A Language for Generic Dynamic Configuration of Distributed Programs

Markus Endler
Departamento de Ciência da Computação,
IME, Universidade de São Paulo,
Caixa Postal 20.570, 01452-990 São Paulo
endler@ime.usp.br

Abstract

Distributed systems are being increasingly used for applications demanding dynamic changes in their functionality and interaction structures. Examples are applications with requirements for high availability, such as in telecommunication, online information services, and process control, and applications with inherently dynamic structures of distribution and parallelism, such as in network management and parallel processing. In this paper we present the reconfiguration language Gerel and its current implementation. Gerel allows for the programming of generic scripts for dynamic reconfigurations which can be used in different applications. The major novelty of Gerel is the provision of powerful precondition and selection mechanisms, which are used to check if the current configuration has the required properties, and to apply the reconfiguration commands only to the objects satisfying these properties. In this paper we show how Gerel is used for defining dynamic reconfigurations in distributed programs implemented according to the master-slaves paradigm.

Keywords: Language, Tool, Dynamic Reconfiguration, Distributed Systems

1 Overview

Distributed systems are being increasingly used for applications which require the ability to dynamically change the system's functionality and communication structures. Such applications either require a high availability, i.e. it is either impossible or very costly to stop the entire program for maintenance, or have a problem-specific dynamic structure of their distribution and parallel processing. This leads to a growing interest on languages and tools that support dynamic reconfiguration of distributed programs.

Essentially, dynamic reconfiguration can be classified according to the kind of change it produces, and to the moment it is defined. In the first case, we distinguish between functional and structural changes. *Functional* changes, also called module implementation changes[9], are reconfigurations where new code is added to the program, thus modifying its functionality. *Structural* changes are reconfigurations which only change the interdependency relation between the system's parts, or create new processes from already available code.

According to the moment of its definition, we distinguish between programmed and ad-hoc changes. *Programmed changes* are reconfigurations which are defined at system design time (and may be compiled to machine code together with the actual application program), and are triggered by the program itself. Due to the possibility for an efficient implementation, programmed changes are well suited for automatic, program-driven modifications, such as dynamic load balancing or fault-recovery mechanisms. *Ad-hoc changes*, on the other hand, are reconfigurations defined only when the application program is already in execution. They are usually supported by tools that allow the user to interact with the system in order to query the current configuration and to execute simple reconfiguration commands. Ad-hoc changes are necessary for software maintenance, and can be of both functional or structural type.

When all kinds of dynamic reconfiguration, and in particular, when ad-hoc and programmed dynamic changes are to be supported together, new requirements for the reconfiguration language and its execution mechanism arise. They come from the problem that the two kinds of reconfiguration have to be coordinated, and that programmed changes must be specified for a dynamically evolving configuration, i.e. a configuration which cannot be predicted at system design time.

In this paper we present the reconfiguration language Gerel, which satisfies these requirements by supporting the implementation of generic dynamic reconfigurations. The major novelty of Gerel is the provision of powerful precondition and selection mechanisms, which allow for querying the current configuration about required properties, and applying the reconfiguration commands only to the objects satisfying these properties. Gerel's execution model further guarantees that programmed reconfigurations are performed only if their precondition is satisfied, and that they are executed in mutual exclusion with other reconfigurations in a same program.

The original goal in the design of Gerel was to have a language supporting the implementation of programmed changes for evolving configurations (i.e. of configurations also manipulated by ad-hoc changes), which is possible due to Gerel's precondition and selection mechanisms. It turned out, however, that these mechanisms also favor the implementation of generic schemas of reconfigurations, which can be used in many application programs. This second benefit of Gerel will be the main concern of this paper.

This paper is structured as follows. Section 2 defines the basic concepts underlying Gerel. In section 3 we present a simple example requiring programmed changes. In

section 4 we introduce Gerel and show how it is suited to implement the programmed changes of the example. Section 5 explains our current implementation of Gerel, and section 6 summarizes related work. Finally, in section 7 we evaluate our work and point to future directions.

2 Basic Concepts

We view a distributed application as a set of interacting components. A component is a run-time object encapsulating part of the application's state and interacting with other components by sending and receiving messages through its communication interface. It is created dynamically from a component type, which is the program describing the component's functionality. The communication interface of a component consists of a set of ports, each of which specifies a message type (which also says whether it is for synchronous or asynchronous communication) and the direction of message flow. Two components can only communicate if they have compatible ports (i.e. with same type and opposite direction), and if these ports are connected.

