SPECIFYING COMPONENT-BASED JAVA APPLICATIONS

S. Cimato

Department of Computer Science, University of Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy.

cimato@cs.unibo.it

Abstract: Large software systems offer to software designers complex problems to solve. 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 propose a interface specification language for Java modules and a methodology to support the design of Java components. We show how such a notation helps in designing software systems built of Java components, like JavaBeans.

INTRODUCTION

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 their applications. The dominant computing paradigm is fastly shifting from monolithic applications running on an isolated computer towards distributed systems running on platforms heterogeneous both for hardware and software used [6].

An important idea to manage this increasing complexity is to recognize and describe the overall structure of a software system, in order to identify its fundamental components and to decompose the original problem in smaller subproblems more easily resolvable. Following this approach already experienced design solutions can be successfully reused, simplifying and making more effective the software development process. *Design patterns*, *software architecture*, and *component based software development* are design techniques that enhance the re-usability of software by applying the practical approach of "divide and conquer" to software design and implementation [7]. The complexity of design patterns and software architectures ranges from solutions

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*
© Springer Science+Business Media New York 1999

to concrete programming problems to reusable structures including "off-the-shelf" components ready to be connected to compose larger applications.

Formal notations should guide software builders in the development of complex applications, in order to make them able to reason on the properties on the design and to ensure a successful result of their work. In particular we are interested in the formal development of modules and components written in the Java programming language [3]. In spite of a multi-faced 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, which have determined its success. Java comes with a set of built-in features that meet the needs of designers involved in the construction of distributed systems. Namely facilities to handle concurrency and distribution have been included in the design of the language. Java uses high level constructs such as monitors to synchronize the concurrent activity of several threads which can be active at the same time. Class libraries providing standard access to network resources and protocols (such as socket and url) are included in the base package and give Java a great potential for the development of dynamic and networked applications. Furthermore, Java extensions like RMI (Remote Method Invocation), Java Beans and JavaSpaces [17], enrich the basic language framework with advanced facilities which have to be mastered in order to design effective distributed applications. However, all their fundamental features are exposed informally, with a number of examples and tutorials.

In this paper we address the formal specification of Java classes and provide a methodology to support the specification of Java components. The framework is based on the Larch approach [8] to the specification of modules. We provide an interface specification language for the specification of modules written in Java, and explore the use of such a notation for the definition of components to be employed to design concrete applications.

SPECIFICATION OF JAVA MODULES

Each module specification is composed of two related parts: an *algebraic part* in which it is possible to define the mathematical abstractions which describe the abstract state of the object and a *behavioral module* which specifies the interface of the object by defining conditions on its state for the application of each operation.

The algebraic layer is based on the Larch Shared Language (LSL) notation. *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. Any trait is structured as follows. An *introduce* section lists the operators and define their signatures. An operator is a total function that maps tuple of values of its domain sorts to a value of its range sort. Then in the *assert* portion, equations define properties of operators stating equalities among terms while *implies* clauses show consequences of the assertions. Each trait defines a theory (i.e. a set of formulas) in a multi-sorted first-order logic with equality which contains the trait assertions, the conventional axioms of first order logic and every logic consequence. Algebraic specifications may be reused and composed by exploiting the constructs for the inclusion and the parameterization of the specifications provided by the LSL notation. For common data structures like integers,

sets, etc, a collection of library traits is provided, 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. Ljala (Larch-interface Java language) has been developed following the approach used for Larch/C++ [10] with the necessary modifications in order to support concurrency. We model Java concurrent features providing each object with a *scheduler*, which controls the execution of each operation, operating on a *waiting set*, containing the threads waiting for synchronization conditions to hold, and with a *lock* to support mutual exclusion among threads.

A single operation may be composed of a sequence of atomic actions; in this case the pre-condition describe the state in which the operation may be invoked, while a sequence of post conditions describes the modifications of the state caused by any single action. 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. To distinguish clients invoking operations a unique identifier *myId* is introduced. As in [5] and [11] concurrency is managed using an additional *when* clause. It provides the mechanism for checking synchronization conditions and refers to the state when the client has invoked the operation which is waiting to be scheduled.

The following example shows a Ljala 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
∀ c:  Counter
   value(newCounter) ==0;
   value(inc(c)) == value(c) + 1;
   dec(inc(c)) == c;
   isZero(c) == c = 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. The behavioral modules act as simple and clear documentation of programs which in first analysis give the implementor strict guidelines.

SPECIFYING THE ARCHITECTURE OF JAVA APPLICATIONS

The *software architecture* concerns the design of the overall structure of a software system, including its dynamic behavior and its decomposition in simpler computational elements. [16]. 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 [15].

As in [1] we recognize *components* and *connectors* as basic elements of an architectural description. Software architecture puts a radical distinction among components and connectors, being the former the basic unit of computation, while the latter are involved in the coordination of the ongoing computation. Complex applications may be thought as collection of interacting components which collaborate to achieve a result. We think of a software component as an "abstraction with plugs" [13], i.e. a component encapsulates both data and independent behaviour with a well-defined way to interact with the external environment and the other components. Furthermore they must be integrated in a component-oriented software development process in which it must be possible to reason on the individual component properties and behavior as well as on the overall system resulting by their composition. We first provide an abstract model of a component and then devise a notation for specification of system made up of components.

The abstract model for components and connectors

In our notation, components like generic modules, have an algebraic specification provided by the included traits, and a behavioral module which define their behaviour.

