An Interface Model for Parallel Components

Milind Bhandarkar and L. V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
(bhandark@csar.uiuc.edu, kale@cs.uiuc.edu)

Abstract. Component architectures promote cross-project code reuse by facilitating composition of large applications using off-the-shelf software components. Existing component architectures are not geared towards building efficient parallel software applications that require tighter runtime integration of largely independent parallel modules. We have developed a component architecture based on Converse, a message-driven multiparadigm runtime system that allows concurrent composition. In this paper, we describe an interface model for this component architecture, which allows reusable component modules to be developed independently of each other, and mediates, monitors, and optimizes interactions between such components.

1 Introduction

Developing scalable parallel applications is a difficult task. Applicability and availability of different computational techniques and programming models demands that these programs be developed as a largely independent collection of software modules based on a common framework. Several parallel languages and application frameworks have been developed to facilitate the implementation of large-scale parallel applications. These frameworks typically focus on a few techniques and applications. No single framework is suitable for all numerical, computational and software engineering techniques employed and under development in academia and research laboratories. Thus, one would need to employ different application frameworks for different modules of a complex application. However, it is currently impossible to efficiently couple together the software components built using different frameworks. This is mainly because these application frameworks have not been built with a "common component model".

One way to couple these independent software modules together is to run them as separate application processes, using mechanisms such as sockets to communicate. This tends to be extremely inefficient. While we want the individual software components to be modular and and independently developed, these components may not always have large enough grainsize to ignore the efficiency of coupling. Also, in order to be scalable to a large number of processors, especially while solving a fixed problem, it is required that the coupling efficiency be maximized. Therefore, it is imperative that the communicating software components be part of the same process in order to have efficient coupling between software components (known in the component terminology as "in-process components".)

In-process components eliminate the inefficiency resulting from separate application processes; do not require the data exchange to be serialized; and are close in efficiency to a procedure call. However, this efficiency must be achieved without sacrificing independence of the individual components and uniformity of data exchange.

In order to achieve this, first we make sure that the software components, which may already be parallel, coexist within a single application. Since these components often use different parallel programming paradigms and different programming languages, this is a non-trivial task. We need to keep the semantics of individual programming paradigms, while allowing these paradigms to cooperate and interact with each other. For this purpose, we need to have a *Common Language Runtime* (CLR) for all the programming paradigms we want to support. Since these programming models differ in the amount of concurrency within a process and the way control is transferred among different entities of the programming model, we have developed a runtime system called Converse, based on message-driven parallel programming paradigm. Converse employs a unified task scheduler for efficiently supporting different concurrency levels and "control regimes". Detailed description of Converse can be found in [4].

While Converse allows us to have multiple "in-process components" within an application, it does not specify how different components interact with each other. For this purpose, we need an interface model that allows components to access other components' functionality in a uniform manner. Converse being a message-driven CLR, traditional interface description languages do not perform well because they assume semantics of a blocking procedure call. Enhancing the traditional interface languages to allow asynchronous remote procedure calls results in other problems, such as proliferation of interfaces. We have developed a different interface model based on separate input and output interfaces, which enables us to overcome these problems. An orchestration language – a high level scripting language that utilizes this interface model to make it easy to construct complete applications from components – is also defined.

This paper describes the interface model for our message-driven component architecture. Our interface model encourages modular development of applications yet allows tighter integration of application components at run time.

1.1 Outline

The next section reviews commonly used component architectures for distributed and parallel computation, and describes their limitations. We list the desired characteristics of an ideal interface model for parallel components in section 3 and show that extensions of existing interface models do not match these characteristics. We describe our interface model, which is a crucial part of our component architecture, in section 4 with several examples, and discuss the runtime optimizations that become possible due to expressiveness of our interface model. We conclude in section 5.

2 Component Architectures

Software components allow composition of software programs from off-the-shelf software pieces. They aid in rapid prototyping and have been successfully used for developing GUIs, database as well as Web applications. A component model specifies rules to be obeyed by components conforming to that model. A software component is a set of objects with published interfaces, which obeys the rules specified by the underlying component model. A component model alongwith a set of "system components" defines a component architecture.