Essentially, in Gerel we adopt the Configuration Based Approach[4], where a distributed program is implemented at two levels: the *programming* and the *configuration* level. While at the programming level one describes the implementation of the components, at the configuration level the interconnection structure between the components of a distributed application is described. Another characteristics of Configuration based approaches is that all kinds of dynamic reconfigurations are described only at the configuration level, e.g. as the creation or removal of components and their interconnections.

3 The Master-Slave Example

Many computational intensive problems have a parallel solution based on the *master-slaves* paradigm. This paradigm advocates that a problem be decomposed into independent subproblems which are solved in parallel. The configuration implied by this paradigm is shown in figure 1. It consists of two types of components. A single *master* component is in charge of decomposing the problem, allocating subproblems to *slave* components, and putting together the partial results received from the slaves.

Some solutions require only a single decomposition, and thus make it possible to determine the number of required slaves at program start. However, other parallel solutions adopt a recursive decomposition of the problem, and therefore require the ability to change the number of slaves while the problem is being solved. Typical examples for such evolving solution structures are Divide-and-Conquer and Branch-and-Bound algorithms. In such cases, the master component must be able to create

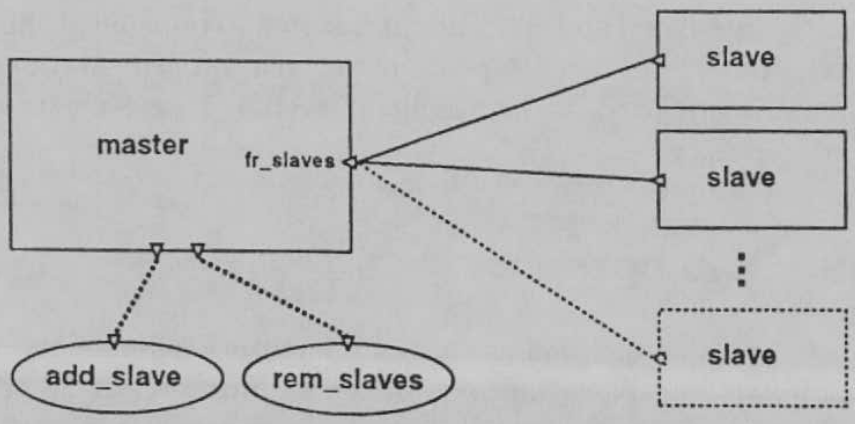

Figure 1: Master-slaves Interconnection Structure

new slaves whenever it decomposes a new (sub)problem. In addition, it must be able to remove all its slaves when the entire problem is solved.

These two extra functions of component *master* are naturally expressed as programmed changes to the Master-Slaves interconnection structure, which are described by Gerel change scripts *add_slave* and *rem_slaves*, presented in section 4.4. As suggested by figure 1, component *master* is capable of invoking these changes since it is logically connected to them. In section 5 we will explain how we implemented this capability.

Figures 2 and 3 show the general structure of the algorithms for the master and slave components. For the interaction between *master* and its *slaves*, we adopt a communication protocol similar to the one described in [5]. In this protocol, a slave first announces its availability by sending a 'null' result to *master*. This message, as well as all subsequent messages containing a result of a subproblem computation, are also requests for new subproblems. The identity of the *slave* is contained in the message sent to *master*, and is used as the destination address for the reply message. This arrangement has the advantage that *master* does not need to know the identity and number of available slaves.

In master's algorithm, *update_set_subpr* is a function which updates the set of open subproblems, based on the receipt of result *r*. This function will typically remove from *open_subpr* the subproblem for which *r* is a solution, and eventually add new subproblems described by message *r*. We further assume that *update_set_subpr* does not modify *open_subpr* when *r* equals 'null'. After updating the set of open subproblems, function *allocate* selects one subproblem, which is then sent to the idle slave. After this, function *requ_slaves* computes the number of extra slaves required, based on the number of open subproblems (#open_subpr) and the number of available slaves (#slaves). These additional slaves are created by repeatedly calling the programmed change *add_slave*. The main loop terminates when all open subproblems are solved. After this, *master* calls change *rem_slaves* to delete all its slaves.

