Modular Heterogeneous System Development: A Critical Analysis of Java

Gul A. Agha, Mark Astley, Jamil A. Sheikh, and Carlos Varela
Department of Computer Science
Univ. of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Phone: (217) 244-3087
Email: agha@cs.uiuc.edu

Abstract

*Java supports heterogeneous applications by transforming a heterogeneous network of machines into a homogeneous network of Java virtual machines. This approach abstracts over many of the complications that arise from heterogeneity, providing a uniform API to all components of an application. However, for many applications heterogeneity is an intentional feature where components and resources are co-located for optimal performance. We argue that Java's API does not provide an effective means for building applications in such an environment. Specifically, we suggest improvements to Java's existing mechanisms for maintaining consistency (*e.g. `synchronized`*), and controlling resources (*e.g. *thread scheduling*). We also consider the recent addition of a CORBA API in JDK 1.2. We argue that while such an approach provides greater flexibility for heterogeneous applications, many key problems still exist from an architectural standpoint. Finally, we consider the future of Java as a foundation for component-based software in heterogeneous environments and suggest architectural abstractions which will prove key to the successful development of such systems. We drive the discussion with examples and suggestions from our own work on the Actor model of computation.*

1 Classifying Heterogeneity

Heterogeneous computing environments arise in practice for a number of different reasons; heterogeneity, however, generates the same basic set of problems: code is not portable, shared data may need to be converted, the utilization of certain resources may be restricted to specific nodes, and so on. Nonetheless, the solution for these problems depends heavily on the types of applications that are deployed in the heterogeneous environment. As an example, consider the following two instances of heterogeneity:

- **System Evolution:** Corporate computing environments are continually evolving as outdated systems are gradually replaced with newer, more powerful systems. However, although the hardware is constantly replaced, corporations are often dependent on monolithic applications that must continue to run correctly in the presence of new hardware.
- **Specialized Hardware:** Certain computing environments are intentionally designed to be heterogeneous in order to utilize specialized hardware. Numeric simulations, for example, may be executed on massively parallel systems while monitoring and analysis is performed on graphics-intensive workstations. As another example, servers with high availability requirements are placed on hardware with large pools of available resources whereas clients execute on low-end workstations designed for single users.

The solution for an evolving corporate system depending on existing software might involve the development of a common execution environment atop each physical node. Thus, as long as existing applications are written in terms of this uniform environment, they will continue to be usable as future improvements are made. On the other hand, specialized hardware might be handled using an environment in which customized objects, targeted for specific hardware, coordinate with one another through a common interface for interactions. Still other environments, may utilize a hybrid of these two solutions.

Many languages and programming environments exist for managing heterogeneous computing environments. The Java programming language is an example

which *directly* addresses the technical problems created by a heterogeneous environment. In the words of its designers [6]:

> Java is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume the minimum of system resources, can run on any hardware and software platform, and can be extended dynamically.

CORBA, COM and other Object Request Broker (ORB) based environments represent so called "middle-ware" solutions. That is, rather than address heterogeneity directly, these environments provide a mechanism for allowing interactions between applications executing in heterogeneous environments.

In general, we may characterize the Java approach as the transformation of a heterogeneous network of machines into a homogeneous network of Java virtual machines. Java makes no effort to abstract over network features or cater to highly-optimized (but non-portable) implementations. However, Java does greatly simplify network access and provides a native method interface as a loop-hole for incorporating non-Java code. On the other hand, ORB-based systems make little or no effort to transform heterogeneous systems into homogeneous ones. Instead, ORBs solve the problem of interactions between heterogeneous environments. While such a solution limits mobility, applications may directly access high-performance implementations executing on dedicated hardware.

Both the Java and ORB-based solution have their merits. However, we argue that in order for Java to become "the answer" for programming heterogeneous computing systems, it must incorporate many of the features already present in ORBs. In particular, to answer the challenge of high-performance systems, Java must make local, optimized servers more available to Java clients. Currently, there are joint efforts between Sun and OSF to link CORBA and Java for precisely this reason [11]. However, we believe that while Java should be more ORB-like, it should also overcome many of the weaknesses of existing ORBs such as the inability to customize interactions between ORB-served objects. Moreover, to effectively support concurrency and distribution, we claim that Java requires more powerful constructs for controlling synchronization and coordination between distributed entities. We find existing Java synchronization (e.g. the `synchronized` keyword) to be too low-level and unsuitable for distributed needs. The lack of control over resource management tasks such as thread scheduling is also undesirable.

We envision Java as evolving to support distributed *collections* of objects executing over heterogeneous computing environments. In such an environment, application developers may specify services consisting of (possibly) distributed collections of Java and native objects. Services would be composed with policies which manage both interactions as well as deployment. These policies would encapsulate many of the solutions currently employed for heterogeneous environments: protocols which marshal arguments, routing mechanisms which link client requests to optimized objects executing on custom hardware, and so on.

