Abstraction and Modularity Mechanisms for Concurrent Computing

Gul Agha, Svend Frølund, WooYoung Kim,

Rajendra Panwar, Anna Patterson, and Daniel Sturman

Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: `<agha|frolund|wooyoung|raju|annap|sturman>@cs.uiuc.edu`

1 Introduction

The transition from sequential to parallel and distributed computing has been widely accepted as a major paradigm shift occurring in Computer Science. It has been observed that the sequential programming model is inadequate for parallel and distributed computing. At a conceptual level, one can compare the fundamental intuition between sequential and concurrent computing as follows:

- The sequential programming model essentially defines a computation as a sequence of instructions which manipulate a global store. The standard abstraction mechanism for sequential programs is procedures. Procedures glue a sequence of instructions; they allow genericity by parameterization and reuse by substitution. While this provides a good building block for much of sequential programming, it is an unsatisfactory model for parallel and distributed computing because it does not provide a communication model and it is not a meaningful abstraction of coordination between concurrent components.

- In a *parallel computation* some actions overlap in time; by implication these events must be *distributed* in space. *Concurrency* refers to the *potentially* parallel execution of programs. In a concurrent computation, the execution of some parts of a program may be sequential, or it may be parallel. Since concurrent programs specify a partial order of actions, it provides us with the flexibility to interleave the execution of commands in a program, or to run them in parallel. Therefore, some of the details of the order of execution are left unspecified. We can instead concentrate on conceptual issues without necessarily being concerned with the particular order of execution that may be the result of the quirks of a given system.

Part of the complexity of reasoning about concurrent programs results from the fact that partial orders allow considerable indeterminacy in execution. In other words, there are many potential execution paths. Furthermore, concurrent programs are complicated by the fact that there are a number of different kinds of design concerns, such as locality and synchronization, that are transparent in sequential execution environments. To simplify the construction of concurrent systems, concurrent abstractions must support a separation of design concerns by providing *modularity*.

The complexity of concurrent systems requires new *abstraction* methods to be developed. There are four important requirements for concurrency abstractions. First, the abstraction must allow specification of the complex organizational and coordination structures that are common in concurrent computing. Second, they must provide genericity and reuse of the coordination patterns, much as procedures do for sequential programming. Third, concurrency abstractions must simplify the task of programming by separating design concerns. And finally, the abstractions must allow efficient execution on concurrent architectures.

This paper describes a number of radical programming language concepts that support abstraction and provide modularity in concurrent systems. Specifically, the constructs we propose allow abstract and modular specification of coordination patterns, temporal ordering, resource management, and dependability protocols. In particular, specifications using these constructs are generic and reusable. The next four sections develop our methodology and apply it to a number of problems as follows:

Actors: we describe the Actor model of concurrent computation. The Actor model provides the basic building blocks for concurrent programming which may be used to build a wide variety of computational structures.

Communication abstractions : three communication abstractions are discussed. These are call/return communication, pattern-directed message passing, and constraints on reception. To provide a concrete representation, we show how call/return communication is transformed to primitive actor message-passing.

Object oriented design: we discuss the use of classes, inheritance, and incremental modification of code.

Modular decomposition: we describe a set of abstractions and discuss how they may be used to factor out multi-actor coordination patterns, resource management strategies, and protocols for dependability.

2 Actors

The universe we live in is inherently parallel and distributed. This suggests that the natural language constructs we use to describe the world may also be useful for modeling computational systems. It can be reasonably asserted that the most important concept we use to model the world is categorizing it in terms of objects. In fact, the first elements of natural language children learn are names of objects.

Computational objects encapsulate a state and an expected behavior. Furthermore, objects provide an interface defined in terms of the names of procedures that are visible. These procedures, called *methods*, manipulate the local state of the object when invoked. In particular, this implies that representations which support the same functionality may be interchanged transparently. This is an important software engineering advantage which has proved its utility in sequential object-based programming.

Traditional object-oriented programming is limited by a mind set which views programming as a sequence of actions. In particular, this mode confounds the natural autonomy and concurrency of objects: sequential object-oriented languages allow only one object to be active at a time. An object's behavior is viewed as a sequence of actions, and this sequence is blocked by invoking a method in another object. This is a rather contrived view: it is more natural to view objects as (virtual) computational agents which may compute *concurrently*.

The *Actor* model unifies objects and concurrency. The model's building blocks can be described and justified in fairly intuitive terms. Actors are autonomous and concurrently executing objects which execute asynchronously (i.e., at their own rate). Actors may send each other messages. Since actors are conceptually distributed in space, communication between them is asynchronous. Asynchronous communication preserves the available potential for parallel activity: an actor sending a message asynchronously need not block until the recipient is ready to receive (or process) a message. If a model requires a sender to block, it reduces the concurrency which may be available.

In response to receiving a message, an actor may take the following sorts of actions:

send: asynchronously send a message to a specified actor.

create: create an actor with the specified behavior.

become: specify a new behavior (local state) to be used by the actor to respond to the next message it processes.