The *slave*'s algorithm consists of an infinite loop, where subproblems are solved

```
component master;
entry fr_slaves: slave_request;
begin
  initialize; #slaves:=0; open_subpr:= global_problem;
  repeat
    receive r from slave_i;
    if r≠null then record(r)
              else #slaves:=#slaves + 1;
    open_subpr:= update_set_subpr(open_subpr, r);
    sub_prob:= allocate(open_subpr);
    reply sub_prob to slave_i;
    for i:= 1 to requ_slaves((#open_subpr−1), #slaves)
              call add_slave(slave_path, fr_slaves);
  until #open_subpr=0;
  finalize;
  call rem_slaves( fr_slaves);
end;
```

Figure 2: Master Algorithm

by function *compute*, results of these computations are sent to *master*, and new subproblems are received.

4 The Reconfiguration Language Gerel

Gerel, which stands for *G*eneric *Re*configuration *L*anguage, is suited both for the programming of change scripts and for performing ad-hoc reconfigurations. The main advantage in the use of Gerel is the capacity to describe *generic* reconfiguration scripts which are based on queries of the current configuration. Gerel scripts are generic in the sense that they can be specified in terms of type and connec-

```
component slave;
begin
  result:= null;
  loop
    send result to master wait sub_prob;
    result:= compute( sub_prob);
  end;
end;
```

Figure 3: Slave Algorithm

tivity properties of the elements of a configuration (e.g. components, component types, ports, etc.) rather than in terms of their concrete names. Due to this possibility, such scripts are flexible, and can be applied to all configurations where its constituent elements satisfy the specified properties. Therefore, Gerel allows implementing programmed changes that maintain their validity despite ad-hoc reconfigurations. Moreover, it is possible to implement change scripts which can be used in different applications, as we will show later.

In the following we present Gerel by first introducing its (basic and structured) commands, and describing the concept of reconfiguration variable (generic symbol). Then we explain Gerel's powerful query language Gerel-SL, and finally we describe the general structure of Gerel scripts by presenting the scripts *add_slave* and *rem_slaves*, of our example.

4.1 Commands

Gerel has two sorts of commands: basic and structured commands. *Basic commands* are used to create and remove components and the connections between their ports. These commands define reconfiguration actions, and can be used both in change scripts and for ad-hoc changes.

create *component* **of** *comp_type* ["(" *arguments* ")"] [**at** *location*]
delete *component*
link *port port*
unlink *port port*

In the above commands *component*, *comp_type*, *location* and *port* are so-called configuration expressions, which either denote the name of an existing configuration object or a generic symbol. A generic symbol (section 4.2) is a sort of reconfiguration variable which has only names of configuration objects as its values. When a generic symbol is used, the corresponding command is applied to the configuration object that is the current value of the generic symbol.

Gerel has three *structured commands*, which define the control flow of reconfiguration programs. Commands **select** and **forall** furthermore implement Gerel's selection mechanism and establish the link between the basic commands and Gerel-SL, which is used for writing the selection formula (*formula*).

select *gs* ":" *formula* **do** *commands* **end**
forall *gs* ":" *formula* **do** *commands* **end**
iterate *i* **in** "[" *low* ":" *high* "]" **do** *commands* **end**

Select and **forall** define the assignment (or binding) of one or more configuration objects to the generic symbol *gs*. Whenever *gs* appears in any Gerel-SL formula or basic command within *commands*, the objects currently bounded to *gs* are used in its place. In **select**, the statements in *commands* are executed for only one (randomly

chosen) object satisfying *formula*. In **forall**, *commands* is executed for every object satisfying *formula*. If the set of selected objects is empty, then *commands* is not executed. The evaluation of *formula* is done only once, before the first execution of *commands*, which guarantees the termination of all **forall** commands. The **iterate** command is a simple for-loop, where *i* is the loop variable, and *low* and *high* are integer expressions holding the lower and upper iteration bounds. This command is needed for specifying reconfigurations on arrays of components or ports.

