Specifying component-based software architectures

P. Ciancarini, S. Cimato

Dept. of Computer Science, University of Bologna,
Via Mura Anteo Zamboni, 7 - 40127 Bologna, Italy
Tel: +39 51 354506
Fax: +39 51 354510
Email: {cianca, cimato}@cs.unibo.it

Abstract

Large software systems offer to software designers complex problem to solve in an efficient and quick way. To manage such complexity several techniques have been developed to make this task easier and to allow the designer to reuse prior experience. However such techniques and frameworks often lack formal notations to support formal reasoning about the resulting products. We extend Larch family languages, by defining Larch interface language for Java modules and argue that such notation should help to design and implement software systems built of Java components.

Keywords: Components, Java, Larch, formal design.

1 Background

The design and the implementation of large software systems is an ever lasting challenge for software developers. Currently, the development of the World Wide Web and internet, and the diffusion of personal computers and local area networks complicates more and more the scenario in which software engineers have to develop new applications. From monolithic applications running on a isolated computer, the dominant computing paradigm is fastly shifting towards distributed systems running on platforms heterogeneous both for hardware and software used [BW96, MM97].

We are engaged in a research project concerning distributed software architectures including components written in Java [CKTV96]. One of the research themes we are mostly interested in consists of studying formal methods supporting the design process of these architectures. In the last three years we have used Larch as formal notation for design, in the context of a development method called Clepsydra [CCP97]. We are exploring the suitability of Larch for supporting Java-based applications.

2 Position

The increasing complexity for present software system compels software builders to research efficient abstractions to simplify and make faster the software development process, by reusing successful solutions to formerly solved design problems. *Design patterns*, *software architecture*, and *component based software development* are techniques that enhance the reusability of software by applying the practical approach of "divide and conquer" to software design and implementation [MKMG97, BW96, Cle96]. The basic idea is to recognize and describe the gross structure of a software system, in order to identify the fundamental components and to reuse for them design solutions already experienced. The complexity of design patterns ranges from solutions to concrete programming problems to solutions involving the overall organization of the system, till the development of "off-the-shelf" components ready to be connected to compose larger applications [MM97].

Formal notations should guide software builders in the development of applications, in order to be more confident in the result of their work. In this work we are interested in the development of modules and components written in the Java programming language [AG96]. In spite of a multifaced growing community of computer users, Java is making software development scenario more uniform, due to its enormous diffusion and to its features of portability, flexibility, and security. Java comes with a set of built-in features that meet the needs of designers involved in the construction of distributed systems. Java extensions like RMI (Remote Method Invocation) [Sun96b], Java Beans [Sun96a], Java-CORBA, and JavaSpaces [W$^+$97], enrich the basic general framework with advanced facilities which have to be mastered in order to build effective distributed applications. However, in the absence of a formal description for these extensions, all their fundamental features are exposed informally, with a number of examples and tutorials.

In this paper we extend the Larch style of specification to modules written in Java, and explore the use of such a notation for the definition of design patterns and components to be employed in the building of concrete applications. The next section introduces Larch/Java notation, while sections 4 and 5 show the application of such a notation for some basic design patterns and for the definition of a simple client-server architecture.

3 Preliminary design of Larch/Java

The Larch family of specification languages consists of the Larch Shared Language (LSL) and a collection of Larch Interface Languages (LIL), each tailored for a particular programming language [GHW85]. The peculiarity of Larch with respect to design specifications is the "two tiers" (or layers) approach it uses. The lower tier is the Larch Shared Language, an algebraic specification language which defines the mathematical abstractions used in the other tier, which is the behavior interface language in which predicate on pre and post condition over the execution of the operations are defined [GH93, Win87].

LSL *traits* are the units of LSL specifications introducing *sorts* and *operators* to represent terms and values and a set of equations to define some relevant properties. For common data structures like integers, sets, etc, Larch provides a collection of library traits, which can be reused to specify new objects or abstract values of some data type. The interface specification module describes the behavior of the operations by defining the relation between the state in which an operation is invoked and the state after the execution of the operation, through the definition of *requires-modifies-ensures* clauses.

