JCMP: Linking Architecture with Component Building

Guoqing Xu, Zongyuan Yang, Haitao Huang
Software Engineering Lab, Department of Computer Science
East China Normal University
{gqxu_02, yzyuan, hthuang}@cs.ecnu.edu.cn

Abstract

Approaches to enforcing communication integrity in the implementation, exemplified by ArchJava, consider only architectural constraints, without taking into account the late integration of pre-built components into the architecture. This may hinder the practice of architecture in the common component building. In this paper, we present an approach to supporting the integration of pre-built components in the context of the architectural constraints. This approach is described in terms of a novel design pattern, an architectural description language (ADL) JCMPL and a toolset JCMP. The language and toolset are designed based on the pattern to dynamically link the architectural constraints with the component building. To achieve this goal, an important step is to automatically translate the connector specification defined in the architecture into the connector implementation, which can serve as glue codes to connect two pre-built components together by transferring the methods invocation between two components.

1. Motivation and introduction

Software architecture [3, 9, 10] describes the structure of a system, enabling more effective design, program understanding, formal analysis, and software product line building [14]. Component development under the control of the architectural constraints can aid in the specification and analysis of high-level designs, facilitate the implementation and evolution of large software systems, and enable architectural reasoning. This makes components defined in the architecture differ from common components in the fact that they must conform to the overall architectural conformance. On the other hand, as the independent modules, they should also be developed individually without dependency on each other for the better reuse.

1.1. The problem

However, these two issues have always been handled separately without any relations. Existing approaches to solving one of these two problems can be no longer available when facing another one.

Methods for keeping architectural conformance. In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. One of the famous works is the three criteria identified by Luckham and Vera [5] for architectural conformance:

- **Decomposition:** For each component in the architecture, there should be a corresponding component in the implementation.
- **Interface Conformance:** Each component in the implementation must conform to its architectural interface.
- **Communication Integrity:** Each component in the implementation may only communicate directly with the components to which it is connected in the architecture.

According to these three criteria, researchers have developed a wide variety of Architecture Definition Languages (ADLs) to describe, model, check and implement software architectures [8]. In these languages software connectors are recognized as an important consideration to adapt one component's interface to the interface of another [12, 5, 3, 11]. One of the good examples is ArchJava [1]. ArchJava unifies architectural abstraction and detailed implementation in one language, using type to enforce the architectural conformance, especially the communication integrity in the implementation. In the system made by ArchJava, component A can directly invoke B's provided method m if their interfaces are connected in the architecture, since it is allowed by the communication integrity rule.

However, we notice there is the direct client-server relationship between A and B. This direct invocation requires that the "requires" method declared in A's interface must have the same name and signature with the "provides" method in B's interface. That is if B evolves changing the name of method from *m* to *n*, or B is replaced by a new component D with it providing a method *p* which has the same function and arguments type with *m* of B, the "requires" interface of component A must also be modified manually.

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

If both concrete components A and B have existed and been built by different software houses, can they be reused in the architecture? The answer might be no since the existing ADLs require the signature of required-provided methods pairs must be completely same. They only focus on architectural conformance problems without considering how to separate the whole architecture into independent reusable building blocks and the integration of pre-built generic components into the architecture-based application. This problem may greatly hinder the component reuse.

Methods for composite adaptation. Generally speaking, there are two major kinds of approaches to supporting system composition from individual modules. One is to use Module Interconnection Language (MIL) which describes the uses relationship between components, such as Jiazzi [6], a component infrastructure for Java and a similar system, knit for component-based programming in C. However, MILs can not be used to describe the architecture and therefore, this kind of implementation does not support architectural reasoning.

Another kind of approaches is like Pluggable Composite Adapters (PCAs) [7] and their predecessor, Adaptive Plug and Play Components (APPCs) which offers different means for on-demand remodularization. The on-demand remodularization means the abstractions and vocabulary of an existing code base are translated into the vocabulary understood by a set of components that are connected by a common collaboration interface. However, this approach has not considered the architectural constraints, and communication integrity can not be enforced.

