Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture

Mark Dixon Wood
Ph.D Thesis

91-1252
December 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
FAULT-TOLERANT MANAGEMENT OF DISTRIBUTED APPLICATIONS USING THE REACTIVE SYSTEM ARCHITECTURE

A Dissertation
Presented to the Faculty of the Graduate School of Cornell University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

by
Mark Dixon Wood
January 1992
FAULT-TOLERANT MANAGEMENT OF DISTRIBUTED APPLICATIONS
USING THE REACTIVE SYSTEM ARCHITECTURE

Mark Dixon Wood, Ph.D.
Cornell University 1992

Distributed applications are becoming increasingly pervasive, and difficult to manage. Examples of distributed applications include operating system servers and clients on a network, programs performing distributed computations, and systems constructed by integrating stand-alone programs. This thesis argues that distributed applications can be managed efficiently by using a reactive system architecture. A reactive system consists of a control component continuously responding to changes in an environment component. This structure is applied to distributed application management by casting the programs making up the application as the environment and superimposing a layer of control. By acting upon conditions sensed in the environment, the control layer can respond to changes in the distributed application, ensuring that it functions in a well-behaved manner. This thesis also presents the Meta toolkit, which provides primitives for controlling distributed applications using the reactive system architecture. The application components are instrumented with sensors and actuators—routines that respectively read and modify the application state. Control of the application is carried out via guarded commands, which are distributed for
execution by either stubs coresident with programs in the application or by special servers. Distributing the control program results in greater responsiveness and efficiency but requires certain consistency problems to be addressed. Furthermore, the Meta toolkit supports fault-tolerant execution of guarded commands through the use of replicated servers. This toolkit has been implemented and is completely functional, and this thesis contains extensive performance figures for the toolkit.
Biographical Sketch

Mark Dixon Wood was born on September 4, 1964 in Burlington, Vermont. His early life was spent in Jericho, Vermont, where he enjoyed the convivial companionship of cats, goats, and turkeys, not to mention his six siblings. After graduating as class valedictorian from Mount Mansfield Union High School in 1982, he studied computer science at the University of Vermont. In 1986 he received his bachelor of science in computer science from that university, graduating summa cum laude. In the fall of 1986, he began his graduate studies at Cornell University, with the master of science in computer science received in 1989. He was granted the doctor of philosophy degree in January 1992.
Acknowledgements

Thanks first go to my advisor, Keith Marzullo, who has provided me with the direction and encouragement needed to bring this work to completion. It has been a privilege to work with Keith, and I have learned much from him about the scientific process. I am also indebted to the other members of my committee: Ken Birman, Keshav Pingali, and Leonard Gross. In addition to giving me considerable assistance with the ISIS system, Ken has provided me with helpful feedback on the design of Meta. Ken has generally been a great source of advice and encouragement, for which I am very appreciative.

Thanks also go to my officemates and friends, Ajei Gopal and Steve Mitchell. Both Ajei and Steve have greatly contributed to making life here more interesting and pleasant.

A number of students have at various times been involved in different facets of the Meta project, and their contributions are gratefully acknowledged. Wanda Chiu deserves credit for having built the first significant application on top of Meta. Nancy Thoman implemented the initial prototype of the Meta guarded command language. Dan Sturman developed a demonstration program on top of Meta. Truls Myklebust implemented the spy program.
I have also benefited from various discussions with other members of the distributed systems group here at Cornell. I would like to specifically thank Navin Budhiraja, Tushar Chandra, Robert Cooper, fellow Vermonter Brad Glade, Ajei Gopal, Guerney Hunt, Fred Schneider, Pat Stephenson, Robbert van Renesse, and Raphael Yahalom. Discussions with Bard Bloom helped in formulating the contents of Chapter 3.

The presentation in this thesis has greatly benefitted from comments made by Keith Marzullo and Ken Birman. In addition, I wish to thank the following people for their helpful comments on various chapters: Pauline Bennett, Navin Budhiraja, Susan Honig, Pat Stephenson, and Robbert van Renesse.

The interest shown in Meta by Guatam Thaker and his colleagues at the General Electric Advanced Technology Laboratory has been most encouraging. They have made extensive use of Meta, resulting in helpful feedback on its design and implementation.

The financial support provided by G.T.E. is gratefully acknowledged, along with the support paid for by American taxpayers, as administered by the Defense Advanced Research Projects Agency (NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904). In addition, this project has received financial support from I.B.M., Siemens, and Xerox.

Lastly, thanks go to all those other people who have helped me get to this point. My parents, Glen and Elizabeth Wood, and my siblings deserve prime recognition. My sister Gretchen, who by virtue of living nearby, had to put up with me far more than she ever had to before. I would also like to thank my high school mathematics teacher, John Bateman, who first introduced me to computers. Then there are all the
friends who have stood by me these years; I would especially like to mention Karen van der Eems, Un-Ku Moon, and Douglas and Donna Stauffer for their friendship and encouragement. I am also grateful for the prayer fellowship with Steve Lantz, Mike Roman, Steve Schaffner, and Jeff Wolinski. Most significantly, thanks go to God who has brought me this far in life.
Table of Contents

1 Introduction .............................................. 1
   1.1 Applications of the Reactive System Model ............... 4
   1.2 Why Reactive Systems? .................................. 8
   1.3 Why Distributed Control? ............................... 9
   1.4 Characteristics of a Good Solution ..................... 9
   1.5 Summary .............................................. 12

2 Related Work ............................................. 13
   2.1 Monitoring Systems .................................... 13
       2.1.1 TQuel ............................................ 14
       2.1.2 Issos ............................................ 14
   2.2 Distributed Debugging Systems ......................... 15
   2.3 Rule-Based Systems ................................... 16
   2.4 Tool Integration Systems ............................... 17
       2.4.1 Forest ........................................... 18
   2.5 Summary .............................................. 19

3 Controlling Reactive Systems ............................ 21
   3.1 Semantics of the Guarded Command ..................... 23
   3.2 Event-based Monitoring ................................ 24
   3.3 Proving Properties about Programs .................... 26
   3.4 Example .............................................. 27
   3.5 Summary .............................................. 29

4 Consistent Detection ..................................... 30
   4.1 Determining the Global State ......................... 32
       4.1.1 Model of the System ............................. 33
   4.2 Protocols for Determining the Global State .......... 36
       4.2.1 Determining the Global State in a Synchronous System ............................................. 36
       4.2.2 Determining the Global State in an Asynchronous System ............................................. 37
   4.3 Summary .............................................. 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.1 Naming</td>
<td>96</td>
</tr>
<tr>
<td>8.2.2 Establishing Connections Between Stubs</td>
<td>97</td>
</tr>
<tr>
<td>8.2.3 Handling Subscriptions</td>
<td>98</td>
</tr>
<tr>
<td>8.2.4 Implementing Aggregates</td>
<td>101</td>
</tr>
<tr>
<td>8.2.5 Replicating Stubs</td>
<td>101</td>
</tr>
<tr>
<td>8.3 Performance of the Meta Toolkit</td>
<td>105</td>
</tr>
<tr>
<td>8.3.1 Experimental Setup</td>
<td>106</td>
</tr>
<tr>
<td>8.3.2 Cost of Subscriptions</td>
<td>106</td>
</tr>
<tr>
<td>8.3.3 Cost of Executing Guards</td>
<td>108</td>
</tr>
<tr>
<td>8.3.4 Executing a Guard Remotely</td>
<td>110</td>
</tr>
<tr>
<td>8.3.5 Executing a Single Remote Action</td>
<td>112</td>
</tr>
<tr>
<td>8.3.6 Transaction Processing</td>
<td>116</td>
</tr>
<tr>
<td>8.3.7 Aggregate Actuations</td>
<td>117</td>
</tr>
<tr>
<td>8.4 Summary</td>
<td>118</td>
</tr>
<tr>
<td>9 Conclusion</td>
<td>121</td>
</tr>
<tr>
<td>9.1 Future Directions</td>
<td>124</td>
</tr>
<tr>
<td>A The Meta Functional Description</td>
<td>126</td>
</tr>
<tr>
<td>A.1 Types</td>
<td>127</td>
</tr>
<tr>
<td>A.2 Instrumenting a Program</td>
<td>129</td>
</tr>
<tr>
<td>A.2.1 Sensors</td>
<td>130</td>
</tr>
<tr>
<td>A.2.2 Actuators</td>
<td>133</td>
</tr>
<tr>
<td>A.3 Front-End</td>
<td>135</td>
</tr>
<tr>
<td>A.3.1 Initialization</td>
<td>135</td>
</tr>
<tr>
<td>A.3.2 Client Operations</td>
<td>135</td>
</tr>
<tr>
<td>A.4 The NPL Programming Language</td>
<td>140</td>
</tr>
<tr>
<td>A.4.1 Executing Guarded Commands</td>
<td>140</td>
</tr>
<tr>
<td>A.4.2 Predefined Sensors and Functions</td>
<td>141</td>
</tr>
<tr>
<td>A.4.3 Expression Evaluation</td>
<td>142</td>
</tr>
<tr>
<td>A.4.4 Predefined Actuators</td>
<td>142</td>
</tr>
<tr>
<td>A.4.5 Aggregate Invocation</td>
<td>142</td>
</tr>
<tr>
<td>A.5 Running Meta Applications</td>
<td>143</td>
</tr>
<tr>
<td>A.5.1 The Machine Program</td>
<td>144</td>
</tr>
<tr>
<td>A.5.2 The Server Program</td>
<td>144</td>
</tr>
<tr>
<td>A.5.3 The MetaCmd Program</td>
<td>145</td>
</tr>
<tr>
<td>A.6 Limitations</td>
<td>146</td>
</tr>
<tr>
<td>A.7 Constant Definitions</td>
<td>146</td>
</tr>
<tr>
<td>Bibliography</td>
<td>148</td>
</tr>
</tbody>
</table>

ix
## List of Tables

7.1  Relational operators on intervals, $[\alpha, \beta] \mathcal{R} [a, b]$ .......................... 83

8.1  Time to receive subscriptions ........................................... 107
8.2  Time to receive subscriptions sent asynchronously .................. 108
8.3  Time to execute a guard locally ........................................ 110
8.4  Time to execute a guard with remote references ..................... 111
8.5  Time to execute a remote action ...................................... 113
8.6  Time to execute a transaction .......................................... 117
8.7  Time to execute an aggregate actuator ............................... 118
# List of Figures

1.1 The reactive system architecture .................................. 3  
1.2 The NMRD distributed application .................................. 5  
3.1 Transformational versus reactive systems ....................... 21  
3.2 Reactive system architecture ..................................... 22  
3.3 Expressing a condition using an automaton ...................... 28  
4.1 A consistent but invalid cut ...................................... 35  
4.2 An inconsistent cut .................................................. 38  
4.3 Inconsistent ordering of message delivery ...................... 39  
7.1 Logical architecture of an instrumented application .......... 69  
7.2 The Meta Server program .......................................... 80  
7.3 Expressing a condition using an automaton ..................... 87  
8.1 Group structure for handling subscriptions .................... 99  
8.2 A breakdown of causal dependencies .............................. 100  
8.3 The S sensor and ToggleS actuator ................................ 108  
8.4 Median execution time to execute a guard with remote references . 112  
8.5 Median time to execute a single remote action .................. 114  
8.6 Median aggregate execution time ................................ 119
Chapter 1

Introduction

As the use of computers proliferates, so too does the use of distributed systems. Characteristically [Mul89], distributed systems consist of multiple processing units interconnected via a network. Each unit is permitted to fail independently but to minimize the effect of failures on other units, each unit contains a certain amount of state information derived from the states of other units in the system.

The use of interconnected computers arises primarily as a matter of economics: it is less expensive to use multiple small machines connected together than to use one large machine. Furthermore, distributing computing resources can put computers where the tasks or users are. For example, installing a graphical display workstation in each office provides individuals with a more powerful work environment than they would have if each were connected via a dumb terminal to some centralized machine. A factory process line may be controlled more economically by using microcomputers located at each station of the line than by controlling the stations using a centralized computer. Furthermore, distributing the global application among different machines can provide opportunities for increased failure resiliency. Using a centralized computer provides a single point of failure; although a distributed system may have more points of failure, individual failures need not bring down the entire distributed
system if it is well-designed. In other words, using a distributed system allows for
the possibility of partial failures, where an individual failure does not prevent the
operation of other parts of the system.

Running an application on a distributed system differs in many respects from
running an application on a centralized system. The different application compo-
nents need to communicate with each other via messages, which takes time, thereby
introducing delays between the time when one application component senses some
condition and the time some other component senses the same condition. Furthermore,
the presence of multiple computers gives rise to multiple points of control,
raising issues of synchronization. And the application as a whole should be able to
gracefully handle the failure of individual subcomponents.

Constructing an application as a set of distributed components raises a number
of management problems, caused largely by the possibility of partial failures and
by the need to synchronize the operation of the various components. Control poli-
cies are required to govern the interaction between the individual subcomponents
of the distributed application. Since distributed computer systems have become so
widespread, effective management of distributed applications has become a pressing
problem.

This thesis presents a theory for managing distributed applications. The Meta
toolkit, described in this thesis, provides a library of routines useful for construct-
ing and controlling distributed applications. The approach adopted in this work
is to view the distributed application as a reactive system, consisting of a control
component reacting to an environment component. The “environment” contains the
individual components of the distributed application. This architecture is shown
in Figure 1.1. Control is embodied in a dynamically changeable set of rules—called
guarded commands—that describe the behavior of the system. The execution of such
guarded commands is performed within the Meta toolkit.
Figure 1.1: The reactive system architecture

In order for a control system to respond to changes in the environment, the environment must first be instrumented both to expose the state of the environment and to provide the control system with a way to operate upon the environment. The Meta toolkit provides a way to instrument the various applications making up the environment with sensors that return some aspect of an application’s state and with actuators that provide a way to control the behavior of an application. Sensors and actuators are simply routines added to the original programs making up the distributed application; these routines present to Meta a uniform interface for reading the state of the application, via sensors, and modifying the state of the application, via actuators.

The structure of a distributed application can be very complicated, involving many components. To provide structure, Meta allows groups of related components to be grouped together into aggregates. An aggregate component inherits sensors and actuators from its member components; additional sensors and actuators may be defined over the aggregate as whole. The sensors and actuators of an aggregate appear to the rest of the system like any other sensor or actuator.
The control of the distributed application is effected via a guarded command language, called NPL. The control of the distributed application is accomplished via a set of NPL programs executing at potentially different components.