4.2 Generic Symbols

Gerel supports the definition of reconfiguration variables, called generic symbols. They are used to relate the configuration property expressed in a Gerel-SL formula with the reconfiguration actions to be applied to the objects satisfying this property. Generic symbols have *configuration types*, and their values are only names of configuration objects. For every kind of configuration object, there is an associated type of generic symbol, e.g. type **inst** (component), **type** (component type), **port**, etc. Additionally, the range of a generic symbol of type **inst** and **port** can be further constrained to a specific object type, as shown in the declaration of the following symbols.

```
symbol
  n: inst abc;          /* components of type abc */
  s: port controlT;     /* ports of type controlT */
  i: inst;              /* components of any types */
  l: location;          /* any location */
  m: type               /* any component type */
```

The first two are called *specific configuration types*, and the latter are called *generic configuration types*. The language defines that all specific configuration types are compatible to their corresponding generic type, and that specific types are only compatible if their generic and specific parts are equal. Gerel's programming-in-the-large types not only allow for strong type checking, but also enhance the efficiency of its selection mechanism, since only the objects of a given type have to be considered.

Besides configuration types, Gerel has also data types, such as integers, booleans, strings, etc. These may be used for defining the script's formal parameters, which in turn can be used as indexes for port or component arrays, or as the arguments of the **create** command.

4.3 Gerel's Query Language

Gerel contains a powerful query language called Gerel-SL. It is a first order logic language used to describe properties of configuration objects. Gerel-SL can be used both for specifying selection formulas and reconfiguration preconditions. While the

evaluation of a reconfiguration condition produces a boolean result, selection formulas (which must contain at least one free generic symbol) bind this generic symbol to the set of objects satisfying the formula.

Gerel-SL formulas are normalized first order logical formulas, where all universal and existential quantifiers (keywords **fa** and **ex**) appear to the left of a quantifier-free propositional formula. These are built by applying the logical negation (**not**), the logical conjunction and disjunction, denoted by symbols & and | to simpler propositional or atomic formulas.

In addition, Gerel-SL supports the use of an open set of primitive predicates and functions (Gerel-SL primitives), each of which expresses a basic configuration property or relation. Since many of these primitives are required for different kinds of configuration objects, we use the prefixes c_, ct_, p_ and l_ to denote, respectively, if the primitive refers to components, component types, ports or locations. In the following list, the prefixes in square brackets describe all the variants of the primitives that are available.

Predicates:

[c,p]_linked(o_1, o_2)	objects o_1 and o_2 are linked
[c,ct,p,l]_exists(o)	object o exists
[p]_free(o)	object o is available (not connected)

Functions:

invoker()	the component calling the change script
[c,p]_type(o)	the type of object o
[p]_owner(o)	the component to whom object o belongs
[p]_def(o)	the definition name of object o
[c,p]_index(o)	the index of the (array) object o
[l]_load(o)	the number of components at location o

The following is an example of a Gerel-SL formula:

fa C: **ex** P: **not** c_type(C)=*def* | c_linked(C, P)

Assuming that C is a generic symbol of type **inst**, and P is a generic symbol of type **inst** *abc*, this Gerel-SL formula expresses that every component of type *def* is connected to at least one component of type *abc*.

4.4 Gerel Scripts

In Gerel, all programmed changes are defined in change scripts. These are a sort of reconfiguration procedure, which may have parameters and define generic symbols that are local to the script. One of the major characteristics of a Gerel script is that it may contain a reconfiguration precondition, which is evaluated whenever a script is invoked. Only if it is satisfied, the script is executed. Otherwise, the invoker is informed about the unsuccessful script execution. By specifying a change

precondition, one is able to design change scripts that are executed only if the configuration to which they are applied satisfies the structural properties described in the precondition. This allows the programming of change scripts that are robust to all 'exceptional' configurations that do not satisfy the precondition.

In the following we show the Gerel scripts for the programmed changes of the example in section 3. In the example, *add_slave* is invoked by *master* whenever this requires a new slave for solving a subproblem. The script for this change is parame-

```
change add_slave(t: type ; masterport: port);
symbol l, m: location;
        new: inst;
        q: port;
condition ct_exists(t) & p_exists(masterport) & p_owner(masterport)=invoker()
execute
  select l: fa m: l_load(l) ≤ l_load(m) do
    create new of t at l;
    select q: p_owner(q)=new & p_type(q)=p_type(masterport) do
       link masterport q end
  end
end.
```