In the next section, we discuss some weaknesses of the current version of Java as well as potential solutions. In Section 3, we describe features of ORB-based models which we believe should be incorporated into Java. In addition, we propose solutions for a Java-ORB system which overcomes many of the current weaknesses of the ORB-based model. In Section 4, we present a future vision of Java as a tool for implementing large grain coordination and management for heterogeneous applications. We describe lessons learned from our research in Actor [2] systems and propose several abstractions to be incorporated in future Java developments. We present concluding remarks in Section 5.

2 Heterogeneity in Java

Software executing in a heterogeneous environment is naturally segmented into a collection of distributed, coordinating objects. As a result, desirable system features such as ease of management and high performance depend on the ability to specify error-free coordination mechanisms which exploit available concurrency. Java uses a passive object model in which threads and objects are separate entities. As a result, Java objects serve as surrogates for thread coordination and do not abstract over a unit of concurrency. We view this relationship between Java objects and threads to be a serious limiting factor in the utility of Java for heterogeneous systems. Specifically, while multiple threads may be active in a Java object, Java only provides the low-level `synchronized` keyword for controlling object state, and lacks higher-level linguistic mechanisms for more carefully characterizing the conditions under which object methods may be invoked. Java programmers often overuse `synchronized` and deadlock is a common bug in multi-threaded Java programs.

Java's passive object model also limits mechanisms for thread interaction. In particular, threads ex-

change data through objects using either polling or `wait\notify` pairs to coordinate the exchange. In decoupled environments, where asynchronous or event-based communication yield better performance, Java programmers must build their own libraries which implement asynchronous messaging in terms of these primitive thread interaction mechanisms. Active objects, on the other hand, greatly simplify such coordination and are a natural atomic unit for system building, but no such alternative is available in the current version of Java.

Finally, we find Java's position on thread scheduling to be inadequate. While it is reasonable to not *require* applications to use fairly scheduled threads, we believe that system builders should have the *option* of selecting fair scheduling if necessary. The lack of fair threads is a particularly devious source of race conditions which makes debugging multi-threaded applications all the more difficult.

In the remainder of this section, we elaborate on each of these criticisms and describe potential solutions.

2.1 Linguistic Support for Synchronization

Synchronization in Java is necessary to protect state properties associated with objects. For example, the standard class *java.util.Hashtable* defines a synchronized *put* method for adding key-value pairs, and a synchronized *get* method for hashing keys. Both methods are synchronized to avoid corrupting state when methods are simultaneously invoked by separate threads. This mechanism works well for classes like *Hashtable* because methods in these classes have relatively simple behavior and do not participate in complex interactions with other classes.

A side-effect of the convenience and simplicity of `synchronized`, however, is that it tends to be overused by application programmers: when software developers are not certain as to the context in which a method may be called, a *rule of thumb* is to make it `synchronized`. This approach guarantees safety in Java's passive object model, but does not guarantee liveness and is a common source of deadlock. Typically, such deadlocks result because of interactions between classes with synchronized methods. For example, consider the threads `t1` and `t2` in Figure 1. The thread `t1` executes the synchronized method `m` which attempts to invoke the synchronized method `n` in class `B`. Similarly, the thread `t2` executes the synchronized method `n` which attempts to invoke the synchronized method `m` in class `A`. In a trace in which both threads first acquire their local locks, this simple example results in a deadlock.

```
class A implements Runnable{
  B b;
  synchronized void m() {
    ...b.n();...
  }
  public void run() { m(); }
}

class B implements Runnable{
  A a;
  synchronized void n() {
    ...a.m();...
  }
  public void run() { n(); }
}

class Test {
  public static void main(String[] args){
    A a = new A();
    B b = new B();
    a.b = b;
    b.a = a;
    Thread t1 = new Thread(a).start();
    Thread t2 = new Thread(b).start();
  }
}
```

Figure 1: A simple example of thread interactions which may result in deadlock.

We view the `synchronized` keyword as too low-level for effective use by application developers. Specifically, requiring developers to implement sophisticated synchronization constraints in terms of low-level primitives is error prone and difficult to debug. Synchronizers [4, 3] are linguistic abstractions which describe synchronization constraints over collections of actors (see Figure 2). In particular, synchronizers allow the specification of *message patterns* which are associated with rules that enable or disable methods on actors. Synchronizers may also have state and predicates may be defined which use state in order to enable or disable methods.