Subsequent chapters of this thesis present both the instrumentation and control aspects of Meta. The Meta toolkit may be used to construct a wide variety of distributed applications, both by providing a way to glue together existing applications and to construct systems from scratch. The next section discusses in more detail some possible uses of the Meta model for constructing and controlling reactive systems.

1.1 Applications of the Reactive System Model

The early stages of the Meta project were guided by one particular application, that of monitoring compliance with treaties banning the testing of nuclear weapons. One proposed solution to this problem, called NMRD, was made up of a number of off-the-shelf components. A signal-processing program SigPro was used to collect and analyze seismological data. This program was a recycled FORTRAN application; it was desired that this program be modified as little as possible. In addition, the application included an expert system called Assess that analyzed the data generated by the SigPro. As part of its analysis, the Assess process would generate additional work for the SigPro components. Buffering between the Assess and SigPro processes was provided by a DataStore service.

By drawing upon preexisting software components, this design required less programming than a design which started from scratch. Moreover, this paradigm corresponded well with the geographical structure of the monitoring problem, having distributed sensing and data analyzing components. At the same time, this system, by being distributed in nature and critical in function, would be difficult to manage. Various policies were to be maintained with respect to the number of SigPro pro-
cesses running and how new application components were to be started up in response to failures; a separate component called **AppManager** managed the application. Due to the complexity of the system, it was desirable that a human interface exist to allow system administrators to observe the state of the system and to adjust the policies governing its behavior if necessary. Consequently, a graphical interface provided a view on to the state of the system and allowed the policy rulebase to be dynamically updated. The logical structure of this application is shown in Figure 1.2. The dotted lines in this figure show the paths taken by control and monitoring information; the solid lines show the paths taken by application data.

![Diagram of NMRD distributed application]

**Figure 1.2**: The NMRD distributed application

This architecture can naturally be cast as a reactive system. The environment consists of the **SigPro**, **Assess** and **DataStore** components. The control component **AppManager** can be handled as a set of Meta guarded commands. The graphical interface may easily obtain the state of the distributed application by referencing
Meta sensors.

The distributed NMRD application illustrates the general problem of distributed application management. The proper functioning of the NMRD system relies upon dynamic management. Like the NMRD example, distributed systems are often formed from existing applications that were designed to be stand-alone processes. Such applications must be “glued” together in some fashion and then managed. The management layer allows the administrator or user to control the performance and reliability of the application. Managing distributed applications constructed in this fashion is one of the primary uses of the Meta toolkit.

Network management provides another good example of a reactive system. Typically, client workstations operate somewhat autonomously, though they are often connected via a network. Also attached to the network are specific services such as printers and file storage devices. The distributed nature of the system can make it difficult for users to effectively use and manage the available resources. For example, one workstation may be overloaded while another has no users, or one file server may appear to be experiencing excessive intermittent errors, yet no record of its past behavior exists. A commonly used printer may have run out of paper, unknown to users until they come to collect their print-outs. Typical configurations of distributed resources provide little support to counteract situations such as these. The Meta toolkit however, provides a convenient way to instrument the resources on the network and to provide a layer of management that operates according to a dynamically changeable rule base.

Using the Meta toolkit, various system services can be instrumented to allow the system to respond to problems with particular resources such as printers. Additionally, sensors may be written to give machine-specific values, such as the processor load, the set of people logged in, or the status of internal long-running programs. Similarly instrumented programs may be grouped together into aggregates, either
statically or dynamically. For example, a free machine group could be maintained to provide a processor pool to use for running new tasks. The aggregate mechanism may also be used to constructed fault-tolerant services, with Meta ensuring that the service is provided by one (or more) members of the aggregate.

Once a collection of workstations and other resources has been adequately instrumented, policies governing the system's behavior are enforced by the Meta guarded command interpreters, which are distributed throughout the system. The policy rules may be dynamically changed, to satisfy changing needs or the whims of individual users. Intermittent problems with particular resources can be watched for, with corrective actions automatically taken or users notified. An appropriately instrumented network of resources may be utilized much more satisfactorily than a network lacking support for monitoring and control. By providing a way to effectively instrument and control the network, Meta enables the efficiency of the individual resources and users to be maximized.

Furthermore, the reactive system architecture is a general model; virtually any long-running application fits into this mold. One particularly promising application of our reactive system architecture is the area of process control. Industrial control systems are naturally monitored and controlled via sensors and actuators. The Meta toolkit provides a way to uniformly define sensors and actuators regardless of the process being controlled. Once the system has been properly instrumented, the Meta guarded command language provides an expressive and dynamically programmable way to its behavior. Often the control of industrial systems is represented as state machines; such machines may be readily represented in Meta.

However, the current implementation of Meta does not address the real-time needs of an industrial process control system; such work is left as a further extension. Ignoring that consideration for the moment though, process control systems provide an additional important and motivating example of the need for systems like Meta.
1.2 Why Reactive Systems?

What makes the reactive system architecture attractive for system monitoring and control? The strength of the model lies in its separation of the control aspects from the rest of the system. Logically isolating the control layer offers a number of advantages.

For one, the separation of policy from the mechanism used to enforce the policy has generally been found to be a good principle, as it prevents the two from becoming confused. Typically, if system policy is interwoven with the mechanism implementing that policy, then changing the policy is quite difficult. By isolating the control layer, system policy can be changed without affecting the mechanism. Similarly, the mechanism may be independently refined and improved. This feature greatly facilitates program maintenance.

A certain amount of mechanism is always needed to just to enforce the policies controlling a system. The reactive system architecture separates the entire control layer from the rest of the system; the control layer therefore encompasses the mechanism for enforcing those policies. If the control mechanism is written to be sufficiently general, then it can be used to manage many different types of applications. This reuse of code simplifies program development.

Furthermore, by having a separate, reusable control layer, optimizations made to that layer are then gained by all applications using that layer. A particular advantage for distributed systems is fault-tolerance. However, constructing a fault-tolerant control system is quite a difficult task, even beyond the reach of many programmers. So having a generic control layer offering fault-tolerance is particularly valuable; application programmers no longer need to spend countless hours attempting to program fault-tolerance into their own particular control mechanism when a generic system is available.
1.3 Why Distributed Control?

If the reactive system itself is distributed, then the control of that system should also be carried out in a distributed fashion. One motivation for distributed systems is fault-tolerance, but of what use is a fault-tolerant system if the control of that system is not also carried out in a fault-tolerant manner? Consequently, replication of the various control components is necessary to provide a corresponding robustness in system control.

Given that a reactive system is distributed, a distributed control layer makes sense for other reasons too. An important factor as the size of distributed systems increases is the matter of scaling; a centralized control layer simply may not be able to keep up with a large and extensive system. By distributing the control layer, we may be able to take advantage of locality of reference. Moreover, distributing the control layer enables different control components to execute in parallel, enhancing the overall power of the control system. Consequently, we present in this thesis a model for controlling a distributed system in a distributed fashion.

1.4 Characteristics of a Good Solution

Any system seeking to solve the problems of distributed control and monitoring should satisfy a number of important properties.

1. First, the system being controlled should behave as if there were a single point of control. This principle is analogous to the serializability theory of database systems [BH87]. We discuss in Chapter 5 the notion of a “virtually” centralized model of execution. Unfortunately, providing this level of consistency requires blocking, which conflicts with our next goal:

2. The process of monitoring should interfere as little as possible with the normal operation of the application being instrumented. Ideally, instrumenting
an environment should not effect the performance of the environment. The presence of sensors within the code should not require a noticeable amount of system resources, to the point of causing the instrumented code to perform differently than it would if the sensors were not present. To achieve this goal to the furthest extent possible would require running the monitoring system on an entirely separate computer system. The code implementing the sensors would have to run on separate machines and the network traffic generated would have to be sent via a separate network. Since running the monitoring system separate from the system being monitored is not usually economically feasible, the monitoring system must generally coexist with the application and so it is especially important that its overhead be low.

For some applications, the presence of a monitoring system will merely effect performance without effecting system behavior. However, if the application consults external variables such as a clock, it may notice its execution to be delayed because of the overhead imposed by the monitoring layer. Moreover, the monitoring overhead may change system timing, causing any race conditions that might exist within the application to be resolved differently from how they would have been resolved had the monitoring layer not been present. Instrumenting a distributed application that both lacks references to external state and is free from race conditions will not effect the behavior of the application; instrumenting other, real-time, applications may unavoidably influence the execution of the system.

In order to correctly control the behavior of a system, the control system may need to deliberately influence the behavior of the system. Obviously the system is affected as the control layer invokes specific actuations on the system. However, it goes beyond this: in order to provide correct behavior, the control component may need to cause subcomponents to block. This issue, discussed in
Chapter 4, arises if the system is to respond to events in the state they were observed. Blocking is also required to prevent distributed controlling components from interfering with each other at times when mutual exclusion is needed.

3. The response time of the control system should be as good as possible. Clearly in a real-time application, the correctness of the application relies upon timely responses. However, to some extent, all applications are real-time systems; for if you didn’t care how long the system took to accomplish its tasks, why bother running the system at all? To be useful, a system for monitoring and control should be able to respond quickly. Achieving this goal requires carefully written protocols.

4. The performance of the system should scale well as additional processing sites and subcomponents are added in. A design that may work well for a small number of components may fail dismally as the number of components is increased; a successful design will scale satisfactorily with the size of the system.

5. Failures of individual subcomponents should not effect the operation of unrelated components. Furthermore, the system should tolerate the failures of components the best it can, with alternative ways of accomplishing the monitor and control tasks.

6. Policies specifying the behavior of the system should be dynamically changeable. As user needs change, administrators should be able to remove old policy rules from the system and install new ones.

1.5 Summary

Economic and fault-tolerance concerns have made distributed systems increasingly popular. Controlling an application becomes especially difficult when the application
is distributed. This thesis presents a toolkit called Meta for constructing reliable systems using the reactive system architecture. A reactive system is one consisting of two layers: a control layer governing the behavior of the instrumented environment layer. By using the reactive system architecture, distributed applications may be conveniently managed through the use of guarded commands, written over the state of the environment.

These applications are constructed by interconnecting various modules together to perform the desired function; the Meta toolkit provides a way to manage the interconnected components. Network management and process control are specific types of distributed applications that may be handled using a system such as Meta.

The next chapter examines some related work in distributed monitoring and tool integration. Chapter 3 presents the semantic model for the guarded commands used to control the reactive system. Chapters 4 and 5 discuss the problems in executing guarded commands in a distributed system. Standard approaches to making distributed systems fault-tolerant are reviewed in Chapter 6.

Having presented the reactive system architecture, we then examine the Meta system in detail. The facilities provided by Meta are presented in Chapter 7. The implementation of Meta is presented in Chapter 8, focusing on the special problems raised in a distributed environment. Chapter 8 also offers performance statistics for the current Meta system. A final summary is provided in Chapter 9. The appendix contains the functional description of the Meta toolkit.
Chapter 2

Related Work

This chapter reviews some research efforts that have been influential in Meta's design. Distributed monitoring systems have provided insight into the problem of instrumentation. Distributed debugging systems with their notion of distributed breakpoints have contributed to our understanding of consistency in a distributed system. Both distributed monitoring and debugging systems include languages for expressing states or events of interest; these languages have influenced Meta's guarded command language. From the field of artificial intelligence, we have examined rule-based systems for approaches to control problems. Tool-integration systems may be viewed as forerunners to distributed management.

2.1 Monitoring Systems

An important monitoring system is TQuel, developed by Richard Snodgrass [Sno88]. One of the significant contributions of this work is its use of the entity-relation model [Che76] to structure the system being monitored [Sno82]. The TQuel work is extended for distributed monitoring in the Issos project [OSS90].
2.1.1 TQuel

TQuel provides a way to monitor the state of a single component. The system provides a methodology for instrumentation, with facilities to enable and disable the gathering of data. The collected data is conceptually viewed by the user as residing in a historical relational database [SA86], a database in which each tuple is tagged with the time in which the tuple was valid. A relational language is used to write queries on the system state. By translating the declarative queries into relational algebra, Snodgrass was able to tap the theory of relational algebra optimizations for efficient evaluation of the queries.

This system is centralized and, being a monitoring system, lacks support for reaction to events.

2.1.2 Issos

The Issos system extends the TQuel approach with an emphasis on providing monitoring information to the application in real-time, thereby enabling the application to dynamically adapt its behavior in response to system changes. Like TQuel, Issos also uses the entity-relation model to present the monitored data. The user writes against this model specifications of conditions to be monitored for; the Issos system generates from these specifications instrumentation code which must be manually inserted into the application. As in TQuel, the collection of data from sensors may be dynamically controlled. This type of instrumentation is static in that sensors are only generated in response to statically specified conditions. Naturally, a user of the system might not have been able to have foresee all the conditions that one might want to monitor for. Consequently, Issos also allows values within a process to be probed using the Unix signal mechanism. However, this mechanism does not work with arbitrary processes; the process must have been linked with a stub supporting this signal mechanism.
Although Issos does have a centralized monitor, it also runs monitors on each processor in the system; work is offloaded to these monitors when feasible. These monitors collect data from the sensors, storing it indefinitely. Sensors report their values to the Issos monitor in one of two fashions. Values may be reported asynchronously, with the sensor sending a message to the monitor and the application continuing execution. Alternatively, values may be reported synchronously, with the sensor blocking after sending a value until it receives an acknowledgement from the monitor. The monitor watches for the specified conditions to be satisfied; satisfaction results in a message being sent to some specified address. Since the focus is on monitoring, no other type of reaction is supported.

Conditions to be monitored for are written as predicates over the state of the system. A qualified notion of simultaneity is supported, in that a "correct-within" value may be specified for a conjunction, stating that the different conjuncts were true at times within the specified value of each other. The emphasis of the Issos project is on the collection of data in a manner that satisfies specified real-time constraints; a major contribution of this work is determining how to gather the necessary state information in a manner that satisfies real-time constraints.

2.2 Distributed Debugging Systems

A debugger provides a programmer with the means for stopping an application during its execution at specified points, called breakpoints, and examining the state of the stopped process. Breakpoints are typically written as some predicate on the state of the process; as soon as the state of the process satisfies the specified condition, the process is stopped. A distributed debugger allows the application to consist of separate, communicating processes. Handling breakpoints for a distributed application is much more complicated, since they are now predicates over a distributed state. Ideally, the system should be halted as soon as the predicate is satisfied, with each
application component being in the state it was at the moment the predicate was satisfied.

The work of [MC88] extends the Chandy-Lamport snapshot algorithm [CL85] to cause the system to halt when the specified condition is observed. To simplify the system, the specification of breakpoints may not include conjuncts on the global state. Although the system does not halt in exactly the same state as the state that first satisfied the breakpoint condition, the halted state is “globally consistent” with the triggering state. In particular, there exists an execution of the system having the same sequence of communication events in which the halted state is the first state satisfying the breakpoint condition.

