

The PAGIS Grid Application Environment

Darren Webb and Andrew L. Wendelborn
Department of Computer Science
University of Adelaide
South Australia 5005, Australia
{darren, andrew}@cs.adelaide.edu.au

Abstract

Although current programming models provide adequate performance, many prove inadequate to support the effective development of efficient Grid applications. Many of the hard issues, such as the dynamic nature of the Grid environment, are left to the programmer. We are developing a programming model that incorporates a familiar, formal computational model and a reflective interface. The programming model, called PAGIS, provides a desirable abstract computer with an interface to introduce and customize Grid functionality. Using PAGIS, an application programmer constructs applications that are implicitly parallel and distributed transparently. This paper describes the basic components of the PAGIS framework for constructing and executing applications, and the reflective techniques to customize applications for computation on the Grid.

1. Introduction

The Grid is used in numerous scientific disciplines to solve large, complex problems. The Grid provides pervasive access to a geographically-dispersed, large scale execution environment. It integrates heterogeneous resources with dynamic availability and capability connected by an unreliable network. Computational scientists and engineers use the Grid to distribute computation to hosts offering various computation and data services.

Grid-enabled programming systems enable familiar programming models to be used in Grid environments[8]. A programming model defines an abstract machine, user libraries and software tools a programmer uses to interact with a system. Grid programming models are generally adapted from established parallel programming models designed for homogenous, tightly coupled supercomputers. However,

there is growing consensus that these programming models are inadequate to support the effective development of efficient Grid applications[16]. Models, such as MPICH-G2[9], are typically low-level, restricted in their applicability, and lack many essential properties and capabilities required for Grid applications.

We believe that current Grid programming models are too complex for many potential Grid programmers. While providing high-performance, the responsibility for managing many of the hard issues, such as the dynamic nature of the Grid environment, fall to the programmer. Many will lack the skills necessary to exploit the full power of the Grid.

Projects including Webflow[3], Symphony[23], Javelin[20], the CCA-based[2] Triana[27] and XCAT[15], JCSP.net[33] and Jini Service Grid[11], are developing high-level programming models suitable for the Grid. Similar to these projects, we focus on a component-based programming model. A programmer visually describes their application by *what* program components are executed. The PAGIS approach is different in the techniques used to describe *how* these program components are executed.

PAGIS is an application building and execution framework that enables scientists to tap into the Grid with little or no Grid programming skills. PAGIS provides two interfaces that emphasize a separation of what an application does from how it does it. The application programming interface uses a formal yet simple, abstract computational model to describe what is executed. The reflective interface enables the introduction of new behaviour to the application, such as its execution in a Grid environment and its dynamic control.

In this paper, we present the PAGIS framework. We first describe the basic components of the framework required for building and executing applications. Second, we discuss the reflective techniques used to customize applications for computation on the Grid. Fi-

nally, we discuss the implementation status and future work.

2. Computational Engine

A computational model describes an abstract computer with certain desirable properties. The model describes the concepts and elements that underly any computer language. The model hides the complexity of real machines, enabling programmers to better understand the performance characteristics crucial to their programs. We define a computational engine as the realization of the computational model.

In this section, we describe the programmer interface to the PAGIS computational engine. We first describe the computational model and its language binding. We then describe the user interface optionally used to visualize the construction and execution of a PAGIS program.

2.1. Process Networks

Kahn's Process Networks[12] is a elementary computational model for parallel programming. The model describes a collection of autonomous computational resources connected in a network of unidirectional communication links. A given computational resource computes on data coming along its input communication links, using memory of its own, to produce output on some or all its output links.

A process network program is stated as a relation between input sequences and output sequences. The structure is a directed graph where a node represents a process and an edge represents a channel able to carry data of a given type. Figure 1 shows a simple process network composed of a number of processes and channels. A process is a sequential program representing successive passes each of which incrementally transforms a stream of data. Processes communicate through first-in-first-out streams of tokens such that each token is produced exactly once and consumed exactly once. Production of tokens is non-blocking while consumption from an empty stream is blocking.

The model has a number of desirable characteristics. A process network program is deterministic. Computation depends only on program state. Program structure determines the schedule, relieving the programmer of the burden of scheduling and guaranteeing consistent behaviour across implementations[26]. In addition, the process network model allows pipelined parallel computation. Future input concerns only future output, hence a process may produce output before all its input is available.