The *message send* primitive is the asynchronous analog of procedure invocation. It is the basic communication primitive, causing a message to be put in an actor's mailbox (*mail queue*). To send a message, the identity (*mail address*) of the target of a communication needs to be specified. Finally, note that although the arrival order of messages is nondeterministic, every message sent to an actor is guaranteed to be eventually delivered.

The *become* primitive gives actors a history-sensitive behavior necessary for shared, mutable data objects. This is in contrast to a purely functional programming model. The *create* primitive is to concurrent programming what procedure abstraction is to sequential programming. Newly created actors are autonomous and have a unique mail address. Furthermore, create dynamically extends computational space, it thus subsumes the functionality of `new` in Pascal or `malloc` in C. Actor primitives form a simple but powerful set upon which to build a wide range of higher-level abstractions and concurrent programming paradigms.

3 Communication Abstractions

Although point-to-point asynchronous message sending is the most efficient form of communication in a distributed system, concurrent languages must provide a number of communication abstractions to simplify the task of programming. Programmers using parallel or distributed computing need to understand the advantages and limitations of different communication abstractions. We describe three basic communication abstractions, namely call/return communication, pattern-directed communication, and constrained reception.

3.1 Call/Return Communication

In call/return communication, an object invokes a number of other objects and waits for them to return a value before continuing execution. A standard mechanism for call/return communication in concurrent programming is *remote procedure call*: a procedure calls another procedure at a remote node and waits for the result. The result is returned to the point where the call is made. RPC extends the sequential procedure call model where procedure calls follow a stack discipline which can be efficiently implemented on sequential processors. In case of high-level actor languages, concurrent RPC-style calls allow a simple expression of functional parallelism. In actor languages, whether two actors are on the same node or on different nodes is transparent to the application code.

Blocking a sender in a call/return communication is generally not desirable: if the actor invoked is on a different node, available concurrency may be unnecessarily lost. If the sender "holds" the processor while *busy waiting* for results, processor time is wasted. Otherwise, extra context switching is needed to change the executing actor from the sender to another actor.

Whenever feasible, we allow the calling actor to continue computation as soon as it has asynchronously *sent* a request. In order to support ease of programming without incurring an unnecessary performance penalty, we transform a program containing a call/return communication to a semantically equivalent one containing asynchronous message sends only. The transformations used preserve the maximal concurrency in a program. Optimizing this form of communication by using a concurrent analog of continuation passing style program transformation avoids incurring unnecessary costs (see inset).

INSET: Program Transformations for Call/Return Communication

We use one of two transformations on call/return communication [9]. First, if the response of a sending actor to the next message is not dependent on the results from a call/return communication, the program is transformed by changing the calls to asynchronous sends and creating a *join continuation* actor [1]. The join continuation actor performs a part of computation of the original sender actor that is dependent on the results. Consider the following expression:

```
send B (v, C.request1(), D.request2())
```

`send` represents an asynchronous send. When executed, actor `C` and `D` receive messages `request1` and `request2`, respectively. Then, actor `B` is sent a message with results from actor `C` and `D` along with `v`. Figure 1 and 2 pictorially represent the execution of the program before and after the transformation, respectively.

Figure 1: Before the transformation creating join continuation

Figure 2: After the transformation creating join continuation

Second, if the response of an actor to the next message is partly determined by the results of the calls to other actors, we separate out the continuation as a method within the original actor. Note that no purpose would be served by creating a join continuation actor: the original sender cannot process other messages until a result is received. The continuation method is triggered by the results of the remote actor invocations. In order to guarantee consistency between state changes, the transformation creates a synchronization constraint (see next section) for the continuation method. This new constraint prevents other messages from being processed until the continuation method has been invoked.

3.2 Pattern-directed Communication

An advantage of point-to-point asynchronous communication mechanism is that it allows locality to be directly expressed and optimized. However, in some cases, it is sufficient to communicate with an arbitrary member of a group. If the recipient must name all potential receivers, the book-keeping involved can be cumbersome. Furthermore, a level of abstraction is lost. The use of pattern-directed communication allows an abstract specification of a group of potential recipients. Thus, the actual recipients may be transparently changed: none of clients needs to know the exact identities of potential receivers or to poll them to determine if they satisfy some pattern.

In the ActorSpace model, a communication model based on destination patterns is defined [2]. An *actorSpace* is a computationally passive container of actors which acts as a context for matching patterns. Note that actorSpaces may overlap; in particular, an actorSpace may be wholly contained in another. Patterns are matched against listed attributes of actors and actorSpaces that are *visible* in the actorSpace. Both visibility and attributes are dynamic. Messages may be sent to one or all members of a group defined by a pattern. An actor may send a message to a single (arbitrary) member of a group, or broadcast it to the entire group. In particular, broadcasting can be used to disseminate common protocols to an entire group.

ActorSpace provides a useful model for many distributed applications. For example, if an actorSpace of servers is defined, none of the clients need to know the exact identities of the potential servers or explicitly poll them to determine if particular ones are suitable. This provides an abstraction that allows replication of services, for example to enhance reliability or increase performance.