Another interesting work on distributed debugging is the system IDD [HHK85]. Although this work was never completed, the approach was to use interval temporal logic for the expression of breakpoints. Assertions however were restricted to be over communication events.

2.3 Rule-Based Systems

Meta is similar to rule-based expert systems in that the policies, or knowledge, of how an application should be controlled are separated from the application itself. Rule-based expert systems consist of three components [CCB88]. The knowledge-base component is a set of rules specifying the behavior of the system. The control component is the inference engine that decides what rules to apply when. Lastly, the data component provides the control component with information enabling it to decide which rules to apply.

Knowledge is represented in a rule-based system as a set of when-do or if-then rules, similar in style to the guarded commands used in Meta. Expert systems often have a complicated mechanism for resolving conflicts arising when multiple rules are concurrently executable. Typically such conflicts are resolved by attaching priorities
to the different rules. The process of deciding which rule to execute is typically rather complicated, making these systems inappropriate for lightweight application management. One common rule-based system is OPS5; this system executes rules at a rate several orders of magnitude slower than Meta [Mos86].

The YES/L1 system [CEF+87] seeks to provide both declarative and procedural representations of policies. This system seeks to be useable in a real-time environment, but that only means that support is provided for timers and communication primitives. This system is typical of expert systems in that it is a heavy-weight system. In this case, the expert system generates PL/1 code from the statically specified knowledge base which can be integrated with existing PL/1 applications.

2.4 Tool Integration Systems

Tool integration systems provide a way to plug independent tools together to solve some problem. A simple example of this approach is the Unix pipe facility. By using pipes, the output of one tool may be passed on to another. This simple form of tool integration only allows one-to-one, unidirectional communication. A general system for tool integration permits arbitrary interaction between the different tools, with the tool integrator controlling how that interaction is to take place.

One approach integrates tools in a client-server architecture, with one tool requesting a service from another tool. This requires tools to have explicit knowledge of other tools. An event-based, or reactive, approach to tool integration removes this restriction. Tools announce events of interest; these events are then observed and handled by the integration level.

[SN90] propose that event-based mechanisms for environment integration should support the following properties, which we compare to the properties of the Meta system.

- Events should be declared explicitly, in the component interface of the tool being
instrumented. The Meta system relies upon the application to report when a state transition has taken place; furthermore, the application may also specify which sensors are likely to have changed in value. However, Meta does not embed within the application which events are of interest; instead events are defined by the conditions being watched for. Since events are defined by dynamic policies, it is not appropriate to (statically) embed event declarations within the tool environment.

- Any component should be able to declare events. Meta allows any application to be instrumented, and thereby enables any component to declare events.

- The names and “signatures” of events should not be limited by the system but rather tools should have complete flexibility in announcing events. As noted previously, the low-level event of a state transition is reported to Meta by the application. The actual events of interest are defined in Meta by the control program, specifically, by the arbitrarily complex propositions associated with the currently active guarded commands.

- The parameters used in invoking a method, i.e., in carrying out an actuation, should be specifiable by the triggering event. In other words, an event causing some method to be invoked should be able to pass parameters to the method. Meta readily supports this concept: parameters may be passed to actuators, with the value of the parameters being bound at the time the enabling predicate was satisfied.

2.4.1 Forest

The Forest system [GI90] for tool integration operates by causing all tool interaction to occur via a centralized message service, called $Msg$. A tool may be integrated into Forest by imposing upon the tool an interface that communicates with $Msg$. A tool
announces an event by sending a message to the service. Msg maintains a policy
definition table containing for each tool a set of message patterns that the tool is
interested in. Associated with each such pattern is a set of condition, action pairs.

If a given message matches a pattern for a particular tool, the conditions associ-
ated with that pattern are then examined sequentially. Conditions are propositions
over variables and functions that are global to the system and whose value is main-
tained by Forest. The action associated with the first satisfied condition is evaluated.
A message may only invoke actions on one particular tool; actuations involving mul-
tiple tools do not appear to be supported.

This work provides a low-cost solution to the specific problem it was intended to
address, namely tool integration. However, even here its flexibility is limited since
interaction between tools is constrained by the centralized message service, and this
service only allows an event to trigger an action at a single tool. Furthermore, events
must be explicitly reported by the tool being instrumented. Consequently, the system
integrator must statically embed within each instrumented tool the knowledge of
what constitutes an event of interest. In the worse case, this requires all possibly
interesting events to be reported to the message service, requiring the condition
predicates to be more detailed.

2.5 Summary

This chapter reviews recent research in the areas of monitoring, rule-based control,
and tool integration. The approaches and solutions to these problems have provided
a starting point for the Meta project. Meta differs from the monitoring systems
discussed here in that it adds control; providing control raises a number of consist-
tency considerations not present in monitoring systems. Rule-based systems come
out of the artificial intelligence field. These systems tend to have more complicated
inference engines than the simple interpreter needed to evaluate Meta’s guarded
commands. In general, rule-based systems lack the simplicity and lean architecture of Meta, making such systems unsuitable for distributed application management. Tool-integration systems on the other hand provide only limited support for application control, focusing more on component interconnection. Although Meta certainly provides a mechanism for interconnection, it goes beyond, providing support for distributed, dynamic control of the application.
Chapter 3

Controlling Reactive Systems

A reactive system is a system that continually responds to external events [HP85]. This is in comparison to the standard transformational system, which when given a specified input produces the corresponding output and then terminates. In contrast, a reactive system dynamically generates output in response to asynchronously occurring events; the output generated may result in yet more events. The system continues to operate indefinitely. Figure 3.1 from [HP85] illustrates the behavior of these two types of systems.

(a) Transformational System

(b) Reactive System

Figure 3.1: Transformational versus reactive systems

Recall that the reactive system architecture consists of two components: the ap-
plication being controlled, which we call the *environment*, and the control layer operating in response to changes in the environment. The environment passes through a sequence of states. In response to a state transition in the environment, the control layer may act upon the environment, which in turn may trigger yet another state transition.

A fundamental premise of this thesis is that reactive systems may be effectively controlled by the use of guarded commands, which are rules of the form

\[(\pi \rightarrow \alpha)\]

where \(\alpha\) is the action to be performed when the predicate \(\pi\) is satisfied. Guarded commands are simple, yet expressive, capable of describing complex sequences of events. With respect to our reactive system architecture as shown in Figure 3.2, the control layer is simply a collection of guarded commands, with the predicates being expressions over the state of the environment.

![Figure 3.2: Reactive system architecture](image)

Using guarded commands to control the operation of a reactive system is a natural adaptation of the Unity model of parallel programming [Mis89,CM88]. Unity is a general theory of computation, capable of expressing many programming concepts relevant to reactive system control. A Unity program consists of a set of variables and a set of assignment statements. Assignment statements may include conditional
expressions; such conditional expressions correspond to guarded commands. The guarded command semantics presented here was largely guided by the Unity model.

3.1 Semantics of the Guarded Command

Before we can define the semantics of a guarded command, we must describe the environment upon which those commands operate. An environment $E$ is characterized by the set of states it may be in, by the set of actuators operating on the environment, and by a relation describing how the environment changes state.

Formally, for a given environment $E$, we let $\Sigma_E$ denote the set of states that the environment may be in. The set $A_E$ denotes the set of actuators operating upon this environment. The next state relation $\mathcal{N}_E$ is a relation of the form $\Sigma \times \Sigma \times (A \cup \epsilon)$. For $\alpha \in A_E$ and $s_i, s_j \in \Sigma_E$, we have $(s_i, s_j, \alpha) \in \mathcal{N}_E$ iff action $\alpha$ applied to the environment in state $s_i$ results in the environment next going to state $s_j$.

The environment in a reactive system is an active entity, capable of changing state on its own. We model this by having the triple $(s_i, s_j, \epsilon) \in \mathcal{N}_E$ if the environment when in state $s_i$ may spontaneously go to state $s_j$.

Controlling the environment is a set of guarded commands $C$ having the form $(\pi_i \rightarrow \alpha_i)$ where $\pi_i$ is a predicate over a state in $\Sigma$ and $\alpha_i$ is an actuator in $A$.

A history of a reactive system $\mathcal{R}$—consisting of environment $E$ controlled by the set of guarded commands $C$—is a (possibly infinite) sequence of states $s_0, s_1, s_2, ...$ where each $s_i \in \Sigma_E$. A valid history is a history in which $s_{i+1}$ is the next state in the sequence after $s_i$ iff one of the following two rules holds:

$$\exists \alpha \in A \text{ such that } (s_i, s_{i+1}, \alpha) \in \mathcal{N}_E \text{ and } \exists (\pi \rightarrow \alpha) \in C \text{ such that } s_i \vdash \pi \quad (3.1)$$

or

$$\exists (s_i, s_{i+1}, \epsilon) \in \mathcal{N}_E \text{ and } \neg (\exists (\pi \rightarrow \alpha_i) \in C \text{ such that } s_i \vdash \pi_i) \quad (3.2)$$
The first rule states that if there is some guard \( \langle \pi \to \alpha \rangle \) executable in state \( s_i \), then the next state \( s_i \) is the state resulting from executing \( \alpha \) in state \( s \). If more than one such guard is executable, then it is unspecified which guard is allowed to execute. The second rule allows the environment to spontaneously change state, if a state change on \( e \) is defined for \( s_i \) and if there is no control guard executable in state \( s \). Note that these rules give higher priority to the execution of guarded commands than to the execution of the environment. Giving the control program higher priority than the environment is necessary if the controller is to be able to respond to state transitions in the environment when they occur.

We define an alternative command to be a set of guarded commands

\[
\langle \pi_1 \to \alpha_1 \parallel \ldots \parallel \pi_n \to \alpha_n \rangle
\]

This is semantically the same as

\[
\langle \pi_1 \to \alpha_1 \rangle, \ldots, \langle \pi_n \to \alpha_n \rangle
\]

but sometimes it will be more convenient to treat groups of guarded commands together. For example, Meta provides a way to enable or disable a set of guarded commands together as a unit, and the alternative command can be used to better show the structure of the control program.

### 3.2 Event-based Monitoring

Reactive system control is event-driven, i.e., an action is taken in response to some event. For example, suppose we have some collection of servers. If the servers’ throughput becomes too low, we wish to create a new server. The act of creating a server may take some time, and even more time may elapse before the throughput is improved. Consequently we wish to take the action at most once per event of the throughput going below some constant. As another example, suppose a valve is to be
opened when a reactor pressure becomes higher than some threshold constant. The action should be only taken once per event; for this particular system, continuing to try to open an already open valve may result in the valve breaking.

The control of a reactive system may include complicated conditions to monitor for, including conditions that specify certain event orderings. For example, a control program might include a condition that becomes satisfied when some event happens during some interval of time, where the interval is also defined by other events. To illustrate this type of condition, consider the condition stating that the load of a machine goes below five within one minute of going above five. This condition would be expressed by a collection of guards watching for the individual events: load going above five, load going below five, and the expiration of a one-minute timer. The time interval for this example is delimited by the events “load going above five” and “timer expired”. (We give an example in Section 3.4 of how such complex conditions may be expressed as automata, which are easily convertible into a set of guarded commands.)

Since much of reactive system control is event-driven, we define an event-driven form of the guarded command. In particular, the guarded command $\langle \pi \rightarrow \alpha \rangle$ (note, $\leftarrow$ instead of $\rightarrow$) is executable in state $s_i$ if $s_{i-1} \not\models \pi$ and $s_i \models \pi$.

The guard $\langle \pi \rightarrow \alpha \rangle$ is only a syntactic shorthand, though; its semantics can be defined in terms of the standard guard construct. For each event-driven guarded command $c \equiv \langle \pi \rightarrow \alpha \rangle$, we introduce a variable $v_c$ having the initial value of $\text{FALSE}$. The command $c$ is then equivalent to the pair of commands:

$\langle \pi \land \neg v_{c_i} \rightarrow \alpha, v_{c_i} \leftarrow \text{TRUE} \rangle$

and

$\langle \neg \pi \land v_{c_i} \rightarrow v_{c_i} \leftarrow \text{FALSE} \rangle$

This definition does raise the question of where the auxiliary variable $v_c$ resides. Clearly, since the control component consists of only programs, it must reside in
the environment. We consequently view the environment $E$ as consisting of the original reactive system environment augmented with any auxiliary state variables. The values of the state variables is therefore part of the state of the environment.

### 3.3 Proving Properties about Programs

A semantics provides a formal specification for the operation of a system. We illustrate through an example in this section how the semantics of the guarded command may be used to prove certain properties about the behavior of a reactive system.

Suppose we have the rule

$$\langle \text{ButtonPressed} \rightarrow \text{TurnLightOn} \rangle$$

This rule states that when the button is pressed, denoted by the predicate ButtonPressed, the light should be turned on by invoking the actuator TurnLightOn; we will leave unspecified for the moment when the light is actually turned on. We might want to formally prove that the light is always turned on after the button has been pressed; this property can be written in temporal logic [MP79] as

$$\text{ButtonPressed} \Rightarrow \Diamond \text{LightOn} \quad (3.3)$$

We prove the validity of Equation 3.3 by showing it to hold over all valid histories of the system. In particular, we must show that for any valid history of the system containing a state $s_i$ such that $s_i \vdash \text{ButtonPressed}$, there exists a subsequent state $s_j$ such that $s_j \vdash \text{LightOn}$.

Let $R$ be a valid history containing state $s_i$. We assume that the actuator TurnLightOn is effective in all states of the environment. Consequently, we have for all $s \in \Sigma$, there exists a triple $(s, s', \text{TurnLightOn}) \in \mathcal{NS}$ such that $s' \vdash \text{LightOn}$. Consequently, by Rule 3.1, $s_i$ is followed by a state $s_{i+1}$ such that $s_{i+1} \vdash \text{LightOn}$. We have thus proved a stronger result, namely $\text{ButtonPressed} \Rightarrow \bigcirc \text{LightOn}$. 
Note in this simple example we relied upon the fact that our control program contained only one rule. Otherwise, we might have had some other rule whose execution would have prevented the light from ever being turned on.

3.4 Example

We present in this section a detailed example using guarded commands. Recall the example in Chapter 1 of the NMRD system. This system includes a number of signal processing components called SigPro's. Part of the management problem is to ensure that an adequate number of SigPro components are running. Since these components are all of the same type, we can treat them as an aggregate. Suppose we have two sensors defined for this aggregate: a load sensor returning some measure of the aggregate workload, and a size sensor giving the number of SigPro processes currently making up the aggregate. We might have a rule stating that when the number of processes in the aggregate goes below two or when the aggregate load remains above five for a minute, then a new process should be created by invoking a Create actuator. The Create actuator is synchronous, not returning until after a new SigPro process has been created or the creation is aborted due to some error condition.

