Specification Supports and Optimizations for Parallel JavaBean Programs

Chung-Kai Chen, Cheng-Wei Chen, Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30043, Taiwan
{cwchen, ckchen}@pllab.cs.nthu.edu.tw and jklee@cs.nthu.edu.tw

Abstract—In this paper, we propose an ontology specification for JavaBean programs, the object component model of Java. Our specification is written using the DAML+OIL language, which is based on the RDF schema and the XML syntax. The vocabulary of this ontology provides a basic terminology to annotate components with the information about the conditions and suggestions of adopting a component for component specializations. It also gives a reference criteria for choosing the most suitable components at a given time and environment for performances and functionality purposes. With our design of the annotations, it’s also possible to automatically retrieve the annotations of object components, connect them by their object-oriented relationships, organize them to form component databases, and discover them in the databases by component characteristics. This will facilitate the sharing of component resources in the internets. We demonstrate through experiments and examples that specification and ontology supports for components provide a new direction for the new arrival of global-grid like architectures.

Keywords—Ontology Specifications, Parallel JavaBean Programs, Runtime Component Compositions, Code Specializations, Runtime Optimizations, Clustered EJB.

I. INTRODUCTION

With the non-stopping spread of the network facilities, billions of computers have been connected to the internet to share their resources. The data distributed on this growing environments are so huge and diverse, it becomes an important issue to find an efficient way to use and adopt them. The technologies in mining the information and searching for useful data on the web are done mainly by searching engines to locate the data of interest. Recent progresses have also adopted the approach to connect the information in the web by their semantic relationship. The semantic web [15] is formed on this purpose. The semantic web approach is an attempt to give more standard terminology and ontology for application domains so that the catalog of products, the contents of literature, and the information for merchandise can be searched and analyzed in internet environments with precision.

In addition to the annotations and ontology descriptions for data in application domains, the building of ontology descriptions for software component are also important. We consider several possible scenarios below. First, application writers can describe the software specification for the software components in the design process of their software systems. If the ontology descriptions for components are annotated properly, many of the component module could be found via web search in the future. Due to object-oriented techniques, the component resources have common information associated with them, the interfaces they implements. Additional descriptions can be added. For example, a MP3 decoding component may have the version number information, the computing power needed for the lowest sound quality acceptable, the memory footprint of its implementation, and the license about its usage. These properties, we called metadata, address the difference between the implementations of the interface. It helps us search not only the kind of components, but also specific realizations. Next, let’s consider another scenario. If the components of applications are allowed to be re-composed at runtime, the selection of a specialized component for specialized architectures, application characteristics, or additional functionality requirements can be done with proper annotations for component characteristics.

In this paper, we propose an ontology specification for JavaBean programs to attempt to address the issues above. Our specification is written using the DAML+OIL[8] language, which is based on the RDF[16] schema and the XML[14] syntax. The vocabulary of this ontology provides a basic terminology to annotate components with the information about the conditions and suggestions of adopting a component for component specializations. It also gives a reference criteria for choosing the most suitable components at a given time and environment for performances and functionality purposes. With our design of the annotations, it’s also possible to automatically retrieve the annotations of object components, connect them by their object-oriented relationships, organize them to form component databases, and discover them in the databases by component characteristics. This will facilitate the sharing of component resources in the internets. Our annotations are with additional interfaces a component implements, and they are embedded into components so that the ontology interfaces annotated in our design can be investigated by Java reflection API’s. In addition, we also give experiments and application scenarios for employing this ontology specification. We demonstrate the employment of this ontology specification for adapting a parallel matrix component for
performance purposes, and for Java RMI adaptations over heterogeneous network environments. This work provides a new direction for automatically enhancing long-running components by allowing components to improve and specialize themselves continuously in terms of performances and functionality.

II. The Ontology

In this section, we give our proposed conventions and ontology classifications in annotating properties for components. Figure 1 gives a sample of our conventions for ontology represented in DAML+OIL language. We model the adoption knowledge of components in the following vocabulary. They are classified into five classes.

1. **SourceInterface** and **hasExtension** model the source interfaces and their extension relationship. They are used to build the source interface hierarchy and assert the logical validity of adoptions.

2. **hasImplementation**, **implement**, and **Component** are used to describe the implementation relationship between source interfaces and components. Through them and the source interface hierarchy, we can traverse candidates of specified interfaces.

3. **withAnnotation** and **AnnotationInterface** define the annotations of components.

4. **Denotation**, **Requirement**, **Preference**, and **Adminicle** model the annotation interface into four kind of annotations. **Denotation** is for the informational annotation. **Requirement** annotation gives the requirement on adoption. **Preference** provides adoption suggestion and **Adminicle** represents the profilers that dynamically deliver useful information for the criteria of adoption judgement.

