Data Handles and Virtual Connections: High-level Support for Anonymous Reconfiguration

Bala Swaminathan
bs@stl.nexen.com
Ascom Nexion
St. Louis, MO 63146

Kenneth J. Goldman
kjg@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130-4899

Abstract

Data handles *and* virtual connections *are presented as a solution to the problem of supporting application-driven reconfiguration without sacrificing the separation of communication and computation. The solution supports* anonymous reconfiguration*, meaning that the module performing the reconfiguration and the modules affected by the reconfiguration need not be aware of each other's presence in the system. The solution allows modules to reconfigure the system within the limits of a specified communication structure while retaining support for dynamic end-user reconfiguration.*

The work is presented in the context of I/O abstraction, a programming model that defines the communication structure of a distributed system in terms of connections among narrow data interfaces of encapsulated modules. I/O abstraction supports dynamic end-user reconfiguration of distributed applications by separating the communication structure from the module definitions.

Keywords: *distributed systems, dynamic reconfiguration, programmable connections*

1 Introduction

The *configuration* of a distributed system consists of modules and their interconnections. Interconnections can be classified into three categories: static connections that are declared within modules and resolved at run-time, connections that are created by end-users, and connections that are created by running modules during the course of the computation.

Systems supporting only static connections cannot be reconfigured without recompilation, whereas the second and third categories of connections enable *dynamic reconfiguration*, the act of modifying the configuration of a system while the modules are executing and interacting. This paper builds on previous work on The Programmers' Playground [7], an implementation of I/O Abstraction that was designed to support dynamic end-user configuration. This paper describes how we have added dynamic module-driven configuration to The Programmers' Playground by introducing three simple mechanisms: *handles*, *internal connections*, and *virtual connections*.

Configuration of distributed systems has been an important area of research, and a number of mechanisms for configuring systems have been developed. For example, the task description of a Durra [1, 2] application has a static number of ports it uses to communicate with other tasks. Darwin [11, 14, 16] supports logical reconfiguration where the programmer adds code that adapts program modules to participate in reconfiguration. Conic [12, 10, 13], Darwin's predecessor, provided a graphical user interface for configuring the structure of communication among system components. Both Durra and Darwin allow adding or deleting interconnections between processes. PROFIT [9] is a language that provides a mixture of RPC and data sharing for communication, and permits dynamic binding of slots in special cases [8]. Dynamic binding of ports in Durra or slots in PROFIT is accomplished externally, and not initiated by the application itself. Darwin allows an application to change its bindings, provided that the application receives the references of service interfaces through messages (RPC's) before making a transactional binding. Argus [15] supports reconfiguration with two phase locking over atomic objects and version management recovery techniques. Since Argus is intended to be used primarily for programs that maintain online data for long periods of time and since Argus programs are configured for sending and receiving messages to and from other Argus programs, there is no need for program-initiated reconfiguration of the communication structure.

In this paper, we consider application-driven dynamic reconfiguration in distributed systems whose modules are written using the *I/O abstraction* programming model. Briefly, I/O abstraction is the view

that each software module in a system has a set of data structures that may be externally observed and/or manipulated. This set of data structures forms the external interface (or *presentation*) of the module. Each module is written independently and modules are then *configured* by establishing *logical connections* among the data structures in their presentations. As published data structures are updated, I/O occurs implicitly ("under the covers") according to the logical connections established in the configuration. I/O abstraction is similar to the dataflow model, except that I/O abstraction modules can produce results autonomously, in addition to being able to react to input events, and communication can be bidirectional.

In I/O abstraction, the communication structure is not hard-coded into the modules and the end-user dynamically creates the system configuration. Sometimes, though, we may want a module within an application to request reconfiguration of the system. For example, a multi-player game application may need to establish communication between opponents dynamically based on winners of previous games. This functionality might be handled by a tournament manager module in the application that makes connection decisions over time on the basis of data received from other modules in the application. However, since I/O abstraction has been designed to separate computation from the communication structure, modules do not know the configuration. Instead, the configuration is imposed externally. So, the need for module-driven reconfiguration results in a paradoxical situation where we want an application that can reconfigure itself, even though each module of the application does not know the system's configuration.

