58355

Tools for Distributed Application Management

Keith Marzullo Robert Cooper
Mark Wood Kenneth Birman

Department of Computer Science
Cornell University, Ithaca NY 14853
isis@cs.cornell.edu

Abstract

Distributed application management consists of monitoring and controlling an application as it executes in a distributed environment. It encompasses such activities as configuration, initialization, performance monitoring, resource scheduling, and failure response.

In this paper we describe the Meta system: a collection of tools for constructing distributed application management software. Meta provides the *mechanism*, while the programmer specifies the *policy* for application management. The policy is manifested as a *control program* which is a soft real-time reactive program. The underlying application is instrumented with a variety of built-in and user-defined sensors and actuators. These define the interface between the control program and the application. The control program also has access to a database describing the structure of the application and the characteristics of its environment.

Some of the more difficult problems for application management occur when preexisting, nondistributed programs are integrated into a distributed application for which they may not have been intended. Meta allows management functions to be retrofitted to such programs with a minimum of effort.

Keywords and phrases: Distributed application management, configuration management, distributed operating systems, dynamic reconfiguration, monitoring distributed systems, rule-based systems, ISIS.

1 Managing distributed applications

There is a great deal of difference between a program that performs correctly and one that performs *well.* A correct program does not fail or produce incorrect results, but a program that performs well makes efficient use of resources and behaves predictably over a range of environmental and operating parameters. Writing *distributed* programs that perform well is especially hard. Distributed programs are often expected to run in widely varying configurations, from a single machine to tens or hundreds, and on machines of widely varying performance or from different vendors. Often they must continue operating when some of the machines on which they are running fail.

We call the activity of producing a distributed program that performs well for a given environment *distributed application management.* Distributed application management involves *configuring* the components of the system for a given hardware and software environment; *initializing* the application in an orderly way; *monitoring* the behavior and performance of the application; and *scheduling* work efficiently among the components of the application. An application must be managed throughout its execution, continually reacting to a varying workload, to changes in the environment, and to failures.

Traditionally, application management is either done manually or hard-wired into the code of the application. A person familiar with the internals of the application must continually monitor and control it, and some adaptations can be made only by reprogramming. In practice, many aspects of application management are ignored, resulting in poorly engineered systems that work most of the time, but often exhibit unpredictable performance, become inconsistent, expose partial failures, and prove fragile when even small changes are made to the hardware or software base. In our work, we seek to avoid the deficiencies of this *ad hoc* approach by creating a framework favoring the construction of robust distributed management software and applications, and a set of tools—the *Meta* system—which directly supports our approach.

A distributed computing environment causes many problems for application management, compared with a nondistributed one. The performance data required for system monitoring is distributed throughout the system, making it hard to access. Variable communication delays mean that the data is less accurate, and is difficult to collate. The potential for improved performance through concurrency is one of the attractions of a distributed system, but this concurrency significantly complicates all aspects of the application. For instance, components of the application must be initialized in a well-defined order that observes the dependencies between components. Failures are a fact of life in distributed systems and greatly complicate management. Most applications do not have strong reliability requirements, but unless special efforts are taken, the overall reliability of a distributed solution will be much lower than that of some equivalent nondistributed program, since the frequency of failures is directly related to the number of hardware components.

Additional problems for application management arise when existing nondistributed programs are re-used in a distributed program. Such re-use is an important way of reducing the cost of distributed software development, but the resulting application often does not perform well and may be difficult to manage. For example, the dependencies among the re-used components may be poorly defined, making program startup and recovery difficult, and the kinds of internal state information necessary for performance monitoring and resource scheduling may not be made available by programs that were not intended to run as part of a distributed application. As we shall explain, the Meta approach is well-suited to applications which re-use existing software in this way.

Throughout this paper, we use the term *application program* to mean a distributed application composed of one or more processes. A *process* is a single nondistributed address space, *e.g.* a Unix process, with one or more threads of control. A *component* is a subsystem of the overall application, comprising one or more processes, or occasionally a component of the environment such as a file server or a workstation.

In the rest of this paper, we describe the mechanisms that Meta provides for application management, how they are implemented, and how they can be used. The next section lays out the Meta application architecture. We show how the management functions are separated from the underlying application, and how these two layers interact with Meta. In Section 3 we present an example of a scientific computing application in order to motivate a more detailed presentation of Meta. The following three sections describe the Meta system in detail. Section 4 is about instrumenting the application program with sensors and actuators which will be used for monitoring and control. Section 5 is about describing the structure of the program in terms of its component processes. Meta stores this structural information in a database. Section 6 presents the rule-based language in which management policy is expressed. Meta translates these rules into sequences of sensor and actuator invocations. In Section 7 we discuss issues of atomicity and consistency. We conclude by comparing our approach with existing technologies for distributed application management and noting the directions along which we are extending Meta.

2 The Meta application architecture

The Meta model of a distributed application is depicted in Figure 1. In this model, the management aspects of an application are separated from its major functional parts, and the interface between these two layers is well defined. In this way, modifying the management of an application is easier, and is less likely to impair the correctness of the rest of the program.

We call the management layer the *control program*. While, the underlying application is built using conventional programming tools, the control program can be programmed in a language called *Lomita*. The Meta system is interposed between the control program and the application, and presents the control program with an abstract view of the application and the environment in which it runs. As shown in Figure 1, not all communication between the control program and the application need go through Meta. The structure of the application

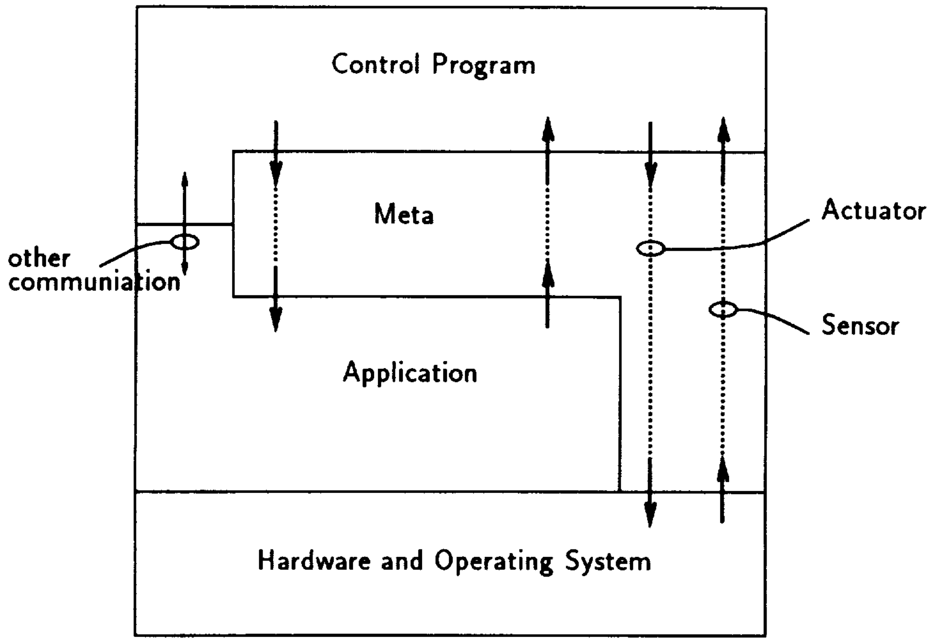

Figure 1: Meta application architecture

program—its constituent components and their interconnections—is declared to Meta in the form of an object-oriented data model.