Summary. From the above discussion, we notice these two issues of enforcing architectural conformance and supporting composite adaptation are totally handled separately. However, if the generic components composition is not supported, software architecture will be far from practical. On the other hand, if architectural conformance can not be enforced, the architectural definition will be meaningless in the components development. These two problems are far from orthogonal although their concern is different.

1.2. Our Approach

We try to band these two problems together for consideration and put forward a novel model Triple-C model to support component reuse in the context of software architecture. Triple-C model stands for

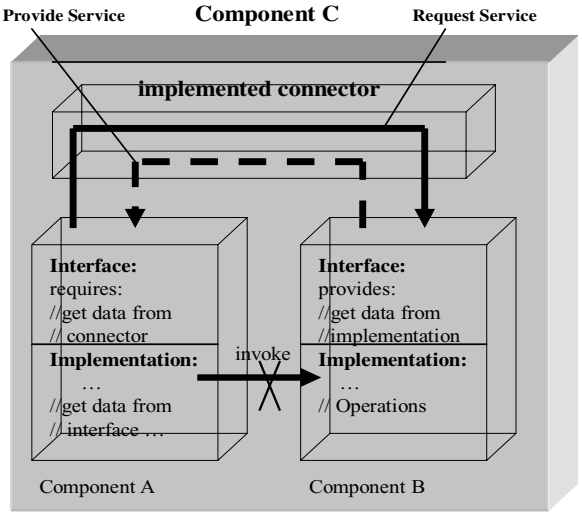


Fig.1. Connector is not only to explicitly specify the communication permission in the architecture but also implemented as a proxy to transfer service in the implementation.

Components-Communicate-through-Connector. Triple-C model is derived from the three criteria in [5], but there are two major differences between them. One is that in Triple-C model, there should be not only a concrete component in the implementation for each one defined in the architecture, but also a concrete connector for each abstract connection. Another one lies with the change of definition of communication integrity. Since we have discussed the harm resulting from the direct client-server relationship in components communication, in Triple-C model, a component can only communicate with another one through the invocation transfer performed in the connector if they are connected in the architecture. Fig.1 illustrates the overview model of component communication allowed by Triple-C model.

Since in Triple-C model, the implemented connector is the direct client and server of two connected components, components are able to be built independently without knowing the detailed methods they want, which supports the integration of generic components. On the other hand, because the connectors are also elements in the architecture, this implementation better reflects the architecture definition, therefore helps architectural reasoning and keep the architectural conformance.

The rest of this paper is organized as follows. Section 2 gives the complete definition of Triple-C model. Section 3 describes the ADL JCMPL and JCMP. Section 4 presents techniques for integrating pre-built components and keeping the architectural constraints, especially the communication integrity. After our approach is evaluated in a case study in

section 5, we conclude and describe the future work in section 6.

2. Triple-C model

We give the definition of Triple-C model as follows.

• **Infrastructure Definition:** In architecture, a system is integrated by components which are then composed by other subcomponents recursively until all their subcomponents are primitive components. We call components other than primitive ones advanced components. Primitive components are distinguished from advanced ones in the fact that connectors are only defined in latter.

• **Decomposition:** For each component in the architecture, there should be a corresponding component in the implementation. For each connection in the architecture, there also should be a concrete connector in the implementation.

• **Interface Conformance:** Each component in the implementation must conform to its architectural interface.

• **Communication Rule:** Each component in the implementation can not communicate with any other components to which it is not directly connected. A component may only communicate with the components to which it is connected through the invocation transfer performed by the connector.

• **Interface Matching:** Component Interface Matching is done in the connector according to a certain strategy when two subcomponents are integrated. The strategies used may include manually specifying in the program or in a configuration file and automated matching. Actually, the interface matching is to find "requires-provides" methods pairs in the connected two ports.