In this paper, we augment I/O abstraction with an *anonymous reconfiguration* strategy that reconciles this paradox. We introduce a special data type called *handle* that can be published by a module for "connection" to items in the presentations of other modules. We may think of a module with handles as a *logical switch*, where the handles represent its external ports. In the same way they establish logical connections between data variables in module presentations, users can establish connections between a handle variable and a data variable of *any* type, and also may form connections between the handles of different logical switches. Furthermore, a logical switch module may create *internal connections* between its own handles dynamically at run-time. These connections can be thought of as internal data paths of a switch. A *virtual connection* between two data variables is created automatically at run-time whenever there exists an appropriate path through the logical switches.

Practically any configurable system permits one to implement a system component that forwards messages received on its input port to various destinations through its output ports. Since the component controls the choice of output ports to which messages are forwarded, it can redirect the message traffic dynamically. Although this kind of component provides switch-like behavior, it has the disadvantage that the data actually must pass through the component on its way to the various destinations.

The Conic configuration tools [13] allow a hierarchy of modules to be constructed, where one module may represent a collection of several other modules. Ports of the internal modules may be exposed as ports of the parent module. Thus, with support for hierarchical connection collapsing, a connection formed between two high-level modules (that expose ports of their respective children) can be mapped to a direct connection between the corresponding child modules. Taking advantage of this structure, one could implement a reconfiguration program in Conic at another layer of the hierarchy. The reconfiguration program could form connections among the ports of several of its children, resulting in direct connections between the modules at the lowest level of the system. This approach is attractive for some applications, but it appears that the configuration program and the underlying modules must be somewhat coupled, and that the system be structured hierarchically in anticipation of which modules' communication may need to be switched.

In the solution we present in this paper, the handles and logical switches are treated as an abstract specification of the communication pattern, instead of as an implementation of the communication itself. Making this separation frees the run-time system to map the specified logical communication pattern onto the real communication network in a more efficient way.

No data communication actually passes through the handles and internal connections of a logical switch. Instead, the existence of virtual connections results in direct communication between the endpoints, as if the user had requested the connection explicitly in the configuration. Furthermore, modules are still written with a local orientation and configured by end-users at run-time. However, by connecting variables to handles of a logical switch, the end-user can leave parts of the reconfiguration to program control, even though the logical switch can be completely generic, needing no knowledge of the specific modules to which its handles are connected. Hence, the term *anonymous reconfiguration.*

Ironically, the fact that I/O abstraction supports the separation of computation from communication helps make it well-suited for supporting module-driven reconfiguration. Rather than writing application programs that are peppered with explicit sends and receives or make explicit calls to remote modules, the

Figure 1: A Playground system

behavior is the sequence of values held by the data items in its presentation. This is the view that the environment has of the module.

Control: The control portion of a Playground module is divided into two components: *active control* carries out the ongoing computation, while *reactive control* carries out activities in response to input from the environment. Active control is defined by the `main()` function, and reactive control is defined by identifying *reaction functions* to be called when particular presentation items are updated externally.

Connections: Connections among presentation data items define the communication pattern. Connections are established separately from the definitions of modules themselves. In this way, each module need not concern itself with the structure of its environment, but only with the behaviors exhibited at its presentation.

Playground supports two kinds of connections, *simple connections* and *element-to-aggregate* connections. A simple connection relates two presentation data items of the same type. The semantics of a connection from item x, in module A, to item y, in module B, is that whenever A changes the value of x, item y in module B is updated with that value as well. Communication is asynchronous and pairwise FIFO. For a discussion of element-to-aggregate connections, see [7, 18].

2.1 Playground Implementation

A basic understanding of the Playground implementation is necessary for an understanding of how anonymous reconfiguration is implemented. An overview of a Playground system is shown in Figure 1. A running module consists of three components: the application, the *protocol*, and the *veneer*. The veneer defines the library of publishable data types and maintains the presentation, including protection informa-

tion. Reactive control information is also registered within the veneer.

The protocol interacts with the operating system in order to exchange data with other Playground modules on behalf of the application. Connections are established by a special Playground application called the *connection manager*. The connection manager enforces type compatibility among the data items involved in a connection and also guards against protection violations by establishing connections only if they satisfy the access privileges defined for the relevant data items.

The protocol interacts with the connection manager in order to make its module's presentation information known to the connection manager and in order to learn about connections affecting its module. The presentation of each Playground module includes an externally readable data structure P that holds a description of the application's presentation and an externally writable data structure L that contains link information for that module. Data structures P and L are used only for communication between the protocol and the connection manager, and are hidden from the application.