The control program observes the behavior of the application by interrogating *sensors*, which are functions that return values of the application's state and its environment. Similarly, the behavior of the underlying application and its environment can be altered by using procedures called *actuators*. Meta provides a uniform, location-independent interface to both built-in and user-defined sensors and actuators. This interface also provides ways to combine multiple sensor values in order to compute more complicated sensors or to provide tolerance of failures.

The particular sensors and actuators that are used depend on the application being controlled. Typical sensors could include:

- *The CPU utilization on a machine.* This is one of a number of built-in sensors provided by Meta.
- *The load on an application component.* This might be the size of the component's input job queue. Such a user-defined sensor can be implemented by supplying a procedure in the component that will calculate the value when needed, or by directly monitoring a variable in the process's address space.

- *The total throughput of the application.* This might be computed by combining the data from a number of more primitive sensors located in each component of the application. Meta provides ways to specify such derived sensors and associates well-defined semantics with them.
- *The status of a component, i.e. failed or operational.* Meta provides built-in sensors that test for the existence of a process, but one can also supply a user-defined sensor that implements an application-specific liveness criterion.

Typical actuators could include:

- *Change a process's priority.* This might be a built-in actuator used to control fine-grained scheduling.
- *Change a lightweight thread's priority.* This might be done by modifying some variable within a designated process's address space, or by invoking a user-specified procedure in some process.
- *Restart a failed process.* This might involve selecting a machine on which to restart a failed process, initializing the process, and integrating the new process into the pre-existing components.

Meta offers several interfaces by which programs can query sensors and invoke actuators. The basic interface is from one of the programming languages in which the application may be written: currently C, Fortran or Lisp. Other higher-level interfaces include the control language, *Lomita*, which combines a real-time interval logic with a rule-based syntax for querying sensors. As described later, the semantics of Lomita cleanly captures the temporal nature of significant complex events in the distributed application. Meta executes Lomita commands using a fault-tolerant distributed interpreter.

Meta is implemented using the ISIS distributed programming toolkit [Birman and Joseph 1987]. ISIS provides primitives for reliable programming including process groups and ordered atomic multicast. On top of these primitives, ISIS provides a toolkit of solutions to common sub-problems in distributed computing such as distributed synchronization, resilient computation, and logging and recovery.

3 An example application: Seismological analysis

To make the discussion of sensors and actuators more concrete, we present an example application and show how it is managed within the Meta framework. The application, NuMon, is a seismological analysis system intended primarily for monitoring compliance with nuclear test-ban treaties. A real nuclear monitoring system, on which this simplified example is based, is being developed using Meta and Isis by Science Applications International Corporation.[1]

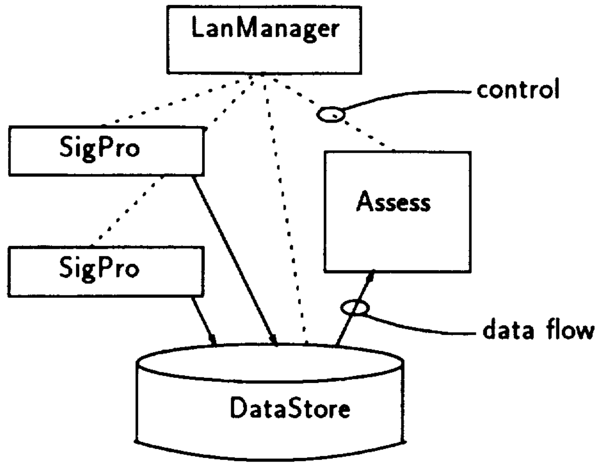

Figure 2: Simplified seismological monitoring application

NuMon consists of four component process types (see Figure 2). The **SigPro** processes collect seismological data and perform signal processing on it. The much smaller resulting processed data is stored in the **DataStore**. The **Assess** process is an interactive expert system that interprets the data produced by multiple **SigPro** processes and forms hypotheses about various events. To confirm these hypotheses, further tasks are assigned to the **SigPro** processes. **Assess** stores its event classifications in the **DataStore**. The structure of the real application is much more complex, with several kinds of **SigPro** processes that are created in response to different events detected by **Assess**.

The **LanManager** contains the control program for **NuMon**. During normal operation the control program schedules work efficiently among the available machines. When individual machines crash it reapportions work automatically, and when total failure occurs it restarts the application. In the remainder of this section we will relate the issues addressed by the **LanManager** to the Meta sensor/actuator model.

3.1 The LanManager

The **LanManager** embodies the control policy for configuration, scheduling and response to failures. This policy is expressed in the form of a rule base. Its second function is to support

[1]DARPA Contract MDA972-88-C-0024.

a graphical user interface which displays the current system state and allows the user to alter policy rules or issue commands to tune the performance of the application. Thus the **LanManager** takes a semi-automatic approach to application management. Common activities such as system startup and shutdown and individual machine failures can be handled without human intervention. But other, perhaps unforeseen, circumstances can be given to the user to handle. Typical examples include a persistent software error that causes some component to crash no matter how many times it is restarted, or a full disk.

3.2 The SigPro performance sensors

The **SigPro** processes are computation engines that service requests from **Assess** and interactive users of the system, and process the input data. The **SigPro** processes derive from large sequential Fortran programs developed by seismologists with little experience in distributed programming. A crucial requirement, therefore, was that application management functions be easy to add to these large programs without requiring substantial modification of the Fortran code.

In order to schedule work among these tasks and start auxiliary **SigPro** processes when needed to improve throughput, each **SigPro** exports two performance sensors: the *load sensor* and the *backlog sensor.* The backlog sensor measures the backlog of input data to be processed. It corresponds to a program variable in **SigPro**. The load sensor is a procedure that returns a measure of **SigPro** load by combining load factors such as the current size of the input task queue and the recent activity within the process. The interface by which these sensors are made available to Meta is explained in Section 4.

The **LanManager** will typically examine sets of sensor values, such as the average of all the **SigPro** load sensors, or the maximum of a load sensor over the last two minutes. These kinds of operations are directly supported by Meta through the notion of *derived sensors*, which may be computed from primitive sensors values using a number of built-in functions. Meta addresses the issues of sampling skew, imprecision and dynamic group membership changes that arise when deriving sensor values in a distributed computing environment.

3.3 SigPro fault tolerance

If the **Assess** process fails (for example the machine on which it is running crashes), a new copy of **Assess** should be started elsewhere. The **LanManager** must therefore monitor the **Assess** process, choose a new location to restart **Assess** after a crash, and reconnect this new process to the **SigPro** and **DataStore** subsystems. Then the work that was in progress at the crashed **Assess** must be assigned to the newly created **Assess**.