Various component architectures such as COM [9], CORBA [8], and JavaBeans [7] have been developed and have become commercially successful as distributed application integration technologies. Parallel computing community in academia and various U.S. national laboratories have recently formed a Common Component Architecture forum (CCA-Forum [1]) to address these needs for parallel computing.

Current component technologies cross language barriers, thus allowing an application written using one language to incorporate components written using another. In order to achieve this, components have to provide interface specification in a neutral language called Interface Description Language (IDL). All component architectures use some form of Interface Description Language. Though these IDLs differ in syntax, they can be used to specify almost the same concept of an interface. An interface consists of a set of "functions" or "methods" with typed parameters, and return types. Caller of a method is referred to as "Client", whereas the object whose method is called is referred to as "Server". Client and Server need not reside on the same machine if the underlying component model supports remote method invocation.

Remote method invocation involves marshaling input parameters (or serializing into a message), "stamping" the message with the component and method identifier, dispatching that message, and in case of synchronous methods, waiting for the results. On the receiving side, parameters are unmarshalled, and the specified method is called on the specified instance of the component. Similar technique is employed for creating instances of components, or locating components, by invoking methods of system component instances, which provide these services. The server machine, which contains the server component instances, employs a scheduler (or uses the system scheduler) that waits continuously for the next message indicating a method invocation, locates the specified component instance, verifies access controls, if any, and calls the specified method on that component instance.

2.1 Limitations of Current Component Architectures

Component architectures in the distributed programming community (such as COM, JavaBeans, CORBA) do not address issues vital for coupling parallel software components. Though all of them support in-process components, they incur overheads unacceptable for the needs of parallel application integration. Microsoft COM has limited or no cross-platform support vital for the emerging parallel computing platforms such as the Grid [3]. JavaBeans have no cross-language support, and needs components to be written only using Java, while FORTRAN dominates the parallel computing community. CORBA supports a wide variety of platforms and languages, but does not have any support for abstractions such as multi-dimensional arrays.

Common Component Architecture (CCA) [2] is one of the efforts to unify different application frameworks. The CCA approach tries to efficiently connect different applications developed using various frameworks together by providing parallel pathways for data exchange between components. They have developed a Scientific Interface Description Language (SIDL) to allow exchange of multi-dimensional arrays between components, and have proposed a coupling mechanism (CCA-ports) using *provides/uses* design pattern. For parallel communication between components, they have proposed collective ports that implement commonly used data transfer patterns such as broadcast, gather, and scatter. In-process components are connected with "direct-connect" ports, which are close in efficiency to function calls. However, these component method invocations assume blocking semantics. Also, CCA restricts itself to SPMD and threaded programming paradigms, and does not deal with coexistence of multiple non-threaded components in a single application. Hooking up components dynamically is listed among future plans, and it is not clear how the efficiency of coupling will be affected by that.

The most suitable way of combining multiple components using CCA is by developing wrappers around complete application processes to perform parallel data transfer, delegating scheduling of these components to the operating system. Heavyweight process scheduling by the operating system leads to coupling inefficiencies. If the communicating components belong to different operating system processes even on the same processor, invoking a component's services from another component requires an inefficient process context-switch. For example, on a 500 MHz Intel Pentium III processor running Linux, invocation of a "null service" (that returns the arguments passed to it) takes 330 microseconds when the service is resident in another process, while it takes 1.3 microseconds for service residing within the same process (Corresponding times on a 248 MHz Sun Ultra SPARC workstation running Solaris 5.7 are 688 and 3.2 microseconds respectively.)

Our component architecture is complimentary to the CCA effort. We support coexistence of multiple components (threaded and non-threaded; based on different programming paradigms) within an application process. Our proposed interface model uses asynchronous method invocation semantics and can be thought of as a lower-level substrate on which CCA ports could be implemented.