5. **judgeCriterion** and **parameter** define how the information from **Adminicle** be used as the criteria of the adoption and the parameter of other annotation.

The vocabulary is formally defined in Figure 1 using DAML+OIL language.

In the vocabulary above, class 1, 2, and 3 are terms directly mapped from the object-oriented design and denoted for the relationship related to the source interfaces and the components. This information can be automatically generated by Java reflection API. For class 4 and 5 above, a naming convention of the annotation interfaces can help translate the methods into the ontology model. Thus given a component with annotation interfaces implemented, we can generate an annotation description in DAML format.

III. Application Examples

A. RMI Component

Java RMI [11] is an important function supported in Java standard library as the communication basis for distributed computing. In addition to the inherent implementation of RMI using TCP/IP sockets, we can also implement RMI over various network architectures. Note that one major part of RMI functionality is carried out by **UnicastRemoteObject**. The RMI remote objects export themselves to provide RMI service by calling **UnicastRemoteObject.exportObject()**. Suppose we have reimplemented **UnicastRemoteObject** to have RMI been connected over VIA, an user-level network architecture. We can have a possible annotation interface shown in Figure 2. Figure 3 then shows the annotation description from **VIARMIImpl** in DAML format. Note the daml format can be automatically generated from annotation interface. Various RMI implementations such as RMI over bluetooth, RMI over nexus, RMI over IB can be annotated with specifications so that the components can be used or adopted for specializations.

B. Parallel Matrix Component

The ontology specification can be used to adapt components at runtime. In our research work, we have been working on enabling the techniques for the run-time composition of parallel components. The ontology specification proposed in this work helps adapt objects dynamically according to application and architecture characteristics. Our runtime support for runtime component compositions is based on the dynamic proxy support of Java [12]. As Java is a statically typed language, objects are created in heap and held by reference variables with a particular type. If we want to re-compose an object, a new object with the same type of subtype can be created in heap and assigned to the original reference variable. However, an object can be referenced by more than one reference variable. In addition, the references can be forwarded as arguments in method invocations. Therefore, assigning an object to a reference can not alter other references pointing to the original object. To avoid problems in component substitution, the proxy object in Java is needed to wrap the original object. Therefore, the reference variables maintain their references to the proxy object, and assigning an object to the proxy object can alter other references to the original object through the proxy object.

IV. Experiment

In the experiments, we enable runtime component compositions in three application scenarios. The examined components are annotated by our component specification. The experimental results thus show the potential benefits

```java
public interface VIARMI { ... } 
public interface Latency extends Denotation {  
    public String getLatency();
}
public interface ViplVersion extends Requirement {  
    public String getRequiredViplVersion();
    public String getViplVersion();
}
public interface PacketSize extends Preference {  
    public int getPreferredPacketSize();
    public int getAvgPacketSize();
}
public class VIARMIImpl implements VIARMI, Latency, ViplVersion, PacketSize {  
    ... 
}
```

Fig. 2. The **VIARMIImpl** Component implements **VIARMI** and several annotation interfaces.
Fig. 1. The Ontology for Optimization and Composition of Parallel JavaBean Programs.
of applying our component technology. The overhead of runtime component compositions is also revealed in these examinations and turns out to be small in experiments.

A. Matrix Manipulation

We use a CFD program as our first application scenario. In the CFD codes for the parallelization of pressure correction method[5], it will need matrix components for the conjugate gradient solver. Possible implementation of the matrix components are dense matrix components, sparse matrix components, and parallel matrix components. We examine the performance of the conjugate gradient solver that incorporating these components. The hardware configuration is a 4-node PC-cluster connected with 100Mbps ethernet, and each node is running on 800MHz AMD Athlon CPU with 256MB memory. The software environ-

Fig. 4. The Parallel Matrix Components implement IMatrix and a variety of annotation interfaces.

Table I first reveals the potential benefits of using a sparse matrix component instead of a dense matrix component in the CG solver with the sparse dataset in [2, page 352, Fig. 4]. As we implement composition with Java proxy mechanisms, Table I also shows the overhead of proxy implementations of matrix components compared to the one without proxy implementations. The overhead of employing proxy mechanisms is very small. As the dataset is actually a sparse dataset in this case, significant performance gain can be expected if the same dataset were executed by a sparse matrix component. We then now incorporate the component composition for runtime compositions. Figure 5 then compares the CG solver running time on the dense matrix component, sparse matrix component, and the dense matrix component which changed in the middle of execution to the sparse one, which is denoted by the AutoSel line. In Figure 5, the x-axis is the iteration number and the y-axis represents the execution time in
TABLE I

<table>
<thead>
<tr>
<th></th>
<th>Direct Ref.</th>
<th>Dynamic Proxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>DenseMatrix</td>
<td>143.318</td>
<td>143.532</td>
</tr>
<tr>
<td>SparseMatrix</td>
<td>6.193</td>
<td>6.213</td>
</tr>
</tbody>
</table>

Fig. 5. The conjugate gradient solver running time on dense matrix, sparse matrix, and the runtime composition from dense to sparse.