The **LanManager** uses the capabilities of Meta and ISIS to accomplish these actions in a fault-tolerant manner. The **Assess** process uses the ISIS spooler tool to log its actions and to periodically checkpoint its state. In this way, when it fails there is a stable record of the tasks it was engaged in and its progress. Some of the built-in Meta sensors are used to detect failure, and to identify suitable alternative machines on which **Assess** can be run. Meta actuators are invoked to restart **Assess** with the correct spool file, and to re-establish connections to the rest of the application.

3.4 LanManager fault tolerance and atomicity

The **LanManager** itself must tolerate failures. If it crashes, it can regain much of its state by sensing the application and environment through sensors. Other important state can be checkpointed using a tool such as the ISIS spooler.

However, if the **LanManager** fails midway through the sequence of sensor and actuator invocations intended to restart an **Assess** process, we may find that two copies of **Assess** or none at all were started. To solve this, each control program rule is executed *atomically* by the Meta rule interpreter, using the facilities of the underlying ISIS system. Thus the programmer is able to concentrate on writing a consistent set of policy rules for the **LanManager**, leaving most of the issues of fault tolerance in their execution to the interpretation algorithm supplied by Meta. However in the case where a rule undertakes real-world actions, the issue of atomicity is more complex. A discussion of this is deferred to Section 7. Having described a simple distributed application and its management requirements, we will describe the Meta system in detail.

4 Instrumenting a distributed application

There are three steps to using Meta to manage an application like **NuMon**, and these are described in the next three sections. First, the programmer instruments the application and its environment with sensors and actuators. These functions, along with a set of built-in sensors, provide the interface between the control program and the application program. Second, the programmer describes the structure of the application using the object-oriented data modeling facilities of the Lomita language. Finally, the programmer writes a control program referencing this data model. The control program may be written as a Lomita script or in a conventional language, such as C, embedded with calls to the Lomita interpreter. The control program can make direct calls on sensors, actuators and other functions in the data model and use higher level policy rules that specify a set of conditions over sensors and the action to take when a given condition becomes true. Meta can be thought of as an object-oriented temporal database,[2] but where the application and environment provide the data values. Figure 3 shows this functional layering of the Meta system.

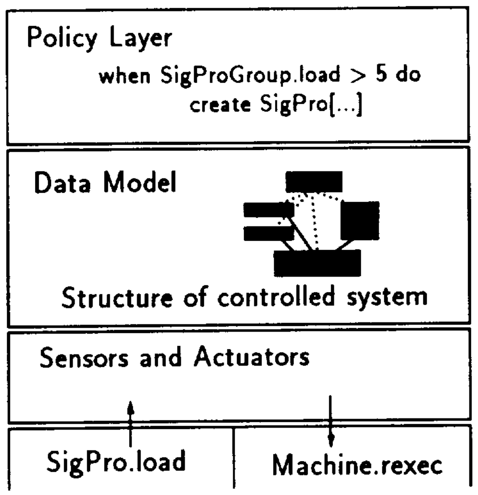

Figure 3: Meta Functional Architecture

We describe these layers from the bottom up. This section describes how sensors and actuators are specified and used by Meta. Section 5 describes Meta's data modeling facilities and Section 6 describes how to write the control program of a distributed application.

4.1 Sensors

A Meta sensor represents part of the state of the monitored application. Each sensor is identified by the kind of application component it monitors (*e.g.* SigPro), the kind of value it monitors (*e.g. backlog*), and the instance of the component it is monitoring (*e.g.* SP1). A sensor can be polled in order to obtain its current value, and a ***watch*** can be set up that alerts the client when the sensor value satisfies some predicate.

Built-in sensors

Meta provides a set of built-in sensors corresponding to information that can be obtained directly from the environment. Examples of these are sensors which return statistics such

[2]By *temporal* we mean simply that the database supports a notion of time. In the terminology of Ahn and Snodgrass [1985], Meta can be thought of as either a historical or a rollback database because the enterprise being modeled by Meta is monotonic.

as the memory and processor usage of a process, obtained from Unix. Furthermore, Meta provides the read_var sensor for reading the values of certain kinds of global variables in an active process. This is implemented with the Unix system call that permits access to another process's address space for debugging purposes. The alive sensor is a built-in sensor that returns *false* if the component it is monitoring has failed.

User-defined sensors

Meta allows the programmer to define and implement primitive sensors. Such sensors correspond to dynamic properties of the application whose values cannot be supplied by simply polling the state of the underlying operating system. Sensors in this class are registered with Meta at run-time. Each application process that contains a user-defined sensor must connect itself to Meta when it is started up by calling meta_init:

```
meta_init (name, instance);
```

The name argument is the component type name (*e.g.* SigPro) and instance is an instance identifier (*e.g.* SP1). An instance identifier can be a uniquely generated name, as in this example, or a more human-understandable quantity such as a machine name. It is the responsibility of the application writer to ensure that instance identifiers are unique for a given component type.

Having issued this call, the process may explicitly export sensors. For example, the following C procedure implements a simple SigPro load sensor:

```
int work_load (int *value)
{
    value = work_queue_size + 2*mbytes_in_use;
    return (0);
}
```

where work_queue_size and mbytes_in_use are global variables maintained elsewhere in the process. This sensor procedure is made available to Meta by calling the new_sensor procedure in the Meta run-time library:

```
s_id = new_sensor (work_load, "load", TYPE_INTEGER, 100);
```

The new_sensor procedure returns an internal identifier for the sensor instance that the client can use for later communication with Meta. The first two arguments establish the binding between the sensor name and the procedure that returns the sensor's value. The third argument specifies the type of the sensor's value and the fourth argument defines the *minimum polling interval* in milliseconds. The sensor should be polled at this (or a shorter) period to avoid missing significant events. Meta uses the polling interval to determine the sampling frequency needed to implement the *watch* operator which waits for a simple predicate on the sensor value to be satisfied.

If the minimum polling interval is specified as zero, then there is no *a priori* minimum polling interval. In this case, Meta must be notified when the value being sensed changes in a significant way. The monitored program does so through the call:

```
check_sensor (s_id);
```

This procedure can be called by the application as a hint to Meta even when a non-zero minimum polling interval was specified.

If the value of a sensor is simply the contents of a single global variable in a process, which is the case with the SigPro backlog sensor, then Meta's built-in read_var sensor can be used. This avoids explicit calls to meta_init and new_sensor in the code for the process, which simplifies adding application management to existing programs.

With both user-defined sensors and the read_var sensor, some thought should be given to synchronizing changes to the variable by the process and accesses to it via the sensor. In the case of sensor procedures, it is possible to use programming techniques such as mutual exclusion locks and semaphores to achieve this. With the read_var sensor, however, the only synchronization provided is the hardware memory interlock. Thus the global variable's value should be represented in one word or less (where "word" is the unit over which the memory interlock operates).

4.2 Actuators

Actuators are named and referenced in the same fashion as sensors. Meta supplies a number of *built-in* actuators. These include an actuator to start up a process with a given argument list, and a global variable actuator which allows a global variable in a process to be modified. As is the case with the global variable sensor, the interactions between the actuator and the internal operations of the process are synchronized only by the hardware memory interlock.