The first part of rule is simple, and is expressed by the following guarded command:

\[(\text{size} < 2 \rightarrow \text{Create})\]

Note that we use the \(\rightarrow\) form of the guarded command rather than the \(\rightarrow\) form. This handles an initial case in which the number of SigPro processes is less than two. Since the Create actuator is synchronous, the system does not enter a new state until after the actuator has completed its work. Should the actuator fail, this rule results in the action being repeated, with the action being retried until two or more SigPro processes have been created.
The second part of the rule is slightly more complex. Its operation is more easily seen by constructing a finite state automaton. Figure 3.3 contains the desired automaton. Each edge of the automaton is labelled with the condition that causes the transition to be taken; the edge is optionally labelled with a desired action. In this case we have three different actuators. The Create actuator creates a new SigPro process. Invoking the StartTimer(60) actuator causes the predicate Timer(60) to become momentarily true 60 seconds later; the timer set by the StartTimer actuator is cancelled by invoking the CancelTimer actuator.

![Automaton Diagram](image)

**Figure 3.3: Expressing a condition using an automaton**

By examining this automaton, we can readily construct the guarded commands needed to enforce this policy. Each transition results in a guard command; we use the auxiliary variable \( state \) to keep track of the current state of the automaton.

For state \( q_0 \) of the automaton, we generate the alternative command

\[
(state = 0 \land load > 5 \rightarrow state := 1, \text{StartTimer}(60))
\]

and for state \( q_1 \) we have the alternative command

\[
\{
state = 1 \land \text{Timer}(60) \rightarrow state := 0, \text{Create}
\mid
state = 1 \land load \leq 5 \rightarrow state := 0, \text{CancelTimer}
\}
\]
Note that in a real-world application, it might take some time after the creation of a new SigProg before the load goes down. Consequently, we might not want to go back to state $q_0$ immediately after creating a new SigProg process. The exact behavior in that case would of course have to be specified more precisely.

### 3.5 Summary

This chapter presents the formal model behind the reactive system architecture used in Meta. A reactive system is controlled by the use of guarded commands. A guard consists of a predicate and an action; when the state of the environment satisfies the predicate of some guard then the corresponding action may be taken.

Since reactive control is typically event-driven rather than state-driven, we extend our guarded command to include an event-sensitive guard. This construct is defined in terms of the normal guard construct by augmenting guarded programs with auxiliary variables.

We explore in Chapter 4 the issues raised by applying this model to a distributed reactive system.
Chapter 4

Consistent Detection

A reactive control system must sense the state of its environment in order to determine when the predicates of guarded commands are satisfied. We would like the controller to be capable of obtaining a simultaneous snapshot of all the state variables within the environment. Such a snapshot allows the controller to determine which guards are satisfied by the global state of the system. Moreover, we would like the controller to react immediately to changes in the global state. As soon as the environment satisfies some condition, the corresponding actions should be taken.

Suppose for the moment that our reactive system consists of a non-distributed, single-threaded program containing both the application being managed as the environment, and the controller, with the controller running as a coroutine with the application. The program is self-contained, not referencing any state defined outside of the program. By calling the controller, the environment in effect is informing the control component of a new state transition for it to act upon. During any interval of time in which the controller is executing, the state of the environment will remain the same as it was when the controller was called. Consequently, the controller may “leisurely” examine the various state variables of the environment; although they are examined sequentially, all the values observed by the controller are
simultaneously valid. Moreover, should the control component decide to take some action upon the environment, the action will be initiated with the environment still being in same state it was in, i.e., the state that caused the action to be taken. The nondistributed, nonparallel nature of this system makes consistent detection and reaction simple. The state of the various environment variables may all be examined in a virtually “simultaneous” fashion. Rule-based reaction can be initiated trivially in the same state that satisfied the rule.

Consider now the execution of a controller in a distributed system. The first problem that arises is determining the global state of the system. Chandy and Lamport make the following perceptive observation in their work on distributed snapshots [CL85]:

The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds—a scene so vast that it can not be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots can not all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define “meaningful” and then to determine how the photographs should be taken.

The scene presented here illustrates a number of the problems we face in determining the state of a distributed system:

- Examining the state of any single component in isolation is insufficient to determine the global state.

- Without reference to some central timepiece, the state of each component cannot be sampled at the same instant in time.

- Sampling the system state should not alter the system’s behavior.
Despite these problems, the observed global state of the system should be consistent, in order for predicate satisfaction to make any sense. This of course requires a precise definition of consistency, which shall be addressed shortly.

A second complication that arises in a distributed reactive system is that since the environment is distributed, the control layer may—and should—also be distributed. By distributing the control layer, control components may reside locally with respect to environment components, permitting greater responsiveness. Moreover, the same desire for fault-tolerance that may have led to the environment being distributed also encourages the distribution of the control program. But distributing the control layer raises even more consistency questions. Not only do we need a consistent global view of the environment, but this view must also be distributed to each control component. Moreover, the actions of the different control components should not be allowed to interfere with each other. A useful benchmark for consistency is that a distributed control program should behave no differently than a centralized control component would behave, except that the distributed version should provide greater responsiveness and robustness.

All these issues are made more concrete and addressed in the next section.

4.1 Determining the Global State

Whereas we previously considered our reactive system to consist of single control layer executing on a monolithic environment, now the reactive system consists of a collection of control and environment subcomponents. We will consider the distributed system to be divided into components, with each component containing a pair of control and environment subcomponents. The pairing is for convenience; we permit either subcomponent to be empty. A component may be thought of as corresponding to a processing unit, for example, a particular process.

Each control component is permitted to execute programs that operate on the
entire distributed environment. Consequently each control component must have a view of the global state. We do not consider the state of the control components to be part of the global state; the global state is only the state of the (distributed) environment.

4.1.1 Model of the System

In the following discussion, we adopt the notation of \[MN91\] and \[BHGP87\] where appropriate. We consider a distributed reactive system to consist of a set of environment and control component pairs. Let \(C_p\) denote the \(p\)th monitoring and control component; let \(E_p\) denote the corresponding environment component. Each environment component \(E_p\) is permitted to communicate with other environment components, but the only control component \(E_p\) communicates with is its local control component \(C_p\). Similarly a control component \(C_p\) is allowed to communicate only with other control components or with its local environment component. If control component \(C_p\) wishes to carry out a remote actuation on environment component \(E_q\), it must communicate with \(C_q\) which carries out the action on \(C_p\)'s behalf. Each environment component must have an associated control component in order for the environment component to be monitored or controlled.

For some execution of the system, the environment component \(E_p\) proceeds through the sequence of states \(s_p^0, s_p^1, \ldots\); this sequence defines the environment's local history. Each state is separated by an event:

\[
s_p^0, e_p^1, s_p^1, e_p^2, s_p^2, \ldots
\]

An event may be local in nature, or it may be a communication event with some other environment component or components. Two types of communication events may occur: a send event denotes the sending of a message to some set of components and a receive event denotes the reception of a message from some other component.
A global state of the system is an n-tuple \((s_1^{t_1}, s_2^{t_2}, ..., s_n^{t_n})\) consisting of some local state value \(s_p^{t_p}\) for the environment component \(E_p\). A cut is an n-tuple of natural numbers \((t_1, t_2, ..., t_n)\) representing the global state \((s_1^{t_1}, s_2^{t_2}, ..., s_n^{t_n})\). A cut is observed if some control component sees that cut as being the state of the global system at some time.

Suppose we were able to simultaneously determine the state of each local component and thereby construct a global state. We define a cut to be valid if it represents a global state of the system that could be seen by simultaneously sampling the different components of the system at some point during the system’s execution. Such a cut is logically consistent, in that we know that it represents an actually occurring global state. However, as illustrated by our sky-watching photographers, obtaining such a view of the system is not generally possible and so we need a more useful notion of consistency. To do come up with such a definition, we first define the notion of causality.

To define causality, we use the happens-before relation of [Lam78]. This relation, denoted by the symbol \(\rightarrow\), is defined over events as follows:

\[
\begin{align*}
\forall j \geq i & \quad e_p^i \rightarrow e_q^j \\
\text{if } e_p^i \text{ is a send event and } e_q^j & \quad \text{is the corresponding receive event.}
\end{align*}
\]

\[\text{the transitive closure of 4.2 and 4.2} \quad (4.3)\]

The happens-before relation can also be defined over states. We define

\[s_p^i \rightarrow s_q^j \text{ iff } e_p^{i+1} \rightarrow e_q^j \quad (4.4)\]

Informally speaking, for states \(s_1\) and \(s_2\), \(s_1 \rightarrow s_2\) if the state \(s_2\) could somehow be dependent upon the state \(s_1\). (Two states \(s_1\) and \(s_2\) are concurrent if neither \(s_1 \rightarrow s_2\) nor \(s_2 \rightarrow s_1\) holds.)

A cut \((t_1, t_2, ..., t_n)\) is causally consistent if the following property holds:
\( \forall p, q, 1 \leq p, q \leq n \ (\forall j, 1 \leq j \leq n, \ (s^j_q \rightarrow s^t_p) \Rightarrow (t_q \geq j)) \) \hfill (4.5)

**Theorem 4.1** Valid cuts are always causally consistent.

**Proof:** Let \((t_1, t_2, ..., t_n)\) be a valid cut. Suppose it is not consistent. Then there exists some \(s^j_q\) such that \(s^j_q \rightarrow s^t_p\) for some \(j, p\) and \(q\) with \(t_q < j\). Since \(s^j_q \rightarrow s^t_p\), we know that \(s^j_q\) happened in real time before \(s^t_p\). Consequently any simultaneously obtained global state that includes \(s^t_p\) must include the state \(s^{t_q}_q\) with \(t_q \geq j\). \(\square\)

The converse unfortunately is not true: cuts may certainly be consistent though not valid. Such a cut is illustrated in Figure 4.1. The states \(s^0_p, s^1_q, \) and \(s^2_r\) are consistent but do not represent a valid cut.

![Figure 4.1: A consistent but invalid cut](image)

Lastly, before exploring protocols for determining the global state, it is useful to define the concept of a *global history*. Recall that the local history \(H_p\) of some environment component \(E_p\) is the states \(s^0_p, s^1_p, ..., \) These states are totally ordered by the relation \(<_{H_p}\) where \(s^j_p <_{H_p} s^k_p\) iff \(s^j_p \rightarrow s^k_p\). We define the global history \(GH\) to be a partial order with the ordering relation \(<_{GH}\):
\[ GH = \bigcup_p H_p \]  \hspace{1cm} (4.6)

\[ <GH \supseteq \bigcup_p <H_p \]  \hspace{1cm} (4.7)

\[ \text{if } s^i_p \rightarrow s^j_q \text{ then } s^i_p <_{GH} s^j_q \]  \hspace{1cm} (4.8)

Given a global history \( GH \), we define a linearization of \( GH \) as a total ordering extending \( <_{GH} \). A totally ordered global history may be specified as a sequence of cuts, \( e.g. \), as a sequence of valid cuts.

### 4.2 Protocols for Determining the Global State

Let the global history for a particular execution of some distributed reactive system be \( GH \). Then the global history \( GH_p \) seen by control component \( C_p \) should satisfy the following properties:

**Consistency** \( GH_p \) is a linearization of \( GH \).

**Agreement** \( \forall p, q, GH_p = GH_q \).

The consistency property states that each control component sees a logical view of what happened. The agreement property states that all control components see the same sequence of cuts. Agreement is important as it prevents different control programs from concluding different things about the order in which events occurred. The agreement property will also be important when we discuss in Chapter 6 methods for fault-tolerance.

### 4.2.1 Determining the Global State in a Synchronous System

First consider a simple distributed system, in which components run in synchrony with each other.
**Protocol 4.1** The execution of the system proceeds in rounds, with each round consisting of the following three phases:

- Each environment component $E_p$ proceeds to its next state $s^{t_p}_p$, and sends the new state value to the corresponding control component $C_p$. (This is the only phase in which the environment component is active. For the initial round, the environment component does not execute a state transition but merely transmits its initial state value to its corresponding control component).

- Then each control component $C_p$ sends the new state $s^{t_p}_p$ to all the other control components.

- Lastly, each control component $C_p$ receives all the new state values sent in the previous phase, and uses these values to form the cut $(s^{t_1}_1, ..., s^{t_n}_n)$.

**Claim 4.2** *Protocol 4.1 satisfies the Agreement and Consistency properties.*

The basis for this claim is obvious. By operating in synchrony with each other, each control component sees the same sequence of valid cuts. Unfortunately, real-world systems often do not have the degree of synchrony required by this protocol. Consequently, we next consider the asynchronous system.

### 4.2.2 Determining the Global State in an Asynchronous System

Now consider an asynchronous system, one in which messages take an unbounded time to be delivered and where each component operates independently, without reference to a global clock. The asynchrony of the system makes it difficult to obtain even a causally-consistent cut. The problem is best illustrated by the use of a figure. In Figure 4.2, component $C_q$ receives a copy of component $E_p$'s state $s_p$ from $C_p$ and as a consequence takes some local action, resulting in a new local environment state
$s_q$. The new state $s_q$ is then sent to control component $C_r$. Nothing in our asynchronous model prevents component $C_r$ from receiving $s_q$ from $C_q$ before receiving $s_p$ from $C_p$. However, the cut then formed by $C_r$ consisting of states $s_p', s_q$ and $s_r$ violates our definition of consistency (Equation 4.5).

Figure 4.2: An inconsistent cut

To solve this problem, the delivery of the message from $C_q$ to $C_r$ should be delayed until after $C_p$'s causally-previous message has been delivered to $C_r$. It is useful to assume the existence of a communications protocol that ensures messages are delivered in an order that respects causality. This protocol, called CBCAST, is implementable for asynchronous systems assuming a fail-stop model [BJ87b]. In particular, if messages $m_1$ and $m_2$ are delivered to the same process, then the CBCAST protocol guarantees that the following property is satisfied:

**CAUSALITY** if send($m_1$) $\rightarrow$ send($m_2$) then $m_1$ is delivered before $m_2$.

To implement this protocol requires inhibition, the delaying of an event that would otherwise take place. If the CBCAST protocol had been used in Figure 4.2, then $C_q$'s message would not have been delivered to $C_r$ until after $C_p$'s message had been delivered; in particular, if the protocol had received $C_q$'s message first, then it would have delayed its delivery. The CAUSALITY property can be trivially implemented by having each message contain all other messages previously received
by the sender. Much more efficient implementations exist, making the protocol not much more expensive than a simple reliable delivery protocol [BSS91].