Note that synchronizers are much more abstract than the low-level synchronization support provided in Java. Synchronizers may be placed on individual actors as well as overlapping collections of actors. Moreover, separating synchronization into a distinct linguistic abstraction, rather than embedding it in class

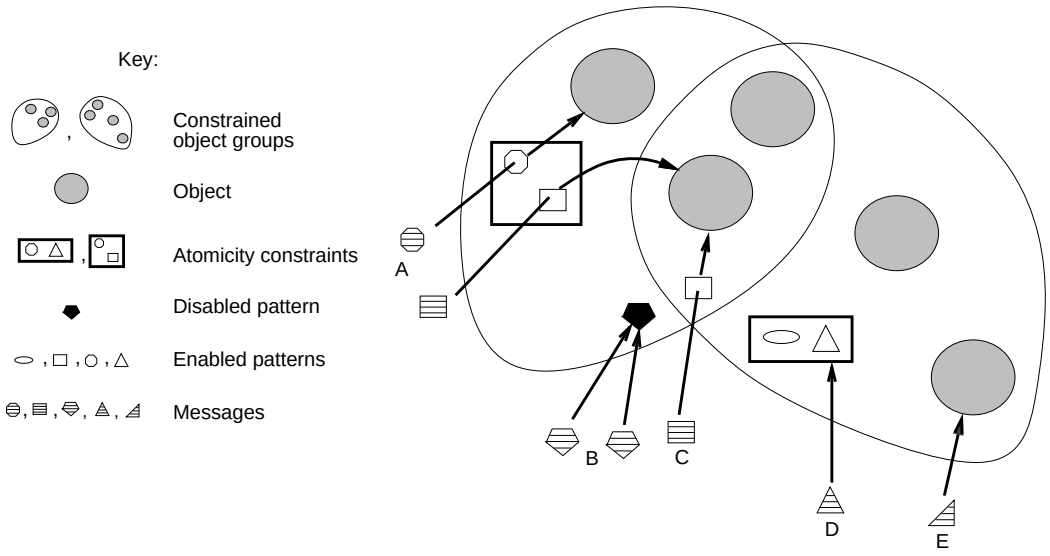

Figure 2: Synchronization constraints over a collection of actors.

definitions, allows constraints to be reused over different classes. As a simple example of how synchronizers may be specified linguistically, consider two resource managers, `adm1` and `adm2`, which distribute resources to clients. We wish to place a bound on the total number of resources allocated collectively by both managers. This can be achieved by defining the synchronizer given in Figure 3. The field `max` determines the total number of resources allocated by both managers.

We believe that heterogeneous environments, in which a wide variety of synchronization properties will be required, argue for an approach similar to synchronizers rather than the current Java solution of embedding low-level synchronization within classes.

2.2 Flexible Interactions

Distributed, heterogeneous systems require the ability to asynchronously participate in interactions in order to take advantage of available local concurrency. Because Java uses a passive object model, threads on a single virtual machine may interact either by polling on shared objects, or using `wait\notify`. Although these heavily synchronized methods of interaction are the most common in Java applications, asynchronous interactions may be implemented by spawning extra threads to handle interactions (see Figure 4).

As in the case of synchronization discussed in the last section, requiring the application developer to explicitly code such interaction mechanisms is prone to error. Asynchronous interactions are an important *basic* service that we believe should be standard in a heterogeneous programming environment. Thus, we argue for higher-level linguistic support in Java which provides such interaction mechanisms.

We believe that asynchronous interactions are best supported by an active object model such as that provided by actors. In such a model, method invocations are buffered in a *mailbox* and handled in a serialized fashion by a dedicated *master* thread. Active objects are thus a natural unit of concurrency and synchronization. Moreover, such objects need not be strictly serialized: intra-object concurrency may be added by allowing the master thread to spawn new threads which access specific internal methods. This form of intra-object concurrency differs from that in Java in that the master thread controls the conditions under which multiple methods may be active, rather than allowing arbitrary threads to execute in an object.

2.3 Resource Control

A final concern with using Java to develop heterogeneous systems is the lack of effective Java support for controlling system resources. A particular example is the ability of application programmers to control thread scheduling. While the Java language specification [5] encourages language implementors to write fair schedulers, this rule is not enforced. Hence, different environments may provide different schedulers emphasizing particular applications. A common solution is

```
AllocationPolicy(adm1,adm2,max)
    { init prev := 0

      prev >= max disables (adm1.request or adm2.request),
      (adm1.request or adm2.request) updates prev := prev + 1,
      (adm1.release or adm2.release) updates prev := prev - 1
    }
```

Figure 3: A Synchronizer that enforces collective bound on allocated resources.

```
class C {
  void m(){...}
  void am(){
    Runnable r = new Runnable {
      public void run(){
        m();
      }
    }
    new Thread(r).start();
    //  Code to continue executing
    //  after asynchronous method call
  }
}
```

Figure 4: A Java class which uses separate threads to handle interactions and execute local behavior.

to favor threads which are responsible for maintaining graphical user interfaces. However, while such an approach may be feasible for certain applications, other applications may fail as a result. Unfortunately, Java provides no mechanism for selecting features of the scheduler, leaving the application developer with the task of implementing custom scheduling if needed.

