ACOEL **on** CORAL
A COmponent Requirement and Abstraction Language

An Extended Abstract

Vugranam C. Sreedhar
IBM TJ Watson Research Center
Hawthorne, NY 10532
sreedhar@watson.ibm.com

ABSTRACT

CORAL is a language for specifying properties of ACOEL, a component-oriented extensional language. The design of CORAL is based on input/output automata and type state. The properties of ACOEL components that need to be verified are specified using CORAL. A verification engine will then crawl through CORAL and verify whether ACOEL can be safely executed or not. In this paper we focus on CORAL, and show how to specify properties of ACOEL. We will also briefly discuss the concurrent modification problem that is commonly encountered in the iterator design pattern.

1. INTRODUCTION

The Internet has revolutionized the kinds of software applications that are currently being developed. These days people are talking about applications as services just as electricity and telephone services. When software are treated as services, it is important to ensure that they are properly packaged as components that can be easily connected to other software components, and it is even more important that (1) software components be certified that it will not do any harm to other components or the environment in which it is deployed, and (2) the clients will properly use the components. ACOEL is a component-oriented extensional language for creating and plugging components together [22, 23].[1] In ACOEL, a component developer can specify and abstract properties and requirements of components using CORAL (a COmponent Requirement and Abstraction Language). Depending on the context in which a component is used, a certification tool will try to certify that the component is well-behaved and is safe for plugging into the system. In this paper we will mostly focus on CORAL.

There are two aspects to CORAL: abstraction and requirements. Abstraction essentially suppresses the irrelevant details of a component so that one can focus just on those properties that we wish to verify. There is definitely a compromise between abstraction and the level of details that one is interested in verifying. Requirements are constraints that are necessary for proper functioning of components. There are many different kinds of requirements that a component will want to enforce. For instance, a square root function $sqrt(x)$ will require that x is not a negative number. For proper functioning of a FTP component, it is required that a client first connects to a file server before getting files from the server. Some of the popular modeling and specification languages and tools in the literature include UML/OCL (Unified Modeling Language/Object Constraint Language) [10], JML (Java Modeling Language) [15], Larch [12], SMV [17], etc. Once the requirements of a component are specified using one of these languages, the underlying system will then encode the specification into a mathematical structure and then prove the required properties.

To ensure usability of an abstraction and specification language, it is important to maintain a close correspondence between the component concrete language and the language used for specifying abstraction/requirement of components. JML, for instance, is tailored to Java [15]. CORAL is a requirement and an abstraction language for expressing and proving properties of ACOEL components. A component in ACOEL consists of a set of typed input ports and output ports. The input ports of a component consists of all the services that the component will provide, while the output ports are all the services that the component require for correct functioning. A port type can be either an interface type or a delegate type. An interface type consists of a set of methods and named constants, whereas a delegate type is an encapsulated signature of a method. The internal implementation of a component in ACOEL is completely hidden from the clients (i.e., a black-box component). In CORAL, the set of input ports of a component are abstracted as a set of *input actions*, the set of output ports of a component are abstracted as a set of *output actions* and the internal implementations of a component

[1] ACOEL was initially called as York.

are abstracted as a set of *internal actions*. The states of a component, and of the environment are encoded using state variables and data types. The above actions when performed on a state will transform the state to another state. In CORAL such state transitions are expressed using a state transition relations. The CORAL model of an ACOEL component is very to close to an *input/output automaton* (IOA) [13].

The rest of the paper is organized as follows: Section 2 gives a brief introduction to ACOEL. Section 3 discusses IOA modeling of CORAL. Section 4 introduces CORAL using a simple example called the concurrent modification problem. Section 5 discusses some of the related work. Finally, Section 6 gives our conclusion and also projects some of the future research direction.

2. ACOEL

The design of ACOEL was motivated by the following component design principles.