The CBCAST protocol alone does not conceal all the anomalies possible in an asynchronous system. Consider Figure 4.3. In this figure, control components \( C_p \) and \( C_s \) are independently broadcasting to components \( C_q \) and \( C_r \). Component \( C_q \) receives \( C_p \)'s message first while component \( C_r \) receives \( C_s \)'s message first. Although this does not violate the causality property, it can still lead to inconsistencies. Suppose that \( C_p \) and \( C_s \) are disseminating new local state values. Then \( C_q \) and \( C_r \) will see these states in a different order. Although \( C_q \) and \( C_r \) may both come up with valid total orderings of the global history, their linearizations will be different.

![Figure 4.3: Inconsistent ordering of message delivery](image)

The ABCAST protocol of [BJ87b] solves this problem. In addition to satisfying the CAUSALITY property, it also ensures

**ORDER** if messages \( m_1 \) and \( m_2 \) are delivered to any two components then they are delivered in the same order at both components.

Using the ABCAST and CBCAST protocols simplifies the development of protocols for disseminating the global state. We first present a protocol based on CBCAST. Although the cuts created by each control component are consistent, different control components may come up with different linearizations of the history. By substituting
the ABCAST protocol for the CBCAST protocol, we then come up with a protocol that provides each control component with the same linearization of the global history.

**Protocol 4.2 (CBCAST-based)** Each environment component notifies its corresponding control component of its initial state. Then after each environment state transition, each environment component notifies its corresponding control component of the new state value. Each control component handles the receipt of new state values as follows:

- Upon receipt of a new local state $s_p$ from $E_p, C_p$
  - forms a new cut containing this value, and
  - broadcasts this state $s_p$ to all other control components using the CBCAST protocol.

- Upon receipt of a new state value from some other component, $C_p$ forms a new cut representing the new global state.

**Theorem 4.3** The use of Protocol 4.2 ensures that each process observes a linearization of the global history.

**Proof:** Let $GH$ be the global history for a given execution of the distributed reactive system. Let $GH_l$ be the global history as seen by component $C_l$. We require that all communication between environment components satisfies the CAUSALITY property.

Each time the control component $C_l$ receives a new state value, either from its local environment component $E_l$ or from some other process, it forms a new cut. This cut consists of the new state of $E_l$ along with the last states seen from the other components. Let this cut be $(t_1, t_2, ..., t_n)$. 
We first show that this cut is consistent, i.e., that Equation 4.5 holds:

$$\forall p, q \forall j \left( s_q^j \rightarrow s_p^{t_p} \right) \Rightarrow (t_q \geq j)$$

We assume that an environment communicates its local state values in chronological (FIFO) order to its corresponding control component. We also assume that each control component initially starts out with a consistent cut, so that Equation 4.5 initially holds. Arguing inductively, let $s_p^{t_p}$ be the newly received state.

Consider first the case where $p \neq l$, so $s_p^{t_p}$ is the state of some remote process. We need to show that

$$\forall q \forall j \left( s_q^j \rightarrow s_p^{t_p} \right) \Rightarrow (t_q \geq j)$$

Suppose not, i.e., $\exists s_q^j$ such that $s_q^j \rightarrow s_p^{t_p}$ and $t_q < j$. If $q = l$, we would have that $s_q^j \rightarrow s_p^{t_p}$ but $t_l < j$; this would imply that some other component has heard a more recent state for $E_l$ than $C_l$ has, an impossibility. For $q \neq l$, the CAUSALITY property ensures that state $s_q^j$ is delivered before state $s_p^{t_p}$. By definition of the cut, the last message $l$ heard from $q$ was the state $s_q^{t_q}$. Therefore, either the state $s_q^j$ was delivered prior to the state $s_q^{t_q}$, or $j = t_q$. Because the CBCAST protocol was used, we know that $s_q^{t_q} \not\rightarrow s_q^j$, so $j \leq t_q$. This contradicts the assumption that $t_q < j$.

The second case is where $p = l$, i.e., the new state is from $C_l$'s own local environment. We need to show that

$$\forall q \forall j \left( s_q^j \rightarrow s_l^{t_l} \right) \Rightarrow (t_q \geq j)$$

Suppose there exists an $s_q^j$ such that $s_q^j \rightarrow s_l^{t_l}$. This implies that there is some chain of communication events beginning with $C_q$ sending out state $s_q^j$ and ending with the local environment $E_l$ sending $s_l^{t_l}$ to $C_l$. Since control components are assumed to not talk directly with remote environments, this sequence must pass through $C_l$. Since $s_q^{t_q}$ is the last state $C_l$ has heard from $C_q$, we must have $t_q \geq j$.

Since each control component reports the states of its associated environment using CBCAST, they are received in order. Consequently $GH_l = \bigcup_l H_l$ and $<_{GH_l} \geq$
\[ \bigcup_l <_{H_l}. \text{ Since each cut is consistent, we also have that if } s_l^i \rightarrow s_q^j \text{ then } s_l^i <_{GH_l} s_q^j. \]
Therefore \( GH_l \) is a linearization of \( GH \). \qed

To understand the nature of the consistent cuts seen by this protocol, it is useful to introduce some more definitions:

**Detect**\((\pi, p, t)\) iff, using Protocol 4.2, the cut formed by monitor \(C_p\) while in state \(t\) satisfies predicate \(\pi\).

**Possibly**\((\pi, p, t)\) iff there exists a consistent cut that includes state \(s_p^t\) in which \(\pi\) is true.

**Definitely**\((\pi, p, t)\) iff all consistent cuts that include state \(s_p^t\) satisfy \(\pi\).

Our consistent cut protocol satisfies the following two properties:

\[
\text{Detect}(\pi, p, t) \Rightarrow \text{Possibly}(\pi, p, t) \tag{4.9}
\]

\[
\text{Definitely}(\pi, p, t) \Rightarrow \text{Detect}(\pi, p, t) \tag{4.10}
\]

However,

\[
\text{Possibly}(\pi, p, t) \nRightarrow \text{Detect}(\pi, p, t) \tag{4.11}
\]

Moreover, as we have already noted, different control components may come up with different linearizations of the total history. If control components are executing programs sensitive to the order in which events occurred, then different control components may reach different conclusions about the order in which events occurred. For some systems, the control components may be managing programs that are sufficiently independent so that this is not a problem. However, if the different control components are replicas cooperating together for fault-tolerance, then this problem becomes fatal. We explore fault-tolerance concerns more in Chapter 6.

The following protocol ensures that the different control components come up with the same linearization of the global history.
Protocol 4.3 (ABCAST-based) Each environment component notifies its corresponding control component of its initial state value and of its new state value after each state transition. Each control component executes the following protocol:

- Upon receipt of a new local state from $E_p$, $C_p$ broadcasts this state to all control components, including itself, using ABCAST.
- Upon receipt of a new state value delivered by the ABCAST protocol, $C_p$ forms a new cut representing the new global state.

Note that in Protocol 4.2, a control component accepted the state of its local environment directly. As a consequence, each control component will always see under Protocol 4.2 its local state changes before those of other components, resulting in each component generating a different linearization. This problem is solved in Protocol 4.3 by having each control component use the ABCAST protocol to disseminate the new state value to all control components, including itself, and not accepting the new state value until the new value is delivered by the ABCAST protocol.

Theorem 4.4 The use of Protocol 4.3 ensures that all control components see the same linearization of the global history.

Proof: That each control component sees a linearization of the global history follows from Theorem 4.3. Each control component sees the same linearization by virtue of the ordering property provided by the ABCAST protocol.

The CBCAST and ABCAST protocols operate by delaying message deliveries to ensure that the ordering constraints are satisfied. However, although the cuts formed by these protocols are consistent, they are not necessarily valid. Consequently, a predicate may be true in a valid cut of the system, yet this fact will never be detected by the controller.
Obtaining a valid cut imposes a larger amount of blocking on the application than the ABCAST or CBCAST protocols. In general, to construct a valid cut, a control component must be able to block its corresponding environment from executing. At worst, the execution of each environment component must be blocked from the time its value is reported to the corresponding control component until the time that all control components have received all the current states of all the environmental components. Since the environments are all blocked, the (valid) cut formed represents the current global state of the system. This protocol essentially forces the asynchronous system to behave synchronously, and may theoretically result in arbitrarily long periods of blocking.

Although the CBCAST and ABCAST protocols can also suffer from arbitrarily long periods of blocking, reasonable implementations of these protocols make excessive delays appear as failures, forcing tardy processes to crash. Moreover, the CBCAST and ABCAST protocols do not require the control component to block the environment, and they only delay the control component from receiving certain events. As a consequence, using the CBCAST or ABCAST protocols to construct consistent cuts will generally impose much less of a burden on an application than would a protocol that constructed valid cuts.

4.3 Summary

This chapter explores the issues behind detecting when the predicate of some guard is satisfied.

Distributing the control and environment layer of a reactive system complicates determining when a predicate is satisfied, since the state of the system is decentralized. Each control component sees a linearization of the global history of the system as a sequence of cuts, representing a sequence of global states the system progresses through. Ideally, cuts should be valid: *i.e.*, they should denote an actually occurring
global state. However, in general, this is only possible in a synchronous system or in a system made to operate synchronously by introducing blocking.

Since valid cuts are often impractical to construct, a control component may instead obtain a sequence of causally-consistent cuts, which preserve causal dependencies between states as defined by the happens-before relation. Such cuts are achievable through the use of causality-preserving communications protocols. However, causality only defines a partial order on the global history of events, and consequently different control components may see different linearizations of the global history. To ensure that each control component sees the same linearization of the global history, the communications protocol must also provide agreement. Achieving causal consistency is considerably less expensive than achieving validity in an asynchronous system.

Detecting that a guard is enabled for execution is only half of the problem. The other half is reaction; we explore in the next chapter the problem of consistent reaction.
Chapter 5

Consistent Reaction

Once it is determined that some guard’s predicate is satisfied, then the corresponding action is to be taken. Reaction raises additional consistency questions; we consider two in particular:

- In what state does the reaction occur in?

  Given that the controller has constructed a cut—representing a global state—and this cut satisfies some guard’s predicate, what state is the system in when the corresponding reaction takes place?

  Note that this question is moot unless the controller has formed a valid cut. If the controller has only constructed, for example, a merely consistent cut, then the observed global state may not have actually existed; worse yet, the guard’s predicate may not have been satisfied by the actual state of the system.

  Suppose however, that the controller correctly determines that some condition predicate is satisfied. Is the system in the same state it was in when it first satisfied the predicate? If not, is the system in a state in which the predicate still holds? Is it adequate to know only that the predicate held at some point in the past?
• How is the action performed?

Distributing the control layer forces extra problems upon reaction. Just as concurrent accesses to a database may interfere with each other, so too can concurrently executing actions interfere with each other. Moreover, any thorough treatment of reaction must at least consider the problem of failures. Coping with failures in a database is a difficult but well-understood problem; coping with actuator failures is much more complicated and greatly depends upon the application.

This chapter explores these issues in greater detail, beginning with the first question.

5.1 Initiating a Reaction

According to the semantics presented in Chapter 3, a guarded command is executed immediately after its guard is satisfied by the system state (assuming the absence of interfering commands). Same-state, immediate reaction requires close synchronization between the environment and the control program; as shown in Chapter 4, such synchronization is expensive. However, in some applications, a controller may be able to predict in advance when to react, and time its reaction so that it is carried out in precisely the right state; we call this predictive reaction. For other applications, the semantics of the system are such that delayed reaction is acceptable. We consider these three different modes of reaction in this section.

5.1.1 Immediate Reaction

To react with the environment being in the same state it was in when a predicate was first enabled generally requires the environment to block upon reporting each state change to the control component. This simply is not possible in most real-world
process control applications. If a controller is to open a valve as soon as the reactor pressure reaches the boiling point, it certainly can not suspend time while the sensor information is processed and the appropriate actuator triggered. However, most non-realtime software applications can reasonably be blocked. As shown in Chapter 4, blocking is crucial to providing consistent control, for without blocking, the controller can not even achieve a causally-consistent cut of the global state.

Achieving immediate reaction requires more than just a causally-consistent cut; it requires a valid cut, which is only possible in a synchronous system or in an asynchronous system that operates in lock-step, as discussed in Chapter 4. A protocol for achieving a valid cut may be trivially extended to provide immediate reaction. In such a protocol, control components keep their corresponding environments blocked even longer than before; in particular, the environment is kept blocked until the control component initiates the next action or determines that there are no actions to be carried out.

There are other complicating factors, however. Consider the case where a given state of the environment satisfies multiple guards. Once an actuator is invoked, the environment proceeds to a new state. If a control program consists of a set of guards which may be simultaneously enabled in the same state, then these rules interfere with each other. Immediate reaction can be guaranteed for only one rule in a set of interfering rules. Determining which rule to execute may be decided by fairness considerations. Additionally, priorities may be assigned to rules, to give some rules preference over others.

The situation is complicated when the enabled guarded commands are in different control components of the distributed system. Then the control components must agree amongst themselves which rule is allowed to execute, again basing the decision upon fairness and priority considerations.

Rather than requiring the system to execute a guard in the same state as the state
that enabled the guard, a more practical requirement would be merely to guarantee that the enabling predicate still holds when the action is taken. This weaker requirement may reduce the amount of blocking required; components whose execution can not possibly effect the predicate in question may be safely allowed to continue.

5.1.2 Predictive Reaction

Aside from the issue of blocking, reaction generally can not occur immediately. For even if a controller could detect a valid cut that satisfies some predicate without blocking the environment, it still takes time for the controller to react. Operating in a distributed environment may further exacerbate the problem, since interprocess communication may be required for the controller to receive state information and/or to carry out the action.

One approach to this problem is to predict when some predicate will be satisfied, and initiate the action in advance, taking into consideration the latencies required to actually perform the reaction. Given a valid cut, such an approach can proceed without further blocking the environment. This approach is especially appropriate for process control systems, where we may often be able to approximate the short term behavior of the system through the use of mathematical functions.

As an example, suppose we know the rate at which the temperature is rising in some component and we desire to open a valve as soon as some temperature is reached. If we also know the exact latencies involved in computation and message transmission, we can initiate an action sufficiently far in advance to ensure that the reaction is carried out when the system actually satisfies the desired condition. Of course, unless the bounds on latencies are exact, the reaction may occur too early or too late. Typically the latencies will be given as intervals of time, and the reaction can be scheduled to occur within some bounded interval of the event of interest. Clearly this type of approach raises many problems, including the problem of interference.
Two rules now interfere if the execution of one will cause the enabling condition of the other to no longer hold.