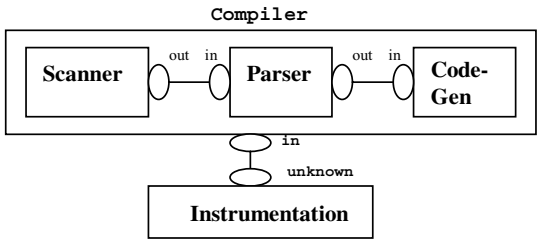
The Triple-C model focuses on two problems. The first four rules are made for keeping architectural conformance in the implementation while the last one is the rule for composite adaptation.

3. JCMPL language and JCMP toolset

A novel ADL JCMPL and a toolset JCMP are designed to support the application of Triple-C model in Java programs.

3.1 JCMPL language

JCMPL is an ADL designed based on the Triple-C model's infrastructure definition. Similar to ArchJava, JCMPL uses the Java-like grammar and explicitly describes the architecture with component, port, interface, and connect clauses, which makes JCMPL not only an Architecture Description Language, but also an Interface Description Language. However, JCMPL only describes the abstraction without any detailed assignment, which makes JCMPL a pure abstract language for the architectural abstraction design.




```
//Parser.jcmpl
public PrimComponent Parser{
 public port in{
   RequireInterface r{
     Token nextToken()throws ScanExceptions;
  }; }
 public port out{
   ProvideInterface p{
      AST generateAST();
   }; } }
// Compiler.jcmpl
public AdvancedComponent Compiler{
  public port in{
   RequireInterface get{ ...
    InputStream getInputStream()throws Exception;
  }; }
  public port out{
    ProvideInterface err{
        Integer getErrCode();...
   }; }
  Scanner scanner = new Scanner();
  Parser parser = new Parser();
  CodeGen codeg = new CodeGen();
  Instrumentation ins = new Instrumentation();
  connect sanner.out, parser.in;
  connect parser.out, codeg.in;
  connect this.in, ins.*;
}
```

Fig.2. A graphic compiler architecture and its representation in JCMPL

Fig.2 gives the JCMPL representation of a compiler architecture which follows the well-known pipeline compiler design [8]. The compiler is integrated with three primitive components: a scanner, a parser and a code generator. A port contains a required interface and a provided interface.

According to the Triple-C model, the compiler is an advanced component in which the ports of instances of scanner and parser, and of instances of parser and codeGen are explicitly connected with connect clauses, which specifies communication permissions in the implementation. Although JCMPL only defines the abstraction, it uses object architecture instead of class architecture, which makes it easy to specify dynamic program structure. Therefore, the only assignment

allowed by JCMPL is the component object creation using a *new* clause. Also, when the architecture is designed, there might be some existing components to be reused. These components may be provided by another software house or their interfaces are not very clear. JCMPL uses the wildcard * to substitute any of these unknown ports.

Architectural abstractions defined by JCMPL can be recognized and translated to the corresponding Java implementation by JCMP toolset.

3.2 JCMP toolset

We have also developed a toolset JCMP to generate codes, implement automated interface matching and enforce the architectural conformance. JCMP consists of four tools: JCMP/Compiler which parses the JCMPL language and generate the corresponding codes skeleton; JCMP/Kernel which contains the prototypes of PrimComponent, AdvancedComponent, port, interface and connector to be extended by components in the implementation; JCMP/Match which performs the automated methods matching in the two connected ports; and JCMP/Checker which checks the validity of implementation codes after programmers finish them.

Fig. 3 illustrates the translated codes skeleton of the compiler model. Two lines in it show the invocation flow. Note that in this model, the connector performs the methods matching and invocation transfer for the parser and scanner. The actual communication happens between methods *nextToken* of the parser and *next* of the scanner although their names are different.