- *Pluggable Units* A component is a unit of abstraction with clearly defined external contracts and the internal implementation should be encapsulated. The external contract should consist of both the services it provides and the requirements it needs when it is plugged or (re-)used in a system.
- *Late and Explicit Composition*. For a component to be composable by a third-party with other components, it must support *late* or *dynamic composition*. During the development phase, requirements of a component should only be constrained by some external contract. Then, at runtime, an explicit connection is made with other "compatible" components (i.e., one that satisfy the constraints) to effect late composition.
- *Types for Composition*. Typing essentially restricts the kinds of services (i.e., operations or messages) that can be requested from a component.
- *Restricted Inheritance.* In OO programming, it is well-known that one cannot achieve both true encapsulation and *unrestricted* class inheritance with overriding capabilities [21]. In ACOEL, classes (which support inheritance) are second-class citizens, and are not visible to the external clients.
- *No Global State.* In ACOEL, there are no global variables and public methods that are visible to the entire system.

Let us briefly illustrate ACOEL by implementing the Iterator design pattern [11]. An Iterator pattern consists of an aggregate (e.g., set, list, array, etc.) and an iterator that traverses the aggregate. The main construct in ACOEL is `component`. A `component` consists of a set of *typed* input ports and output ports. A `List` component, defined below, consists of two input ports: one port is used by the client code to add / get / remove list elements and for creating an iterator, and the other port is used by the iterator to add / remove / get list elements. A client uses the following type to access services from the `List` component.

```
interface CLIntf {
  void add(int index, Elem e) ;
  void remove(int index, Elem e) ;
  Elem get(int index) ;
  ListIter iterator() ;
}
```

An iterator component interacts with the `List` component using the following interface.

```
interface ILIntf {
  void remove(int index, Elem e) ;
  Elem get(int index) ;
  void start() ; // start of the iterator
  void end() ; // end of the iterator.
}
```

The `start()` method and `end()` are basically used to start and end an iteration, and `iterator()` is a factory method that returns an iterator component.

Next we define the `List` component.

```
component List {
  in CLIntf clin ;
  in ILIntf ilin ;
  ListNode head = null ;
  int count = 0 ;
  List(){head = null ; count =0 ;}
  class ListNode {
     Elem e ;
     ListNode n ;
     ListNode(){} ;

  }
  class CLCls implements CLIntf, ILIntf {
    void add(int index, Elem e) { ...};
    void remove(int index, Elem e) {...};
    Elem get(int index) {...} ;
    int count(){return count ;}
    ListIter iterator() {
      return new ListIter(This) ;
    }
    void start() { ...}
    void end() { ...}

  }
  attach clin to CLCls ;
  attach ilin to CLCls ;
}
```

The `attach` statement essentially attaches an input port to a particular implementation class inside the component. Any messages that arrive at an input port is forwarded to the instance of the class that is attached to the input port. The class instance will either process the message or it will delegate to another class instance inside the component.

Next we define the `ListIter` component. It consists of one input port and one output port. The output port `ilout` is used to connect to the input port `ilin` of `List`. A client component uses the input port `clin` for accessing services of the `ListIter`. First let us define the type `CIIntf` of input port `clin`.

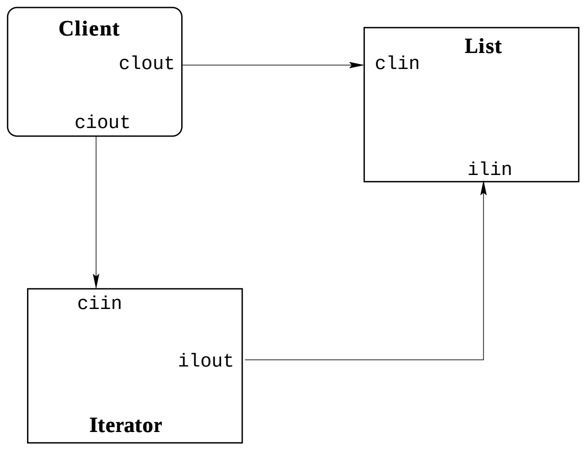

Figure 1: Various components in Iterator pattern

```
interface CIIntf {
  Elem next() ;
  Elem start() ;
  boolean hasNext() ;
  void remove() ;
}
```