The interested reader is referred to [Sho88] for a theoretical discussion of predictive reaction.

### 5.1.3 Delayed Reaction

Rather than providing any guarantees about the timeliness of reaction, a system may merely guarantee that reaction will eventually take place after the event of interest. By allowing delayed reaction, the process of detecting that a guard is satisfied may be separated from the process of reaction. In particular, the environment need not be blocked after a local state is obtained. Nor do we need to concern ourselves with the effects of other actions that may have been executed in the interim. This freedom is especially desirable in a distributed system, since control components may execute rules concurrently.

Clearly this approach offers simplicity, and for certain types of applications it may prove to be quite adequate. In particular, there are many predicates which, once true, will remain true until the controller takes corrective action. For example, consider a rule that tests if some part of the system is deadlocked, and if so, initiates an action that breaks the deadlock. Deadlock is a stable property; once the system is deadlocked, it remains so, until some external action—in this case, the action of the rule—is taken to break the deadlock. Executing this rule with the delayed reaction semantics is sufficient to ensure the liveness of the system. This example illustrates a case where it may not be necessary to know exactly when the reaction will be invoked; we may be content to know that the expected reaction time is "small" and that the variance is "low".

As an other example, suppose we have a rule that takes corrective action when a printer runs out of paper. It probably doesn't matter if one second versus five
seconds elapse before the controller redirects the print job elsewhere or notifies an operator. The predicate such as "printer out of paper" can also be viewed as being stable—remaining true until some corrective external action is taken.

The two examples just cited specify predicates that remain true until the controller takes corrective action. The work of [Spe89,SK88] seeks to formalize these concepts. This work defines three classes of predicates, which we present here with a slight change of terminology. A monotonic predicate is a pure stable property: once true, forever true. Such predicates typically test to see if some condition ever held. A dependent monotonic predicate is one which under certain conditions behaves as if it were monotonic. For example, the predicate "the system is deadlocked" is dependent monotonic in the presence of a deadlock detection and recovery mechanism; the predicate remains true as long as the condition "corresponding corrective action taken" does not hold. The last type of predicate is the nonmonotonic event; the fact that such a predicate held in some past state does not mean it will hold in any subsequent states.

Predicates which are dependent monotonic because they are stable—remaining true until some external action is executed—are of particular significance to a controller, since for such predicates the controller is guaranteed that the predicate will still hold when the reaction takes place. Many real-world reactive systems do exhibit this type of stability. Recognizing this behavior allows for the less expensive delayed reaction semantics to be used instead of a more expensive semantics that imposes additional requirements upon the state of the system at the time of reaction.

5.2 Making Actions Atomic

Once a reaction has been initiated, it should be completed without interference: a reaction should appear as an atomic action—a single, indivisible action despite failures or other concurrently executing guards. Even if the action of a guarded
command executes not just one action, but a sequence of actions, the collection as a whole should appear as an atomic action.

The notion of atomic actions is much like the transaction concept in database theory and so it is useful to examine for a moment the traditional transaction. Typically, a transaction is characterized by the following set of properties [HR83,GMB+81]:

**Consistency.** A transaction preserves the set of “invariant” properties which the system must satisfy. Although in the course of execution, a transaction may momentarily cause some property to be violated, all such properties are again satisfied by the time the transaction is completed.

**Isolation.** Each transaction executes in isolation, as an indivisible action; different transactions do not interfere with each others’ operation.

**Failure Atomicity.** Either the transaction completes successfully, or leaves the system in a state as if the transaction had never run.

**Durability.** Once completed, a transaction’s effects persist even after subsequent failures.

This list of properties is not perhaps exhaustive nor is it minimal as some of these characteristics are related. Moreover, not all of these properties apply directly to reactive system control. The consistency property is an abstract property and raises the question of granularity. The rules governing a reactive system may be thought of as describing a set of “invariants” the system is to always satisfy; guards are executed whenever the system violates those rules.

The failure atomicity and durability properties are closely related. The failure atomicity property states that in the presence of failures, either a transaction completes with lasting effect or the transaction has no effect; the durability property emphasizes that failures occurring after a transaction completes have no effect on the results of that transaction.
5.2.1 Failure Atomicity

Formally stated, the failure atomicity property requires that for a given action $\alpha$ and a failure $\dagger$,

$$a || \dagger \equiv a \uplus \text{skip}$$

where $||$ denotes parallel execution and $\uplus$ denotes nondeterministic choice [Bla91]. To satisfy this property, one of two approaches may be taken. The first approach is to undo whatever partial effects a transaction may have had before it failed. This requires for each actuator or action a corresponding undo action. Unfortunately, this may not always be possible; a classical example is the dispensing of money from an automatic teller machine.

Alternatively, a failure can be compensated for by advancing the system to the state it would be in if the action had finished. If actions are idempotent, then this may be achieved by restarting the action. For non-idempotent actions, a more complicated approach may be necessary, to compensate for the partial effects of the failed transaction\(^1\).

Coping with failures in a reactive system is an inherently difficult problem. One approach is reduce the probability that failure can occur. By running multiple replicas of a given component, with each replica failing independently of the other replicas, we can decrease the probability of a total component failure. Achieving fault-tolerance through replication is discussed more in Chapter 6.

5.2.2 Isolation

The action of a single guarded command may reference not just one, but a sequence of actuators, with the actuators located a different components in the system. The

\(^1\)Our all-or-nothing failure atomicity property precludes partial effects. However, for some applications it may be reasonable to weaken the failure atomicity property, only requiring that the transaction leave the system in some "safe" state, a state that satisfies the system consistency properties.
isolation property requires that if two such compound actions are executed concurrently, then their executions do not interfere with each other. Formally, we require that for any two actions $\alpha_1$ and $\alpha_2$,

$$\alpha || \beta \equiv (\alpha; \beta) \sqcup (\beta; \alpha)$$

where the semicolon denotes sequential execution [Bla91]. In other words, the execution of two actions in parallel should be equivalent to some serial execution of those two actions.

Serializing the execution of concurrently executing programs automatically ensures that they execute without interference. However, in eliminating parallelism, the strict interleaved execution also effectively destroys whatever increased throughput that might have otherwise been offered by a distributed system. One could even go so far as to say that preventing nonconflicting actions from executing concurrently is in itself a form of interference—this is certainly true if timing constraints are part of the specification.

So then, to achieve maximal parallelism, we need to know when actions may be executed in parallel without conflict. In a database system, this problem is fairly straightforward: two update actions interfere if they both attempt to write the same object. However, in a distributed reactive system, changing one physical variable may result in changes cascading throughout the system, even in remote components. Consequently the effects of each action must be thoroughly understood in order to determine if two actions interfere.

We assume that each environment component offers atomic actions, and that all actions upon an environment component are channeled through the corresponding control component. These assumptions provide low-level atomic actions, much like an atomic “test-and-set” hardware instruction. However, the action of a guard may be specified as a sequence of such actions. If we have two actions $\alpha \equiv a_1, a_2, ..., a_n$
and $\beta \equiv b_1, b_2, ..., b_m$, then although the individual $a_i$ and $b_i$ may be executed concurrently without interference, we may still not have the property that

$$\alpha \parallel \beta \equiv \alpha; \beta \sqcup \beta; \alpha$$

This property will be violated if the individual atomic actions conflict, where two actions $a$ and $b$ are defined to conflict if

$$a; b \not\equiv b; a$$

i.e., if the outcome is dependent upon the order of execution. Given an execution history of executing any two action sequences $\alpha$ and $\beta$ in parallel, we can construct a graph with a node for each atomic action and a directed edge between any two nodes that conflict indicating the order in which they were performed. If this graph is acyclic, then transaction theory [BH87] tells us that there is an equivalent serial execution history, in which the actions of $\alpha$ are either all ordered before those of $\beta$ or all after those of $\beta$. If the execution is serializable, then the two actions were executed without interference.

To ensure noninterfering executions, we can adopt approaches used in transaction systems. For example, two-phase locking [BH87] may be used to ensure that execution histories are serializable. Two-phase locking does require knowing explicitly which actions may conflict with each other. This can either be stated explicitly for each action, or in an axiomatic manner. For example, it may be assumed that actions on different environment components don’t interfere. Given this assumption, to execute an action sequence $\alpha \equiv a_1, a_2, ..., a_n$ using two-phase locking requires only that all the components referenced in $\alpha$ be locked.

The assumption that the actions of different environment components do not interfere may be too weak, since the different environment components are interconnected and acting upon one environment affects others. Alternatively, if our system permits multiple actions to be done in parallel on the same environment, then locking
at the component level is too coarse a granularity, stifling parallelism. Determining how to prevent interference for the arbitrary reactive system is clearly a difficult problem.

“Correct” control of a distributed reactive system can be defined in terms of the behavior of a single controlling task. Even though a system may contain multiple control components, its execution should be equivalent to an execution that a uni-threaded control component could give. Such an execution would automatically be free from interference, since each action would be generated serially. Of course, a uni-threaded controller is not able to exploit the parallelism present in a distributed system. Consequently, we desire that the overall history of the system, including both the actions taken and the global states seen, to be serializable. Combining the ABCAST protocol of Chapter 4 and a system such as two-phase locking will provide a “virtually-centralized” pattern of control.

5.3 Summary

This chapter has explored the problems of consistent actuation in a distributed reactive system. The first issue considered is what state does reaction occur in. A guarded command causes some action to be taken because the system state satisfies some predicate. A strong semantics is to require the system to be in that same state when the reaction occurs. Weaker semantics require only that the predicate still hold, or that the reaction occurs within some bounded amount of time. The weakest useful semantics is that the reaction eventually occurs.

Once reaction is initiated, reaction be performed in a consistent manner. The problem of consistency has a number of facets to it. One is the problem of concurrent actions interfering with each other. This problem can be solved by using some protocol such as two-phase locking that ensures that action sequences are done in a serializable order. Another issue to consistent reaction is the problem of failures.
Coping with failures is a hard problem for reactive systems; one approach is to minimize the chance of failures via fault-tolerant protocols, as discussed in the next chapter.
Chapter 6

Tolerating Failures

To prepare for the discussion of the Meta implementation, we review in this chapter some methods for achieving fault-tolerance in distributed systems. Different failure models have been defined to describe the ways the components in a distributed system might fail [PSL80]. In the most benign model, the fail-stop model [SS83], a processor fails by ceasing operation at some time and remaining quiescent from then on; however, the other processors are notified of the failure. The crash failure model is like the fail-stop model, except that the other processors are not notified of the failure. More general failure models allow for omission failures, where processors intermittently fail to send or receive individual messages. At the extreme end of the spectrum we find the Byzantine failure model, in which processors may exhibit arbitrary, even malicious, behavior.

We assume in this work the most benign model, the fail-stop model. We chose this model since it is generally realizable in the typical computing environment. In particular, the ISIS toolkit upon which Meta is built provides the abstraction of the fail-stop failure model. ISIS transforms a system capable of omission failures into a fail-stop system by providing for retries in the case of omissions and by incorporating a failure detector that notifies other processors when a processor crashes. Since ISIS
operates in an asynchronous environment, the failure detector is not guaranteed to be correct: it may announce a processor to have crashed whereas in reality the processor is just slow. ISIS covers up this mistake when it is discovered by forcing the processor in question to crash. The ISIS toolkit does not address the problems encountered in a Byzantine environment, in which processors behave in an arbitrary manner. However, the assumptions made by ISIS have proven to be quite satisfactory and workable for the typical networked environment. The facilities provided by ISIS are summarized in Chapter 8.

By assuming a fail-stop model, the only failures we need concern ourselves with are announced processor crashes. In general, fault-tolerance against any failure type is achieved via replication, either in space, i.e., by replicating components, or in time, i.e., by redoing computations on backup components. In a centralized system, a failure of the (single) processing element results in a total failure. Consequently, achieving fault-tolerance in a centralized system is only possible by applying a recovery mechanism to the system once it has been made operational again. This amounts to replication in time. Users of the system must wait for the recovery procedure to finish before they can use the system again.

Adding spatial replication to an already distributed system is natural, since the system already contains the infrastructure to support multiple processing elements. Individual components may fail, but such partial failures need not result in the total failure of the system. If components fail independently of each other, then having multiple replicas concurrently performing some function will decrease the probability of a total system failure.

Our model of a distributed reactive system consists of a set of environment and control subcomponents. In this chapter we extend that model to permit subcomponents to be replicated. To the rest of the system, a replicated subcomponent will behave as an unreplicated but nonfaulty entity.
Replication is generally discussed in the context of providing a fault-tolerant service. Instead of a single server handling requests from clients, the service consists of a set of servers appearing to the rest of the system as a single server. In a reactive system, each component can also be viewed as providing a service. In particular, the Meta control components provide services to the other control components, with each control component serving as the interface to the corresponding local environment in addition to executing some part of the global control program.

We examine two approaches to replication in this chapter. The first is passive replication, in which a single replica is active and the other replicas are stand-bys. The second scheme is active replication, in which each replica independently carries out the computation. Passive replication trades off some replication in space for replication in time. In the absence of failures, the passive replicas need not do as much work as an active replica would do. However, recovering from a failure will take more time. Note that passive and active replication are really just endpoints of a spectrum of techniques for replication. After presenting the active and passive replication schemes, we will also present a hybrid technique, the coordinator-cohort method.

6.1 Passive Replication

In passive replication, one replica serves as the primary, providing the desired service. The other replicas are backups, prepared to takeover in case the primary fails. An early paper outlining this method is [AD76] and an example of its use in a real system is presented in [Bar81]. We present here the basic primary-backup protocol.

6.1.1 The Primary-Backup Protocol

In the basic primary-backup protocol, a designated primary replica alone receives all messages on behalf of the replicated service. These messages may require the
server to perform some computation and then return a value; the computation need not be deterministic. Alternatively, a message may simply be for the service's own information. In either case, the message causes some computation to be performed, potentially resulting in a new state for the service. Before this state can be made visible to clients—through a reply message or some other message sent out from the service—the primary must ensure that the backups will know the new state of the service. This guarantees that if the primary should fail, a new backup will be able to bring its state up to the last externally visible state held by the (now deceased) primary. However, only the primary performs the actual computation required to process the request for service.