terized with the component type (the name of the executable) for the slave and the (address) of the master's port, whose values are provided by the invoking component (*master*). Its precondition (keyword **condition**) requires both that slave's type is available, and that port *masterport* really exists at the interface of the invoking component. In the execute section (which must be present in every Gerel script), first the machine with the less number of processes is selected and bound to generic symbol *l*. Then a new slave is created (with the type passed as parameter) at the selected machine. Finally, the port at the new slave which has a type compatible with *masterport* is connected to this port, i.e. the master and the new slave are connected.

```
change rem_slaves( masterport: port);
symbol s: inst;
        p,q: port;
execute
  forall s: c_linked(s, invoker()) do
    select p: p_owner(s) & p_linked(p, masterport) do
     unlink p masterport end;
    delete s
  end;
end.
```

In the script for change *rem_slaves* first all the components connected to *master* are selected and bound to generic symbol *s*. Then, for every of such component, its port is disconnected from port *masterport*, and then it is removed.

An important aspect here is that both *add_slave* and *rem_slaves* define all their reconfiguration actions only in terms of their formal parameters and of generic symbols, and are therefore independent of names, types or the number of configuration objects. This independence from a particular configuration makes these scripts robust to dynamic evolution. For example, both script are still valid if some new slaves are arbitrarily added or removed. Moreover, because of their genericity, these scripts can be used in any distributed program implemented according to the master-slave paradigm.

5 Implementation

The current implementation of Gerel takes the form of a configuration management tool. This tool, called `gerel`, was built on the top of the Conic[7] reconfiguration support, and performs dynamic reconfigurations on distributed programs running on a network of Sun Workstations under Unix. The main feature of this tool is that it supports both the execution of programmed changes (Gerel scripts) and of ad-hoc changes.

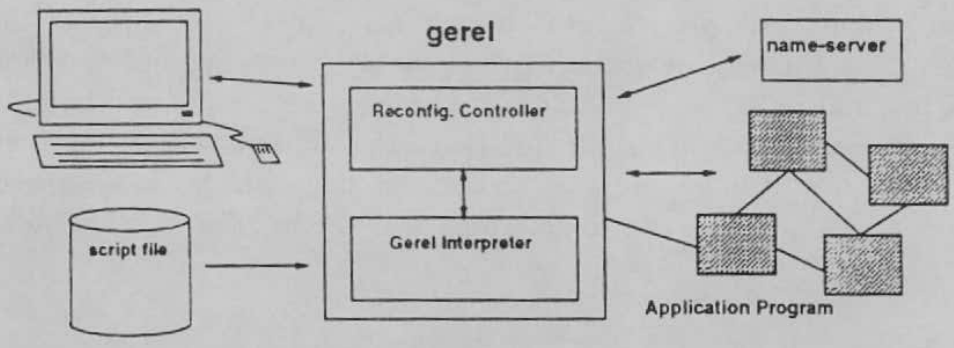

Figure 4: Architecture of `gerel`

As suggested by figure 4, `gerel` uses a nameserver for obtaining the location of the application program components, and is composed of two interacting processes: an interpreter for gerel scripts and a reconfiguration controller. The interpreter process reads Gerel scripts from a file, parses them, performs the static semantic checks, and evaluates the scripts upon requests. The controller process is in charge of managing the interactions with the user, monitoring the current configuration of the distributed application and coordinating the programmed and ad-hoc reconfigurations.

Gerel scripts may be invoked either by the user or by any application process connected to `gerel`. For the first case, the user must enter the command

exec *scriptid* [*arguments*]

where *scriptid* can be either the name of the script or its sequence number in the script file, and *arguments* is an optional list of data type constants or configuration object names. For the latter case, we have implemented a group of standard procedures which perform the connection to `gerel` and make the (remote) invocation of a script and the synchronization with its result transparent for the application component. These procedures have to be imported by every component which is supposed to invoke any Gerel script.

Our configuration manager `gerel` provides also a command by which the user makes available new component types (i.e. executables), which are to be used for further creation of components. With this, it is possible to do functional reconfigurations, e.g. to add and replace component implementations in an executing program.

Besides its feature of guaranteeing the mutually exclusive execution of programmed and ad-hoc changes, `gerel` also monitors all modifications to the configuration, like changes in the availability of workstations, or termination of components. By maintaining all information about the current configuration in local data structures, `gerel` implements an efficient evaluation of the Gerel-SL formulas. Although Gerel-SL formulas are evaluated by brute force, i.e. by testing all the objects of the specified type on all the formula's sub-formulas, it turned out that the major delays are due to the network communication in executing Gerel's basic commands.