A client component uses the `start()` method to start a new iteration. Here is the `ListIter` component for iterating over the elements of a list.

```
component ListIter {
  in CIIntf clin ;
  out ILIntf ilout ;
  ListIter(List l ) {
    connect ilout to l.ilin ;
    pos = 0 ;
  }
  int pos = 0 ;
  class ILCls implements ILIntf {
    Elem next() {
      Elem e = ilout.get(pos) ;
      pos++ ;
      return e ;
    } ;
    Elem start(){
      ilout.start() ;
      pos = 0 ;
    }
    boolean hasNext() {
       if (pos < ilout.count())
         return true ;
       ilout.end() ;
       return false ;
    } ;
    void remove() {...} ;
  }
  attach clin to ILCls ;
```

A client component has to first explicitly `connect` to a component before obtaining the services. Notice that `ListIterator` can invoke services of `List` component via its own output port `ilout`, and this output port is connected to input port `ilin` of `List` component instance.

Finally, here is a client code that wants to access the `List` component and the `ListIterator` component.

```
component Client {
  out CIIntf ciout ;
  out CLIntf clout ;
  main() {
    List l = new List() ;
    connect clout to l.clin ;
    // add a bunch  of elements ...
    ListIterator li = l.iterator() ;
    connect ciout to li.ciin ;
    ciout.start() ;
    while(ciout.hasNext() {
      Elem e = ciout.next() ;
    }
  }
```

Figure 2 shows the overall structure of the iterator pattern. There can be more than one iterator that is simultaneously active. A client will typically use the iterator created by the list component (through the factory method `iterator()`). The list component can ensure that it will only interact with iterators that it created for a client. Whenever there are multiple simultaneous iterators, there is a potential for concurrent modification of the list by multiple iterators (which will lead to inconsistent states). We will discuss this problem later in the paper.

3. MODELING COMPONENTS

A component in ACOEL consists of (1) an external contract made of typed input and output ports, and (2) an internal implementation consisting of classes, methods, and data fields. A client can only see the external contract and the internal implementation is completely encapsulated. A component provides services via its input ports, and specifies the services its requires via its output ports. In ACOEL, a `connect` statement makes an explicit connection between an output port of a component to a "compatible" input port of another component. Let *connect* $\check{c}_1\langle p_o\rangle$ *to* $\check{c}_2\langle q_i\rangle$ be a connect statement. For this connection to be *compatible*, it is necessary that $q_i <: p_o$. The sub-type relation ensures that any message sent over the connection by $\check{c}_1$ can be processed by $\check{c}_2$. But the sub-type relation is not sufficient to ensure port compatibility. In ACOEL, we enforce other kinds of constraints using CORAL.

We use a framework that is similar to input/output automaton (IOA) to model ACOEL components. Abstractly, a component automaton (CA) consists of a set of actions, a set of state, and a set of transitions. The set of actions are classified as either *input actions* $in(A)$ (corresponding to messages arriving at input ports), *output actions* $out(A)$ (corresponding to the requirements at output ports), and *internal actions* $int(A)$ (corresponding to internal calls). Let $acts(A) = in(A) \cup out(A) \cup int(A)$.

Similar to IOA, a CA A consists of the following four components:

- $sig(A)$, a signature
- $states(A)$, a set of states (not necessarily finite)
- $start(A) \subseteq states(A)$, a set of start or initial states
- $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$, a state-transition relation, such that for every state s and every input action π, $(s, \pi, s') \in trans(A)$.

An action π is enabled in a state s if $(s, \pi, s') \in trans(A)$. Input actions are enabled in every state (i.e., a component cannot block messages arriving at its input ports). This is not a big restriction, since almost always we can throw an error condition for messages that a component cannot handle (also, we can use the type system to ensure that no arbitrary message arrives at input ports of a component).