4. Communication integrity and composite adaptation

As what we discuss in section 1, our approach bands two issues of keeping architectural conformance and supporting composite adaptation together for consideration. This section presents how to solve these problems in our implementation.

4.1 Keeping communication integrity

Fig.4. illustrates the detailed implementation of Parser component. There is an inner class RequireInterface which defines the required interface methods. This class only collects the signatures of methods exposed in the components' interfaces. The collection of signatures is important in the methods matching which will be presented later.

The decomposition and interface conformance criteria are easily to be implemented since the generated codes skeleton implements every element including port, connector, component, and interfaces defined in the architecture. But how can we keep the *nextToken* in the parser. Since Triple-C model prohibits the direct client-server relationship, we do

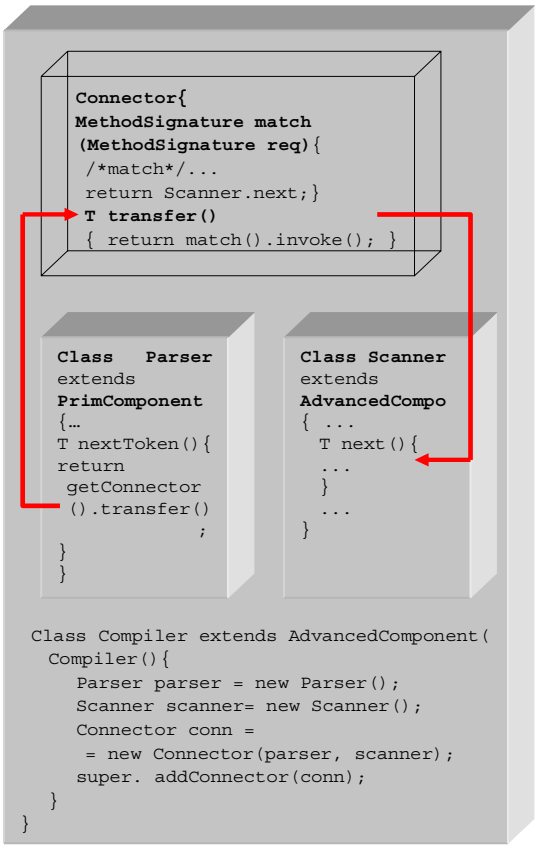


Fig. 3. The graphic description of the parser and scanner integration and the pseudo codes for generated skeleton

not generate any public methods if they are in component's interface. Therefore, no interface methods can be directly visited except through connector since the connector can invoke any methods.

by suppressing the Java access checking. Connector conforms to the connection in the architecture, therefore, only methods in ports connected directly in the architecture can communicate with each other. ArchJava uses language level type checking to enforce communication integrity while in our approach, it is much easier to be done since we use Java access checking to keep the integrity. Finally, JCMP/Checker checks the validity of implementations including whether interface methods are protected, whether corresponding ports, interfaces and connectors are established in component's constructor, etc.

4.2 Automated interface matching

In the implementation following Triple-C model, connector is considered as a composite adaptor to perform the methods matching and invocation transfer, which makes it possible to automate the matching

```
Class Parser
  extends PrimComponents{ //component Parser
  Class ParserReqInterface //interface
        extends RequiresInterface{
     // only signature here
     public String nextToken(){return null);
  }
  //constructor
  Parser(){
    ParserReqInterface req
       = new ParserReqInterface();
    Port in = new Port("in");// port in
    in.setRequiredInterface(req);
    super.addPort(in);
  }
  // the required method
  protected String nextToken(){
   RequireInterface req = //get interface
     super.getPort("in").getReqiredInter();

     MethodSignature ms //create its signature
      = new MethodSignature("nextToken",
        new Class[]{},String.class);

     //invoke the requires$$ method with
     //signature and parameters. This method
     //then passes the parameters to the
     //connector.
     Object result
       = req.requires$$(ms,new Object[]{});
     if(result instanceof String)
       return (String)result;
     else // report errors
  }
}
```