Linda [5] defines a communication abstraction similar to that of actorSpace; however, the semantics of Linda, unlike ActorSpace, require explicit read operations by recipients. This results in at least two significant differences. First, race conditions may occur as a result of concurrent access by different processes to a common space. Second, communication cannot be made secure against arbitrary readers – for example, there is no way for a sender to specify that a process with certain attributes may not consume a message (called a tuple in Linda). By contrast, in ActorSpace, the attributes of a message's potential recipient are determined by the sender.

3.3 Synchronization Constraints

In sequential programs, there is a single thread of control and the programmer must explicitly fix a calling sequence for all objects – essentially by calling them one at a time and passing the control to them. Generally a stack discipline is used and control returns to the calling object once the called object has finished executing. In fact, this calling discipline corresponds quite poorly to the nature of many computations. In concurrent systems, more distributed forms of synchronization supporting partial orders are needed.

Consider producer actors and consumer actors which communicate through a buffer actor. Since actors are autonomous and asynchronous, they do not know if a particular message they send may be meaningfully processed in the current state of a receiving actor. Therefore, it is possible for a producer actor to send a `put` request to a full buffer, or, for a consumer actor to send a `get` request to an empty buffer (Figure 3). Thus, it is necessary to specify *when* different computational objects may be invoked. Rather than *reject* messages which may not be processed in the current state we *defer* the request.

Figure 3: Producers and Consumers. The producer actor deposits a data item into the buffer by invoking `put` method defined in the buffer actor. The consumer actor retrieves a data item from the buffer by invoking `get` method of the buffer actor. The producer actor may not invoke `put` method on a full buffer. And the consumer actor may not invoke `get` method on a empty buffer.

Our view is that constraints which limit invocations of a concurrent object – i.e., its *synchronization constraints* – should be a part of the actor's interface. A programmer may associate such constraints with each method. The separation of synchronization constraints frees the programmer from explicitly specifying code to manage messages received by an actor which it is not in a state to process. A message not satisfying the constraint is buffered until such time when the actor's state satisfies the constraint. Essentially, synchronization constraints provide an abstract representation of the reactive behavior of an actor. Using synchronization constraints, it is possible to reason about the effect of composing actors.

Actor programs, unlike iteration in sequential programming languages, do not use *global* loops for sequencing actions; the usual semantics of such loops implies that an iteration should be completed before the next one may be initiated. Instead, actors support message driven programming. Synchronization constraints may be used to *locally* ensure consistent sequencing of iterations. By not creating unnecessary data dependencies, message driven programming maintains the maximal concurrency available in an algorithm (see inset).

INSET: Overlapping Communication and Computation

Actor programming naturally leads to efficient parallel execution in a number cases. We illustrate this by an iterative matrix algorithm for the Cholesky Decomposition (CD) of a dense matrix ([4]). Specifically, the matrix is represented by a collection of actors which execute different iterations in response to messages they receive. Since the arrival order of messages is indeterminate, local synchronization constraints are used to ensure that the iterations are executed in the correct order.

Table 1 compares the results of an implementation of CD algorithm which pipelines the execution of iterations with an implementation which completes the execution of one iteration before starting the next. Note that the pipelining naturally follows from the fact that control is distributed between actors.

Matrix	32×32		64×64		128×128		256×256	
Nodes	*Seq*	*Pipe*	*Seq*	*Pipe*	*Seq*	*Pipe*	*Seq*	*Pipe*
1	.131	.123	.873	.857	6.45	6.41	49.6	49.6
2	.127	.111	.788	.760	5.75	5.68	44.1	43.9
4	.103	.080	.581	.518	3.93	3.76	29.8	29.3
8	.101	.048	.445	.308	2.53	2.18	17.5	16.6
16	.133	.029	.451	.173	1.88	1.20	10.7	8.9
32	.222	.019	.645	.099	1.98	.93	8.0	5.6

Table 1: Results from an implementation of the CD algorithm on an Intel iPSC/2. The columns *Seq* represent the implementation which completes the execution of one iteration before starting the execution of the next iteration. The columns *Pipe* show the values obtained by pipelining the execution of iterations. The times shown are in seconds.

4 Object-Oriented Design

A natural progression from naming individual objects is to name categories or classes of objects. Object-oriented design provides a hierarchical framework which naturally models the world using classification. Such classification allows us to build categories which provide shared attributes and functionality. By providing a tool for parsimony of representation, classification simplifies our ability to model the world.

In more concrete terms, a class may be thought of as a category of computational objects which can be specialized. A canonical example of a class is a `vehicle` which has certain attributes such as `position, velocity, occupants, weight,` etc., and allows certain method invocations to change some of its attributes. A member of this class is defined by specifying its initial attributes (state). Some of these attributes may never change if the class does not contain operations to change them.