Several LIL have been developed for the most common programming languages such as C [GH91], ADA [GHW85], and Modula-3 [BGHL91]. The most recent LIL are all object-oriented: Smalltalk [CL94b] and C++ [CL94a]. Interestingly, a Generic Interface Language has been proposed as a generic basis for the definition of new languages, which has been extended for the specification of concurrent objects with the definition of a generic concurrent interface language (GCIL) in [Ler91]. Finally, a preliminary proposal for Larch/CORBA-IDL has been recently published in [Siv95].

Larch/Java has been developed starting from the design of Larch/C++ with the modifications necessary to support concurrency features of Java. Leaving out most design details, in Larch/Java an additional *when* clause provides a mechanism for synchronization. A *when* clause states the condition for scheduling the corresponding operation once that the preconditions have been met. The identifier *myId* identifies the client which has invoked the operation; each object has a *lock* and a *waiting set* to represent Java mechanisms for thread support and mutual exclusion. In particular, the scheduler is responsible for the assignment of the lock to a waiting thread, while the client may release the lock and modify the set of threads waiting for a condition to be met. The following example shows a Larch/Java specification for a simple concurrent counter which uses the *BCounterTrait* LSL trait:

```
BCounterTrait: trait
  includes Natural(Nat)
  introduces
    newCounter: -> Counter
    inc: Counter -> Counter
    dec: Counter -> Counter
    value: Counter -> Nat
    isZero: Counter -> bool
    MAX: -> Nat
  asserts
    Counter generated by newCounter, inc
    Counter partitioned by value
    \forall c: Counter
```

```
value(newCounter) ==0;
value(inc(c)) == value(c) + 1;
isZero(newCounter);
0 < MAX;
```

This trait defines a bounded counter abstract data structure. Such an abstract definition is used to specify a *CCounter* class as follows:

```
public class CCounter {
uses BCounterTrait;
invariant value(self) <= MAX ∧ value(self) > 0;
public int CCounter() {
    modifies self;
    ensures self' = newCounter; }
public synchronized int value() {
    requires lock=myId;
    modifies lock;
    ensures result = value(self^) ∧ lock=nil; }
public synchronized void inc() {
    when lock=myId ∧ value(self) < MAX;
    modifies self,lock;
    ensures self' = inc(self^) ∧ lock=nil; }
public void synchronized dec() {
    when lock=myId ∧ ¬ isZero(self);
    modifies self,lock;
    ensures self' = dec(self^) ∧ lock=nil; }
}
```

The CCounter class defines a bounded counter which can be shared among several threads. The class specification includes the specification of three synchronized methods; namely these methods operate in mutual exclusion.

4 Larch/Java for design pattern

The technique of design patterns aims to classify "best practice" solutions to problems recurring in the development of applications, in order to help and guide software builders. They can successfully exploit prior experience in the solution of a given design problem by applying the right pattern of design [MKMG97, GHJV95]. The usual notation used to describe design patterns is a mixture of informal and formal notations with annotated examples written in a particular programming language or in pseudocode. In [GHJV95] a catalog of most important design patterns is provided, each classified with a text template, which illustrates the motivation, the condition for the applicability, the structure and other information, and with class and object diagrams based on the OMT technique.

The use a formal notation as Larch/Java to describe design patterns has the usual advantages coming from the application effective formal techniques. Unlike other too abstract formal notations, the two tiered approach of Larch provides the designer with an effective guide for the concrete implementation of program modules.

Design patterns such as those described in [Lea97a] which deal with constructs and techniques to solve common concurrent design problem, can be formally studied in our framework. For example, let us consider the design pattern in which pairs of actions cannot co-occur, referred to as *conflict sets* in [Lea97a]. Suppose that $\{(op1, op2), (op2, op2)\}$ is the conflict set for the two operations $\{op1, op2\}$, i.e two $op1$ operations can execute concurrently, while an $op2$ should not run concurrently with an $op1$ or another $op2$ operation. A Larch/Java specification for such pattern is the following:

```
public class ConflictOperations {
uses oper1, oper2;
protected int op1=0;
protected int op2=0;
public void op1(...){
    composed of s1,do,s2;
    action s1{
        when op2=0 ∧ lock=myId;
        modifies op1,lock;
        ensures op1=op1+1 ∧ lock=nil;}
    action do {
        modifies self;
        ensures result=doaction(self);}
    actions s2 {
        when lock=myId;
        modifies op1,lock,ws;
        ensures op1'=op1ˆ1 ∧ (if op1'=0 then ws'={}) ∧ lock=nil}
    }

public void op2(...){
    composed of r1,make,r2;
    action r1 {
        when op1=0 ∧ op2=0 ∧ lock=myId;
        modifies op2;
        ensures op2=op2+1;}
    action make {
        modifies self;
        ensures result=makeaction(self);}
    action r2{
        when lock=myId;
        modifies op1,ws;
        ensures op2'=op2ˆ1 ∧ (if op2'=0 then ws'={}) ∧ lock=nil}
    }
}
```