A client of a service that is replicated using the primary-backup protocol is usually aware of the replication. It sees the service in terms of the primary replica with whom it communicates. When a client makes a request for a service, it must keep track of the status of its request until it knows the request is stable, i.e., until it knows that the primary has forwarded the request on to the backups. Once the request is stable, the client need not keep track of it any further; the resiliency provided by replication ensures the permanence of the request. Note that if a client is making a synchronous request (i.e., a request requiring a reply) then the client knows that the request is stable when it receives the reply, since the reply message can not be generated until after the primary has forwarded the request to the backups. However, if the client is making an asynchronous request (i.e., a request not requiring a reply) then the client must explicitly keep track of the request until it is known to be stable. This is typically implemented by having the primary send the client an acknowledgement after it has forwarded the request to the backups.

Should the primary fail, a new primary is chosen according to some protocol. Clients of the service must now talk to the new primary. Before the new primary is able to service requests, it must first recover its state up to the last externally
visible state that the previous primary had. This can be accomplished in a number of different ways. The simplest is when a primary checkpoints its state to the backups each time it takes an externally visible action. In that case, the new primary has the correct state. However, if the state being maintained by a service is large, a primary may only pass on requests to the backups. Then to recover, a backup must replay the sequence of events seen by the old primary.

The failure of a backup does not generally require any special treatment. However, any failure results in decreased resiliency of the service. To compensated for this, new replicas must be added to the service. A new replica comes on as a backup.

When a backup takes over after the primary has failed, there may be uncertainty as to whether the last message that should have been sent by the old primary was actually transmitted. Two approaches can be used to overcome this uncertainty. The new primary can retransmit the message, potentially resulting in a duplicate; the client must in that case be prepared to handle the duplicate. Alternatively, when the new primary informs clients of its existence—so they will know who to send subsequent requests to—clients can also resubmit any pending requests to the new primary; this method can also result in duplicate replies being generated (potentially a duplicate reply for each resubmitted request). Note that clients must in either approach retransmit any requests which were not stable.

Schemes for making databases fault-tolerant are much like the primary-backup scheme [BHG87]. Typically a database system will need to write the effects of transactions to stable storage before committing; this is analogous to the primary replica making a request stable by giving copies of it to the backups. To recover a database, a checkpoint is used from stable storage, and/or a log may be replayed to reproduce the state the system was in when the failure occurred. This is much like the action taken by a backup taking over as the primary.

For a replicated service to be correct, the behavior of the service should be iden-
tical to the behavior of an unreplicated service in the absence of failures\(^1\). Correct, single-copy behavior is straightforward to implement in the primary-backup scheme, since a single site, the primary, determines the sequence of actions taken. Care of course must be taken to ensure that when a backup takes over as primary, it does so in the same state as the previous primary.

### 6.2 Active Replication

In active replication, each replica is on an equal footing with the other replicas. All replicas service requests from the clients. In order for the replicated service to appear as a single, fault-tolerant service, we must ensure that each replica proceeds through the same sequence of states.

A program whose behavior is completely determined by the sequence of requests received is called a state machine in the literature [Sch90]. To make a state machine fault-tolerant, it is sufficient to run multiple replicas of the state machine, ensuring that each replica starts in the same initial state and receives the same subsequent sequence of requests. Since the behavior of each replica is deterministic, a client of the replicated service will receive an (identical) reply from each replica.

A classical work in using active replication to provide a fault-tolerant service is the replicated procedure call of the Circus system [Coo85]. In this work, a normal synchronous RPC call is transformed into a one-to-many call, in which the client calls a set of replicas, with the multiple identical responses seen as one. Since clients are themselves servers, and can therefore be replicated, a service must handle individual RPC calls coming not from a single source but a set of clients. Consequently, servers need to handle many-to-one RPC calls. The combination of the many-to-one and on-

\(^1\)In the presence of failures, the behavior of a replicated service can be defined as being correct if it is identical to the behavior of a \((k, \Delta)\)-bofo (bounded-outages, finitely often) service [BMST91], a service that can mask a limited number of failures, or outages.
to-many calls results in a many-to-many RPC call, enabling one replicated service to call another replicated service.

### 6.2.1 The State Machine Protocol

Implementing active replication is at least in theory quite simple. Each replica’s behavior must be deterministic and each replica must see the same sequence of events. Given those properties, an active replication protocol must simply account for the redundant requests and responses generated. Failures of individual replicas require no special action to be taken.

If each event or request for service is broadcast to the entire set of server replicas by each client replica, then each replica of the service will receive $n$ messages, where $n$ is the number of (surviving) client replicas. By having each client attach identically determined sequence numbers to messages, the members of the service can filter out duplicates. If a message is synchronous—requiring a reply—then each server replies, again attaching a sequence number. Clients use these sequence numbers to filter out duplicate replies. Clients may themselves be replicated services in which case a server replies to the entire service constituting the client. This approach can result in a slow client replica receiving a response to a request that it had not yet generated. Since messages are processed strictly in order of arrival, such premature responses do not cause problems; the message will not be processed until after the slow client has processed the message that resulted in the request being generated in the first place.

Note that each individual replica operates autonomously of the other replicas. The correctness of the system in terms of providing a single-server behavior hinges upon the fact that each replica proceeds through the same sequence of events. This can be achieved by using a broadcast protocol that satisfies the ORDER and ATOMICITY properties of Chapter 4.
As with the primary-backup protocol, failures result in decreased resiliency which is compensated for by adding new replicas. To incorporate a new replica into the service, there must be some point at which the new replica has the same state as the other replicas relative to the events the other replicas have processed; from that point on the new replica must receive all messages like any other replica. This means that the new replica must atomically acquire the state of the service from the other replicas, which does require the cooperation of the other replicas.

6.2.2 Optimizing the State Machine Protocol

A variation of the state machine approach is one in which all parties receive the requests, but agree via some implicit mechanism which replica will actually service the request. That replica, called the coordinator, performs then necessary computation and then informs the other replicas, the cohorts, of its action. The replicas decide who is the coordinator for each message by using some deterministic algorithm that does not require any explicit communication. However, the coordinator does need to inform the cohorts through an explicit broadcast what the result of the action is.

Now if a failure occurs, some recovery action is called for, as in the primary-backup scheme. In particular, if a coordinator for some action fails before sending out the termination message to the cohorts, a new coordinator must be elected to run the action to completion. If the action resulted in some message being sent to some process(es) external to the replicated service, then care must be taken to handle the possibility of duplicates.

This approach, called coordinator-cohort, is basically a cross between active and passive replication. Only one replica is active for each request, but by having the request go to each replica, the states of the coordinator (the primary) and the cohorts (the backups) are kept very closely synchronized. Consequently, recovery is straightforward. Furthermore, clients of the a service replicated by the coordinator-cohort
mechanism need not know who is serving as the primary; the request is presented to
the group as a whole. The \emph{coordinator-cohort} mechanism has the benefit of reducing
the actual amount of computation performed in the absence of failures.

6.3 Summary

This chapter reviews the concepts used to provide fault-tolerance in a distributed
system. In passive replication, a primary replica is responsible for processing events.
The other replicas are backups. The primary keeps the backups informed of the
primary’s state so that if the primary fails, a backup is able to continue from the same
state that was last externally visible. In active replication, all replicas independently
process events and proceed through the same sequence of states. The complexity
in passive replication lies in ensuring that the backups are kept informed of the
primary’s state. The complexity in active replication is in ensuring that all replicas
see the same sequence of events. The primary-backup method requires some special
mechanism to be invoked in the case of failure; active replication handles failures
transparently. Active replication requires each replica to behave deterministically;
passive replication does not have this requirement.

A cross between active replication and primary replication is the coordinator-
cohort mechanism. In this approach all replicas know of all requests, but only one
actually performs the required computation.

This chapter finishes the foundation required to construct a reliable reactive sys-
tem. We now turn our attention to a specific implementation of these ideas, the
Meta system.
Chapter 7

The Meta Toolkit

A complete toolkit for reactive system control should provide a way to instrument the environment (i.e., the application) and it should provide facilities for carrying out control. This chapter presents the Meta library, which contains these two types of routines. Instrumentation requires explicit sensor and actuator routines to be added to the existing application components and then declared to Meta. The Meta library provides the necessary infrastructure to support such sensors and actuators, with the infrastructure being provided by a stub that is coresident with each application component.

In addition to providing access to the instrumented component via sensors and actuators, the stub allows the distributed application to be controlled via a guarded command language. Guarded commands are explicitly given to a stub for execution; the commands are represented as strings which are parsed by the receiving stub. The language used to express guarded commands is a postfix-based language designed for efficient translation.

We expect that control programs will usually be generated from a single, high-level description of the system's intended behavior. A dedicated service then would be responsible for handing out the resulting guarded commands to the appropri-
ate components. This architecture gives rise to a single control layer governing the
behavior of a unified environment. Alternatively, the Meta toolkit allows for each
process—each application component—within the encompassing distributed applica-
tion to explicitly specify rules governing the system’s behavior. This approach
results in each application component having its own control layer, with the rest of
the world being seen as its environment. This approach allows for greater autonomy
of application components; at the same time, the collective system’s intended behav-
ior can no longer be characterized by a single high-level description. Which strategy
is more appropriate to use depends upon the application.

7.1 The Meta Stub Library

By linking in the Meta library, a program using the Meta instrumentation routines
will also include the Meta stub. This stub is analogous to an RPC-style stub in
that it handles requests from other components. This stub also includes the guarded
command interpreter, which executes the control rules given to the stub. The overall
structure of an instrumented application is shown in Figure 7.1.

A given program component corresponds to a component of the environment.
Each program component defines a specific base context, consisting of all the sensors
and actuators defined in that context. These base objects are defined by C routines
which the programmer declares to the Meta stub. Each context is an instantiation
of a context class; a context class is the class of components that all have the same
set of base sensors and actuators. (This structure is similar to the notion of entities
in the entity-relation model of database theory [Che76], with a context analogous to
an entity and with sensors and actuators being entity attributes. The E/R model
has been used elsewhere to present monitoring information in a database framework
[Maz90,Sno82,OSS90].)

Contexts are named according to the form context-class(instance). For example,
Figure 7.1: Logical architecture of an instrumented application

the Meta toolkit includes a program called **machine** that provides sensors giving for a computer its load, the people logged on, etc. This program may be run on different machines; the program running on **svax** defines a specific context named **machine(svax)**, while the program running on **bullwinkle** implements the context **machine(bullwinkle)**.

A program must declare to the Meta library which context class it belongs to and which instance of that class it implements. This is accomplished by calling the routine **meta_stub_init**\(^1\), passing as string parameters the context class and context instance. Since the different members of a context class are (typically) just different copies of the same program, each program must compute a unique instance name for itself. In addition to declaring its identity to the stub, the application program must also declare its sensors and actuators.

By instrumenting an application, certain aspects of the application state are exposed via sensors to the monitoring system. Since it is not feasible for a monitor to examine the application state after each instruction is executed, the monitor also

\(^1\)A complete description of all Meta routines is contained in the Appendix.
needs to know when the values of sensed objects change.

The general problem of application instrumentation may be tackled in a number of different ways. The most convenient approach for the user is the automatic generation of instrumentation routines, including the automatic insertion of code that notifies the monitor whenever a sensor changes value. The method of automatic insertion is best handled by a compiler, since knowing when variables change value requires an analysis of the application similar to the kind of analysis done by optimizing compilers.

A simpler approach is to require that the programmer insert manually-generated instrumentation code into the application at the appropriate points. In particular, if some sequence of application code could result in a sensor changing its value, then the programmer is responsible for inserting additional instructions into the application at that point notifying the application monitor of the change. The Issos system [OSS90] follows this strategy of manual insertion but the actual instrumentation routines are generated instead from high-level user specifications.

Issos' focus is on application monitoring while Meta's focus is on application control. Although automatic code generation could be added to Meta as a straightforward extension, Meta currently requires the programmer to both write the instrumentation routines and to insert them into the program. If at some point in a program the value of a sensor function might have changed, then the programmer must add instructions notifying the Meta stub of that fact in order for the change to be observed. While stopping short of automatic code generation, Meta does provide routines to facilitate writing sensor and actuator functions in a uniform manner for different types of values.
7.1.1 The Type System

The Meta system supports a variety of sensor value and actuator parameter types. The base types supported are integers, represented as C longs; reals, represented as C doubles; and arbitrary length strings. Sets may be formed over these base types. To provide support for imprecise sensors, an interval constructor allows for two-element arrays of integers or reals to be formed. Interval values permit a range to be represented, with one value representing the left endpoint and the other value representing the right endpoint.

An additional constructor allows composite types to be formed; we defer discussion of composites until the section on aggregates, Section 7.2.2.

7.1.2 Sensors

Sensors are functions that expose some part of the state of the application to Meta. A simple sensor function might just return the value of some application variable. Alternatively a sensor routine may compute some function. For example, a sensor function might return the value of some flag or compute the difference between the head and tail pointers for some list structure. A sensor may also sample some environmental value external to the process; such is the case with the various sensors provided by the standard Meta machine program. For example, the machine load sensor operates by reading the CPU utilization factor from the kernel.

An application declares a sensor to the Meta stub by calling the `meta_new_sensor` routine, passing as parameters the address of the function, the sensor's name, the type of the value returned by the sensor, and the polling period. The `meta_new_sensor` routine returns a handle, whose use is described shortly.

The polling period is the maximum time in milliseconds which may elapse between successive samplings of the sensor. It is assumed that if the sensor is sampled at least this often, then all significant state transitions will be seen. Periodic polling is
usually only appropriate for sensors measuring some external value such as machine load. For most applications, the programmer not only knows at what points a sensor changes value, but if the code is well-written, such changes occur at only a few, well-defined places in the program. In these cases, the programmer specifies the constant NEVER_POLL as the polling period. To notify Meta of sensor changes, the programmer inserts calls to the Meta library routine meta_notify, passing the sensor handle returned by the meta_newsensor function as a parameter.

The programmer uses the meta_notify routine to inform Meta when sensor values have changed. This routine takes as a parameter a vector of sensor handles identifying those sensors whose value may have changed, with the handle for a given sensor being the value that was returned by the meta_newsensor routine. The vector can contain zero, one, or more handles. When the vector contains multiple sensor handles, all the corresponding sensors are seen by Meta as changing in a single, atomic event. The meta_notify routine may be invoked with a zero-length vector just to give the Meta stub the opportunity to execute; we will come back to this point in Section 7.1.4.

For variables whose value is updated in many places by the program, it may be more convenient to specify a non-zero polling period rather than adding explicit calls to the meta_notify routine. Since this approach may result in Meta missing changes in sensor value, it should only be used if the control policies are insensitive to intermittent changes.