3 Interface Models

Software components in an application interact with each other by exchanging data and transferring control. An interface model defines the way these components interact with each other in an application. The ideal interface model for a parallel component architecture should have the following characteristics:

- It should allow easy assembly of complete applications from off-the-shelf components. An interface description of the component alongwith the documentation of the component's functionality should be all that is needed to use the component in an application. Thus an interface model should be able to separate the component definition from component execution.
- It should allow individual components to be built completely independently, i.e. without the knowledge of each other's implementation or execution environment or the names of entities in other components.
- Components should make little or no assumptions about the environment where it is used. For example, a component should not assume exclusive ownership of processors where it executes.
- It should be possible to construct parallel components by grouping together sequential components, and this method of construction should not be exposed to the users of the resultant parallel component.
- It should not impose bottlenecks such as sequential creation, serialization etc on parallel components. In particular, it should allow parallel data exchange and control transfer among parallel components.
- It should enable the underlying runtime system to efficiently execute the component with effective resource utilization.
- It should be language independent and cross-platform.

Traditional component architectures for distributed components use a functional representation of component interfaces. They extend the object model by presenting the component functionality as methods of an object. Thus, a component interface description is similar to declaration of a C++ object. Components interact by explicit method calls using the interface description of each other.

A straightforward extension of such interface models for parallel components would be to provide a sequential component wrapper for the parallel component, where functionality of a parallel component is presented as a sequential method invocation. This imposes serialization bottleneck on the component. For example, a parallel CSE application that interfaces with a linear system solver will have to serialize its data structures before invoking the solver. This is clearly not feasible for most large linear systems representations that need hundreds of megabytes of memory.

Another extension of the traditional interface models is to treat each parallel component as a collection of sequential components. Within this model, the interaction between two parallel components takes place by having the corresponding sequential components invoke methods on each other. While this model removes the serialization bottleneck, it imposes rigid restrictions on the structure of parallel components. For example, a parallel finite element solver will have to partition its mesh into the same number of pieces as the neighboring block-structured CFD solver, while making sure that the corresponding pieces contain adjacent nodes.

A major disadvantage of extending the interface models based on method-calls is that they make data exchange and control transfer between components explicit. Thus, they do not provide control points for flexible application composition and for effective resource management by the runtime system.

Lack of a control point at data exchange leads to reduced reusability of components. For example, suppose a physical system simulation component interacts with a sparse linear system solver component, and the data exchange between them is modelled as sending messages or as parameters to the method call. In that case, the simulation component needs to transform its matrices to the storage format accepted by the solver, prior to calling the solver methods. This transformation code is part of the simulation component. Suppose, a better solver becomes available, but it uses a different storage format for sparse matrices; the simulation component code needs to be changed in order to transform its matrices to the new format required by the solver. If the interface model provided a control-point at data-exchange, one could use the simulation component without change, while inserting a transformer component in between the simulation and the solver.

Lack of a control point for the runtime system at control transfer prevents the runtime system from effective resource utilization. For example, with blocking method invocation semantics of control transfer, the runtime system cannot schedule other useful computations belonging to a parallel component while it is waiting for results from remote method invocations.

Asynchronous remote method invocation provides a control-point for the runtime system at control-transfer. It allows the runtime system to be flexible in scheduling other computations for maximizing resource utilization. However, extending functional interface representations to use asynchronous remote method invocations pose other problems. It introduces "compositional callbacks" as a mechanism for connecting two components together.

When a component (caller) invokes services from another component (callee) using asynchronous remote method invocation, it has to supply the callee with a its own unique ID, and the callee has to know which method of the caller to call to deposit the results (see figure 1.) This is referred to as the "compositional callback" mechanism.

Compositional callback mechanism is equivalent to building an object communication graph (object network) at run-time. Such dynamic object network misses out on certain optimizations that can be performed on a static object network [10]. For example, if the runtime system were involved in establishing connections between objects, it would place the communicating objects closer together (typically on the same processor).