A subclass may further specialize the operations of a superclass. For example, a `car` may have more specific attributes and functions as well as those of a `vehicle`. By allowing a `car` to *inherit* code from a `vehicle`, object-oriented languages support incremental refinement and code reuse. Depending on the object oriented language, a method may be extended or redefined in a subclass.

Actors, like any objects, may be organized into classes and include notions of inheritance. Note that because we specify synchronization constraints in a modular fashion, they may be inherited and incrementally modified. Specifically, a synchronization constraint may be further strengthened, weakened or overwritten in a subclass independently of whether the constrained methods are changed.

Alternately, actors may use forms of inheritance only to support method code reuse. Specifically, *delegation* is a variant of inheritance which allows the code of a prototype object to be reused. For example, it may not be meaningful to think of a stack as a kind of an array, but a stack may be defined by using the representation and operations of an array. A stack may delegate invocations of its methods to an array.

A good survey of research in object-oriented programming can be found in [12]. A description of concurrent object-oriented programming can be found in [1].

INSET: Incremental Modifications of Synchronization Constraints

Consider the example of producers and consumers which communicate through a buffer (see Figure 3). The buffer defers requests from consumers if it is empty. The code for such a buffer may be specified as follows:

```
class  Buffer
  var  first, count
  restrict  get()  with  (count > 0)
  init ()
    count := 0
    ...
  end
  method  put(x)
  ...
  method  get()
  ...
end
```

Now suppose we want to create a buffer which takes in requests as long as the cumulative size of the pending requests is smaller than its buffering capacity. If the buffer represents a fast cache accessed by a number of printers, it could be defined as an instance of the `sizedBuffer` class which extends the buffer class definitions as follows:

```
class sizedBuffer  inherits  Buffer
  var usedCapacity, totalCapacity
  restrict  put(x)  with  (x.size + usedCapacity <= totalCapacity)
  init (bufferCapacity)
    usedCapacity := 0
    totalCapacity := bufferCapacity
  end
  method  put(x)
    super.put(x);
    usedCapacity := usedCapacity + x.size;
  end
  method get()
  ...
end
```

5 Separating Design Concerns

We describe three mechanisms to develop modular and reusable components for concurrent systems. These mechanisms allow:

- The use of abstractions to specify multi-actor coordination patterns. The coordination patterns include atomicity and temporal ordering.
- Separation of functionality and resource management strategies. For example, policies for actor placement may be specified in terms of an actor group abstraction independent of the representation and invocations of a particular group satisfying the abstraction.
- The ability to develop generic, application independent code for protocols which increase dependability. In particular, the architecture we propose allows the dynamic installation or removal of protocols to change the fault-tolerance and security characteristics of a running system.

5.1 Synchronizers

Synchronization constraints provide modular expression of constraints which need to be satisfied by a *single* actor before it may process a communication it has received. Although synchronization constraints are often promoted as a way to describe coordination of concurrent objects (e.g., [6]), they are unsatisfactory when describing multi-actor coordination: synchronization constraints depend only on the local state of a single actor.

In distributed computing, a group of object invocations often must satisfy certain temporal ordering or atomicity constraints. Conventional programming languages do not allow multi-object constraints to be specified in a modular and reusable manner. This creates a number of problems. Considerable programming effort is required to express multi-object constraints in terms of low level message passing. Moreover, expressing these constraints by explicit message passing "hard wires" both the constraints and their implementation into the application software. Thus, the same abstract multi-object constraints must be reprogrammed for use with different objects. Finally, the implementation of the multi-object constraints may not be transparently changed. These difficulties suggest that new high-level coordination language constructs are needed to simplify the task of programming.

We have made some progress in this area. Specifically, we have developed high level language constructs which allow *multi-actor constraints* to be directly expressed [7]. We define two types of multi-actor constraints: temporal orderings on, and atomicity of, invocations of shared distributed actors. Multi-actor constraints are described in terms of conditions that must be satisfied for a group of method invocations to be accepted: if the conditions are not met, the invocations are delayed. Thus multi-actor constraints coordinate concurrent objects by restricting their activation. As the following examples suggest, a large class of coordination schemes can be efficiently expressed using invocation constraints:

- Consider a group of cooperating resource administrators who must share a limited resource. The administrators must therefore adhere to a collective policy limiting the total number of resources allocated at a given point in time. Enforcement of a collective policy can be expressed as a multi-actor constraint on invocations that request resources: an allocation request can only be serviced if there are resources available.

- A group of dining philosophers is organized so that each philosopher shares her two chopsticks with two others. The number of philosophers is equal to the number of chopsticks and a philosopher needs the two chopsticks next to her in order to eat. Deadlocks may be avoided by enforcing a multi-actor constraint that requires atomic invocation of the pick method in two chopstick actors by a single philosopher.