One possible solution is to include standardized thread scheduling libraries which may be invoked by applications desiring more control over scheduling. However, a user-level approach may not apply to certain critical threads in a system. For example, Java's RMI [12] package handles remote invocations using a separate, non-user controlled thread which invokes methods on user-defined objects. Because this thread is not under user control (and hence not subject to a user-level scheduling solution), unexpected pre-emption and deadlock may result[1]. As a specific solution, we favor the inclusion of lower-level policy selection which allows application developers to specify their scheduling needs. At a more general level, application developers should be able to specify abstract policies which govern more general classes of resources (see Section 4).

[1]It is possible to "hack" around this problem by modifying the RMI-created thread's properties once within a user-defined method. However, this may have unexpected side-effects since the thread was created for use by RMI.

3 Object Request Brokers

As of JDK 1.2, Java will incorporate an interface to the Common Object Request Broker Architecture (CORBA). The inclusion of ORB-based technology in Java indicates the widespread acceptance of Java as a platform for distributed computing, as well as the acceptance of CORBA as an appropriate technology for building component-based systems. In considering this recent combination of technologies, it is interesting to compare the Java Transaction Services (JTS) to the Object Transaction Services (OTS) used in CORBA. These two services are used to manage issues which arise in handling interactions between distributed objects. For example, marshaling data types, handling remote references, etc.

The design decisions evident in the JTS and OTS are a symptom of the relevant strengths and weaknesses of Java and CORBA, and attempt to combine the best of both worlds in a single package. Both Java and CORBA have their strong points and both have been used to develop successful applications. As discussed in the introduction, Java is a rich language with many features designed to simplify programming in heterogeneous environments. However, Java does not provide extensive support for matching clients to servers based on a service description. CORBA, on the other hand, facilitates service location and interaction in a heterogeneous environment. In particular, CORBA allows service description in terms of an Interface Definition Language (IDL), and provides mechanisms for locating services based on IDL descriptions. IDL specifications are an abstract specification of service which are independent of low-level system features such as resource requirements, procedural behavior, control-flow and so-on. Unfortunately, CORBA limits the types of data that can be commu-

nicated in interactions, and prohibits the passing of object references which is required to take advantage Java's more powerful features. The combination of Java and CORBA is intended to alleviate many (but not all) of these problems, while carrying over as much functionality as possible from existing remote interaction mechanisms in Java and CORBA.

In the remainder of this section we discuss some of the motivation behind combining ORB-based technology with Java. While we favor this marriage of technologies, we argue that such a combination still lacks many important features necessary for effective heterogeneous programming. Specifically, CORBA and its relatives still provide a closed model for interactions, and force application developers to embed interaction protocols within client and server code. Encryption protocols, for example, can not be defined as a property of the connection. Instead, both the client and server must embed appropriate endpoints for the protocol within the existing code for handling interactions. We propose an alternative approach in which these types of protocols may be factored out of application code and specified independently on a per-interaction basis.

3.1 Why Add ORB Technology?

Providing services among a collection of objects accessible via a shared network requires a common interaction layer which links clients, which request services, to servers, which implement those services. CORBA and related ORBs enable the construction and integration of distributed applications by providing such a layer. In particular, CORBA allows the dynamic placement and update of objects which implement services in a distributed, heterogeneous network. Moreover, these objects may be accessed using a common data exchange framework with many features critical to the development of heterogeneous systems. These features include:

- Multi-threading
- Debugging and Network Monitoring
- Connection Groups
- Synchronous and Asynchronous calls to servers
- Virtual Callbacks from the server
- Asynchronous operation
- Location Brokering for location transparency
- Naming Service †
- Event Service
- Life Cycle Service †
- Transaction Service †
- Concurrency Control Service
- Relationship Service †
- Query Service †
- Licensing Service
- Security Service †
- Object Trader Service †

Those items marked with a † indicate features that are present in the JTS as well as the OTS. A detailed description of each of these features is not within the scope of this paper. We refer the interested reader to [7] for more details.

In addition to the features described above, ORBs provide several other features which simplify system development. Among these are the ability to quickly design and implement larger object oriented systems, and a communication backplane with consistent semantics regardless of whether a system executes on a heterogeneous network or a single machine. However, as we discussed in the introduction, ORBs make no attempt to transform heterogeneous systems into homogeneous environments. As a result, although ORBs have been used for some time, it is only recently that issues such as load balancing, security, and transactions have received appreciable attention.