Another problem associated with the callback mechanism is that it leads to proliferation of interfaces, increasing programming complexity. For example, suppose a class called `Compute` needs to perform asynchronous reductions using a system component called `ReductionManager` and also participates in a gather-scatter collective operation using a system component called `GatherScatter`. It will act as a client of these system services. For `ReductionManager` and

```
Client::invokeService() {
  ServiceMessage *m = new ServiceMessage();
  // ...
  m->myID = thishandle;
  ProxyServer ps(serverID);
  ps.service(m);
}

Server::service(ServiceMessage *m) {
  // ... perform service
  ResultMessage *rm = new ResultMessage();
  // ... construct proxy to the client
  ProxyClient pc(m->myID);
  pc.deposit(rm);
}

Client::deposit(ResultMessage *m) {
  // ...
}
```

Fig. 1. Asynchronous remote service invocation with return results

`GatherScatter` to recognize `Compute` as their client, the `Compute` class will have to implement separate interfaces that are recognized by `ReductionManager` and `GatherScatter`. This is shown in figure 2. Thus, for each component, this would result in two interfaces: one for the service, and another for the client of that service. If a component avails of multiple services, it will have to implement all the client interfaces for those services.

4 Our Interface Model

Our interface model requires components to specify the data they use and produce. It takes the connection specification (glue) between components out of the component code into a scripting language that is compiled into the application. This provides the application composer and the runtime system with a control-point to maximize reuse of components. Asynchronous remote method invocation semantics is assumed for dispatching produced data to the component that uses them, thus supplying the runtime system with a control-point for effective resource utilization.

We mandate that the interface of a component consist of two parts: a set of input ports, and a set of output ports. A component publishes the data it produces on its output ports. These data become visible (when scheduled by the runtime system) to the connected component's input port. Connection between an input port of an object and an output port of another object are specified

```
class ReductionClient {
  virtual void reductionResults(ReductionData *msg) = 0;
}

class GatherScatterClient {
  virtual void gsResults(GSData *msg) = 0;
}

class Compute : public ReductionClient, public GatherScatterClient
{
    // ....
    void reductionResults(ReductionData *msg) { ... }
    void gsResults(GSData *msg) { ... }
}
```

Fig. 2. Proliferation of interfaces

"outside" of the object's code, e.g. using a scripting language. Each object can be thought of as having its own scheduler which schedules method invocations based on the availability of data on any of its input port, and possibly emits data at the output ports. Input ports of components have methods bound to them, so that when data become available on the input port, a component method is enabled[1].

For example, a simple producer-consumer application using this interface model is shown in figure 3. Note that both the producer and the consumer know nothing about each other's methods. Yet, with very simple scripting glue, they can be combined into a single program. Thus we achieve the separation of application execution from application definition. Individual component codes can be developed independently, because they do not specify application execution. They merely specify their definitions. The scripting language specifies the actual execution.

In figure 3, the data types of the connected ports of producer and consumer match exactly. For base types, wherever transformations between data types is possible, the system will implicitly apply such transformation. For example, if the type of `producer::Data` is `double`, and the type of `consumer::Data` is `int`, the system will automatically apply the requested transformations, and will still allow them to be connected. However, if `producer::Data` is `Rational` and `consumer::Data` is `Complex`, then system will not allow the requested connection. Thus, the application composer will have to insert a transformer object (see figure 4) between the producer and the consumer. A performance improvement

[1] Enabling an object method is different from executing it. Execution occurs under the control of a scheduler, whereas a method is enabled upon availability of its input data. This separation of execution and enabling objects methods is crucial to our component model.

```
class producer {
  in Start(void);
  in ProduceNext(void);
  out PutData(int);
};
producer::Start(void) {
  data = 0;
  PutData.emit(0);
}
producer::ProduceNext(void) {
  data++;
  PutData.emit(data);
}
```

(a) A Producer Component

```
class consumer {
  in GetData(int);
  out NeedNext(void);
};

consumer::GetData(int d) {
  // do something with d
  NeedNext.emit();
}
```

(b) A Consumer Component

```
producer p;
consumer c;

connect p.PutData to c.GetData;
connect c.NeedNext to p.ProduceNext;
connect system.Start to p.Start;
```

(c) Application Script

Fig. 3. A Producer-Consumer Application

hint "inline" can be interpreted by the translator for the scripting language to execute the method associated with the input port immediately instead of putting it off for scheduling later. This hint also guides the runtime system to place the transformer object on the same processor as the object that connects to its input.

The real power of this interface model comes from being able to define collections of such objects, and with a library of system objects. For example, one could connect individual sub-image smoother components as a 2-D array (figure 5) to compose a parallel image smoother component (see figure 6).