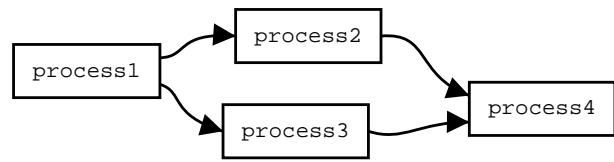


Figure 1. A simple process network of four nodes and four channels.

The process network model is defined by a recursive schema. An important property of this schema, and a fundamental element of process network semantics, is deterministic reconfiguration of the program graph. Reconfiguration enables a program graph to evolve to produce results. In fact, an application can be described as a single process that dynamically evolves, in a top-down fashion, as computation proceeds[13]. Process networks may evolve by replacing, inserting, or removing a node from the process graph.

The process network model is beneficial for many applications. The model is used in applications ranging from image analysis in Geographical Information Systems (GIS)[7, 31] to real-time sonar beamforming[1]. In addition, many other computational models, such as dataflow networks[17] and synchronous dataflow[22], are special cases of the process network model[17]. Results obtained from this model are directly applicable to these special cases.

2.2. A Process Network API

The process network model describes an abstract computer. A language defines the interface for communicating with the computer. A programmer uses the language to instruct the computer to perform certain tasks. In process networks, such tasks include receiving and sending data.

An Application Programming Interface (API) defines an abstract system. It provides, much like a language, an interface between application programmers and an abstract system. A suitable API allows applications and the system implementation to vary independently.

The Process Network API (PNAPI) follows a port-centric architecture. Programs are described by composition using ports to represent communication endpoints. Ports define a communication contract or interface with the process. Channels are defined by connecting ports. The approach is used in many graph-based syntaxes, including Ptolemy[5], the Process Graph Method (PGM)[14] and the Common Com-

ponent Architecture (CCA)[2].

The port-centric approach provides several software engineering benefits. Processes are modular, self-contained and reusable components. The level of indirection introduced by the ports enables control of access to the channel and allows the channel structure to change independent of the end-points.

The API provides abstractions for describing both network construction and execution. The API provides *Network*, *Process*, *Port* and *Channel* abstractions to describe the structure of a process network. A programmer composes a process network using a *Builder*. Reconfiguration operations are achieved using a special *Builder* called a *Framework*. These are further discussed in [28].

We have described the language for communicating with the abstract system. Next, we describe a graphical tool for visually composing and executing process networks.

2.3. Application Builder

The Process Network model lends itself well to a visual programming interface. Like any model that syntactically can be described as a graph, the elements of a process network map nicely to visual components. The PAGIS Application Builder is our user interface tool enabling programmers to interact visually with the PNAPI. It visually maps the concepts and abstractions of the PNAPI.

The Application Builder is comprised of two environments: the *Designer Environment* allows a programmer to visually construct a process network, while the *Execution Environment* enables a programmer to visualize the dynamic structure of a process network application in execution.

The *Designer Environment* is a composer of processes and channels into a processing graph. It enables a programmer to build applications by the composition of general, widely applicable processes organized into domains or palettes. It enables the user to create processes and channels, manipulate their initial properties and add them to a process network. Finally, the Designer enables programmers to execute a completed process network.

The first step in constructing a process network is to add processes. The user selects processes from the Process Palette. The Process Palette presents a set of domain-specific set of operations that address the functional concerns of an application. They are presented as a categorized view of processes available to the user. Processes are categorized in a tree structure, but may appear in more than one category. Default categories

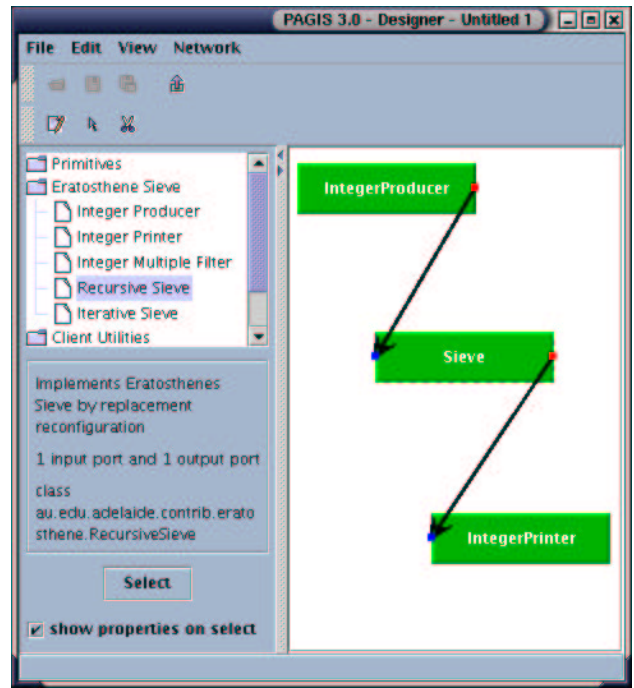


Figure 2. The PAGIS Designer Environment.