3.2 Other ORB-based Systems

CORBA is the most well-known ORB and is based on the Object Management Group's (OMG) Object Model. This model is backed by a large consortium of commercial system developers and hence has a significant role to play in the future of system development. However, although CORBA has achieved widespread success, several other systems have been developed which support a variety of object models (including CORBA).

The *Top-ORB* system from NCR will allow the connection of CORBA objects, Java Beans, DCOM objects and many other type of objects using the *Top End* framework as the underlying infrastructure. Top End is part of the Top End Service Interface Repository (TESIR) model designed by NCR for supporting access to legacy applications, and which defines a general object service mechanism [1]. NCR plans to launch the underlying infrastructure of Top-ORB in 1998.

The *Solaris NEO* system from Sun is similar to CORBA and designed around the same object model. JOE is another Sun product which provides for distributed client-server applications, and complies with the CORBA 2.0 standard. While supporting CORBA standards, both NEO and JOE also allow for the connectivity of Java applets to applications running on distributed servers. In particular, the object request broker used in JOE may be automatically downloaded

into web browsers, and used to connect Java applets to remote NEO objects. Another useful feature provided by JOE is an IDL compiler which generates Java classes from interface definitions of CORBA objects.

Finally, Java's *Remote Method Invocation* (RMI) provides for more primitive client-server functionality. In particular, RMI is not CORBA compliant, but does support interoperability among Java objects in distributed environments. However, RMI does not provide any explicit support for incorporating legacy (*i.e.* non-Java) objects. Such objects may only be included by adding a Java front-end which interacts with RMI.

3.3 Adding ORB Functionality to Java

The current release of Java supports RMI and JavaBeans and hence does not allow for integration with CORBA-like models of object systems. Despite the various other benefits of ORBs, however, ORB vendors including the OMG and Sun have placed technical emphasis on incorporating several object models within a single framework, rather than attempting to increase the functionality of ORB models as a whole. This trend is expected to continue as no single standard (*i.e.* object model) has been adopted for ORB-based systems.

Thus, while the next release of Java will provide greater flexibility in terms of incorporating existing object models, several key problems with ORBs are inherited with the new approach. Specifically, remote procedure call (RPC) remains as the primary mechanism for building distributed interactions. As with the `synchronized` keyword discussed in the previous section, RPC is often abused in the context of distributed interactions and leads to heavily synchronized, and therefore poorly performing applications. We have already argued for asynchronous modes of interaction in the previous section. More importantly, however, ORBs currently do not provide a mechanism for flexible specification of connection properties. Applications requiring specific policies must either use a custom coded ORB implementation, or embed policy code within clients and servers. Both approaches are error-prone and make systems less modular.

Our research in Actors has lead to a novel approach for separating communication policies from application code. Communicators [10] rely on a meta-architecture to abstract over the communication behavior of Actors. In particular, actor interactions are represented abstractly in terms of three operations (see Figure 5):

- A *transmit* operation is invoked when an actor attempts to send a message;
- A *deliver* operation is invoked when the system receives a message on behalf of an actor; and,
- A *dispatch* operation is invoked when an actor is ready to process the next message.

The communication behavior of actors are customized by installing meta-actors which redefine one or more of the basic actor operations. This technique may be used to implement a wide variety of protocols. For example, consider a simple protocol for implementing a FIFO channel between two actors. Figure 6 gives a Communicator specification which defines such a protocol.

Communicators effectively separate protocol code from application code allowing system designers to pick and choose the protocols necessary for interactions, without complicating code development by changing clients and servers. We believe that an ORB-Java combination must include similar abstractions in order to be an effective tool in distributed, heterogeneous environments.

4 Component-Based Systems

In the previous sections we have discussed the near-term limitations of Java as a tool for building heterogeneous systems. In this section, we present a future vision of software for heterogeneous systems and the features we expect to be incorporated into Java to make it a viable development environment.

The next logical step for component-based heterogeneous system development is higher-levels of granularity in which distributed *collections* of objects are managed as individual components and services. Currently, this is an active area of research in the software architecture community in which such systems are viewed as consisting of a collection of components, which encapsulate computation, and a collection of connectors, which describe how components are integrated into the architecture [9]. This separation of design concerns favors a compositional approach to system design; a methodology which is particularly important when specifying architectures for heterogeneous distributed systems. Heterogeneity, failure, and the potential for unpredictable interactions yield evolving systems which require complex management policies. Allowing architectural specifications in which these policies are separated into abstract connectors has clear advantages for system design, verification and reuse.