The composite `ImageSmoother` component specifies connections for all the `InBorder` and `OutBorder` ports of its `SubImage` constituents. Note that by specifying connections for all its components' ports, and providing unconnected input and output ports with the same names and types as `SubImage`, `ImageSmoother` has become topologically similar to `SubImage` and can be substituted for `SubImage` in any application. Code for `ImageSmoother` methods `InBorder` and `InSurface` is not shown here for lack of space. `InBorder` splits the input pixels into sub-

```
class transformer {
  inline in input(Rational);
  out output(Complex);
};

transformer::input(Rational d) {
  Complex c;
  c.re = d.num/d.den; c.im = 0;
  output.emit(c);
}
```

Fig. 4. Transformer Component

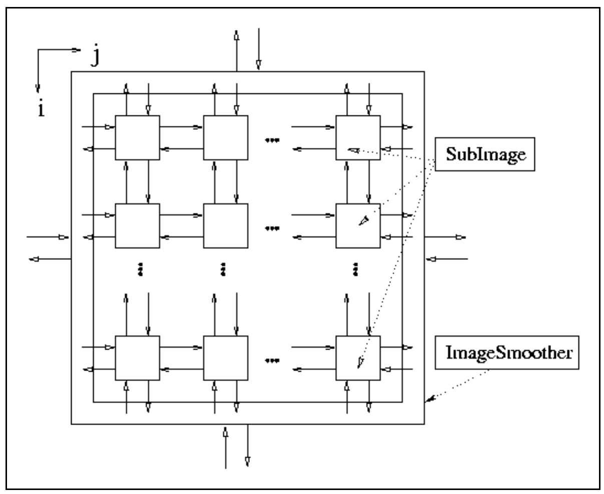

Fig. 5. Construction of a 2-D array component from elements

arrays and emits them on `OutSurface` ports. `InSurface` buffers the pixels until all the border pixels are handed over to it from a particular direction. It then combines all the pixels into a single array, and emits them onto corresponding `OutBorder` port.

Also, notice that the `ImageSmoother` component can configure itself with parameters `N` and `M`. `N` and `M` determine the number of rows and columns in a 2-D array of `SubImages`. They can be treated as attributes of the class `ImageSmoother`, which can be set through a script.

In this example, the interface of `SubImage` was serial. User of this component was expected to feed and receive one pixels array in each direction. The point of this example was to demonstrate the even after parallelization, the component interface can remian identical to the sequential component. If parallel interface is desired, that is also feasible, as our next example illustrates.

```
enum {EAST=0, WEST=1, NORTH=2, SOUTH=3};

class SubImage {
  in[4] InBorder(Pixels *);
  out[4] OutBorder(Pixels *);
}

class ImageSmoother <int N, int M> {
  in[4] InBorder(Pixels *);
  out[4] OutBorder(Pixels *);
  in[2*N+2*M] InSurface(Pixels *);
  out[2*N+2*M] OutSurface(Pixels *);

  SubImage si[N][M];
  // Make the east and west elements connections to surface
  for(int i=0; i<N; i++) {
    connect si[i][0].InBorder[WEST] to this.OutSurface[i];
    connect si[i][0].OutBorder[WEST] to this.InSurface[i];
    connect si[i][M-1].InBorder[EAST] to this.OutSurface[i+N];
    connect si[i][M-1].OutBorder[EAST] to this.InSurface[i+N];
  }
  // Similarly connect north and south border elements to surface
  for(int j=0; j<M; i++) {
    // ...
  }
  // Now, make internal elements connect to each other
  for(int i=1; i<=(N-1); i++) {
    for(int j=1; j<=(M-1); j++) {
      connect si[i][j].InBorder[0] to si[i-1][j].OutBorder[2];
      // ...
    }
  }
}
```

Fig. 6. A Parallel image smoother construction from sub-image smoother components