Each operation is composed of several atomic actions. The scheduler may arbitrarily interleave actions of one operation with the actions of another one, provided that the execution order is maintained and the conditions of the when-clause are satisfied. The first operation, when invoked, waits for being executed, until it gets the lock and the variable $op2$ is zero, meaning that no other $op2$ operation is running. At this point the counter on $op1$ operations is incremented, the lock can be released and the second action *do* can be executed as soon as the scheduler gives it the control. The next action must instead wait to get the lock in order to properly modify the variable $op1$ after the execution of the operation and to awake the other threads in the waiting set if no other *op*1 operation is running. The specification for the other operation *op*2 is similar apart from the condition in the *when* clause, stating that no other operation can be executing concurrently as desired.

Note that in Larch/Java the synchronization constraints among the different methods are clearly specified and the pairs of pre/post conditions for the execution of each method define the modification of the state of the object as computation goes on.

5 Larch/Java for Software Architecture

The *software architecture* concerns the design of the gross structure of a software system, including its overall behavior and its decomposition in simpler computational elements. [GS96]. An architectural description singles out the components from which a system is built, describing their functional behavior and providing a complete description of their interactions [S^+95, AAG95]. Several type of components and connectors have been analyzed and classified by defining architectural styles, which characterize family of systems sharing the same structural properties.

As in [AAG95, GMW96] we recognize *components* and *connectors* as basic elements of an architectural description. In the following we show how Larch can be used for the design of a software architecture, with Java as target language for the design of the interface of the modules. The LSL can be used for the formal description of the basic properties of components and connectors. Traits for a general component and a general connector respectively are given below:

```
component:trait
  includes Set(iport, Set[iport]),  Set(oport, Set[oport])
  component tuple of inports: Set[iport], outports: Set[oport]

connector:trait
  includes Set(irole, Set[irole]),Set(orole, Set[orole])
  component tuple of inroles: Set[irole], outroles: Set[orole]
```

Basically, components and connectors are active elements, each owning a set of *ports* and *roles*, respectively. A *port* is an interface between each component and its environment; a *role* is an interaction point among participating components. Let us concentrate on a particular type of architecture, namely a simple client server system. We start defining the trait for the basic components of the system, client and server, and the trait for the connector describing the interactions among components, in our case a remote procedure call. These traits are obtained reusing the general traits given above and the traits defined in the LSL library [GH93]. Client and server traits are similar but have distinct behaviors.

```
Client:trait
  includes component,
     Queue(data, inputstream)
     Queue(data, outputstream);
  Client tuple of comp:component, inputs: inputstream,
```

```
                 outputs:outputstream
  introduces
     Initclient: -> Client
     Isconnected: Client -> Bool
     IsValid: Client -> Bool
  asserts
     \forall c: Client
       IsValid(c) == size(c.comp.inports) =1 /\
                     size(c.comp.outports) = 1;

Server:trait
  includes component,
     Queue(data, inputstream)
     Queue(data, outputstream);
  Server tuple of comp:component, inputs: inputstream,
                 outputs:outputstream
  introduces
     Initserver: -> Server
     Isconnected: Server -> Bool

Rpc: trait
  includes connector,
     Queue(data, inputstream)
     Queue(data, outputstream);
  Rpc  tuple of conn:connector, inputs: inputstream,
                 outputs:outputstream
  introduces
     Initrpc: -> Rpc
     Initinputstream: -> inputstream
     Initoutputstream: -> outputstream
     IsValid: Rpc -> Bool
  asserts
     \forall r: Rpc
       IsValid(r) == size(r.conn.inroles) =1 /\
                     size(r.conn.outroles) = 1;
```

The LIL specification modules describe the functional behavior of each component. A client is responsible for starting the communication and the retrieval of result data. The server, instead, is listening for incoming requests; it services a client as soon as possible. The *rpc* connector sets up the connection for the exchange of data between the client and the server.