Traits for a generic component and a generic 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])
connector 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. Starting from the traits for a generic component (connector), more specific descriptions of components can be derived by extension and/or parameterization of generic theories, exploiting the usual mechanism for the inclusion or instantiation of theories. The idea is to provide a hierarchy of theories whose leaves are the components (connectors) one wants effectively to use for the needed specification for the architecture of the system (fig. 1).

Configuration modules

At the architectural level of design, a system is described through its constituting elements and the interaction patterns occurring among them. Then a *configuration* module lists the instance of design elements which form our system, the Ljala modules which are used for the specification and the attachments between ports of the components and roles of the connectors. The structure of a configuration module is the following:

system-name:configuration
component:
      list of component instances
connector:
      list of connector instances

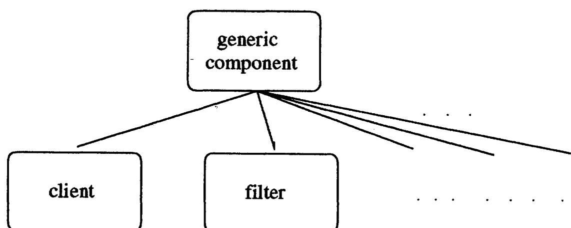

Figure 1 Hierarchy of components

attached:
list of connections among ports and roles
behaves:
activation rules

The configuration module defines the topology of the system being built. The *component* and *connector* parts of the module serve to name the instance of the components and the connectors, respectively, used for the description of the system. For each type a trait must have been provided. The *attached* part lists the connections among components and connectors. A number of *behave* clauses can be stated for each component in order to specify its active behaviour. Each clause is composed by a precondition on the state of the component which acts as a trigger for the activation of the rule. When components are connected together to build a complete system, the theory of the system is obtained through simple composition of the theories of the individual components. Semantics of composition rests about the assumption that parts shared by the attachments clauses represent the same state component. Consistency of the specification ensures that the combination of the individual specifications of components and connectors still make sense when put together and give a well-formed description of the system.

Modeling beans

Our methodology can be successfully applied for the modeling of applications exploiting the JavaBeans framework, i.e. the Java extension providing component based application services. The beans' interaction paradigm fulfills the event-driven programming model. A bean acts as a source of events and maintains a list of listeners which are other objects potentially interested in the firing of particular events. Adapters are event listeners which are intermediate objects between the event source and the object willing to react to event occurrences.

In our model, a generic bean obviously will be a component with particular structural characteristics. The connectors will be *adapters* which have the task to match events fired by the source component and to forward them to the listener objects waiting to react to such events. In the following we give the LSL trait for a bean and model a simple bean implementing a button.

```
bean:trait
includes component,
     Queue(event,oport),
     Queue(event,iport),
     Set(listener,Listeners)
bean tuple of comp:component, lis:Listeners
introduces
     assoc:  Listener -> Oport
asserts
     ∀ l,l':listener
     assoc(l)≠assoc(l')
```

```
button:   trait
includes bean,String
button tuple of b:bean, label:String

adapter:  trait
includes connector,
     Queue(event,Inrole).
     Queue(event,Outrole)
adapter tuple of conn:connector,source:Inrole,dest:Outrole
```

A bean has a set of listeners which are the other components interested in the firing of the events. Each listener is notified of the occurred event through one of the output ports of the component; each listener is associated to a different output port. The call is then forwarded by the adapter connector to the destination component. The traits above give only the abstract model of a button bean, whereas the behaviour and the interactions between collaborating beans can be described by opportune interface and configuration modules, respectively.

CONCLUSION AND RELATED WORK

The specification of software architectures is a relatively new field of application for Larch formalism. A similar approach was carried on by Baraona and Alexander in [4] where VSPEC, a Larch based interface language for VHDL, is combined with VHDL constructs to support the formal description of architectures.In [14] architectures are described as theories to specify the behavior of a system through a collection of axioms describing the behavior of each component. Several architectural description languages have been developed to provide software designers with notations and tools to specify and reason about architectural designs. In [15], architectural descriptions can be compiled to obtain executable code or a rapid prototype for the simulation of system behavior can be constructed as in [12]. General specification notations have also been exploited in the study of software architectures such CSP [2] and CHAM [9] to give behavioral specification for the system being developed.

The main problem with these approaches is the "implementation gap" which software designers have to face when passing from the abstract specification to the implementation of a system. We think that our notation provides a right level of abstraction between formality and practice supporting a refinement process which guides the implementor, from the abstract model to the concrete realization of a system.

Future work involves the evaluation of our method for the description of the architecture of concrete software systems, improving the analysis capabilities of the model. Furthermore, we are studying an operational semantics model both for Ljala specifications and Java programs, such that the relationship between specifications and its correct implementations can be investigated. A simple parser of Ljala specification has been developed; we plan to make this tool the front-end of a system to perform static semantics checks. Finally, a more ambitious goal consists of defining a complete ontology for recurrent components and connectors in order to provide software designers with well specified and functioning modules to be easily combined together.

References

[1] 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.

[2] R. J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1997.

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

[4] P. Baraona and P. Alexander. Abstract Architecture Representation using VSPEC. In *to appear in VLSI Design*, 1996.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

[13] O. Nierstrasz. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object Oriented Software Composition.* Prentice-Hall, December 1995.

[14] J. Penix and P. Alexander. Declarative specification of software architecture. In *to appear in the Proceedings of the 12th International Automated Software Engineering Conference*, November 1997.

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

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

[17] Sun Microsystems Inc. http://www.sun.com.