A process can be instrumented with user-defined actuator procedures in a similar way to sensors. For example, we might have a SigPro.reset_file actuator:

```
void reset_file (char *file)
{
    cancel_current_work = TRUE;
    new_file = file;
}
```

that causes the process to stop work on its current data file, and begin work on a different file. The SigPro program declares this actuator by calling:

```
reset_id = new_actuator (reset_file, "reset_file", ACT_SERIAL);
```

The last argument specifies whether multiple concurrent invocations of the actuator are permitted (ACT_CONCURRENT), or as in this case, that multiple actuator calls must be mutually excluded in time (ACT_SERIAL). Clearly, triggering the **reset_file** actuator also raises questions of atomicity to failures. We defer this discussion to Section 7.

5 Describing an application and its environment

Once an application is instrumented with sensors and actuators, the programmer writes a control program to manage the application. The control program is usually written in the *Lomita* language. Lomita comprises two sub-languages: one is an entity-relationship data modeling language used to describe the structure of the application and the sensors and actuators with which it is instrumented. The other is a rule-based language for expressing management policy rules. In this section, we describe the Lomita data model, and show how sensors can be combined to form more complex sensors. In the following section we present the Lomita rule-based language.

5.1 The Lomita data model

In order to describe an application, the programmer develops a schema using the Lomita data modeling language. For exposition, we show in Figure 4 part of the Lomita description of the seismological monitoring application.

Components in the application and the environment are modeled by *entities*, following entity-relationship database terminology [Chen 1976]. An entity is similar to a record or object in a programming language. The example includes a **Machine** entity that models a computer, and a **Process** entity that is a process running on a computer. Since one of the functions of the control program is to allocate environment resources such as processors and peripherals to the application, all these must be represented in the data model. Like entities are grouped together in *entity sets* which are similar to data types or classes.

The "fields" or "instance variables" of an entity are called *attributes*. Lomita supports three kinds of attributes: *properties*, *sensors*, and *actuators*. Sensors and actuators were introduced above. Properties are static attributes whose values are stored in an internal database, rather than being sensed directly from the application or its environment. For example, the processor-type of a machine is a property. One or more of an entity set's

```
Machine: external entityset
   attributes
      key name: string
      sensor load: real
      sensor users: {string}
      sensor jobs: {string}
      actuator exec(string, string)
   end
end

Process: dependent Task entityset
   attributes
      key instance: UID
      property params: string
      property executable_file_name: string
      sensor alive: integer
      sensor read_var(string): any
      actuator write_var(string, any)
      actuator exit
   end
end

Task: relation Process - > Machine
   operations
      create
         Process.instance := new_uid()
      do
         Machine.exec(executable_file_name, params)
      end
      delete do Process.exit() end
   end
end

SigPro: Process entityset
   attributes
      sensor load: integer
      property executable_file_name: string := "/usr/numon/bin/sigpro"
      property params: string
      actuator reset_file (string)
   end
end

SigProGroup: SigPro aggregate
   attributes
      key program_name: string
      sensor load: integer := mean(Process.load)
   end
   select all
end

FreeMachines : Machine aggregate
   attributes
      key unique
   end
   select m suchthat size(Machine[m].jobs) <= 1 & Machine[m].load <= 0.5
end
```

Figure 4: Data model for seismological application

properties, called the *primary key*, uniquely identifies each entity. For instance, the primary key of the machine entity might be its name.

Lomita provides ways to specify a rich set of connections and groupings between components. A *relationship set* specifies one-to-one, many-to-one and many-to-many relations between components. For instance, there is a many-to-one relationship between processes and the machine on which they run. Thus the definition for the entity set Process specifies that a Process entity is dependent upon the existence of some Machine entity through the relationship set Task. A dependent entity may not exist by itself; it can be created only by creating a relationship in the specified relationship set. In the seismological monitoring application, a process's sensors can not be accessed unless it is running on some machine.

The definition of an entity set or a relationship set may contain *data model operations* to be executed when an entity or relationship is created or deleted. For example, Figure 4 shows create and delete operations for the Task relationship. The create operation ensures that the Process.instance property is initialized, and then invokes the Machine.exec actuator to start up the process. Not all entities need contain creation and deletion operations. Those that do not, for instance the Machine entity set in Figure 4, must be added and removed outside the control of Meta, via the database.

Often, one entity will be an extension of another. This is the case with the SigPro entity set in Figure 4, which is based on the Process entity, with a few additional fields. This is modeled in Lomita with a *subtype* entity set. The subtype inherits all the original entity's attributes and optionally adds new ones. An inherited attribute that had type any in the parent may be refined to a specific type in the new entity set.

An *aggregate* groups together related entities into a single new entity, which may define attributes of its own. For example, the SigProGroup entity set is an aggregate consisting of all SigPro processes. The FreeMachines aggregate collects together a subset of the Machine entities that are lightly loaded. The members of an aggregate may be specified by a select operation, and must be drawn from the same entity set.

Although Lomita describes the modular structure of the program that it is controlling, Lomita itself lacks a notion of modularity. We wish to allow several Lomita control programs to co-exist within one distributed system without name conflicts arising over the names of entities and relationship sets. But we also want Lomita programs controlling related but independent subsystems to be able to interact in well-defined ways. To this end we are experimenting with ways of structuring multiple Lomita programs. This may be possible entirely within the current Lomita language definition, or it may require additions to the language. In the meantime, multiple Lomita programs can be completely insulated from each other using the facilities of Isis.

5.2 Derived sensors

Primitive sensor values obtained from the application program can be combined in the form of *derived sensors*, which provide a higher-level view of the behavior of the program. A derived sensor may combine values from a number of different primitive sensors, or from a single sensor over a time interval. Derived sensors are defined by simple arithmetic expressions augmented with some more powerful combining operations.

The following SigProTask relationship set definition illustrates the use of an arithmetic expression to define a sensor giving the ratio of SigPro load to the Machine load:

```
SigProTask: Task relation SigPro -> Machine
   attributes
      sensor load_ratio: real := SigPro.load / Machine.load
      ...
   end
end
```

The SigProTask relationship set consists of (Machine, SigPro) pairs. Since the load_ratio sensor is composed from primitive sensors in these two entity sets, it is natural that the sensor definition appears in the SigProTask relationship set.

function	description
size(s)	Number of elements in set s
max(s)	Maximum value of numeric set s
min(s)	Minimum value of numeric set s
range(s)	max(s) – min(s)
mean(s)	Mean value of numeric set s
deviation(s)	Standard deviation of numeric set s
choose(a)	Return an arbitrary element of aggregate a

Table 1: Meta sensor functions

In addition to simple arithmetic operations, Lomita provides the functions listed in Table 1 that operate over sets of values. With the exception of choose, these functions have three overloaded meanings. First, they may be applied to a set returned from a set-valued sensor such as Machine.users. The following Machine entity set definition includes a sensor that gives the current number of users on a machine during the last ten minutes:

```
Machine: entityset
   attributes
      key name: string
      sensor users: {string}
      sensor n_users: integer := size(users)
      ...
   end
end
```