For establishing logical connections between the presentations of various modules, the connection manager publishes a connection request data structure R which may be updated externally by any module in the system, usually by a graphical front-end module for the connection manager. For each connection request placed in R, the connection manager (through a reaction function) checks for type compatibility, verifies that the connection obeys the access protections established for the endpoint data structures, and adds the connection to its published link update record L'. This change is reflected in the L data structures of the protocols corresponding to the endpoints of the connection. In this way, each protocol is aware of each logical connection in which it is involved. Note that the connection manager is not a communication bottleneck. The connection manager simply sets up the connections that are then handled individually by the protocols associated with the connected modules. It is also interesting to note that the connection management system itself uses the I/O abstraction mechanism to establish links.

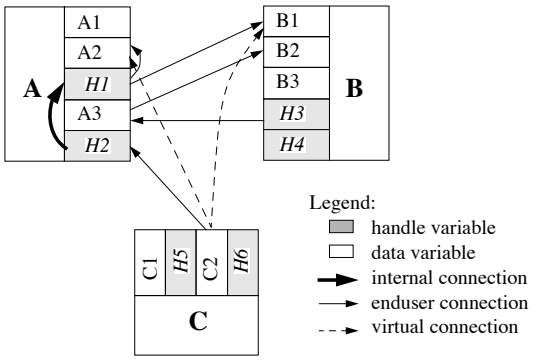
3 Anonymous Reconfiguration

Recall that Playground modules are written with a local orientation and then configured, by end-users, at execution time. The communication structure of a Playground system is through the connections between the data-interfaces of these modules. Users may establish connections through a graphical front-end by drawing an arrow from an entry in one module's presentation to an entry in another module's presentation, where the arrow's direction indicates the direction of communication.

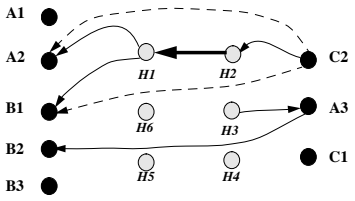
In this section, we present a mechanism through which arbitrary user applications can initiate reconfiguration of the communication structure of the system without compromising the notion that each module is written to interact with an abstract environment. We define a new Playground data type called *handle* that carries no data information. A handle is a simple Playground data type, an instance of which can act as an alias for one or more data or handle variables in other modules and/or for data variables in the same module. The alias relationship is created by end-users when they establish a connection between a handle and another presentation variable. Since a handle may be thought of as part of a "conduit," as opposed to a variable, a given handle may be connected to variables of different types simultaneously.

The set of handles in a system can be construed as a *logical switch* with the handles as its external ports (see Figure 2). Ports of a switch become internally linked when the switch requests a connection to be added between two of its own handles. Such connections between handles of the same application are called *internal connections*, to distinguish them from *external connections* that are outside the module. When an internal connection is established, this may complete certain circuits of the switch. The connection manager resolves the configuration of the system (see Section 3.1), adding connections between variables that are indirectly connected through the switch. These connections, called *virtual connections*, are established if the types match and access protections are not violated. Similarly, when an internal connection is deleted, the connection manager deletes all the virtual connections that are no longer connected through an internal connection. We emphasize that none of the data actually passes through the logical switch. Instead, the switch is simply an abstraction that provides information to the run-time system in order to establish direct communication between the endpoints.

Figure 2(a) shows a collection of three modules and the configuration among their presentations. In this figure, variable *C2* is connected to handle *H2*, handle *H1* is connected to variables *A2* and *B1*, and so on. Figure 2(b) shows the corresponding configuration graph. Suppose module A requests an internal connection (shown as a dark line) from handle *H2* to handle *H1*. Our semantics require that the connection manager resolve this virtual connection into two connections: one virtual connection from *C2* in module C to *A2* in module A, and another from *C2* in module C to *B1* in module B, assuming that types match and



(**a**) Modules and their configuration.



(**b**) Corresponding configuration graph.

Figure 2: Creating virtual connections.

access protections are not violated.

Since we think of the handles and internal connections of a switch as forming conduits through which connections may be established, we expect that each virtual connection should involve two handles of that switch that are connected internally. We would not expect, for example, that two data variables in the presentations of different modules are connected if one has a connection going into a conduit and the other has a connection coming out of the same end of that conduit. Instead, we would expect them to be connecting with items at the other end of the conduit through an internal connection of the switch. In the next section, we define carefully those situations in which virtual connections occur.