Fig 4. The detailed generated implementation of the Parser component

process. The policy we use in JCMP/Match integrates component testing and interface matching. Given a "requires" method R and a "provides" method P, we want to test if P is behaviorally equivalent to R. That is to see whether P and R have the same observable behavior. Since program invariants are easily specified manually with formal specifications or generated automatically by a certain tool like Daikon, we determine whether two methods can be matched by dynamically discovering the runtime state of program invariants. We define the match strategy formally as follows:

$$match_{jcmp/match}(P, R) = (R_{pre} \Rightarrow P_{pre}) \wedge ((P_{pre} \wedge P_{post}) \Rightarrow R_{post})$$

That is P and R match iff. P's precondition is implied by R's precondition and P's postcondition implies R's postcondition. In addition, P's postcondition must be in accordance with its precondition, otherwise, there exist errors in P's implementation.

In JCMP/Match, we use JML specifications [2] as the program invariants. Based on the above definition, a specific testing approach is designed to finish the match as shown in Fig.5. This approach can also be

```
boolean match(MethodSignature r, MethodSignature p)
{
  try{
  //test if r and p have the same arguments and return
  //type
    ...
    for(; …;… ){ …  //get test inputs from JMLAutoTest
    //use the test cases as arguments to invoke r
    r.invoke(testInputs);  …
  }catch(JMLEntryPreconditionError ex){
    try{ // r's pre is false
        // directly invoke p to check if p's pre is false
       p.invoke(testInputs);
    }catch(JMLEntryPreconditionError ex1){
      //both r'pre and p'pre are false, so continue
      continue;
    }
     //r's pre is false while p'pre is true
    return false;
 }catch(ProvidesPreconditionError ex){
   //r's pre is true while p'pre is false
    return false;
 }catch(ProvidesPostconditionError ex){
    //p's post is false which means there are errors
    //in p
   throw new Exception("Errors exist in method p");

 }catch(JMLPostconditionError ex){
  // r's post is false
    return false;
 }
 }catch(java.lang.throwable ex){
    continue;
 }
 }  return true;
}
```

```
//exception mapping for p's pre or post violation
  T r (Object[] args){
    try{
   //indirectly invoke the method p through the
   //connector.
   this.getConnector().transfer();
    }catch(JMLEntryPreconditionError ex){
     //p'pre is false;
    throw new ProvidesPreconditionException();
    }catch(JMLPostconditionError ex){
      //p's post is false.
      throw new ProvidesPostconditionError();
    }catch(throwable ex){
       throw ex;
    }
 }
```

Fig 5. Psedo codes for automated matching Strategy in JCMP/Match

implemented in components made with other languages and formal specifications. JMLAutoTest [13]is an automated testing framework for Java programs annotated with JML specifications. This tool is used here to automatically generate a large number of test cases. We invoke the "requires" method r in which the "provides" method q is invoked indirectly through the connector and use the exceptions thrown by JML runtime assertion checker [6] to check whether they match.

Note that we must distinguish the pre and post condition violation in method r and q. We make an exception mapping in r's body. That is a *ProvidesPreconditionError* is thrown if p's precondition is violated and a *ProvidesPostconditionError* is thrown if p's postcondition is violated. If p's postcondition is false, there exist errors in p's implementation. If r's precondition exception is caught, we must invoke p explicitly outside since p will not be invoked by r.

5. Evaluation

In order to determine whether Triple-C model meets its design goals, we undertook a case study to answer the following experimental questions:

- Does Triple-C model help to keep the architectural conformance in the implementation?
- Does the application of Triple-C model enhance communication and help components integration?
- Can our approach of interface matching discover the potential "requires-provides" methods pairs effectively?
- What is the cost of using invocation transfer performed by the connector?