in this example include “Primitives”, “Client Utilities” and “Eratosthene Sieve”. The user clicks on the category then the process they wish to use. The palette displays information regarding the role of the process, its input and output. The user selects the process by clicking the “Select” button. The programmer uses the palette to create a new instance of the selected process and adds it to the Designer Environment.

The user creates channels by clicking on an output port, dragging the pointer to and then releasing on an input port. The channel is represented as a line between the output port and input port. The arrow on this line represents the direction of data flow (as opposed to demand).

Figure 2 shows the result of constructing the Sieve of Eratosthenes as a process network. The Sieve is a simple example that uses process reconfiguration to generate prime numbers. The Designer shows three processes: an integer generator, a sieve, and an integer printer. The integer generator produces a sequence of integers (from 2 to n) as output. The sieve reads a prime from its input, reconfigures the network to filter future multiples of this prime, and writes the prime to output. The integer printer displays integers it reads from input. We have developed and demonstrated more complex examples, such as the manipulation of satellite imagery.

The *Execution Environment* enables a programmer to visualize the real-time execution of a process network. It enables the programmer to view the active structure and state of a process network, including the results of reconfiguration, in the computational engine.

The Execution View shows the processes and channels of a running network. When started, the network contains no processes or channels and therefore the view is blank. Processes and channels are added to the network using Designer Environment.

The user opens the Designer Environment from the Execution Environment. When the network described using the Designer Environment is triggered, the processes and channels are added to the Execution Environment for visualization.

The Application Builder is a simple tool that provides two views of a process network program. It is used to provide programmers with a simple, graphical interface and to demonstrate the concepts of the model. Next, we describe an implementation of the model.

2.4. Implementation Layer

The computational engine provides the implementation structures that execute the processes and coordinate communication. The engine implements the syntax of the PNAPI and the semantics of the model. As an abstract model, process networks permit many potential implementations. We have previously described[31] an implementation, much like Symphony[23] and Javelin[20], that distributes processing using RMI. Here we describe a lightweight base implementation that addresses only the functional concerns of applications as defined by the model and PNAPI. Additional functionality, such as distribution, is introduced later, and only introduced as required.

Our base system is a multithreaded application written in Java. Similar to [26], the system creates a separate thread for each process. The thread iteratively invokes the process firing sequence. Channels are implemented as a producer-consumer pair of buffers. An intermediary *transfer thread* is responsible for moving tokens from the producer buffer to the consumer buffer. This channel design, called *Half-Channels*[32], is primarily used to simplify reconfiguration operations. A half-channel segregates the producer and consumer processes so reconfiguration operations involve only the reconfiguring process and intermediary threads.

This is a lightweight implementation that enables programmers to construct, run and reconfigure process networks. But how do we best approach the execution of process network applications in a Grid environment?

One alternative is to change the implementation of

the computational engine. The implementation will undoubtedly benefit some Grid application domains, but will very likely adversely affect others. A new implementation for each application domain will diversify the code base, complicating portability and reusability of the system.

A second alternative is to leave Grid concerns to the programmer. The programmer contorts their application code to use Grid libraries that distribute the computation and communication of a process network on the Grid. Now programmer the programmer is mixing what the application does (its functional concerns) with how it does it (its non-functional concerns), thereby increasing complexity, and reducing portability and reusability.

Our choice was to investigate the application of reflective techniques that allow a programmer, in an organized way, to introduce new Grid behaviour to their applications. The technique, called Metalevel Programming, allows us to expose selected aspects of the programming model the application programmer is likely to need to introduce such behaviour.

Next, we describe the technique of metalevel programming, the metalevel architecture introduced to our programming model, and how the metalevel is used to introduce Grid behaviours.