Note that policies for managing such systems (*e.g.* reliability protocols, load balance and placement, security constraints, coordination, etc.) not only assert

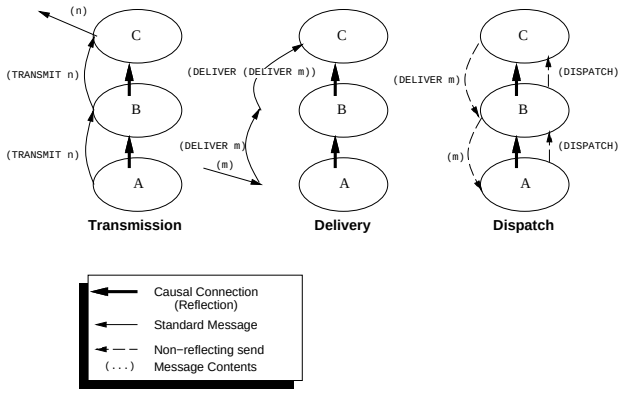

Figure 5: Customizing the communication operations of an actor. Actors B and C are meta-level customizations of actor A. Each operation of A results in an operation on B and/or C.

properties on the connections between component interfaces, but must also enforce constraints on how resources are allocated to components. For example, a reliable server may be developed by adding a backup to an existing server and installing an instance of the primary backup protocol. In addition to recording interactions at the backup, the primary backup protocol must also ensure that the backup and server use separate, failure-independent resources (*e.g.* they must execute on separate processors). The resulting collection of policies is quite different from those required to manage interactions in, for example, ORB-based models, and therefore requires new abstractions with the goal of fitting components to architectural *contexts*, rather than defining interconnections between component interfaces. Specifically, component interfaces abstract over functionality but not resource management. In the remainder of this section, we elaborate further on this point, and describe recent research using the Actor model which proposes a solution to these problems.

4.1 Extending Component Interfaces and Architectural Policies

Current notions of component interfaces are based on a functional representation of the services provided by a component. This abstraction is a natural extension of the object model. However, when placing an object in a heterogeneous architecture, this model fails to describe many important features such as:

- **Locality properties:** The distribution and communication behavior of internal computational elements.
- **Resource usage patterns:** Distinctions such as computation bound versus I/O bound elements, degree of concurrency, hardware dependencies, and the resources corresponding to critical and transient state.
- **Inter-level dependencies:** The relationships between management policies at various levels of granularity.

In general, components should provide a comprehensive model of *architectural context*: the relationships between component behavior and architectural features such as those described above. A natural solution would be to extend current interfaces with additional functional entry points for selecting, for instance, placement policies, reliability features (*e.g.* fault-tolerance protocols), and so on. However, such an approach complicates component code by embedding orthogonal, context-specific concerns. The more preferable approach would be to design generalized components which may be customized to particular architectural contexts. Connectors would encapsulate these customizations, preserving compositional system development. Note that such a solution solves both sides of the heterogeneity problem: general components may be adapted to new environments by composing them with appropriate policies, while hardware-sensitive components may be used in a general context by adding policies which guarantee appropriate resource allocation to this class of components.

A key challenge for specifying more general, resource-based policies is the problem of composing policies while respecting object-integrity. The connection-oriented customizations we described in

```
protocol FIFO_channel {

  Installation asymmetric;
  Isolated-Interaction;

  role local-client { }


  role client {
   int tag;


   method init() {
     tag = 0;
   }


   method out(msg m) {
     server.tagged_in(tag, m);
     tag = tag + 1;
  } }

  role server {
   MsgBag delays;
   int intag;

   method init() {
     intag = 0;
   }

   method tagged_in(int t,msg m) {
     msg next;


     if (t == intag) {
       next = m;
       while (next) {
          deliver next;
        intag = intag + 1;
        next = delays.get(intag);
       }
     } else delays.put(t,m);
} }
```

Figure 6: The Communicator specification for a FIFO channel between actors.

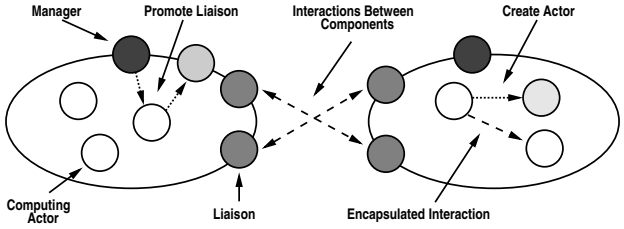

Figure 7: Components are an encapsulated collection of actors. *Liaisons* are a subset of the collection which may participate in external interactions. The *manager* negotiates new connections and promotes actors to liaisons.

Section 3 avoid this problem because they operate strictly on component interfaces. However, specifying policies which control the allocation of resources may require access to component internals. Thus, abstractions which support these policies must be carefully designed to avoid exposing object features which are not normally exported through an interface. We describe our model for such policy composition in the next section.