5.1 Methodology

We used JCMPL and JCMP toolset to design a Java system. What we looked for was a system small enough that we could describe the whole design and implementation process in this section. Another reason that we chose a small system is that what we want to get and prove is the soundness including the correctness and performance of our principle, rather than the experience of using our tool in large systems as shown in most case studies. Finally we chose the quick sort program described in a data structure book[4]. The function of this program is to give a quick sort on the input linked list which is made up of a couple of nodes. To demonstrate the flexibility of interface matching resulting from Triple-C model, two developers made components respectively with the help of JCMP toolset. Then the third one was in charge of the late integration.

5.2 Experience

Hypothesis 1: JCMPL can support the separation of architecture design.

Two sub-components. Our target system includes only two components. One is the linked list andanother is a list sorter which receives the list as input and uses the quick sort algorithm to sort it. These two developers use JCMPL respectively to describe the two components' interfaces. Fig.6 illustrates the ports and interfaces of List and Sorter. We can find that these two developers made the interfaces based on their own understanding of the

```
public PrimComponent LinkedList{
 public port operation_on_node{
  ProvideInterface pro1{
    void append(Node node);//append a node
    void remove(Integer num);//remove a node
    Integer retrieve(Integer num);//get a node
    void clear(); //clear the list
  };
 }
 public port operation_on_whole_list{
  ProvideInterface pro2{
   constructList(Nodes[] nodes);
  };
 }
}
public PrimComponent Sorter{
  public port Sort{
    ProvideInterface pvdInterface{
       LinkedList sort();//quick sort
       void setList(LinkedList list);//set list
   };
 }
  public port List{
    RequireInterface reqInterface{
       //get data of node indexed by ID
       Integer getNodeData(Integer ID);
       void append(Node node);//append a node
      //get the node indexed by Id and delete it
       Integer getandDelete(Integer ID);
  };
 }
}
```

Fig. 6. The two components' JCMPL representation made by two developers.

system which resulted in the fact that most of methods declared in the interfaces were totally different in their names and signatures and could not match directly. Especially, the *getandDelete* method required in the Sorter could not even be satisfied by any individual method provided by the List.

In addition, the use of JCMPL to design a part of architecture individually reminds us that using JCMPL, the design of architecture itself can also be separated into parts and each part can be designed without dependency on each other, since JCMPL does not require there must be a corresponding provided method for each required method. The architectural abstraction itself is also available for reuse.

Late Integration. The third developer integrated two sub components into one advanced component. The architectural abstraction of the system is described in JCMPL as Fig.7. The port *list* in the component Sorter and *operation_on_node* in Linkedlist are connected since their operations have the similar functions. A new port *in* is added to the advanced component QuickSort to contain its own published *requires* and *provides* methods. Then the port *in* is connected with Sorter.Sort and LinkedList.operation_on_whole_list.

```
public AdvancedComponent QuickSort{
  public port in{
     RequireInterface req{
        void constructList(Vector nodes);
        LinkedList sort();
     };
     ProvideInterface pvd{
        void startSort();
        void setListtoSorter(List list);
     };
   }
  Sorter s = new Sorter();
  LinkedList l = new LinkedList();
  connect
     s.List, l.operation_on_node;
  connect this.in, s.Sort;
  connect
    this.in, l.operation_on_whole_list;
}
```

Fig. 7. The architectural abstraction of QuickSort

```
Starting...
Trying to match methods getandDelete and retrieve…
Fails!...
Pre1 is true: 100     Pre2 is true:100
Pre1 is false:1       Pre2 is false:1
Post1 is true:0       Post2 is true:1
Post1 is false:1      Post2 is false:0
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLNormalPostcondi
tionErr
or: by method Sorter.getandDelete regarding
specifications at
File "sample\Sorter.java", line 31, character 35
when
       '\old(size())' is 6
       'ID' is 4
       '\result' is 3
       'this' is sample.Sorter@1a8c4e7
```

Fig. 8. Attempt to match getandDelete and retrieve fails

Hypothesis 2: Each required method in one port will be satisfied as long as the operations it requires can be performed by a method or the combination of a set of methods provided in another port, no matter whether their signatures are the same.