3. Customizing for the Grid

The semantics of the process network model are defined formally, and execution can be performed by any computational engine conformant with those semantics. Hence, it is desirable to express any given computational engine as a customization of a generic computational engine. One example of this is customizing a computational engine for application to the Grid.

3.1. Metalevel Programming

One view of metalevel architectures, shown in Fig. 3, is to treat the system as a black box, with separate interfaces that address the functional and non-functional concerns of an application. The *baselevel* interface, implemented by the system, describes the functional behaviour of a program, and comprises “ordinary” objects and classes called baseobjects and baseclasses. The *metalevel* interface exposes particular aspects of the underlying system to customization. The metalevel consists of objects that *reify* elements of the system, and carry out *reflective computation* on them.

Reflection is the ability of a system to observe and change its own execution[24]. It is described by

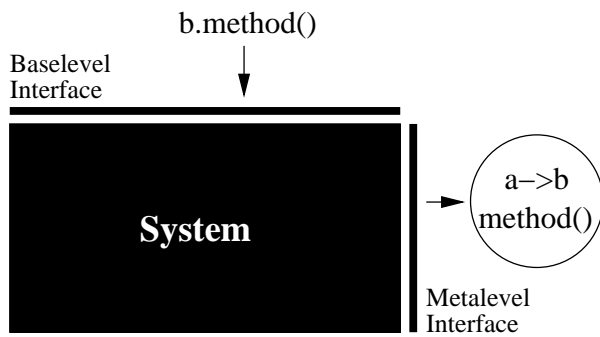


Figure 3. The system provides both baselevel and metalevel interfaces to address application concerns.

Maes [18] as being “the activity performed by a computational system when doing computation about its own computation”. The goal of reflection is to allow programs to reason about their own execution state and alter it to change its own meaning. Reification is essentially the process of converting some component of system state into a representation that may be computed upon[25], using *metaobjects*[18].

We identify two forms of reification: *structural reification* and *behavioural reification*. Structural reification is the process of converting some element of a system into a metaobject. Each baselevel object has a corresponding *MetaObject*, which contains reflective information about the implementation and interpretation of the baseobject, and reifies structural elements of a baseobject including inheritance information, methods and attributes. A program may access this meta-information and make modifications to it. Modifications to the meta-information result in actual modifications to the behaviour of the system, through the property of causal connectivity[18].

We now consider issues of metalevel design, under the assumption that we reify (behavioural reification) a method invocation, and thus have available, as meta-information, details of the invocation.

3.2. Metalevel Design

The metalevel interface evident in most metalevel architectures, including 3-KRS[18], ProActive[4], and MetaXa[10], requires the implementation of a *trap method* of the target metaobject. The trap method is a hook used to access the message metaobject, called with the reification of the actual method invocation when that invocation takes place.

This enables a metalevel programmer to apply metalevel behaviour to all messages sent to a target baseob-

ject. However, it is common to require the application of multiple metalevel behaviours. This design does not easily accommodate the application of multiple metalevel behaviours. The provision of one method of the target metaobject leads the programmer to mix metalevel behaviour code which complicates maintainance and reuse of otherwise independent metalevel behaviours.

Certain other metalevel interfaces, such as Friends[6] and Guarana[21], represent individual metalevel behaviour as first class objects that can be composed and applied to each message, allowing two or more metalevel behaviours to be used at the same time as one combined metalevel behaviour.

These metalevel interface designs allow the composition and application of multiple metalevel behaviours to each message. Additionally, the representation of metalevel behaviours as first class objects promotes reuse of metalevel behaviour.

Some metalevel architectures apply more structure to the metalevel interface, especially for concurrent and distributed programming. CodA[19] uses an operational decomposition of the method invocation process to define the metalevel interface. The metalevel interface decomposes method invocation into various orthogonal phases including send, receive, accept, queue and execute. ProActive and MetaXa share some aspects of this structure.

An operational decomposition provides a model that is clear and simple to understand. However, a given metalevel behaviour may require customization at many phases of the method invocation process. This splitting up of the metalevel behaviour into several parts, and coordination between the phases, complicates reuse and composition. Metalevel behaviour that applies to the same phase can not be composed without the programmer manually solving conflicts between overlapping semantics.

We now show how we extend this previous work towards an approach that we think facilitates a compositional approach to defining Grid-relevant behaviours, usable as both a model to guide the “software engineering” of grid applications, and as a technique for Grid-enabling existing applications.