There are few differences between a regular IOA and the kinds of programs that we are dealing with in ACOEL. First, components in ACOEL can be dynamically created and destroyed. Also, each component has its own state. In ACOEL there are no global variables and methods. Since component instances are dynamically created and destroyed, an IOA model should include actions for creation and destruction of automaton and for modeling system of automaton. To model dynamic creation of components, we introduce a *create action* $crt(A)$ that corresponds to creation of an automaton A. The $crt(A)$ will also invoke the constructor function that modifies the state of A. The create action $crt(A)$ can be thought of as an input action to the newly created automaton A, and the input action will invoke the constructor methods of the corresponding component. The create action will be executed by another automaton for creating a new automaton. At any instance, only a finite set of automaton exists. We can think of a configuration $\mathcal{C}$ as a finite set $\{\langle A_1, s_1\rangle, \ldots \langle A_n, s_n\rangle\}$, where A_i, for $1 \leq i \leq n$, is automaton identifier and s_i is the state of A_i. An action π essentially changes a configuration $\mathcal{C}$ to a new configuration $\mathcal{C}'$, a create action will add a new automaton to $\mathcal{C}$, and all other action will simply change the states of existing automaton.

4. THE CORAL LANGUAGE

In this section we will briefly introduce CORAL using the iterator pattern example. Our intention is only to expose the core ideas behind CORAL. A component automaton (CA) consists of two main parts: (1) `states` and (2) `transitions`. The `states` part consists of a set of state variables, whose types can be either primitive or composite data types. Primitive data types include `char`, `string`, `int`, `float`, and reference type. Composite data types can be either in-built types or user-defined types. In a `types` part one can define new data types (see Figure 2). The `transitions` part consists of a set of state transition written in the style of *pre-condition-effect-error* for each action. This is illustrated in `ListAutomaton`, a CORAL automaton for the `List` component (see Figure 2). For each action, we list the `pre`-condition part, the `eff`ect part, and the `error` part. Whenever the pre-condition part is satisfied, the `eff` part is executed otherwise the `error` part (if defined) is

```
coral ListAutomaton {
  types:
    Iter {
       int id ;
       enum st = {active, passive} ;
    }
  states:
    int srcId ;
    List l ; // list type
    Iter iter[] ; // a hash of iterators.
  transitions:
    input void CLIntf.add (int index, Elem e) {
      pre:
        (forall i iter[i].st==passive) &&
           (l.length < index)
      eff:
         l.insert(index, e) ;
      error:
          throw AddException ;
    }
    input void CLIntf.remove(int index, Elem e) {
      pre:
        (forall i iter[i].st==passive) &&
           (l.length < index)
      eff:
        l.remove(index, e) ;
      error:
        throw RemoveException ;
    }

    input Elem CLIntf.get (int index) {
      pre:
          (l.length < index)
      eff:
      error:
        throw GetException ;
    }
    input CLIntf.iterator () {
      pre:
      eff:
        // add a new iterator to iter
 int newsrcId = create ListIterator
         iter.add(newsrcId) ;
 return newsrcId ;
    }
    input ILIntf.start() {
      pre:
      eff:
        iter[srcId].st = active ;
    }
    input ILIntf.end() {
      pre:
      eff:
        iter[srcId].st = passive ;
    }
    input Elem ILIntf.get (int index) {
      pre:
         (iter[srcId] == active)
         (l.length < index)
      eff:
      error:
        throw GetException ;

    }
    input ILIntf.remove() {
       pre:
         (iter[srcId] == active) &&
         (forall i  and i!=srcId
           {iter[i].st==passive} ) &&
         (l.length < index)
       eff:
         l.remove(index, e) ;
       error:
         throw RemoveException ;
    }
}
```

Figure 2: CORAL for `List` component.

executed. The eff part essentially performs state transformations.

For the example in Figure 2, the states part consists of three states: srcId is the identity of the source component that is invoking the input action. l is a list with operations such insert, remove, etc. An insert operation will add an element to l and changes the state of l to a new l. The iter state keeps track of all iterators that a client created. A create operation will essentially create a new iterator identity and saves it in iter. Consider the input action CLIntf.add(), the eff part will be executed only if all the iterators in iter[] are passive and the index is less than the length of the list. Otherwise the error part is executed.