Interfaces Matching. The integrator uses the JCMP/Match to automatically search for the matched methods pairs in three ports connected in the architecture. JCMP/Match tries on every possible *requires* and *provides* methods in the connected ports. It starts with scanning the arguments types and return type of each possible pair to prune large portions of search space, so most of pairs with different arguments types and return types are eliminated very quickly. JCMP/Match then uses the matching algorithm discussed above to find the most accurate methods pairs. Fig.8 illustrates the output when we generated integers from -100 to 100 to test the method *getandDelete* and *retrieve*. The automated matching process stopped as soon as the post condition of

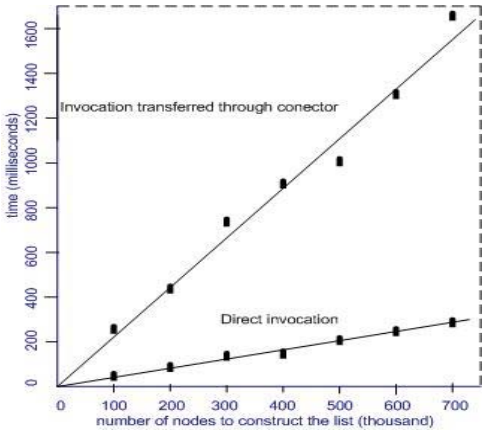


Fig.9. The performance comparison of invocation transfer based on triple-C pattern and common direct invocation when the list is constructed with number of nodes changing.

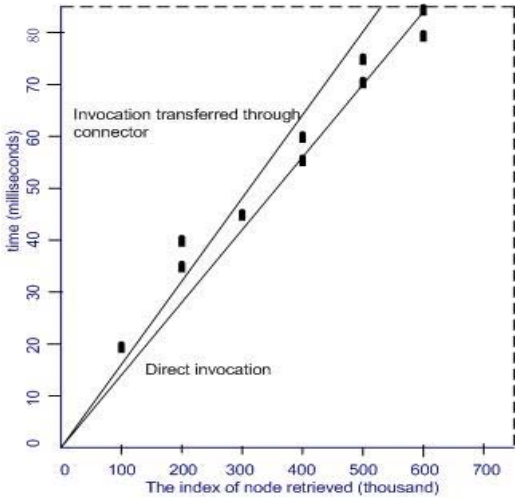


Fig.10. The performance comparison when the different nodes are retrieved.

getandDelete was violated. Finally, JCMP/Match found four pairs, {operation_on_node.retrieve, list.getData}, {operation_on_node.append, list.append}, {Sort.sort, in.sort} and {in.setListtoSorter, Sort.setList}. The matched results were saved in the XML file.

5.3 Performance

The major cost of running the Triple-C model based implementation has two components: the runtime cost of type reflection and the extra cost of invocation transfer through the connector. The type reflection includes three parts: the primitive or array arguments encapsulated into objects at the entrance of the connector[1], releasing at the exit, and the reflection

[1] JCMP instruments the methods with arguments of primitive and array types and encapsulate them into objects since the primitive and array types do not support Java Reflection.

from methods name to method class and from class name to instance performed by JVM. The cost of invocation transfer mainly consists of three extra function call time including the call of the RequiredInterface method *requires$$* in the required methods, the call of the Connector method *transfer$$* in *requires$$* and the call of the ProvideInterface method *provides$$* in *transfer$$*.

Fig.9 illustrates the performance comparison of invocation transfer based on Triple-C model and common direct invocation when the list is constructed with increasing numbers of nodes. It is very clear that the time cost increases much more quickly in the invocation transfer than the common direct invocation. Fig.10 illustrates another comparison when different nodes are retrieved in the list. This test is done in a 800,000-nodes list. Note that in this test, the time increase of invocation transfer is very close to that of common invocation.