In their second form, these functions take two arguments (s and t). The function is computed over the set of values that the sensor s took during the last t seconds. For example, the following SigPro definition contains a derived sensor **high_load** that is the maximum load in the last ten minutes.

```
SigPro: Process entityset
   attributes
      sensor load: integer
      sensor high_load: integer := max(load, 600)
      ...
   end
end
```

The third interpretation applies to the individual sensors of the components comprising an aggregate entity. Thus **max(s)** is the maximum value of the sensors s of each component of the containing aggregate entity. The following **SigProGroup** aggregate definition includes a sensor that gives the maximum load of all the **SigPros** in the group during the last ten minutes:

```
SigProGroup: SigPro aggregate
   attributes
      key port: integer
      sensor max_load: integer := max(high_load)
      ...
   end
end
```

5.3 Meta-tarsis: A prototype schema

In some ways, Lomita is *too* general to be convenient. The structure of most distributed applications will include common components such as machines, processes, process groups and so on. We are developing a Lomita data model called *Meta-tarsis* that can be used as a prototype for most distributed applications, and modified as necessary for a particular setting. Meta-tarsis provides some basic entity and relationship sets that will be common to most application areas. It includes entities for workstations, file servers, networks and processes. There are relationships for common process organizations such as a server and its clients, a pool of processing slaves controlled by a single master process, and a group of processes replicated for resilience.

Meta-tarsis contains a much more complete notion of **Machine** than the example just presented. It includes attributes such as physical location, owner, processor type, floating point coprocessor type, operating system type and version, and local sub-network identification. These permit a reasonably intelligent allocation of machines to tasks, for instance, by assigning floating-point-intensive jobs only to machines with a floating point coprocessor. The **Process** entity set contains all the built-in sensors and actuators mentioned in Section 4.

There is a relationship set **FailureClass** which is intended to identify machines that may experience common-mode failures. For instance, two diskless workstations that share a common disk server machine would both fail if the disk server failed. When replication is used to improve reliability it is vital that the replicas execute on machines in different failure classes. These kinds of constraints can be expressed using a **select** clause such as that which appears in the **FreeMachines** aggregate in Figure 4.

6 Expressing policy rules in Lomita

We now describe the highest level of Meta: the rule-based control language. Using the Lomita data model of the application, the programmer writes a description of the intended behavior of the system consisting of a set of Lomita policy rules. A policy rule is written using the statement:

when *condition* **do** *action*

Intuitively, this statement declares that when the specified condition is observed, the stated action is to be taken. The condition part of each rule is a predicate that is expressed on the underlying data model. The action component is simply a sequence of actuator invocations and data model operations.

Part of the control description for the **NuMon** application is shown in Figure 5. The **when** rule states that when the number of **SigPros** becomes too low or their collective load becames too high and remains continuously high for at least sixty seconds, a new **SigPro** is to be started. The form of conditions is limited to simple predicates, optionally appearing in a temporal logic expression. By limiting conditions to this form, they may be easily translated into watches on sensors. Both primitive and derived sensors may be referenced.

```
when size(sig_pro_group[program_name]) < 2 ior
    (during SigProGroup[program_name].load > 5 to 60
        always SigProGroup[program_name].load > 5)
do
    create Task (
        Machine := choose(FreeMachines)
        Process := create Sigpro (params := "/usr/numon/data" )
    )
end
```

Figure 5: Rule for creating new **SigPros**

The body of the **when** statement specifies a sequence of actions to be carried out. An action may be an invocation of an actuator or a **create** or **delete** statement, which will invoke the appropriate operation associated with that entity set. The **create** expression provides values for any properties that are not initialized in the entity declaration. While Lomita permits only a linear sequences of actions, more complex flow of control can be achieved either by triggering multiple rules with more complex **when** conditions, or by writing more complex actuators (*e.g.* in the C language). In the extreme, the control program can be written almost entirely in C with embedded calls to Lomita for condition evaluation.

6.1 The Lomita rule interpreter

Lomita control programs are executed by an interpreter, which is replicated in order to provide fault tolerance. To enforce rules, Lomita needs to communicate with the individual sensors and actuators in the application processes and the environment, such as the **Machine.jobs** and **SigPro.load** sensors. Lomita uses ISIS process groups to structure this communication.

Lomita maintains a database about the current structure of the application. This database contains the members of entity sets, the tuples in each relationship set, and the membership lists for each aggregate. Most entities correspond to processes in the application being controlled. In our example, a **SigPro** entity is a process running on some computer. The **Machine** entity is implemented by a process that contains procedures for the sensors and actuators in the **Machine** definition. One copy of this **Machine** process must be started up on each computer which Meta will manage. For each of these entities, the database keeps a reference to the process, and maintains the values of any properties associated with the entity. Entities that do not correspond to processes, along with relationship sets and aggregates, are maintained entirely within the database.

6.2 Mapping entity sets to process groups

Entity sets that correspond to processes are represented by ISIS process groups. In our example, all the **Machine** entities belong to a process group with a name such as **/Meta/Machine**. All groups associated with Meta have **Meta** as the first component of their names, followed by the set name (**Machine** in this case).

An ISIS group provides an easy way to organize the elements in an entity set and to communicate with them. ISIS multicast is used to simultaneously access all copies of a sensor or actuator in an entity set. ISIS multicast is atomic: an actuator invocation is received by all group members or by none of them. In addition, concurrent multicasts are ordered consistently at all group members. ISIS group semantics also ensure that Lomita has accurate knowledge of the current membership of an entity set. Changes to the membership of a group, either planned, such as when a new entity joins a group, or unplanned, such as a failure of an entity process, are serialized with group communication.

The group structure of our example is shown in Figure 6. The replicated Lomita rule interpreters all belong to a process group named **/Meta/Lomita**, so that application processes have a well-known address to which to send sensor updates. The Meta database resides in the same processes as the Lomita interpreters and thus exhibits identical fault tolerance.

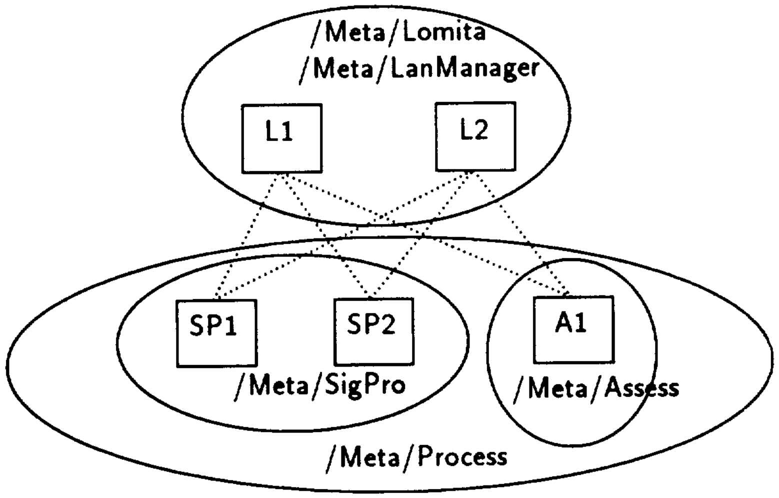

Figure 6: Lomita process group structure

6.3 Rule interpretation

The details of Lomita interpretation will be explained by describing the execution of the rule shown in Figure 5. The **when** condition has two parts which are *or*'ed together. The first notices when the number of **SigPros** drops below two, and the second notices when the composite load on the **Sigpros** exceeds five for one minute or more. To interpret the first condition, a list is formed of the tasks belonging to the **SigProGroup** aggregate and the **size** function is evaluated on that aggregate. Lomita re-evaluates **size** whenever an entity is added to or removed from the corresponding entity set, for example, when a process is created or terminates. Process creations are detected via the **Process.create** operation, and process terminations with the **Process.alive** sensor.