We essentially translate a CA to an IOA, and then verify properties in IOA. An input action in CA also returns a value (which can either a normal value or an error condition). So an input action in CA is translated into a input action followed by an output action in IOA. The purpose of the output action is to return a value or an error condition back to source component. We do the same for an output action in CA (i.e., it is also broken into an output action followed by an input action).

Unlike in IOA, in CA we typically do not perform composition operation explicitly—we typically verify whether a composition is a valid composition, and the actual composition is effected by subtype relation between ports via connect statement. There are two kinds of verification we are interested: *invariance* and *reaching an error state*. An invariance is a property that is true in all reachable states. Reaching an error state means that a pre-condition fails and an "error" state is reached. An execution of an automaton is a finite sequence of $s_o, \pi_1, \ldots, \pi_n, s_n$, with s_0 being a start state of the automaton. A state is reachable if it occurs in some execution. Our main goal is verification of safety properties (rather than liveness or fairness properties).

Let us briefly illustrate one kind of verification problem, called concurrent modification problem (CMP). This problem was motivated from Ramalingam et al. [18]. We have simplified the problem from what is described in Ramalingam et al. [18]. The main problem with CMP is that when an iterator is active a modification to the underlying aggregate structure can cause an inconsistency between the iterator and aggregate structure. Most implementation of an iterator pattern will allow modification to an aggregate structure only through the iterator (especially when an iterator is active). Let us slightly modify the client code given in Section 2 and include the statement l.add(0,e) in the while-loop.

```
component Client {
  out CIIntf ciout ;
  out CLIntf clout ;
  main() {
    List l = new List() ;
    connect clout to l.clin ;
    // add a bunch  of elements ...
    ListIterator li = l.iterator() ;
    connect ciout to li.ciin ;
    while(ciout.hasNext() {
      Elem e = ciout.next() ;
      l.add(0,e) ;
    }
  }
```

In the ListAutomaton the precondition for l.add will fail since an iterator in iter state *may* still be active. Although the above example looks trivial there are many non-trivial phases that one has to go through before coming to the conclusion. For instance, we need alias analysis information to disambiguate different iterators. We need to use theorem proving techniques to verify invariants defined in the pre-conditions. We have used end() method to explicitly terminate an iterator. Compared to Ramalingam et al., our approach gives very conservative result. It is to be noted that our intention in using CMP is only to illustrate the use of IOA for verifying this, albeit simplified, problem.

5. DISCUSSION AND RELATED WORK

Verifying software system is an age-old, but certainly not a solved problem. Many specification and verification techniques have been proposed in the literature for ensuring that software systems are safe and well-behaved [2, 14, 24, 3, 12, 7, 20]. With the advent of the Internet-based applications it is even more important to ensure safety and security of software system. In this paper we presented CORAL for abstracting and specifying requirements of ACOEL components. We used IOA for modeling ACOEL components. Typically in the past, IOA has been used to model distributed system. In CORAL we use IOA to verify whether a component when plugged into a system will behave correctly, and also whether a client of the component will use the component correctly or not. CORAL can be used to verify other kinds of constraints such a protocol verification [25]. We can simply encode the correct sequences of method calls using an automaton.

This paper presents a preliminary experience of using IOA for software verification. There are many open-ended problems that needs to be resolved. Handling aliasing, sub-type polymorphism, etc. presents some interesting challenges. Recently Attie and Lynch proposed dynamic IOA that can handle dynamic creation and destruction of automaton. Rather than thinking in terms of single automaton, dynamic IOA goes one step further and defines a configuration of interacting automata [4]. We are currently exploring on how to use the full potential of dynamic IOA in CORAL. For verification purposes we have to deal with practical programming languages which typically include aliasing and polymorphism. Unlike IOA, our main goal is verification of components. One component can be connected to another component through their ports if the corresponding port types have a sub-type relation (i.e., the input port should be a subtype of the output port). We use CORAL to go beyond subtype relation and verify other kinds of constraints [16].