We were so curious on such large differences of performance behaviors of our system between two tests. We divided the whole process of the first test into three phases. The first phase was the period before the connector getting the arguments, the second phase was from the connector getting the data to passing them to the provided method and then the third phase included the rest part. We made statistics on the average time cost of each phase and found that the average time costs of three phases are 38.9%, 17.8% and 43.3% respectively after testing it for eight times. It is not strange that the time of third phase is largest since the specific operations are performed in it. However, the abnormally large time cost of the first phase reminds us the major cost of the process lies with encapsulating the nodes array into a vector since the original argument of method *construct* has the type *Nodes[]* and it has been instrumented to encapsulate the array into a vector. In the test shown in Fig.10, the type of argument of the *retrieve* method is integer, and therefore it does not need the instrumentation and encapsulation of arguments.

Then it has been clear that the cost of invocation transfer is not obvious while the type reflection takes most of time. The experimental results demonstrate that our approach of implementing connector as proxy to transfer service works and is practical. We will make every effort to optimize the encapsulation of primitive and array type arguments, but for right now, we suggest Objects, rather than primitive values should be used in running our system.

6. Conclusions

In this paper, we define a novel model, the Components-Communicating-through-Connector (Triple-C) model, which provides guidelines for the methodology of how to implement the architectural abstraction into the detailed code and solves component communication problems for composite adaptation.

We introduce the JCMPL language and toolset JCMP which are designed to help apply Triple-C model in Java implementations. JCMPL only defines the abstraction without any detailed implementations and JCMP parses the architectural definition of JCMPL and translate it into Triple-C model based implementation. Finally, we evaluate the Triple-C model in a small case study.

Reference

[1] Jonathan Aldrich, Craig Chambers and David Notkin. ArchJava: Connecting Software Architecture with Implementation. *Proc. International Conference on Software Engineering*, FL, 2002

[2] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). Technical Report 02-05, Department of Computer Science, Iowa State University, March 2002.

[3] David Garlan and Mary Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering, I (Ambriola V, Tortora G, Eds.)* World Scientific Publishing Company, 1993.

[4] Robert L. Kruse and Alexander J. Ryba. Data Strutures and Program Design in C++, Pearson Education, 2001

[5] David C. Luckham and James Vera. An Event Based Architecture Definition Language. *IEEE Trans. Software Engineering 21(9)*, September 1995.

[6] Sean McDirmid, Matthew Flatt and Wilson C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. *Proc. Object Oriented Programming Systems, Languages, and Applications*, Tampa, FL, October 2001.

[7] M. Mezini, L. Seiter, and K. Lieberherr. Component I ntegration with pluggable composite adapters. In M. Aksit, editor, Software Architectures and Component Technology: The State of the Art in Research and Practice. Kluwer, 2001. University of Twente, The Netherlands

[8] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Software Engineering*, 26(1):70-93, January 2000.

[9] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17:40-52, October 1992.

[10] David S. Rosenblum and Rema Natarajan. Supporting Architectural Concerns in Component-Interoperability Standards. *IEE Proceedings-Software 147(6)*, 2000.

[11] João C. Seco and Luís Caires. A Basic Model of Typed Components. *Proc. European Conference on Object-Oriented Programming*, Cannes, France, June 2000.

[12] Mary Shaw, Rob DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. Software Engineering,* 21(4), April 1995.

[13] Guoqing Xu, and Zongyuan Yang, JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit. *Proc. The 3rd International Workshop on Formal Approaches to Testing of Software (FATES03)*, Montreal, Canada, Oct 2003, also in LNCS vol. 2931, Springer-Verlag, Jan 2004

[14] Paul Clements and Linda Northrop, Software Product Lines: Practice and Patterns, Addison-Wesley, 2002