The other condition is the interval temporal logic expression:

```
during SigProGroup[program_name].load > 5 to 60
   always SigProGroup[program_name].load > 5
```

Such expressions are converted to finite state automata, in which the state transitions occur when part of the predicate becomes true or false. This expression involves an aggregate sensor whose value Meta maintains by periodically polling the members of the group. Meta then applies the aggregate function in order to compute the value of the aggregate sensor. The finite state automaton for this expression is shown in Figure 7. The first state sets a watch on the aggregate sensor in order to be notified when the load goes above a threshhold. When this event occurs, the automaton transfers to the next state, arms an interval timer, and waits for either the interval timer to go off or the watch to arrive noting that the load has gone below the threshhold. If the next event received is the timer event, then the guard is satisfied and the action is initiated.

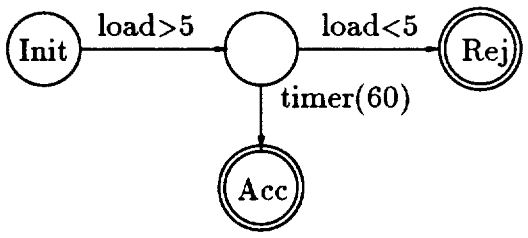

Figure 7: Finite State Automaton

If either the temporal logic expression or the condition on process group size is satisfied, the action part of the rule is obeyed. In this case, Lomita uses Meta to obtain a lightly-loaded machine An entry in the **Task** relationship is created and Lomita attempts to run the job on the chosen machine. The action is synchronous, in that Lomita waits for actions to be completed to enable it to detect failures. Upon failure the action is retried.

6.4 Fault tolerance of the Lomita implementation

The Lomita interpreter is replicated to provide fault tolerance. One of the Lomita processes is chosen to have primary responsibility for executing the control program, while the extra processes are backups. These processes cooperate using the ISIS *coordinator-cohort* mechanism. The coordinator is the active party, sending out messages to effect the desired application behavior. The other processes, cohorts, remain passive unless the coordinator fails, at which point one of them is chosen as the new coordinator. While passive, the cohorts receives all messages involving the coordinator and maintain an up-to-date view of the system being monitored.

If the Lomita interpreter is replicated n times this architecture can survive $n-1$ process failures. The degree of fault tolerance required will vary depending upon the application. Total failure occurs if all n Lomita processes fail. Lomita can recover from total failures if the database is checkpointed to disk periodically, and updates to it are logged to disk. Meta leaves the activation of this mechanism as an option for the application programmer, because there are two other reasonable ways to cope with total failure of the control program that avoid the need for logging.

First, each application process can monitor for the total failure of the /Meta/Lomita group. If Lomita fails then the application could terminate itself. Simply restarting Lomita from its initialization files would restart the application in an orderly way. By having the application terminate itself, there is no possibility of "orphan" application processes surviving the failure of the control program. Such orphan processes would generate much confusion when the application was restarted.

The second option is to leave surviving application processes running, and arrange that when the control program restarts it first searches for existing application processes from a previous execution. In our example it would do this by looking for any existing members in the /Meta/SigPro group. Once the orphans had been identified, they could either be terminated, or re-initialized and integrated in the new instance of the application. Currently we have not provided support for orphan detection in the Lomita language, however an ISIS program could perform this function at application startup.

7 Consistency and atomicity considerations

With Meta, distributed application management is a soft real-time reactive system. When a condition becomes true, the control program notices this and reacts in a timely manner. The control program should incorporate a model of how long its actions should take to produce noticeable changes in the application that it is controlling.

A number of properties are required of the Meta implementation to satisfy this model. Meta must provide a predictable and preferably short delay between the occurrence of some condition and its notification to the control program. For compound conditions, the set of sensor value readings must be *consistent*. And, when multiple conditions might be triggered concurrently there is a need for *atomicity*.

7.1 Real-time consistency

For the kind of monitoring involved with application management, a weak kind of real-time consistency is needed. Several factors can affect the level of consistency achieved. First there is the inherent inaccuracy in the sensor itself. Then there is the delay in communicating new sensor values to Meta. Sensor values can be timestamped at their source so that no matter what the delay, the time at which the sensor reading was taken will be known. However inaccuracies between clocks on different machines (really just another kind of sensor inaccuracy) introduce further uncertainty. To accommodate these inaccuracies, Meta represents a sensor value (or a clock value) as an interval in which the true value lies. Thus a sensor reading is treated as a two-dimensional interval: for a certain time period, the sensor value is known to have fallen within a measured range. Under the assumption that sensors are piecewise continuous, Meta can interpolate an estimate of the value that the derived sensor took during a given interval of time. This approach is discussed fully elsewhere [Marzullo 1989, Marzullo and Chew 1990].

Decision making in the presence of inaccurate sensors represented by intervals is more complex than with point sensor values. With many sensors, taking the midpoint of the interval and treating it as a point sensor value is appropriate. But this is not always the case. Suppose we have a *diskspace* sensor that returns the number of bytes free on a file system. A rule that is triggered when the available diskspace reaches a certain level allows two interpretations: that the action be taken if at *least* the given level of diskspace is available, or if at *most* the given level is available. A rule that waited for the disk to become almost full before deleting temporary files might use the upper bound of the interval. Conversely a rule that waited for the required space to become available before performing a large file transfer should use the lower bound of the diskspace interval before proceeding. Meta has logical operators for both of these interpretations. The user is expected to select the one appropriate to the situation. Distributed evaluation of the resulting expressions is discussed by Wood [1991].

Proper handling of issues such as the accuracy intervals for sensors and the latency before actuators take effect is essential for producing a robust control program.

Real-time consistency on Unix and ISIS

In the Unix environment in which Meta runs, clocks are approximately synchronized by the Berkeley time protocol [Gusella and Zatti 1985], which achieves an accuracy of about 1 millisecond. Although ISIS does not provide real-time message delivery guarantees, the communication delay imposed by the ISIS multicast primitives is normally around 10 to 30 milliseconds. However Unix provides unpredictable scheduling delays and virtual memory waits. These can affect the responsiveness of ISIS and the application itself to the notification of new sensors values. Thus, to be conservative, we assume that the end-to-end sensor latency can be as large as several seconds. It remains for the application programmer to specify the accuracy intervals for sensors, and the minimum meaningful polling period. For the kinds of applications which use Meta, polling intervals of several seconds, and up to a minute are reasonable. In many cases the dominant consideration is that producing sensor values should impose little overhead on the underlying application.