3.1 Virtual Connection Management

The connection manager maintains an RGB graph representing the configuration of the system.[1] An RGB graph is a directed graph $G = (V, E)$, where V is the set of all presentation variables in the system, and an edge $(a \in V, b \in V)$ is included in E exactly when there is a connection between a and b. We let $color(e)$ denote the color of each edge e, as follows. Edge $e = (a, b)$ is *white* (W) if both a and b are data variables; e is *red* (R) if one of a and b is a data variable and the other is a handle; e is *green* (G) if a and b are handles in the same module, and e is *blue* (B) if a and b are handles in different modules.

If $p = (e_1, e_2, \ldots, e_n)$ is a path in G, we define $color(p)$ to be the sequence of colors $color(e_1), color(e_2), \ldots, color(e_n)$.

White edges are of primary interest because they represent the actual communication pattern in the system. Whenever a white edge exists in the graph, the connection manager informs the endpoints so that direct communication between the two endpoints may be established (see Section 2.1). A white edge is considered *user-defined* if the edge represents a connection created by the end-user through a graphical front-end. A white edge is considered *virtual* if it represents the existence of a path between the endpoints that was established through a logical switch. Moreover, it is possible for a white edge to be *both* user-defined and virtual. For example, an application can request an internal connection that results in a virtual connection to be realized between two variables that were already connected by the end-user.

Virtual white edges are created by the connection manager according to the following rule. We say that a directed path p from u to v in an RGB graph is an *rgb-path* iff $color(p)$ is a string in the language defined by the regular expression RG(BG)*R. The connection manager keeps a virtual white edge from u to v in the RGB graph if and only if there exists an *rgb-path* from u to v, provided that access restrictions and type compatibility rules are satisfied.

Thus, all edges in the RGB graph are added explicitly by the user, except for the green edges and the virtual white edges, which support anonymous reconfiguration. The green edges are internal connections created by logical switches under program control, and the virtual white edges are created by the connection manager in response to paths through those switches. Communication occurs between endpoints whenever there is a white edge, either virtual or user-defined.

4 Examples

In this section, we present two examples of the use of anonymous reconfiguration: a shared cursor application and an application-driven module migration application.

4.1 Shared Cursors

This section describes the shared cursor mechanism used in a a distributed radiological multimedia conference application that has been constructed recently using the Playground. The application uses cursor sharing to enable a group of geographically separated physicians to discuss an x-ray image [3].

[1] Currently, the connection manager is centralized, but we are planning to distribute its functionality in the next version of the implementation.

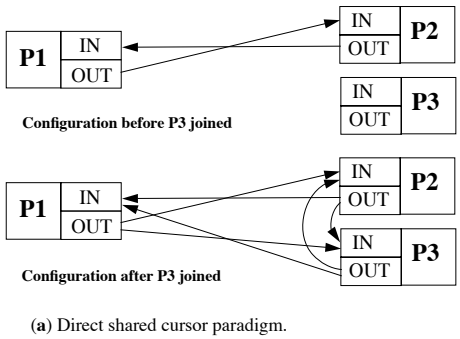
In the shared cursor application, two or more users run modules that display an image and a cursor. The "active" cursor and the user's name are highlighted in every user's image. In this section, we show two approaches to cursor sharing. We first describe the direct shared cursor approach where a new physician module that wants to join the shared cursor application either should know the "address" of other participants in order to be able to connect to their cursors or should be connected by the end-user. We then describe the switched shared cursor approach where the physician modules simply connect to a switch that is associated with a discussion of interest. The switched cursor approach simplifies the radiology conference set-up.

Direct shared cursor approach: Figure 3a shows two physician modules exchanging their cursor information between themselves. The cursor information includes the name of the module and the position of the cursor. Suppose physician module P3 wants to participate in the discussion. With the direct shared cursor approach, the *IN* data of P3 should be connected to the *OUT* data of P1 and P2, and the *OUT* data of P3 should be connected to the *IN* data of P1 and P2. One way to achieve these connections is for P3 to know the addresses of P1 and P2 and connect to them directly during initialization. Alternatively, one could use the Playground end-user configuration tool (the graphical front-end to the connection manager) to create the connections.