We use the Java socket interface as protocol of communication between hosts on the network [OH97]. The LIL modules for the *Server* component and the *Rpc* connector are given in the following:

```
public class Server{
uses server;
public void setup_server(port){
   requires ¬ Isconnected(self) ∧ port ∈ self.comp.inports;
   ensures self' =Initserver;}
```

```
public data get(port){
    requires Isconnected(self);
    when ¬ self^.inputs = empty;
    modifies self;
    ensures result=head(self^.inputs) ∧
            self'=set_inputs(self^, tail(self^.inputs));}
public data write(data,port){
    requires Isconnected(self);
    modifies self;
    ensures self'=set_outputs(self^, append(data, self^.outputs));}


public class rpc_sock(host,port){
uses rpc;
public void setup_socket(host,port){
    ensures self'=initrpc ∧ self'.inputs = initinputstream
            ∧ self'.outputs = initoutputstream;}
public void send_data(data){
    modifies self;
    ensures self'=set_outputs(self^, append(data, self^.outputs));}
public Byte get_data(data){
    modifies self;
    ensures result=head(self^.inputs) ∧
            self'=set_inputs(self^, tail(self^.inputs));}
public void disconnect(){
    modifies self;
    ensures self'.inputs = empty ∧ self'.outputs = empty;}
}
```

The meaning of the interface specification modules is a collection of Java code files which satisfy their specification [Lea97b]. The following is one Java code module implementing an rpc-socket:

```
public class rpc_sock(host,port){

public void setup_socket(host,port){
    Socket rpc = new Socket(host,port);
    Inputstream in = rpc.getInputStream();
    OutputStream out = rpc.getOutputStream();}
public void send_data(data){
    out.write(data);}
public Byte get_data(data){
    in.read();}
public void disconnect(){
    rpc.close;}
```

At this point another level of specification is needed to describe the effective interactions among the computational elements.

A *configuration* module lists the instance of design elements which form our system, the LIL modules which are used for the specification and the attachments between ports of the components and roles of the connectors. Additionally, functional properties may be added to describe and constrain the system behavior. In our case we have an instance for each component and connector we previously have defined, and state input-output connections between rpc connector and the two components.

```
c/s-system:configuration
component:
   c1:client,s1:server;
connector:
   r1:rpc
attached:
   r1.conn.irole = c1.comp.oport;
   r1.conn.orole = s1.comp.iport;
```

The implementation of a configuration module may be thought of as a main application module which use the other components of the system and define their behavior when computation progresses

6 Related work

Several architectural description languages have been developed to provide software designers with notations and tools to specify and reason about architectural designs. Some frameworks give the designer an environment to build system according a specific architectural style [GAO94], where a style prescribes the type of components and connectors that can be used and the rules for their interconnections. Other general-purpose architectural languages support the description of the architecture of a system through behavioral specification. In [S$^+$95], architectural descriptions can be compiled to obtain executable code or a rapid prototype for the simulation of system behavior can be constructed as in [L$^+$95]. General specification notations have also been exploited in the study of software architectures such CSP [GAO94] and CHAM [IW95] to give behavioral specification for the system being developed.

Our approach aims to formalize the design of modules written in Java to ease the development and the reuse of previous experience for the concrete realization of specific instance of a system. The Larch notation provides a formal framework which guides the development of applications through the composition of preexisting components and in which it is possible to define and reason about properties of system being developed. We start from the definition of generic components which can be adapted and refined to satisfy the particular requirements encountered, exploiting the modularity supported by Larch notation. The behavior of such components can be described and analyzed individually in each LIL module while their interrelationships can be defined in the configuration module. In this way we have another level of formal design which allows an overall vision of the system as formed by interacting components and puts in evidence global constraints and behavioral properties.