Although real-time responsiveness is clearly important, Meta currently offers no explicit real-time guarantees. Instead, statistics characterizing the typical system response times under various conditions are provided, and the designer is expected to ensure that the application operates well away from the region at which deadlines might be violated. We see little chance for improving on this approach as long as Meta remains a Unix application. We are considering porting ISIS and Meta to an operating system kernel such as Chorus [Armand *et al.* 1989] that provides support for real-time scheduling. This should reduce these performance figures and make them more predictable. However real-time multicast [Cristian *et al.* 1986] and group membership algorithms are required to achieve the full potential of the Meta sensor model.

7.2 Virtual synchrony

In addition to real-time consistency, Meta provides a high degree of logical consistency through the semantics of ISIS. By logical consistency, we mean a consistent total ordering on events such as sensor readings, actuator invocations, and process failures. We call the model that underlies ISIS ***virtual synchrony*** because events appear to happen one-at-a-time. The ISIS implementation permits events to overlap or be re-ordered where such changes have no effect on the correctness on the application.

The consistent ordering of failures with other events is particularly important. For instance, we can be sure that after the control program learns of a process failure, it will not receive any further sensor readings from the process. This simplifies the conditions attached to rules in the control program. The ramifications of this model and the synchronization protocols needed to implement it efficiently are discussed by Birman and Joseph [1987]. Details of the adaptation of this model to cover the logical consistency issues that arise in Meta are presented by elsewhere [Wood 1991].

A difficulty with virtual synchrony is that it requires the system to occasionally reorder events in an execution to fit the logical requirements of the model. For example, it proves necessary to impose an ordering on failures that might not correspond to the order in which failures actually occurred, or to pretend that a failed process did (or did not) see some event before it failed. If there is external evidence establishing otherwise (such as a file written by a process just before it died) or if processes have some way of communicating outside ISIS, then the illusion of virtual synchrony will be violated. In most of our previous uses of ISIS this has not been a problem since all communication takes place via ISIS. It is possible that there are distributed control algorithms that cannot be represented in Meta, as it currently exists, because of its use of virtual synchrony. We leave this as an open question for future study.

7.3 Atomicity

When Meta operations trigger actions, problems of atomicity are raised. For example, suppose a Meta rule reacts to a failure by selecting a lightly loaded machine, reserving it, and instantiating a program on it. If several such rules are triggered simultaneously, one must prevent the machine being reserved more than once.

Meta provides a simple default policy for controlling concurrency, in which the action part of each rule is a critical region protected by a single mutual exclusion lock for the entire control program. Thus, in situations where a rule might trigger a sensor-actuator feedback loop, rules that monitor sensors or trigger actuators will be strictly serialized. In particular, this is the case for the machine reservation rule mentioned above.

The style of locking used by Meta can result in inefficient synchronization patterns when sets of rules do not in fact overlap on the sensors and actuators that are referenced. More sophisticated users may choose to disable Meta's mutual exclusion lock on actions, and provide their own mutual exclusion primitives coded as actuators.

8 Extending Meta to physical control systems

So far, we have discussed the Meta sensor/actuator model in the soft real-time world of managing software applications. Sensors can also correspond to measured quantities in the physical world, in which case Meta might be used to control the external environment. For instance, a system for monitoring the environment in a machine room might monitor air temperature and chilled water pressure. If the air conditioning system becomes overloaded, the least critical pieces of computer equipment could be turned off in an orderly way, in the hope of offering a degree of continued operation. A more ambitious example would be controlling an industrial plant.

We believe that there are many applications in which Meta could be used to monitor and control real-world physical processes, and which would be difficult or impossible to solve in the absence of such a tool. Examples of areas in which our user community is using or experimenting with Isis and Meta include stock and bond trading systems, factory-floor job scheduling in an industrial plant, clinical information systems in a hospital, and management of the orbiting space laboratory NASA has proposed to build. All these applications have real-time, external consistency requirements that are much more stringent than the kind of application management described in this paper.

Physical quantities have different characteristics from internal, computer generated measures. Their values are often taken from a continuous domain and change slowly and predictably. Whereas sensors corresponding to internal measures normally fail by crashing, physical sensors may also fail by giving erroneous values. To accommodate applications of this sort, the Meta architecture includes techniques for using collections of inaccurate physical sensors to implement more reliable, accurate abstract sensors. This material is explored elsewhere by Marzullo [1989].

9 Comparison with other control technologies

The problem of distributed application management has been largely ignored in the past and seldom has system-level support for it been provided. In this section we look at related work in the area. A significant amount work has been done on tools for monitoring the behavior of distributed applications, while comparatively little attention has been paid to mechanisms for controlling distributed applications. We are aware of no work that combines the two: using the results of monitoring to automatically control an application.

9.1 Distributed performance monitoring and debugging

Many systems have been developed for instrumenting distributed programs to obtain performance figures, or for distributed debugging. Several researchers have recognized the benefits of viewing the data gathered from monitoring as a temporal database. Foremost among these is Snodgrass [1988]. In his system data is extracted and analyzed by posing queries in the temporal query language, *TQuel*, which is based on the relational calculus. Lomita conditions, based on real-time interval logic expressions, are equivalent in expressive power, but arguably easier to use. More importantly, by adding control to monitoring we have closed the loop and produced a feedback control system.

9.2 Distributed operating systems

Many of the functions of what we call the control program are functions that are more usually associated with an operating system: in particular scheduling and resource management. Indeed, our choice of the term *control program* is intended to reinforce this similarity. In a general purpose distributed operating system, such as Locus [Walker *et al.* 1983], the set of resources and control parameters is fixed by the operating system, and usually limited to the lowest common denominator of the applications envisaged.

A common facility is *remote execution*, in which unrelated nondistributed programs are allocated to or migrated between machines in order to share the available load more evenly [Douglis and Ousterhout 1987, Litzkow *et al.* 1988]. Load sharing is easily implemented using Meta, but Meta provides much richer facilities for describing the inter-relationships and dependencies between the processes making up a true distributed application. With Meta, the programmer can build operating-system-like facilities, but at the application level.

9.3 Configuration of distributed applications

There are a small number of systems for controlling or configuring distributed applications that take a more structured view of the application, permitting a finer degree on control.

The *RM2* distributed resource manager [Craft 1983], permits the construction of compound software resources, which are similar to our notion of a distributed application. Although resource requirements and preferences can be specified when configuring an application with RM2, only static attributes such as memory size and processor type may be used. There is no counterpart to Meta sensors so that dynamic control such as load sharing is not possible. Finally, the RM2 implementation has a small number of fixed resource allocation policies, whereas with Lomita the policy may be specified by the programmer.

The Conic language and system [Kramer *et al.* 1985] is perhaps the most thorough attempt at dynamic reconfiguration for distributed programs. A Conic application is structured as a set of modules and communication ports. Configuration consists of creating instances of modules and connecting input and output ports in a type-safe way. Conic supports dynamic reconfiguration by allowing new modules to be created and existing ports to be reconnected at run-time. While Conic provides some of the functionality necessary for application management, there is no general notion corresponding to either sensors or actuators. Thus configuration scripts cannot react to changes in the application or it environment, and reconfiguration is restricted to modifying module connections and creating new modules.