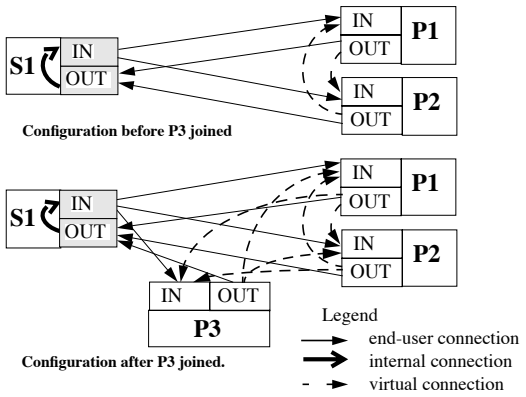
With this approach, the number of connections that need to be manually made by the end-user is linear in the number of participants. Also, if a given physician module wants to join a different group of physicians, then all of its connections to the previous group must be severed and connections with the new group must be established. This can be cumbersome and slow when physician modules keep changing groups.

Switched shared cursor approach: Figure 3b shows two physician modules P1 and P2 connected to a switch S. Since the switch has an "internal connection" from its *OUT* handle to its *IN* handle, all legal virtual connections are established between P1 and P2 as shown by the thick lines.

Consider the physician module P3 that wants to join the sharing of the image with P1 and P2. All P3 needs to do is to connect to the *IN* and *OUT* handles of the switch, and the connection manager establishes virtual connections between P1 and P3 and between P2 and P3. With the switched shared cursor approach, the number of connections that needs to be made is a constant and does not depend on the current number of participants. Further, when a physician module decides to join a different group, this can be achieved simply by disconnecting it from one switch and by connecting it to another switch. The radiological conference application mentioned above can be enhanced by introducing a switch for each conference.



(a) Direct shared cursor paradigm.



(b) Switched shared cursor paradigm.

Figure 3: Shared cursors example.

It is interesting to note that the radiological conference application was originally built using the direct shared cursor approach. When anonymous reconfiguration was added to Playground, the connection set-up was not only simplified, but the virtual connection mechanism was incorporated into the application without modification of the existing physician modules. Also, since the switch provides an abstraction for a cluster of virtual connections, the physician modules need not have any knowledge of the number and the locations of other participants.

4.2 Application-Driven Module Migration

In the module migration mechanism of The Programmers' Playground, end-users specify migration by establishing a connection from a distinguished presentation entry called `MIG` in the *running* module to a similar entry in the *waiting* module [19]. We assume that the presentation of each relocatable module is

static.

Since `MIG` is a simple data variable, it can be connected to a handle variable in the same module or in another module. A serendipitous advantage of virtual connections is that migration can now be initiated by the *running* module. Suppose the *running* module's `MIG` variable and the *waiting* module's `MIG` variable are connected to two of the running module's handles. The *running* module can initiate migration by requesting a connection between these handles.

In certain applications, a group of relocatable modules may try to achieve load balancing by migrating between processors back and forth. Or, we can imagine a load balancing module that has published data variables that receive load information from other modules and has published handles which are connected to the `MIG` variables of other modules. The migration mechanism can be modified so that when an application initiates migration, through virtual connections, the veneer can change the state of the migrating program from *running* to *waiting* instead of terminating the program. This way, the load balancing module can move the modules without the necessity to restart them at the destination processors.

4.3 Discussion

In the previous examples, we presented anonymous reconfiguration in the context of The Programmers' Playground. However, the anonymous reconfiguration concept is applicable to a wide range of distributed systems. In order for our anonymous reconfiguration mechanism to be applicable in a distributed system, the system must satisfy the following properties.

1. The logical configuration of the system is separate from the module definitions.

2. The connection management system can detect the paths between any two data points in the current logical configuration.

If a distributed system satisfies these properties, it can be extended with handles, internal connections, and virtual connections to support anonymous reconfiguration. In Playground, the modules are written with a local orientation and are later configured together through external connections made by the end user. The connection manager in the Playground system maintains the logical configuration graph.

5 Summary and Future Work

The properties of I/O abstraction, particularly the declarative nature of communication and the clear separation of computation from communication, provide advantages for application driven reconfiguration. In this paper, we identified a design paradox of allowing application modules to participate in reconfiguration while retaining a separation of communication from computation. This problem was resolved by designing an anonymous reconfiguration strategy. Our strategy employed the concepts of *handles*, data types that act as conduits; *internal connections*, connections made within a module; *logical switches*, in which handles behave as ports and connections as internal data paths; and finally, *virtual connections*, the connections that are created by the presence of an internal connection made by the application.