We are evaluating the use of Larch/Java notation for the description of the architecture of concrete software systems through the completion of non toy-examples. The specification of the architecture of a Web based application to support financial transactions has been object of study in an undergraduate course on software engineering. Much work remains to be done to extend and integrate

Larch/Java notations in order to obtain more complete specifications. First of all, we are studying a semantic model for Larch/Java specifications through the mapping of objects and states on solutions, molecules and chemical rules as provided by the Chemical Abstract Machine framework [BB92]. A simple parser of Larch/Java specification has been developed; we plan to extend this tool to make it possible to perform static semantic checks. Finally we aim to define an ontology for recurrent components and connectors in order to provide the designer with well specified and functioning modules to be easily combined together. Reference architectures as Java/Corba or Javabeans should in this way find out a formal framework guiding where the design of the architecture of the system being developed.

References

[AAG95] G. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[BB92] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BGHL91] A. Birrell, J. Guttag, J. Horning, and R. Levin. Thread synchronization: a Formal Specification. In G. Nelson, editor, *System Programming with Modula-3*, chapter 5, pages 119–129. Prentice-Hall, 1991.

[BW96] A. Brown and K. Wallnau. Engineering of Component-Based Systems. In Alan W. Brown, editor, *Component Based Software Engineering*, pages 7–15. IEEE Computer Society Press, 1996.

[CCP97] P. Ciaccia, P. Ciancarini, and W. Penzo. Formal Requirements and Design Specifications: The Clepsydra Methodology. *Int. Journal on Software Engineering and Knowledge Engineering*, 7(1):1–42, March 1997.

[CKTV96] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace : An Architecture to Coordinate Distributed Applications on the Web. *Computer Networks and ISDN Systems*, 28(7-11):941–952, 1996.

[CL94a] Y. Cheon and G. T. Leavens. A Quick Overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.

[CL94b] Y. Cheon and G. T. Leavens. The Larch/Smalltalk Interface Specification Language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.

[Cle96] P. Clements. From Subroutine to Subsystems: Component-Based Software Development. In Alan W. Brown, editor, *Component Based Software Engineering*, pages 3–6. IEEE Computer Society Press, 1996.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In D. Wile, editor, *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, volume 19:5 of *ACM SIGSOFT Software Engineering Notes*, pages 175–188, New Orleans, USA, December 1994.

[GH91] J. Guttag and J. Horning. A Tutorial on Larch and LCL, A Larch/C Interface Language. In S. Prehn and J. Toetenel, editors, *VDM91: Formal Software Development Methods (Tutorials)*, volume 552 of *Lecture Notes in Computer Science*, pages 1–78. Springer-Verlag, Berlin, October 1991.

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, Berlin, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[GHW85] J. Guttag, J. Horning, and J. Wing. The Larch Family of Specification Languages. *IEEE Software*, 2(5):24–36, 1985.

[GMW96] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description and Interchange Language. Technical report, Carnegie Mellon University, Pittsburgh, 1996.

[GS96] D. Garlan and M. Shaw. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[IW95] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[L+95] D. Luckham et al. Specification and Analysis of System Architecture using RAPIDE. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[Lea97a] D. Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[Lea97b] G. Leavens. Larch/C++ Reference Manual. Version 5.1, January 1997.

[Ler91] R. Lerner. Specifying Objects of Concurrent Systems. Technical Report CMU-CS-91-131, Carnegie Mellon University, Pittsburgh, 1991.

[MKMG97] R. Monroe, D. Kompanek, R. Melton, and D. Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1):43–52, Jan 1997.

[MM97] T. Mowbray and R. Malveau. *CORBA Design Patterns.* John Wiley & Sons, 1997.

[OH97] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA.* Wiley, 1997.

[S+95] M. Shaw et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[Siv95] G. Sivaprasad. Larch-CORBA: Specifying the Behavior of CORBA-IDL Interfaces. Technical Report TR95-27, CS Dept., Iowa State Univ., 1995.

[Sun96a] Sun Microsystems Inc. Java beans 1.0, October 1996.

[Sun96b] Sun Microsystems Inc. Java remote method invocation specification, December 1996.

[W+97] J. Waldo et al. Javaspace specification 1.1. Technical report, Sun Microsystems, JavaSoft Lab., March 1997.

[Win87] J. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.