4.2 Specifying Policies for Connection and Context

In order to reason about architectural context, we require a model of component computation which represents component behavior in terms of interactions with a set of default system services. Relative to computational behavior, the semantics of these services will remain the same regardless of architectural context. However, the semantics of the *implementation* of these services will vary as components are placed in different architectures. This distinction allows compositional development, in which generalized components are fitted to particular architectures, not by changing their computational behavior (which would break encapsulation), but by customizing the interactions between components and the particular implementation of underlying services.

We build on the actor model extensions described in previous sections by modeling components as encapsulated collections of actors in which a distinguished subset, called *liaisons*, are used for interactions with other components (see Figure 7). Interactions between liaisons in different components define component connection properties. In particular, by customizing these interactions, specific protocols may be enforced. Moreover, the architectural context of a component is represented by the service invocation behavior of internal (*i.e.* non-liaison) actors. Thus, the

collective behavior of a component relative to architectural features is captured by the interactions through its liaisons and the resource access patterns of its internal actors. Both behaviors are represented uniformly in terms of invocations of the basic actor primitives, providing a clean representation for architectural customization.

Components are customized by designing policies which define how components access a collection of basic system services (see Figure 8). Liaisons are the only externally visible elements of a component. Thus, connectors which specify protocols between components are naturally represented in terms of customizations applied to individual liaisons. However, connectors which specify resource management policies are more challenging because they customize internal component elements. In particular, we would like to specify arbitrary customizations of internal actors while respecting the encapsulation properties of a component. To this end, policies are constructed from two types of meta-level behavior:

- **Roles:** A role is a specific customization applied to one or more liaisons. Roles are used to implement protocols on connections between components. For example, an encryption protocol may be implemented by customizing the "send" behavior of one liaison (*e.g.* to encrypt outgoing messages) and the "receive" behavior of another (*e.g.* to decrypt incoming messages). Roles are installed explicitly on a set of liaisons.

- **Context:** A context is a single meta-level behavior which customizes *all* actors within a component and is automatically installed on any dynamically created actors. Contexts are used to manage the allocation of resources. For example, a local load balancing strategy may be implemented by customizing the "create" behavior of all actors within a component.

Because roles are installed on liaisons, there is no danger of compromising object integrity as liaisons are already exported by components. Contexts, on the other hand, must be installed on internal component members. However, the structure of the meta-level architecture and the encapsulation properties of components prohibit contexts from destroying internal component elements or exporting non-liaison addresses. Specifically, a meta-level customization may only modify actor interactions with system services, and may not change the internal behavior of an actor. Similarly, regardless of meta-level customizations, managers control component namespaces and determine which actors may participate in external interactions.

A remaining open issue is the question of whether or not policies are composable (both with components or with other policies). In particular, as component compositions encompass larger systems, there is a greater potential for detrimental interactions between existing policies on sub-components. We are currently in the process of extending our abstractions to model and reason about such interference.

5 Conclusion

We have discussed the Java approach to solving the heterogeneity problem and identified several areas for improvement in the current release of Java. In particular, we claim that relative to the needs of heterogeneous computing, current synchronization mechanisms in Java are too low-level and hence prone to misuse. Similarly, we argue that Java does not provide enough control over resource usage, particularly threads, and that existing interaction mechanisms between Java tasks (*i.e.* threads) are too heavily synchronized and lack an alternative communication medium such as asynchronous messaging. We presented several examples from our own work on Actors which demonstrate the utility of more powerful synchronization constructs.

We have considered the recent marriage between Java-based computing and existing CORBA-like systems in the context of heterogeneous computing. While incorporating ORB-based technology into Java is a significant step, we argue that ORBs are still too closed with respect to interaction policies. We presented several examples of policies which may be factored out of object code and applied to the endpoints which implement the connection itself. Such an approach simplifies debugging and makes components more reusable. Moreover, system designers may select only those policies appropriate to their environment, rather than having to pay the price of layering policies atop an existing interaction mechanism.

Finally, we discussed the future of Java in the realm of component-based software development and described our preliminary work on policies for resource management in a distributed, heterogeneous setting. We model components as hierarchical collections of actors with interfaces defined as dynamic sets of actors called liaisons. Components are customized according to the needs of a particular environment by accessing an open implementation of the interface between actors and their underlying system services. We factor customizations into two categories: *roles* are ex-