3.3. Enigma Metalevel

In our Enigma metalevel[29], we present a metalevel architecture that is customizable and composable and is applicable within a distributed setting. In the abstract metalevel, each baseobject has a representative metaobject. This metaobject provides access to information and operations concerning the baseobject.

Figure 4 depicts the course of a method invoca-

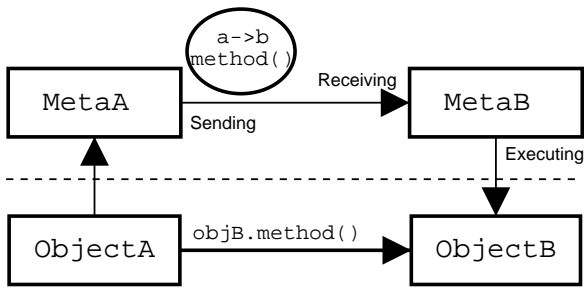


Figure 4. A model reifying the three phases of method invocation.

tion through the abstract metalevel. The metaobject *MetaA* receives all reified method invocations sent from *ObjectA*. The metaobject *MetaA* directs the message to the target metaobject, in this case metaobject *MetaB*. The metaobject *MetaB* executes the invocation upon the target baseobject *ObjectB*.

This identifies the responsibilities of the metaobject for an object during a method invocation. When the baseobject is the source of the message, the metaobject exposes the process of identifying the target metaobject and sending the message. When the baseobject is the target of the message, the metaobject exposes the process of receiving and executing the message. These responsibilities are made accessible by the metaobject. Exposing the process of method invocation allows us to influence system behaviour to better suit baselevel behaviour. The metalevel programming task of customizing a particular behaviour is carried out by associating particular actions with one or more of the three “points of responsibility” identified above.

A metalevel programmer composes chains of metabehaviour at selected points of responsibility: each metabehaviour is given the opportunity to reflect and customize the method invocation. Metabehaviours delegate the responsibility to invoke the method code to the next metabehaviour in the chain.

We now show briefly how we can design a new metabehaviour using Enigma. A tracing behaviour, useful for debugging, is readily defined (similar to [18], for example) in terms of the “executing” hook, where we show the handling of a reification of a message invocation: the metalevel program can simply extract and print the method name, invoke the method code, then extract the results and print those.

3.4. Enigma applied to Process Networks

One critical design decision in doing this with Enigma is: what is the best way to represent, from the point of view of reflective programming in Enigma, the various aspects of a process network? To make a poor decision leads to a cumbersome and complex meta-program; a separate reification of individual components turned out to be the wrong approach. Hence, we introduce, by a process of structural reification, a metaobject termed *MetaComputation*, that reifies the abovementioned components as one structure.

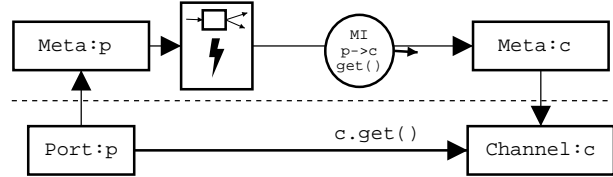


Figure 5. The reification of computation provides a logical grouping of a computation's components.

Figure 5 illustrates how such a *MetaComputation* routes a method invocation. The figure illustrates the behavioural reification of a method invocation *MI*. (Here, *MI* represents the invocation of a the *get()* method of a *Channel* object *c* by a *Port* object *p*; in the process network model, this might take place during the retrieval of a data token by a process from a channel.) It is important to understand the semantics of the diagram. *Meta : p* is a metaobject corresponding to the base level object *Port : p*. It receives the reification of the method invocation, *MI*. *Meta : c* has encoded within it the action to take on receiving *MI*: the *MetaComputation* (pictured just to right of *Meta : p*) is used by this code to give access to the required components of a process network sub-computation. For example, the *MetaComputation* includes an object reference to the receiver, which identifies the recipient, *Meta : c*.

The *MetaComputation* metaobject enables the components of a computation to be treated as a single entity. Customization affects all components of the *MetaComputation*. Next, we discuss Grid behaviours that make use of the *MetaComputation* metaobject.