Consider the problem of interfacing Fluids and Solids modules in a coupled simulation. Each of the Fluids and Solids components is implemented as a parallel object. (Constituents of these modules, namely `FluidsChunk` and `SolidsChunk`, are not shown here for the sake of brevity.) A gather-scatter component `FSInter` specific to the application-domain can be used to connect an arbitrary number of Fluids chunks to any number of Solids chunks by carrying out the appropriate interpolations. The core interface description of this situation is shown in figure 7.

```
class Fluids<int N> {
  in[N] Input(FluidInput);
  out[N] Output(FluidOutput);
};
class Solids<int N> {
  in[N] Input(SolidInput);
  out[N] Output(SolidOutput);
};
class FSInter<int F, int S> {
  in[F] FInput(FluidOutput);
  out[F] FOutput(FluidInput);
  in[S] SInput(SolidOutput);
  out[S] SOutput(SolidInput);
};
```

(a) Component Interfaces

```
Fluids f<32>;
Solids s<64>;
FSInter fs<32,64>;

for(int i=0;i<32;i++){
  connect f.Output[i] to fs.FInput[i];
  connect fs.FOutput[i] to f.Input[i];
}
for(int i=0;i<64;i++){
  connect s.Output[i] to fs.SInput[i];
  connect fs.SOutput[i] to s.Input[i];
}
```

(b) Component Connections

Fig. 7. Fluids-Solids Interface in Coupled Simulation

Though most application compositions can be specified at compile (or link) time, for some applications (such as symbolic computations, branch-and-bound) it is necessary to dynamically specify connections, or to dynamically create objects. For such applications, apart from a scripting language, an API must be provided. This API will be available as an interface between the creator of the component and a "system" component. For example, creator's output port connects to the system's input port (system component is special in that it has infinite input and output ports), and emits the class type to be created, and also specifies connection information.

4.1 Runtime Optimizations

Several new runtime optimizations become feasible with our interface model because the composition and connectivity information is explicitly available to the runtime system. A few of them are described here.

Consider an example in a molecular dynamics application based on spatial decomposition [6], where each pair of subdomains (*patches*) has a compute object associated with it, which is responsible for computing pairwise cutoff-based interactions among atoms in those patches. If the patch neighborhood information is specified using our interface model (by constructing a 3-D grid of patches with a scripting language), these patches could be placed automatically by the runtime system on the available set of processors by taking locality of communication into account. Further, specification of connections between patches and compute object would allow the runtime system to place the compute objects closer to the patches they connect. Also, information about communication volume available from the Converse load balancing framework would enable the runtime system to place a compute object on the same processor as the patch that sends more atoms to it. We provide such runtime optimization techniques by incorporating the connectivity information in the Converse load balancing framework.

Connection specification also enables the runtime system to optimize data exchange between components that belong to the same address space. This can be achieved by allowing the components to publish their internal data buffers to the output ports. In the normal course, the published data will be sent as a message directed at the input port of the connected component. If the connected component belongs to the same address space, the runtime system may pass the same buffer to the input port of the connected component in the same process, when the corresponding input port declares itself to be readonly.

We are currently implementing the scripting language on top of the Charm++ runtime system [5], which is a C++ binding for the message-driven execution environment provided by Converse. An interface translator is being developed for translating the class definitions into C++ classes, and a small compiler is being written for the scripting language.

5 Conclusion

Efficient and scalable integration of independently developed parallel software components into a single application requires a component architecture that supports “in-process” components. We have developed a component architecture based on Converse, an interoperable parallel runtime system that supports message-driven execution.

We described an interface model and a scripting language to make the application composition and connectivity among components explicit. We are currently implementing this interface model and the scripting language, and carrying out optimizations enabled by them on top of the Charm++ language.

References

1. Common component architecture forum. See `http://www.acl.lanl.gov/cca-forum`.
2. Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, pages 115–124, Redondo Beach, California, August 1999.
3. I. Foster and C. Kesselman (Eds). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
4. L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.
5. L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
6. Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.
7. Richard Monson-Haefel. *Enterprise Javabeans*. O'Reilly and Associates, 2000.
8. Alan Pope. *The Corba Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1998.
9. Dale Rogerson. *Inside COM*. Microsoft Press, 1997.
10. Joshua Yelon. *Static Networks Of Objects As A Tool For Parallel Programming*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1999.