: farm, pipe, divide and conquer, etc.

References

1. Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web services agreement specification (ws-agreement). Draft Version 2005/09. `http://forge.gridforum.org/projects/graap-wg`.

2. L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer Verlag, 2005.
3. F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.
4. Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. Resource management in legion. Legion Winter Workshop, 1997.
5. Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 62–82, London, UK, 1998. Springer-Verlag.
6. D. Fensel and C. Bussler. The web service modeling framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, Summer 2002.
7. Global Grid Forum. `http://www.gridforum.org/`.
8. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit, 1996.
9. Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *CCGRID*, pages 35–39. IEEE Computer Society, 2001.
10. R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.
11. ProActive. `http://proactive.objectweb.org`.
12. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.
13. R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource coallocation with gangmatching. In *Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12)*, 2003.
14. INRIA OASIS Team and ETSI. 2nd grid plugtests report. `http://www-sop.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf`.
15. W3C. Xml schema: Formal description. `http://www.w3.org/TR/xmlschema-formal/`.
16. S Zhou. Load sharing in large-scale heterogenous distributed systems. In *Proceedings of the Workshop on Cluster Computing, 1992.*