Since a handle is also a Playground data type and since any Playground module can have handle instances in its presentation, the combination of handles, internal connections, and external connections involving handles can be used to achieve any desired configuration. For example, the current implementation of the connection manager front-end does not have a built-in mechanism for specifying broadcast connections. However, a broadcast-switch module can be designed with handles externally connected to the participating variables. This switch can be initialized with a complete graph of internal connections between its handles, thus effecting a broadcast connection. Every participating variable will be connected to every other variable through virtual connections, provided the types match and access protections are obeyed. Furthermore, if true multicast is supported by the network, the logical switch mechanism enables the runtime system to provide a transparent mechanism that maps the virtual connections onto a multicast connection in the network for more efficient bandwidth utilization.

We considered application driven reconfiguration in the context of modifying the communication pattern. Our mechanism for reconfiguration only ensures that the resulting communication pattern will be in agreement with the internal connections, but does not specify the order in which these connections will be made nor does it specify when communication must start. It may be useful to extend the mechanism so that all the virtual connections formed as a result of a given internal connection occur *atomically*, in the sense that data would be sent on these connections only after all of the connections have been established.

Acknowledgments

We thank Paul McCartney for his ideas during the initial discussions and Naeem Bari for his input on implementation during his work on distributed radiological multimedia conference application. We thank Ram Sethuraman for his comments on earlier drafts.

This research was supported in part by the National Science Foundation under grant CCR-94-12711 and the Advanced Research Projects Agency under contract DABT-63-95-C-0083.

References

[1] M. R. Barbacci, D. L. Doubleday, C. B. Weinstock, and J. M. Wing. Developing applications for heterogeneous machine networks: The Durra environment. *Computer Systems*, 2(1), March 1988.

[2] Mario R. Barbacci and Jeannette M. Wing. A language for distributed applications. In *International Conference on Computer Languages, New Orleans, LA, USA*, pages 59–68, March 1990.

[3] Naeem Bari and Jerome R. Cox, Jr. Distributed radiological multimedia conference. Technical Report WUCS–95–28, Washington University in St. Louis, 1995.

[4] Kenneth J. Goldman. Welcome to The Programmers' Playground! *http://www.cs.wustl.edu/cs/playground*.

[5] Kenneth J. Goldman, Michael D. Anderson, and Bala Swaminathan. The Programmers' Playground: I/O abstraction for heterogeneous distributed systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 363–372, January 1994.

[6] Kenneth J. Goldman, T. Paul McCartney, Bala Swaminathan, Ram Sethuraman, and Todd Rodgers. Building interactive distributed applications in C++ with The Programmers' Playground. Technical Report WUCS–95–20, Washington University in St. Louis, July 1995.

[7] Kenneth J. Goldman, Bala Swaminathan, Michael D. Anderson, T. Paul McCartney, and Ramachandran Sethuraman. The Programmers' Playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

[8] Brent Hailpern and Gail E. Kaiser. Dynamic reconfiguration in an object-based programming language with distributed shared data. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 73–80, May 1991.

[9] Gail E. Kaiser and Brent Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, 14(2):201–264, April 1992.

[10] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):1293–1306, April 1985.

[11] Jeff Kramer and Jeff Magee. Evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[12] Jeff Kramer, Jeff Magee, and Anthony Finkelstein. A constructive approach to the design of distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 580–587, May 1990.

[13] Jeff Kramer, Jeff Magee, and Keng Ng. Graphical configuration programming. *IEEE Computer*, 22(10):53–65, October 1989.

[14] Jeff Kramer, Jeff Magee, and Morris Sloman. Configuring distributed systems. In *5th ACM SIGOPS European Workshop, St. Michel, France*, September 1992.

[15] B. Liskov. Distributed programming in Argus. *CACM*, 31(3):300–313, March 1988.

[16] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. In *International Workshop on Configurable Distributed Systems*, pages 102–117, March 1992. Imperial College of Science, Technology and Medicine, UK.

[17] T. Paul McCartney and Kenneth Goldman. Visual specification of interprocess and intraprocess communication. In *Proceedings of the 10th International Symposium on Visual Languages (VL'94), St. Louis, MO, USA*, pages 80–87, October 1994.

[18] Bala Swaminathan. Connection management in reconfigurable distributed systems. D.Sc. dissertation, Washington University, St. Louis, MO, September 1995.

[19] Bala Swaminathan and Kenneth J. Goldman. Dynamic reconfiguration with I/O abstraction. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing (SPDP'95), San Antonio, Texas, October 25–28, 1995*, pages 496–501, October 1995.