Model checking is a classical approach to verification of software systems [8]. Bandera is a collection of tools for model-checking concurrent Java programs [9]. It takes Java source code, compiles them, and generates code for verification tools like SMV and SPIN. SLAM project is very similar to Bandera project, except that SLAM also uses predicate abstraction and discovery to point errors in C code [5]. Strix is specification language for expressing business process and a Strix compiler once generates code for SMV model checker [6]. CANVAS uses EASL specification and translate them to a 3-valued logic for verifying program properties [18]. JML is a Java Modeling Language and it uses design-by-contract and Larch theorem prover to verify program properties [15]. There are several other projects related to software verification.

Another important, but related, area is the Architecture Description Language (ADL) [19]. A software system is typically starts off with a requirement and a design phase. During this phase, the implementation details are typically ignored and the focus is on understanding and developing software architecture. ADLs are typically used at this phase to specify the structure and the requirements of a software system. ArchJava is an example of integrating ADL with Java [1]. CORAL can be used as a ADL. One can express the requirements of ACOEL components, even before implementing them using CORAL. To use as an ADL, we need a way to compose component automaton. For this we rely on IOA theory of composing automaton.

6. CONCLUSION

In this paper we briefly introduced CORAL as a language for abstracting and specifying ACOEL components. CORAL is based on IOA. Unlike classical IOA, our intention in using the theory of IOA is for verification of software components. We are currently working on three aspects of CORAL. First we are refining on the syntax and semantics of CORAL. Second, we are focusing on the dynamic IOA model for CORAL. Finally, we are looking at ways to model aliasing, sub-typing, classes, and other states within IOA. Both ACOEL and CORAL are at design stages, and we are at initial stages of implementation. We expect to publish more details of CORAL in the near future.

Acknowledgement

I thank Deepak Goyal for valuable discussions and comments on an earlier draft of the paper.

7. REFERENCES

[1] Jonathan Aldrich and Craig Chambers. ArchJava: connecting software architecture to implmentation. Technical Report UW-CSE-01-08-01, Univ. of Washington, August 2001.

[2] Dean Allemang. Extending the applicability of formal verification techniques. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997*, pages 1–10, September 1997.

[3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[4] Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: a formal model for dynamic systems. In *CONCUR'01: 12th International Conference on Concurrency Theory*, LNCS. Springler-Verlag, 2001.

[5] T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[6] B. Bloom. Seeing by owl-light:Symbolic model checking of business application requirements. Technical Report ????, IBM T.J. Watson Research Center, 2001.

[7] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[8] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[10] Martin Fowler and Kendall Scot. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, New York, NY, 1995.

[12] S. Garland, J. Guttag, and J. Horning. An overview of Larch. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 329–348. Springer-Verlag Lecture Notes in Computer Science 693, 1993.

[13] S. Garland and N. Lynch. Using i/o automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[14] M. Goedicke, H. Schumann, and J. Cramer. On the specification of software components. In Jean-Pierre Finance, editor, *Proceedings of the 6th International Workshop on Software Specification and Design*, pages 166–174, Como, Italy, October 1991. IEEE Computer Society Press.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, November 1994.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts,

1993.

[18] G. Ramalingam, A. Warshavsky, J. Field, and M. Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Technical Report RC22145, IBM T.J. Watson Research Center, August 2001.

[19] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[20] Murali Sitaraman, Lonnie R. Welch, and Douglas E. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledege Engineering*, 3(2):207–229, 1993.

[21] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Workshop on Object-Oriented Programming*, pages 165–188, Camebridge, MA, 1987. MIT Press.

[22] Vugranam C. Sreedhar. ACOEL: A component-oriented extensional language. Technical report, IBM T.J. Watson Research Center, 2001.

[23] Vugranam C. Sreedhar. York: Programming software components. In *Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001. Poster session.

[24] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[25] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.