6 Related Work

Some other projects have developed reconfiguration languages and support environments. However, none of them has considered the problems of supporting both programmed and ad-hoc reconfigurations, and of describing generic and reusable scripts for dynamic reconfiguration.

Conic[7] was one of the first Configuration Based approaches to support ad-hoc reconfiguration. Conic provides both a textual and graphical interface for the interaction with a reconfiguration manager. However, its facilities for defining change scripts are limited to Unix-shell programming features. Conic's successor, Darwin[6] is a configuration language supporting only programmed changes. Change scripts in Darwin are always shaped to the particular characteristics of the configuration within a so-called composite component. Therefore, Darwin lacks the abstraction mechanisms necessary for implementing generic reconfiguration scripts. For similar reasons, Durra[1] is also suitable only for implementing program-specific dynamic reconfigurations.

Polylith[9] provides a library of primitives for performing dynamic reconfigurations. Since these primitives are called from within the program components, this approach

is only suited for programmed reconfigurations. Moreover, because this library lacks querying primitives, it is not possible to implement dynamic reconfigurations which are adaptive to changes in the program's configuration.

Meta[8] follows a rule-based approach for dynamic reconfiguration, and its reconfiguration language consist of guarded commands. Although Meta's approach has the advantage of supporting an automatic change invocation mechanism, and provides some form of ad-hoc reconfigurations, its language is not suited to implement generic reconfiguration scripts.

7 Conclusions

In this paper we presented the language Gerel, which supports the implementation of generic scripts for dynamic reconfiguration. Such genericity is needed when programmed and ad-hoc reconfigurations are to be supported together. In addition, it is useful for defining scripts which describe common patterns of dynamic reconfiguration, and therefore can be used in many application programs. This is the case, for example, of scripts *add_slave* and *rem_slaves* in section 4.4, which can be used in every program structured according to the master-slaves paradigm.

Since the evaluation of Gerel-SL formulas is exponential in the number of elements of a configuration, the execution of Gerel scripts is only feasible for programs with a small number of components, such as in coarse-grained parallel programs, multi-service servers, and small scale process control systems. For large scale parallel and distributed applications Gerel requires that the program be structured in a hierarchic configuration, where small groups of components are encapsulated in composite components that can be manipulated as simple components. Actually, Gerel has been defined for such hierarchic configurations[3, 2], but our current implementation of Gerel is an interpreter which handles only flat configurations of distributed programs. However, we are planing to implement a Gerel compiler supporting reconfigurations in hierarchic program structures.

Till now Gerel also does not support synchronization of program execution and dynamic changes, and the preserving the program's state across a reconfiguration, but we plan to extend our approach in this direction.

References

[1] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, M.J. Gardner, and R.W. Lichota. Building Fault Tolerant Distributed Applications with Durra. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 128–139. IEE, March 1992.

[2] M. Endler. *A Language for High-Level Programming of Dynamic Reconfiguration.* PhD thesis, Technische Universität Berlin, Franklinstraße 28/29, 1000 Berlin 12, Germany, November 1992. (published as GMD Bericht Nr. 210, by R. Oldenbourg Verlag).

[3] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 68–79. IEE, March 1992.

[4] J. Kramer. Configuration Programming - A Framework for the Development of Distributable Systems. In *Proc. IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro90)*, Tel Aviv, Israel, May 1990.

[5] J. Magee and S.C. Cheung. Parallel Algorithm Design for Workstation Clusers. *Sotware Practice & Experience*, 21(3):235–250, March 1991.

[6] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. In *Proc. of the Int. Workshop on Configurable Distributed Systems*, pages 102–117. IEE, March 1992.

[7] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, SE-15(6), June 1989.

[8] K. Marzullo, R. Cooper, M.D. Wood, and K.P. Birman. Tools for Distributed Application Management. *IEEE Computer*, 24(8):42–51, August 1991.

[9] J.M. Purtilo and C.R. Hofmeister. Dynamic Reconfiguration of Distributed Programs. In *Proc. of the 11th Int. Conf. on Distributed Computing Systems*, pages 560–571. IEEE Computer Society Press, May 1991.