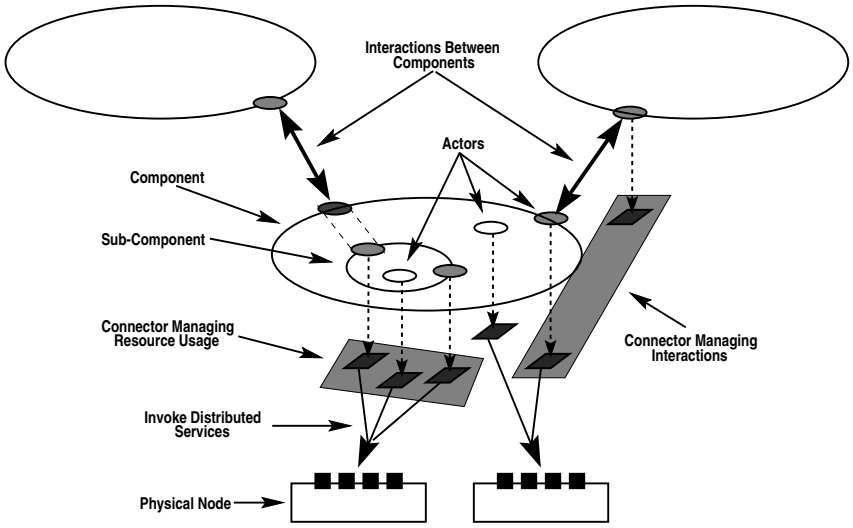

Figure 8: Components are customized by policies which redefine interactions between liaisons, and the invocation of basic system services.

plicit customizations of liaisons, while *contexts* are implicit customizations of all actors within a component. Roles allow the enforcement of interaction policies over connections between components. Contexts support component-wide resource management and coordination. Composition at the meta-level allows multiple customizations to be applied to a single component.

Despite our reservations, we believe that Java is an important step towards developing appropriate tools for building heterogeneous systems. In particular, we have used Java as the development environment for a prototype actor system which incorporates many of the abstractions described above [8].

Acknowledgments

We thank past and present members of the Open Systems Laboratory who aided in this research. The research described has been made possible in part by support from the National Science Foundation (NSF CCR-9619522) and the Air Force Office of Science Research (AF BASAR 2689 ANTIC).

References

[1] YOU'RE THE TOP: A research note from the standish group. Available at http://www.ncr.com/product/integrated /analyst_reports/standish-your/index.htm.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[4] S. Frølund and G. Agha. *Object-Based Models and Languages for Concurrent Systems*, chapter Abstracting Interactions Based on Message Sets. Lecture Notes in Computer Science. Springer-Verlag, 1995.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[6] J. Gosling and H. McGilton. The java language environment: A white paper. Technical report, Sun Microsystems Inc., May 1996. Available at `http://www.javasoft.com/docs /white/index.html`.

[7] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. Available at http://www.omg.org/corba.

[8] Open Systems Lab. The actor foundry: A java-based actor programming environment. Available for download at http://www-osl.cs.uiuc.edu/~astley/foundry.html.

[9] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support

them. *IEEE Transactions on Software Engineering*, April 1995.

[10] D. C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[11] The Java Team. JDK 1.2 Beta specification. Available at http://developer.javasoft.com/developer/earlyAccess/jdk12.

[12] The Java Team. Rmi specification. Available at ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.ps.

Biography

Gul Agha is director of the Open Systems Laboratory at the University of Illinois at Urbana-Champaign and an associate professor in the Department of Computer Science. He serves as editor-in-chief of IEEE Concurrency, associate editor of ACM Computing Surveys, and associate editor of Theory and Practice of Object Systems. His research interests include models, languages and tools for parallel computing and open distributed systems. Agha has received the Incentives for Excellence Award from Digital Equipment Corporation in 1989, and he was named a Naval Young Investigator by the US Office of Naval Research in 1990. He received an MS and PhD in computer and communication science, and an MA in psychology, all from the University of Michigan, Ann Arbor, and a BS in an interdisciplinary program from the California Institute of Technology.

Mark Astley is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. His research interests include visualization, and architecture description languages and environments for open distributed systems. He received an MS in 1996 in computer science from the University of Illinois at Urbana-Champaign, and a BS in computer science and BS in mathematics, magna cum laude with honors, in 1993 from the University of Alaska - Fairbanks.

Jamil A. Sheikh is a visiting scholar in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign and doctoral candidate at Quaid-e-Azam University - Islamabad. Previously he served as a Computer Systems Engineer at the Department of Nuclear Power. His research interests include real-time distributed computing, concurrent object-oriented programming and high-speed computer networks. He received an MS in 1994 in Nuclear, Computer & Control Engineering from Quaid-e-Azam University, and a BS in Computer Systems Engineering in 1992 from NED University - Karachi.

Carlos Varela is a doctoral candidate and research assistant in the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. His research interests include web-based applications and concurrent programming in Java. He received an MS in 1997 in computer science, and a BS in 1992 in computer science with honors, both from the University of Illinois at Urbana-Champaign.