We build on the observation that multi-actor constraints can be specified independent of the representation of the actors being coordinated. Specifying multi-actor constraints in terms of the interfaces enables better description, reasoning and modification of multi-actor constraints. Specifically, utilizing only knowledge about interfaces to describe multi-actor constraints allows code for coordination to be separated from that for the actor's functionality. This separation enables system design with a larger potential for reuse. Actors may be reused independent of how they are coordinated; conversely, multi-actor coordination patterns may be reused on different groups of actors. In particular, it is possible to abstract over coordination patterns and factor out generic coordination structures.

We describe multi-actor constraints using *synchronizers*. Conceptually, a synchronizer is a special kind of actor that observes and limits the invocations accepted by a group of actors. The functionality of synchronizers is illustrated in Figure 4. Operationally, synchronizers are implemented using primitive actor communication. The advantage of synchronizers is that the involved message-passing is transparent to the programmer who specifies multi-actor constraints in a high-level and abstract way. The implementation of synchronizers may either involve direct communication between the constrained actors, indirect communication with a central "coordinator," or a hybrid. Thus, by using a high-level specification of multi-actor constraints, we provide the flexibility to map the same multi-actor constraint to different implementations.

A synchronizer can be defined and instantiated by a client actor when accessing shared servers. Thus clients can use constraints to enforce customized access schemes. Alternately, a synchronizer can be permanently associated with a group of servers when the servers are first put into operation. In this case, the constraints can express the default interdependence between servers.

Two other approaches to constraints are developed in the systems Kaleidoscope [6] and Rapide [10]. Constraints in Kaleidoscope capture relations between instance variables of multiple objects. Thus, Kaleidoscope constraints are formulated in terms of the representation of the constrained entities rather than their abstract interfaces. The Rapide prototyping system developed by Luckham et al. [10] involves pattern-based triggering of concurrent objects. Thus, in Rapide, it is not possible to express constraints on the invocations accepted by the involved actors.

Figure 4: *Overlapping actor groups defined by two synchronizers.* Messages sent to actors in the two groups are placed outside the synchronizer boundaries. In this figure, pattern matching is depicted as matching shapes between patterns and messages. Synchronizers may disable certain patterns: disabled patterns are black and enabled patterns are white. The messages at `B` are disabled whereas the message at `C` is enabled. The `E` message is unconstrained. Patterns may be grouped together into atomicity constraints that ensure that multiple invocations are scheduled as an atomic action (depicted by boxes around the groups). The messages at `A` satisfy the atomicity constraint whereas the message at `D` is blocked waiting for another message.

INSET: Cooperating Resource Administrators

Consider two cooperating resource administrators (spoolers) which manage a common printer pool. Suppose the pool has n printers. When an administrator receives a print request, it performs some bookkeeping computations and then sends the request through a common bus so that one of the free printers can grab the request and start printing. The use of two spoolers allows greater concurrency and increases availability. We use a coordination constraint to ensure that requests are not relayed when there are no free printers. The constraint also ensures that the two spoolers cooperate to maintain the correct count of available printers.

Maintenance of common information about the number of free printers can be described external to the spoolers as a synchronizer. Figure 5 contains a synchronizer which enforces the global allocation policy. The names `spooler1` and `spooler2` are references to the two

```
AllocationPolicy(spooler1,spooler2,numPrinters) =
{ numUsed := 0;

   numUsed = numPrinters  disables (spooler1.print  and spooler2.print),
   (spooler1.print  or spooler2.print)  updates increment(numUsed)
   (spooler1.done  or spooler2.done)  updates decrement(numUsed)
}
```

Figure 5: A synchronizer which coordinates two print spoolers. A synchronizer has an encapsulated state that is updated through an `updates` operator. The state of the above synchronizer is held by the variable `numUsed`. The `disables` operator delays invocation of the constrained actors. Delays of invocations are expressed as conditions over the state of the synchronizer.

constrained spoolers. The synchronizer prevents the processing of a request when there is no free printer in the pool.

Synchronizers are general tools for describing interdependence between servers performing a service. Using synchronizers, interdependence is expressed independent of the representation of the servers. The resulting modularity makes it possible to modify the coordination scheme without changing the servers and vice versa. In particular, it is possible to dynamically add new printers to the printer pool or add new administrators to the system without changing codes for already existing printers or administrators; new synchronizers may simply be instantiated.

5.2 Modular Specification of Resource Management Policies

Expressing a parallel algorithm in terms of primitive actors provides a logical specification of the algorithm. Such a specification may be called an *ideal algorithm* [8]. The time taken by the ideal algorithm, in the presence of unbounded resources and zero communication cost, is determined by the sequential depth of the longest path in the partial order defined by the actor computation. However, neither of these assumptions is realistic.

In particular, communication costs for an algorithm are a function of the latency and bandwidth of an architecture. Latency is the time taken to send a message from one node to another and bandwidth is the rate at which information may be transmitted between two halves of an architecture. For example, if a problem, such as sorting, requires half the data on a distributed computer to be moved, the performance of an algorithm solving the problem will be bound by the bandwidth. In any physically realizable architecture, the bandwidth may grow by at most $P^{2/3}$, where P is the number of processors. This follows from the fact that space is three dimensional, therefore a given technology yields a constant bandwidth per unit area, and a bisecting plane may grow by at most $P^{2/3}$. There is a similar theoretical bound on I/O. In case of sorting, this means that the speed up is bound by $P^{2/3}$ in general (and $\sqrt{P}$ on a two dimensional network) [13].