3.5. Grid Metabehaviours

We now consider the implementation of metabehaviours for the Grid. In Fig. 6, we show how we can effect a form of migration. Here, we use

another instance of the method invocation reification “pattern” to customize the “receiving” point of the *MetaComputation* metaobject itself. In terms of the process network computational engine, the customization essentially transfers to another (possibly remote) metaobject the responsibility for handling method invocations upon the elements of the computation. This effectively distributes individual computations of the computational engine, transparently affecting how the computation engine is deployed. In terms of Enigma, note that we have introduced a meta-meta-level for the purposes of customization of the behaviour of how we handle the reification of the original method invocation.

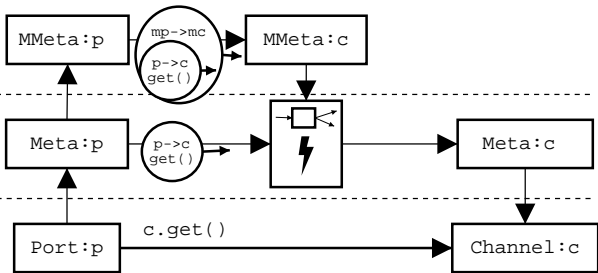


Figure 6. The introduction of a new metalevel to customize communication with all components of the computation.

Next, we consider potential metabehaviours that generate adaptation events. We can extend the tracing behaviour described earlier to provide a “performance monitoring” behaviour that records information about the method call (destination, timing, and so on) in a suitable repository, perhaps an NWS (Network Weather Service) memory. We envisage the incorporation of a software “agent” that uses this information and Grid resource information services as the basis for detection and forecasting of large-scale changes in system behaviour, and the generation of events to the metaprogram to trigger an adaptive response. At this time, the generation of events is invoked through user action

At CCGrid 2002, we demonstrated an interface through which a user could view non-functional (implementation oriented) aspects of the runtime system, and also influence the behaviour of the system in execution. The interface is via a pop-up GUI associated with elements, such as processes, of a process network. It allows introspection of implementation attributes such as physical location. The GUI can also be used to invoke Globus tools to find suitable compute resources (the

GUI provides the ability to construct Resource Specification Language (RSL) expressions, and use Globus to run them over a set of Grid directory servers), make a selection from the set returned, and cause that process, its ports and channels to migrate to the newly selected resource.

4. Conclusions

In light of the low-level nature of current Grid programming models, we have developed a high-level programming model with an interface to a simple, intuitive computational model and an interface to customize aspects its implementation. Programmers use the application interface to describe the functional requirements of their application. Programmers optionally use the metalevel interface describe non-functional requirements, such as the application’s execution on the Grid.

The model forms a unifying framework for describing a computational engine independent of its deployment. We have described deployment within a Grid environment. The model supports other types of metabehaviour for other types of deployment. For example, we are interested in using Enigma to develop an agent-based computational engine using Aglets. The *MetaComputation* is implemented as an Aglet that coordinates execution, communication and migration. Composing this behaviour with the Grid behaviours described earlier provides a flexible approach for combining agent and Grid services within a common framework.

Although we use Enigma to effect a migration behaviour, migration is restricted in actual use because the code it needs must be already present on the JVM to which it migrates. This is not easily achieved with current Java/Globus tools. Our Coglets project [30] develops a more suitable secure peer to peer framework: its implementation is in progress, to be completed by March 2003. We are also making use of the PAGIS environment as the basis for a demonstrator project in middleware for mobile applications on a grid.

We have developed a form of metabehaviour composition that we believe is useful for moving existing systems to the Grid, for building new Grid applications, and especially for supporting adaptive software. We intend to continue to build better abstractions for Grid metabehaviour composition, and improve our understanding of how to use them.