Fig. 6. The conjugate gradient solver running time on dense matrix, parallel dense matrix, and the runtime composition from dense to parallel dense.

Fig. 7. One of the primitive testings of RMI in the Java Grande Forum MPJ Benchmarks using different RMI implementation.

seconds. Note that the number of iteration is large due to the requirements of numerical accuracy. The selector automatically selects the sparse matrix component during the CG solver at early stage of computations after examining sparsity of programs. The runtime composition overhead including proxy, selection, and object serializations and the overhead is amortized. We can observe that the version with the runtime composition gains around 7 times speedup than the version with dense matrix components.

We now look at the aspects of optimizations with parallel environments. We now have the diagnoser to observe if a sequential application is a long running execution. In this case, the selector then picks up the parallel dense matrix component due to the match of the names `getPreferredParallelNode()` in the annotation interface it implements and `getAvailableParallelNode()` in the environment interface PVM. The result in Figure 6 shows the composition of parallel dense matrix components (the `AutoSel` lines) does improve the original solver. In Figure 6, the upper line represents the long execution time of the one with sequential matrix component. Next, it shows both 2-node and 4-node execution. Each case is with two lines quite close to each other. One of the two lines in the low parts are the one running initially with parallel configurations. The `AutoSel` one is the with initially executed by sequential programs, and then the selector composes the program into a parallel matrix version. Object-serializations are needed here as parts of overheads to provide consistency state. We can see the performance benefits out-perform the overheads.

B. Cluster Computings with Specialized RMI

The second application scenario is the RMI communications in the cluster computings. The high-performance clusters are usually equipped with special high-speed, low-latency inter-networks. The cluster used in the previous experiments is also equipped with VIA [10] network. Since the traditional intercommunications of parallel programs like barriers, broadcastings, etc. are usually latency-sensitive, we will demonstrate our component technology by adapting RMI components over ethernets to RMI components over VIA for gaining the low-latency benefits from the specialized hardware. In our clustered environments, we support three RMI components including RMI component over TCP/IP and ethernets, KaRMI [7] over TCP/IP, and KaRMI over VIA. The KaRMI is adopted from the work in KaRMI [7] and is designed to be easily ported to different networks. It abstracts the transport layer by the interface called Technology. The KaRMI had successfully incorporated technologies to reduce the time to do object serialization. This can reduce latency. On top of that, we build a KaRMI component over the VIA cluster. To switch
Fig. 8. Running realistic applications from Java Grande Forum MPJ Benchmarks with RMI components adaptations.

between the ethernet RMI and the VIA RMI, we simply change the technology object used by KaRMI. The VIA technology object may implement the annotation interface so as the environment investigator. The additional match of the returned value of getRequiredVipl() and getCurrentVipl() called on the VIA technology object and the environment investigator will find it as a suitable one and compose it in.

We experiment with runtime compositions of Java RMI components with the Java Grande Forum MPJ Benchmarks. The cluster configuration is the same as above, except for adding VIA networks from Giganet’s implementation. The softwares include Sun Java SDK 1.4.1 and KaRMI 1.07b. Figure 7 compares the all-to-all primitive testing of MPJ using the two kinds of technology components on the 4-node cluster. When the communication messages are small, the VIA technology object performs better than the original one. The RMI version is the RMI component over TCP/IP and ethernets. Initially, all three cases running this version. After first RMI communication, the system quickly adapt the RMI component into the KaRMI and KaRMI over VIA, respectively. We see performance gains in those two cases.

Focus is now directed to real-world applications in the Java Grande Forum MPJ Benchmarks. We experiments with the benchmark suite including successive over-relaxation (SOR), sparse matrix multiplication (SparseMatmult), IDEA encryption (Crypt), and monte carlo simulation (MonteCarlo). Again all applications were initially running RMI components, but then adapts to KaRMI component and KaRMI component over VIA at early stages. The performance results showed the applications running time using TCP/IP socket technology object (KaRMI), applying the component composition to adopt VIA technology object in the very beginning (KaRMI + VIA), and using general RMI for references (RMI).

References