Since the performance of an algorithm is dependent on how many messages have to be sent and to which nodes, the efficiency of execution depends in part on the placement and scheduling of objects. In general, the problem of finding an optimal placement policy is intractable. However, for a given algorithm, a user may be able to determine the most efficient placement policy.

Specifications of resource management policies, such as placement, introduce a new layer of complexity to programming concurrent architectures. In particular, the same ideal algorithm executed on the same architecture may yield a different efficiency depending on the resource management policy used. Specifically, the efficiency obtained may depend on a number of factors such as:

- the problem or input size,
- characteristics of the concurrent computer including its latency, bandwidth, size, and processor speeds,
- the placement policy used to map objects to physical resources,
- scheduling strategies used to manage the concurrency.

Current programming methods for concurrent computers intermix specification of resource management policies with the code specifying the ideal algorithm. The resulting conflation of design goals complicates the code and reduces its reusability. We propose to separate the specification of an ideal algorithm from the strategies used to map it to a concurrent architecture. Specifically, we describe a mapping policy in terms of *ActorSpace Type* (AST). An AST may be thought of as a group of actors together with both the abstract operations and the concurrent access constraints on them.

For example, consider an $n \times n$ array. The array can be mapped on a two-dimensional mesh of $p \times p$ processors in a number of ways including:

- *Block placement policy:* the $(i, j)^{th}$ element is assigned to the $(i \text{ div } k, j \text{ div } k)^{th}$ processor, where $k = n/p$.
- *Shuffle placement policy:* the $(i, j)^{th}$ element is assigned to the $(i \bmod p, j \bmod p)^{th}$ processor.

Although the abstract operations and concurrent access constraints of an array are the same, different ideal algorithms may be executed more efficiently using different mapping policies. Linear equation solution techniques, such as Gaussian Elimination or Cholesky Decomposition, are generally efficient when the matrix is mapped using a shuffle placement policy. On the other hand, algorithms for low-level image processing applications, domain decomposition techniques for solving Partial Differential Equations, perform efficiently when their matrix representation is mapped using the block placement policy.

Figure 6: Combining ideal algorithm specification with a placement policy to obtain architecture specific, efficient executable code.

Note that the correspondence between an algorithm and the optimal placement of its AST is not one to one. Different placement policies may be more efficient for the same ideal algorithm on different architectures, and, sometimes for the same algorithm and architecture but a different input size. Organizing a computation in terms of its AST's provides modularity and promotes reuse. The task of programming can be simplified by composing and reusing modules from a repository of placement policies for a given AST.

INSET: Reuse of Resource Management Strategies

Although we are developing better language support for resource management functions in terms of ASTs, the functionality we will provide may, in some cases, be mimicked using current tools. For example, consider a dense matrix representation. A simple set of functions may be used to specify a placement of matrix elements which may be simply used to execute, for example, a Gaussian elimination code or a matrix multiplication code, on a distributed memory machine. Changing the definitions of these functions changes the placement policy and can affect the performance drastically.

The following C-like code specifies the block placement policy (`N` is the number of rows and `P` is the number of processors):

```
row_mapping(global_row_num) { return(global_row_num / N) }
local_index(global_row_num) { return(global_row_num % N) }
```

```
global_index(local_row_num) { return(local_row_num + me * N) }
```

The above functions may be easily redefined to implement a different placement policy, such as the shuffle placement policy, without changing the code for an ideal algorithm using the mapping policy.

```
row_mapping(global_row_num) { return(global_row_num % P) }
local_index(global_row_num) { return(global_row_num / P) }
global_index(local_row_num) { return(local_row_num * P + me) }
```

5.3 Customizing Dependability

Currently, a protocol for dependability must either be built into the system architecture or be re-implemented for each application. Moreover, development of dependable software is expensive: the increased complexity caused by mixing the code for a set of dependability protocols with that of the application code is itself a source of bugs. A significant savings in software development and maintenance costs may be realized if abstract, application-independent specifications of dependability protocols are possible.

We have developed a methodology which allows the code for a dependability protocol to be specified independently of the application specific code [3]. The methodology has been implemented in an experimental kernel called *Broadway*. Our reflective model allows *compositionality* of dependability protocols. Compositionality means that we can specify and reason about a complex dependability scheme in terms of its constituents. Thus, logically distinct aspects of a dependability scheme may be described separately resulting in a methodology which allows dependability protocols to be implemented as generic, composable components.

We employ *reflection* as the enabling technology to allow modular specification and dynamic installation of dependability protocols. Reflection means that an application can access and manipulate a description of its own behavior and execution environment. The actors representing such a description are called *meta-level* actors. For our purposes, the meta-level contains a description sufficient to model the dependability characteristics of an executing application; reflection thus allows dynamic changes in the execution of an application with respect to dependability.