9.4 Rule-based systems

Rule-based techniques, like those in Meta, are commonly used in debuggers for concurrent programs. Usually these have taken the form of production rule breakpoints in which a condition over the values of variables and the program counters of several processes triggers an action such as suspending the program being debugged. Bruegge's *Pathrules* language [Bruegge and Hibbard 1983] is a good example. Debugging researchers seem not to have recognized the general utility of production rule systems for other kinds of monitoring and control of distributed applications.

Rule-based techniques have been used widely in expert systems. The approach used by Meta resembles some expert systems of the distributed blackboard model, especially those used in soft real-time control applications [Cruise *et al.* 1987]. However ISIS provides a sounder basis for handling failures and achieving consistency that is more appropriate to the environments in which Meta will be used. The production rules provided by Meta are similar in structure to expert system rules, but we believe that the actual rules written for an application management system will be much less complex, and less ambitious than those in typical expert systems.

9.5 User interfaces for distributed control

A graphical user interface complements a textual language for distributed application management. The information derived from application monitoring can be much easier to comprehend when displayed in graphical form, and a graphical editor interface provides a particularly powerful way of experimenting with different control policies. ConicDraw [Kramer *et al.* 1989] is such a graphical interface to the Conic configuration system. It displays the current structure of the system, and provides graphical counterparts to the facilities of the Conic configuration language.

Magic Lantern is a graphical application management tool being developed in concert with the Meta project [VanRenesse 1990]. It provides a comprehensive set of graphical objects such as strip chart recorders, bar graphs, scroll bars, buttons and text objects. These objects can derive information from Meta sensors, and invoke Meta actuators. The layout of the display and the connections to Meta are completely programmable by the Magic Lantern user. Magic Lantern may also be used independently of Meta and ISIS. With this tool one can experiment interactively with different control strategies and receive immediate feedback in terms of the performance of the application. This gives a fuller understanding of the behavior of the application, leading to a better automatic control policy in the form of a Lomita script.

10 Conclusions

We have described the Meta system, which provides a sound basis for implementing distributed application control mechanisms. Although high level in approach and structure, it has proved possible to express and maintain rigorous semantics in managing a distributed application.

The widespread availability of distributed computing and operating systems has made it increasingly important to focus on the technologies by which large systems can be composed from sets of components. By offering a programming methodology for distributed control, Meta makes it easier to build robust distributed applications using components that, individually, are incapable of tolerating faults. The approach also makes it easier to reason about and establish the correctness of the resulting control structures. These are important steps towards the open, heterogeneous distributed operating systems that will characterize the next generation of distributed programming environments.

Availability

Version 1.2 of Meta has been implemented using the ISIS system and is being distributed in source code form within the ISIS user community of about 300 sites. This version contains a complete implementation of the Meta sensor and actuator subroutine interface described in Section 4, and the built-in **Machine** and **Process** sensors. The Lomita language, as described in Sections 5 and 6 is still being implemented. The 1.2 release contains an earlier query language based on the relational algebra. We have also produced an experimental C-Prolog interface to Meta that we are no longer releasing. A preliminary version of the Magic Lantern system is also included in the ISIS release.

Acknowledgements

Our work has profited from the comments and suggestions of many colleagues and users of the first releases of the system. Particular thanks are due to Cris Kobryn, Jerry Jackson and Jim Wong of Science Applications International Corporation, architects of the nuclear seismological application, and to Miriam Leeser, Fred Schneider and Robbert Van Renesse who read and commented on earlier drafts of this paper.

References

[Armand *et al.* 1989]
François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or

Distributing UNIX Brings it Back to its Original Virtues. Technical Report CS/TR-89-36.1, Chorus systèmes, 6 Avenue Gustave Eiffel, F-78182, Saint-Quentin-en-Yvelines, France, August 1989.

[Birman and Joseph 1987]
Ken Birman and Thomas Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pp. 123–138. ACM Press, New York, NY 10036, Order No. 534870, November 1987.

[Bruegge and Hibbard 1983]
Bernd Bruegge and Peter Hibbard. Generalized Path Expressions: A High Level Debugging Mechanism. In *Proceedings of the ACM/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pp. 34–44. ACM, 1983.

[Chen 1976]
P. P.-S. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, **1**, 1, pp. 9–36, March 1976.

[Craft 1983]
Daniel H. Craft. Resource Management in a Decentralized System. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pp. 11–19. ACM SIGOPS, 1983.

[Cristian *et al.* 1986]
Flaviu Cristian, Houtan Aghili, and Ray Strong. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. Technical Report RJ 5244 (54244), IBM Almaden Research Laboratory, July 1986.

[Cruise *et al.* 1987]
A Cruise, R Ennis, A Finkel, J Hellerstein, D Klein, D Loeb, M Masullo, K Milliken, H Van Woerkom, and N Waite. YES/L1: Integrating Rule-Based, Procedural, and Real-time Programming for Industrial Applications. In *Proceedings of the Third Conference on Artificial Intelligence Applications*, pp. 134–139. IEEE Computer Society, February 1987.

[Douglis and Ousterhout 1987]
Fred Douglis and John Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pp. 18–25. IEEE Computer Society, 1987.

[Gusella and Zatti 1985]
R. Gusella and S. Zatti. The Berkeley UNIX 4.3 Time Synchronization Protocol: Protocol Specification. Technical Report UCB/CSD 85/250, Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, June 1985.

[Kramer *et al.* 1985]
Jeff Kramer, Jeff Magee, and Morris Sloman. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, **SE-11**, 4, pp. 424–436, April 1985.

[Kramer *et al.* 1989]
Jeff Kramer, Jeff Magee, and Keng Ng. Graphical Configuration Programming. *Computer*, **22**, 10, pp. 53–65, October 1989.

[Litzkow *et al.* 1988]
Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—A Hunter of Idle Workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pp. 104–111. IEEE Computer Society, June 1988.

[Marzullo 1989]
Keith Marzullo. Implementing Fault-Tolerant Sensors. Technical Report TR 89-997, Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853, May 1989.

[Marzullo and Chew 1990]
Keith Marzullo and Paul Chew. Efficient Algorithms for Masking Sensor Failures. In preparation, 1990.

[Snodgrass 1988]
Richard Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, **6**, 2, pp. 157–196, May 1988.

[Snodgrass and Ahn 1985]
Richard Snodgrass and Ilsoo Ahn. A Taxonomy of Time in Databases. In *Proceedings of the International Conference on Management of Data*, pp. 236–246. ACM SIGMOD, May 1985.

[VanRenesse 1990]
Robbert Van Renesse. The Magic Lantern Distributed Management System. In preparation, 1990.

[Walker *et al.* 1983]
Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pp. 49–70. ACM SIGOPS, 1983.

[Wood 1991]
Mark Wood. *Distributed Monitoring and Control.* PhD thesis, Cornell University, Department of Computer Science, 1991. In preparation.