References

- [1] G. Allen, B. Evans, and D. Schanbacher. Real-time Sonar Beamforming on a Unix Workstation using Process Networks and POSIX Threads. In *Proc. of 32nd Asilomar Conf. on Signals, Systems & Computers*, pages 1725–1729, November 1998.
- [2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Computing. In *Proc. of HPDC8*, 1999.
- [3] D. Bhatia, V. Burzevski, M. Camuseva, G. Fox, W. Furmanski, and G. Premchandran. WebFlow - a visual programming paradigm for Web/Java based coarse grain distributed computing. *Concurrency: Practice and Experience*, 9(6):555–577, 1997.
- [4] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, Sept–Nov 1998.
- [5] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muiadi, S. Neuendorffer, J. Reekie, N. Smyth, and Y. Xiong. Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California Berkeley, 1998.
- [6] J.-C. Fabre and T. Pèrennou. FRIENDS: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications. In *Proc. 2nd European Dependable Computing Conf. (EDCC-2)*, Taormina, Italy, 1996.
- [7] J. Flack. *On the Interpretation of Remotely Sensed Data Using Guided Techniques for Land Cover Analysis*. PhD thesis, Curtin University of Technology, November 1995.
- [8] I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *LNCS*, 2150:1–??, 2001.
- [9] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-Area Implementation of the Message Passing Interface. *Parallel Computing*, 24(12), 1998.
- [10] M. Golm and J. Kleinöder. MetaJava - A Platform for Adaptable Operating-System Mechanisms. In *Proc. of ECOOP 97 Workshop on Object-Oriented and Operating Systems*, volume 1357 of *LNCS*, June 1997.
- [11] Z. Juhasz, A. Andics, and S. Pota. JM: A Jini Framework for Global Computing. In *Proc. of 2nd Intl. Sym. on Cluster Computing and the Grid (CCGrid 2002), GP2PC Workshop*, May 2002.
- [12] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of IFIP Congress 74*, pages 471–475. North Holland Publishing Company, 1974.
- [13] G. Kahn and D. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Proc. of IFIP Congress 77*, pages 993–998. North Holland Publishing Company, 1977.
- [14] D. Kaplan and R. Stevens. Processing Graph Method 2.0 Specification. Technical report, Naval Research Laboratory, September 1995.
- [15] S. Krishnan, R. Bramley, D. Gannon, M. Govindaraju, J. Alameda, R. Alkire, T. Drews, and E. Webb. The XCAT Science Portal. In *Proc. of SC2001*, 2001.
- [16] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A grid programming primer. Technical report, Global Grid Forum Programming Models Working Group, August 2001.
- [17] E. Lee and T. Parks. Dataflow Process Networks. *Proc. of IEEE*, 83(5):773–801, May 1995.
- [18] P. Maes. Concepts and Experiments in Computational Reflection. In *Proc. of OOPSLA 87*, 1987.
- [19] J. McAffer. Meta-level Programming with CodA. In *Proc. of ECOOP85*, volume 952 of *LNCS*, 1985.
- [20] M. Neary, A. Phipps, S. Richman, and P. Cappello. Javelin 2.0: Java-based parallel computing on the Internet. In *Proc. of Euro-Par 2000*, Munich, Germany, 2000.
- [21] A. Oliva, I. Garcia, and L. Buzato. The reflexive architecture of Guarana. Technical report, Institute of Computing, State University of Campinas, 1998.
- [22] J. Pino and E. Lee. Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors. In *Proc. of Intl. Conf. on Acoustics, Speech, and Signal Processing*, May 1995.
- [23] A. Shah. *Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources*. PhD thesis, Virginia Tech, 1998.
- [24] B. Smith. Reflection and Semantics in a Procedural Language. Technical report, MIT, 1982.
- [25] J. Sobel. An introduction to Reflection-Oriented Programming. In *Proc. of Reflection'96*, April 1996.
- [26] R. Stevens, M. Wan, P. Lamarie, T. Parks, and E. Lee. Implementation of Process Networks in Java. Technical report, University of California Berkeley, July 1997.
- [27] Triana. <http://www.triana.co.uk>.
- [28] D. Webb. *The Application of Metaobject Protocol to Grid Programming*. PhD thesis, University of Adelaide, 2003. In preparation.
- [29] D. Webb and A. Wendelborn. A Meta-Object Protocol for Grid Behaviours. Submitted to the Intl. Workshop on Java for Parallel and Distributed Computing.
- [30] D. Webb and A. Wendelborn. Java Coglets. In *Proc. of 2nd Intl. Sym. on Cluster Computing and the Grid (CCGrid 2002), GP2PC Workshop*, Berlin, May 2002.
- [31] D. Webb, A. Wendelborn, and K. Maciunas. Process Networks as a High-Level Notation for Metacomputing. In *Proc. of 13th Intl. Parallel Processing Sym. Workshops: Java for Distributed Computing*, pages 718–732. Springer-Verlag, April 1999.
- [32] D. Webb, A. Wendelborn, and J. Vayssière. A Study of Computational Reconfiguration in a Process Network. In *Proc. of 7th Workshop on Integrated Data Environments Australia (IDEA'7)*, February 2000.
- [33] P. Welch, J. Aldous, and J. Foster. CSP networking for Java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *LNCS*, pages 695–708. Springer-Verlag, April 2002.