The most general form of reflection leads to interpretation and is costly. For our purposes, we use a limited reflective model in which each actor has three meta-actors: its *dispatcher*, its *mail queue* and its *acquaintance list*. The acquaintances meta-actor represents the current state (behavior) of the actor. The dispatcher and mail queue meta-actors implement the communication primitives of the runtime system so that the interaction between actors can be modified to change the dependability characteristics of an application.

Specifically, a dispatcher meta-actor is a representation of the implementation of an actor's transmission behavior. When customized, messages sent by the corresponding base actor are rerouted to the customized dispatcher. An actor's mail queue meta-actor represents the mail buffer holding messages received by the actor. If a customized mail queue meta-actor is installed, all messages to the base actor are rerouted through it. A customized mail queue may alter the order of messages to the base actor, e.g., to enforce local synchronization constraints.

A number of protocols which increase dependability of a system can be expressed in terms of a customized mail queue, dispatcher and acquaintance list. These protocols include two phase commit, three phase commit, primary back-up, full replication, check-pointing, and encryption. Composition of dependability protocols is achieved by transparently manipulating the meta-actors of the meta-actors. The resulting system allows not only dynamic installation of generically specified protocols but their dynamic removal. The limited form of reflection we use supports incremental compilation and increases execution time by only a very small constant.

INSET: A Replicated Server

Figure 7: A replicated service protocol. `Client` A initially sends a message to `Server` S_1. The message is routed to the mail queue `Broadcaster`. Broadcaster passes a copy of the message off to S_1 and sends a copy to S_2. Both servers eventually reply and their replies are tagged by the two `Tagger` dispatchers. The `Eliminator` mail queue then passes only one copy of the two messages onto A.

To illustrate how *Broadway's* reflective architecture may be used to support dependability protocols, we describe the meta-level implementation of a replication protocol. The protocol involves replicating a server to resist crash failures.

Our original system is a service with two clients. Figure 7 shows the result of installing the meta-actors for this protocol in the system. A clone is made of the service and the appropriate meta-actors are installed at the service and at the clients. The meta-actors are designed to manipulate generic messages. As a result, in combination with the system's ability to clone actors, this protocol is implemented transparently of the base actors. Furthermore, since the protocol is generic, it may be reused with any application.

For each client, a meta-actor — a customized mail queue (`Eliminator`) — is installed at its node. This mail queue will eliminate duplicate messages from the copies of the server. For the server and its replicated copy, customized meta-actors are installed to handle the transmission and reception of messages. The dispatcher `Tagger` tags all outgoing messages so that the clients (using the mail queue `Eliminator`) may eliminate duplicate responses. The mail queue `Broadcaster` copies to server S_2 all messages sent to server S_1. The repetition of messages is necessary to keep the state of the two servers consistent. When a message is sent to the service, the `get` method of the customized meta-mail queue is invoked instead. When the service requires a new message to process, the `Broadcaster` mail queue is sent a `put` message. The code for the `Broadcaster` actor class is shown below. Notice that the transparency of the protocol is preserved by the methods `get` and `put`. These methods manipulate entire messages, never needing to inspect the message contents:

```
class Broadcaster
  var S2, Base;
  /* Setup this actor */
  init(copy,orig)
    S2 := copy;
    Base := orig;
  end
  /* A new message is received */
  method get(msg)
    send S2 msg;
    myqueue.enqueue(msg);
    end
  end
  /* The base actor requests a message */
  restrict put() with (!myqueue.empty());
  method put()
    send Base myqueue.dequeue();
  end
end
```

In Figure 7, the results of a sample message transaction are shown. Note that additional dispatcher meta-actors are required to correctly handle messages that may be sent by the client after the server crashed, but before that crash was detected. To keep this example simple, these additional actors were not shown.

6 Conclusions

Because of the costs of learning new programming languages and rewriting old code, the conversion to new programming paradigms has been slow. This has led some observers to downplay the importance of research in programming languages. At the same time, there is a perception of a software crises as the cost of maintaining programs has escalated. In fact, what exasperates the software maintenance problem is the use of old languages and methodologies which are insufficiently expressive and provide little support for software maintenance.

The acceptance of new programming paradigms is now likely to come more rapidly. There are two reasons for this. First, the cost of developing code in newer languages often outstrips the cost of maintaining old code. For example, consider the increasingly deployed object-oriented software technology. The technology enables programmers to reduce development time by providing support for design and to reduce software maintenance costs by allowing them to incrementally modify their code. Second, the increasing computational power available and, the ever lowering cost, of concurrent computers implies that at least portions of the code need be rewritten to take advantage of concurrent computers.

The need for new programming paradigms is by no means the dominant force in parallel or distributed computing research. For example, considerable effort has gone into the development of parallelizing compilers which attempt to extract parallelism from existing sequential code and then automatically determine the mapping to concurrent computer architectures. The parallelizing compilers approach suffers from two limitations. First, code based on sequential algorithms cannot be generally translated to the best parallel algorithms. Second, no general techniques can allow efficient placement and scheduling strategies for arbitrary algorithms on a concurrent computer.

The development of new programming paradigms should allow more complex programs to be written with less effort. Furthermore, it should make the expression of potential parallelism simpler. However, to be practical, new paradigms must not place unrealistic restrictions on expressiveness. For example, although purely functional, or state-less programming, has some nice concurrency properties, shared mutable state is an essential requirement of distributed computing.

Gains in programmer productivity can only be realized by the greater use of new abstractions and modularity mechanisms. Modularity is gained by separating design concerns: code for different purposes should be independently specified and composed. Abstraction allows increased genericity and reuse. Furthermore, it raises the granularity of programming by allowing code to be expressed in terms of more intuitive structures.

We have discussed a number of ways in which modular construction of multi-component concurrent systems can be supported. These include the use of constraints for expressing coordination patterns for over distributed objects; ActorSpace Types for abstracting over resource management strategies for groups of actors; and meta-programming for dependability protocols. Although the resulting modularity allows concurrent programming to be simplified, this is only a small part of the gain. More importantly, the application independence provides a basis for constructing software repositories:. for example, code stored in repositories can include specifications and implementations of constraints, placement and scheduling policies, and dependability protocols. The executable specifications may then be dynamically linked with different application code without the need for reimplementing any of them. The average application developer need not understand the details of the representation – rather she needs to know only the relevant properties of the abstraction (including properties such as the performance characteristics of certain access patterns).

The concurrent programming language abstractions and modularity mechanisms we propose are certainly not a complete set. They do, however, suggest ways of drastically reducing software development and maintenance costs, scaling up software systems, and making it feasible to use the power of parallel and distributed computing. We believe that the successful application of these methods will further stimulate research in the development of a new generation of realistic high-level programming languages.

INSET: Furhter Reading

Following is a list of further readings.

The Actor model was originally proposed by Hewitt (for example, see [1]), and later developed by Agha ([2]).

[1] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

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

A formal theory of actors, including proof techniques for establishing the equivalence of actor systems, appears in [3].

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Third International Conference on Concurrency Theory (CONCUR '92)*, pages 565–579. Springer-Verlag, August 1992. LNCS.

Separation of representation and description of system and reasoning in terms of meta objects are described in [4].

[4] N. Venkatasubramanian and C. Talcott. A MetaArchitecture for Distributed Resource Management. In *Proceedings of the Hawaii International Conference on System Sciences.* IEEE Computer Society Press, January 1993.

The use of actors for message driven programming of multicomputers is described in [5].

[5] W. Athas and C. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, pages 9–23, August 1988.

A number of research efforts in object-oriented programming are described in [6].

[6] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, Massachussets, 1987.

AST's generalize and abstract over Concurrent Aggregates ([7]).

[7] A. A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs.* MIT Press, 1993.

For actor-based computer architecture such as J-machine, see [8].

[8] W. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Press, 1986.

[9] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1993. (to be published).

Acknowledgments

The authors' work is supported by the Office of Naval Research (ONR contract number N00014-90-J-1899), by the Digital Equipment Corporation, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195). The authors would like to thank contributors to the Open Systems Laboratory including Christian Callsen, Shingo Fukui, Chris Houck, Shakuntala Miriyala, Shangpin Ren, R.K. Shyamasundar, Nalini Venkatasubramanian and Takuo Watanabe. contributions to the laboratory. The research described in here has strongly benefited from the first author's discussions with Carl Hewitt and Carolyn Talcott, among others.

References

[1] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[2] G. Agha and C.J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *Principles and Practice of Parallel Programming '93*, 1993. Sigplan Notices (To be published).

[3] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Dependable Computing for Critical Applications III, IFIP Transactions.* Elsevier Science Publisher, 1993. (to be published).

[4] G. Agha, C. Houck, and R. Panwar. Distributed Execution of Actor Systems. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 1–17. Springer-Verlag, 1992. Lecture Notes in Computer Science 589.

[5] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[6] Bjorn N. Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 268–286, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[7] S. Frølund and G. Agha. A Language Framework for Multi-Object Coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, 1993. To appear in LNCS.

[8] L. H. Jamieson. Characterizing Parallel Algorithms. In R. J. Douglass L.H. Jamieson, D.B. Gannon, editor, *The Characteristics of Parallel Algorithms*, pages 65–100. MIT Press, 1987.

[9] W. Kim and G. Agha. Compilation of a Highly Parallel Actor-Based Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*. Yale University TR DCS RR-915, 1992. to appear in LNCS, Springer-Verlag.

[10] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent Timed Systems. In *Proceedings of the 1992 DARPA software Technology Conference*, April 1992.

[11] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1993. (to be published).

[12] B. Shriver and P. Wegner (Eds.), editors. *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass., 1987.

[13] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Scalability of Parallel Sorting on Mesh Multicomputers. *International Journal of Parallel Programming*, 20(2), April 1991.

Contents