### 7.1.3 Actuators

Actuators act upon the instrumented application, changing its state. This can be as simple as setting a variable to some value. Alternatively, an actuator may be a complex procedure possibly involving external actions, such as dispensing money at an automatic teller machine or opening a valve in a process control system.
The Meta stub will only call one actuator at a time, running it until completion. An actuator procedure returns either the constant `META_SUCCESS` or the constant `META_FAILURE` depending upon whether the actuation succeeded or failed.

### 7.1.4 The Thread of Control

The Meta stub operates as a coroutine to the instrumented application. As such, it must be explicitly called from time to time to enable it to run. The `meta_notify` routine is used to pass control to the stub. This routine was mentioned in connection with sensors; it can be used to inform the stub that specific sensors may have changed value and should be sampled. More generally though, calling `meta_notify` implies to the stub that the application is in an internally consistent state and that sensors can be sampled as needed.

It is important that sensors be called by Meta only when the system is internally consistent. For example, suppose a program contains two list structures, with sensors giving the size of each list. Suppose furthermore that the program contains a routine that moves an element from one list to the other. If the sensors are sampled after an element has been removed from one list but before it has been added to the other, then the element will not be counted with either list. This type of behavior prevents the move operation from being viewed as an atomic action. The problem described here is often presented in terms of bank transaction systems, for example, moving money from one account to another. The system state is seen to be inconsistent if money intermittently disappears or appears twice.

The notion of internal consistency is intrinsic to the application, and is defined according to the application writer's notion of correctness. By defining a notion of internal consistency, rules written on the state of the system are simplified, with the programmer not having to worry about transient conditions. Meta assumes that the notion of internal consistency assumed in the writing of guarded commands
is satisfied by the application's state when the application calls the `meta_notify` routine.

This non-preemptive execution model eliminates concerns about sampling the system in an internally inconsistent state. To do otherwise would require some synchronization mechanism, some way for the application to selectively enable and disable the stub from executing.

In calling the `meta_notify` function, the application gives Meta the opportunity to poll periodically-polled sensors. Recall that each such sensor has specified for it a maximum allowable interval between successive samples. The application program should call `meta_notify` often enough so that Meta can honor those polling periods. However, regardless of how often `meta_notify` is called, Meta may still be unable to poll often enough, since the underlying operating system may swap out the entire application at some inconvenient moment. For real-time sensors, such indiscriminate swapping presents a serious problem. However, pauses in execution do not alter the execution of nonreal-time applications. The values of nonreal-time sensors do not change if the execution pauses nor do they change while the Meta stub is executing.

In addition to sampling sensors, each time the stub gains control it performs a number of other functions. Since guarded commands may reference both local and remote objects, the stub must obtain new sensor values from other stubs. Furthermore, other stubs may have references to values from this stub. The mechanism of receiving remote sensor values is called `subscription`. When a stub receives a guarded command referencing a remote sensor, it `subscribes` to that sensor, registering itself with the stub handling the context containing that sensor. By subscribing to a sensor, the stub will be informed of new values of the sensor when they are observed by the stub servicing the subscription. When `meta_notify` is called, the stub processes any pending subscription requests and it sends out to existing subscribers any new sensor values. If multiple sensors change value atomically, for example, because mul-
tiple sensor handles were passed to the \texttt{meta\_notify} routine, then subscribers will also see the changes atomically.

Each call to \texttt{meta\_notify} defines a state transition. The accompanying change in sensor values may cause some guards to become executable. The stub maintains the set of alternative commands in a circular list. Beginning at a distinguished starting point, the stub searches for an alternative command having an executable guard. When such an alternative command is found, the first of its executable guards is executed. Although Meta permits multiple guards within an alternative command to be simultaneously executable, only one is executed. The set of guards for the given command is rotated, however, so that the old last guard becomes the first. This provides a degree of fairness for subsequent guard selections.

The execution of a guard may trigger a new state transition, if the action invokes some actuator which in turn calls \texttt{meta\_notify}. The occurrence of a state transition causes the stub to repeat the process of obtaining sensor values and reevaluating guards. However, if an action does not trigger a state transition, the stub will go on to the next executable alternative command, repeating the process of selecting a guard for execution until a state transition does occur or until there are no more executable guards. Before processing the next state transition, the starting point with the circular list of active alternative commands is advanced, again to provide some degree of fairness.

Since actuators are only called by \texttt{meta\_notify}, any call made by an actuator to \texttt{meta\_notify} must be a recursive invocation. Recursive calls are recorded and processed by the top-level invocation of \texttt{meta\_notify} in an iterative fashion. In addition to any recursive calls to \texttt{meta\_notify}, the reception of sensor subscriptions and actuation requests also trigger state transitions.

A Meta stub's current view of the global state is defined by its \textit{sensor cache}. The stub caches the values of sensors as it receives them; each sensor value is assumed
to remain valid until the next time `meta_notify` is called with the associated sensor handle as a parameter or, for periodically-polled sensors, until the time has come for the sensor to be polled again. The stub also caches the subscribed sensor values received from other components. Values from remote contexts are assumed to be valid until a new value is received, or until the remote context fails. Note that sensor values from the local environment are delivered immediately. That, combined with the fact that sensor values are not sent atomically to remote subscribers, means that different control components may see different linearizations of the global history. However, all components do see a causally-consistent linearization, because the causality property is enforced through the use of the CBCAST protocol.

Furthermore, we see that a Meta stub only offers a delayed reaction guarantee, since a stub does not block its associated application from executing after the stub has sent out sensor values to subscribers. Although different control components are allowed to execute concurrently, a transaction protocol is used to ensure that actions execute in a serializable order, as was discussed in Chapter 5. Note in the special case where all sensor and actuator objects are local, the cut (restricted to those sensors) is indeed valid, and reaction takes place immediately.

It may be inconvenient to instrument an existing application program, so Meta includes as a standard utility a `spy` program that allows the state of running programs to be examined and modified without requiring that the program be linked with the Meta library. This utility operates by using the Unix `ptrace` facility. Global variables of type integer, real, and string may be examined and modified. This utility operates by sending the program a stop signal, reading or writing the specified variable, and continuing the program. Consequently, this program uses a `preemptive` approach. It is not possible to control when the program state is sampled, so the program's state may not be internally consistent. For example, the program might be in the middle of updating some multi-word data structure, such as copying a string. The
main advantage of this program is that existing applications may be “instrumented” without modification to the source code.

In contrast to an uninstrumented application, a Meta program may also consist entirely of sensors and actuators. The Meta machine program is an example of such a program; the program does not directly compute anything but exists only to host the sensors and actuators “instrumenting” the host machine. Since such a program has no main thread of control, it must call the meta_mainloop routine after executing its initialization sequence. This routine, which does not return, can be thought of as forever invoking meta_notify.

In a multithreaded environment, a programmer may be tempted to fork off a thread that, like the meta_mainloop routine, repeatedly calls meta_notify. In such a setting, Meta would no longer be running as a coroutine; sensor and actuator routines could be called by Meta at anytime the thread scheduler permitted Meta to run. Consequently, the programmer would have to insert synchronization code around the body of sensor and actuator routines to ensure that they only execute when the application is in an internally consistent state; such synchronization code raises the possibility of deadlock. These problems are avoided by always treating the Meta stub as a coroutine.

7.2 The Meta Client Interface

The instrumented collection of applications making up the Meta environment may be seen by clients using the Meta library client interface. This interface includes routines to obtain the current value of a sensor and to subscribe to a sensor. It also provides a way to organize subsets of a context class into aggregates. Lastly, it includes routines needed for effecting environment control.
7.2.1 Accessing Sensor Values

A client may subscribe to a sensor using the routine meta_subscribe, resulting in
the client being notified via a call-back routine whenever the sensor value changes.
The call-back routine is also called if the context fails, with an indication that the
sensor value is unavailable, denoted as BOTTOM.

A client program may obtain the "current" value of a sensor by calling the
meta_current_value function. This routine takes a context and a sensor name,
and polls the stub implementing that context to get the sensor’s value from that
stub’s cache. Consequently, this value is not truly current; it is however, a value
present in the cache of the stub implementing that context at the point the request
was received. Just how old the value is depends upon the message latency and how
long ago that sensor was actually sampled by the host stub.

7.2.2 Aggregation

Meta permits members of a context class to be grouped together into aggregates. For
example, a rule might exist stating that all machine programs whose load sensor reads
less than one are to belong to a free machine aggregate. In general, an aggregate
consists of zero or more members of a context class; this aggregate itself becomes a
context. As a context, it contains sensors and actuators; it inherits all the sensors and
actuators characteristic of the context class over which the aggregate is formed. The
type of these objects becomes composite. In particular, if an aggregate is formed
over a context class C containing a sensor S, then the value of the aggregate sensor
$S_C$ is the set of all the members’ values for the sensor S, with each member’s value
tagged with the context name—the instance—of the member providing the value.

If the base context class for the aggregate contains an actuator A, then the
aggregate contains a composite actuator $A_C$. Such actuators are invoked differently
than non-composite actuators. They require two extra parameters: a preference list
of member contexts and the number \( n \) of member contexts on which the action is to be invoked. The aggregate actuation operates by invoking the action on the first \( n \) members of the preference list. If \( m \) of the specified contexts are no longer members of the aggregate or crash while doing the actuation, then the next \( m \) members are invoked from the preference list. The process repeats until \( n \) members complete the actuation. Note that if any actuator returns \texttt{META\_FAILURE} then the invocation is aborted and the aggregate actuator fails; the aggregate actuator succeeds only if it successfully completes on \( n \) members of the aggregate\(^2\).

An aggregate is implemented by assigning some stub to handle the aggregate context; to define an aggregate, a client must assign some stub with the aggregate definition. This is accomplished through the \texttt{meta\_endow\_aggregate} call. This routine takes the name of a stub—named by its base context—along with the name of the new aggregate context and the name of context class over which the aggregate is being formed.

Note that in assigning an aggregate context to a stub, that stub becomes responsible for multiple contexts. This is because all stubs implement their base context, the context that was specified to the \texttt{meta\_stub\_init} call. In addition, stubs may implement any number of aggregate contexts.

Rather than burdening down some specific application program with handling an aggregate definition, Meta provides a standard server utility which may be endowed with aggregate definitions. Such a program is trivial to write; the code for the utility

\(^2\)An actuator may return \texttt{META\_FAILURE} for two types of failures. The first type is a usage failure, \textit{e.g.}, a problem with the parameter types. This type of error prevents the actuator from executing at all. Aborting the aggregate action is the correct behavior in this case, since all other aggregate actuators would likewise return \texttt{META\_FAILURE}. However, an aggregate actuator may also return \texttt{META\_FAILURE} because of an error condition that arose while carrying out the action. In this case, the action might succeed if it was carried out another aggregate member. This argues that Meta should support an additional return status, \texttt{META\_REJECT}, which denotes the case where the action failed locally but could be reasonably retried elsewhere.
is shown in Figure 7.2. The include file `meta.h` contains general definitions for the Meta library; the include file `meta_stub.h` contains definitions that are specific to application instrumentation.

```c
/*
   Bare Bones NPL server
*/

#include "meta.h"
#include "meta_stub.h"

void main()
{
    meta_stub_init("server","",0);
    meta_mainloop();
}
```

Figure 7.2: The Meta Server program

### 7.2.3 Replicating Aggregate Servers

Meta offers the ability to replicate the server that is endowed with an aggregate definition. This is accomplished via an extended form of the `meta_stub_init` routine that takes an extra parameter indicating that the stub should run in the replicated mode. The replicas are managed using the active replication strategy presented in Chapter 6. The implementation of replicated servers is discussed in Chapter 8. By running $n$ replicas, up to $n - 1$ failures may be tolerated.

Meta imposes some restrictions on the server program. In particular, the server program is not permitted to have any sensors or actuators in its base context; its only sensors and actuators are those belonging to the aggregates the server is endowed
with. The replication strategy ensures only that all remote events are seen in the same order at all replicas, hence the restriction on local sensors or actuators.

7.3 The Guarded Command Language

Clients specify control of the environment by giving to Meta alternative commands—sets of guarded commands. The Meta guarded command language is called NPL in honor of the implementor of the first language prototype\(^3\). An NPL guarded command is evaluated by some stub with respect to some context implemented by that stub. A client enables a guarded command \(G\) to be evaluated in context \(C\) by passing the string representation of \(G\) along with the context name \(C\) to the \texttt{meta.npl} routine. This routine sends the guard to the stub implementing that context.

Guarded commands may reference any sensor or actuator object. Objects within the context \(C\) may be referenced simply by name; objects in other contexts are referenced by prefacing the object name with the name of the context. Since each stub contains a guarded command interpreter, commands may be evaluated at any stub. However, remote references are much more expensive than local references. Consequently a guarded command should be written with respect to the context that offers the best performance, minimizing the number of remote references. Meta provides great flexibility in allowing a guard to be evaluated at any stub; however, it relies upon the user to determine where a guard should be evaluated.

The basic alternative command has the form:

\[
\text{predicate Guard actions [ALTERNATE predicate Guard actions]}^*\]

In particular, an alternative command consists of a sequence of clauses, with each clause separated by the \texttt{ALTERNATE} keyword. A clause consists of a predicate, the keyword \texttt{GUARD} (or its variant \texttt{TGUARD}), and the action sequence. The predicate is a

\(^3\)Nancy Thoman; NPL is an acronym for Nancy's Postfix Language.
postfix expression evaluating to an integer. Several variations of the guard keyword are permitted. The command \textit{predicate GUARD actions} has the intuitive meaning that when the predicate is true, the actions may be executed; it is semantically equivalent to the guarded command construct \(\langle\text{predicate} \rightarrow \text{actions}\rangle\) presented in Chapter 3. The variant \textit{predicate TGUARD actions} requires a transition from false to true to occur for the actions to be executed; it is formally equivalent to the expression \(\langle\text{predicate} \rightarrow \text{actions}\rangle\).

The action sequence consists of a sequence of guard operators or actuator invocations. Guard operators govern the enabling and disabling of guards for execution. The \texttt{EXIT} operator states that the entire alternative command is to be removed. Two operators provide a way for one guard to enable another alternative command. The \texttt{NPL} operator takes two parameters, an alternative command and a context, both represented as strings. The alternative command is given to the specified context for execution. In addition to the \texttt{NPL} operator, there is also the \texttt{LNPL} operator; this operator gives the single string parameter to the current context for execution. Since the \texttt{LNPL} operator does not explicitly require a context parameter, it can be used to write general alternative commands that can run in different instances of a context class.

An actuator invocation consists of a postfix expression terminated by the name of an actuator object. The postfix expression need not evaluate to a single parameter; by leaving multiple terms on the postfix evaluation stack, multiple parameters may be passed to an actuator. Meta does not keep type information for actuators, so it can not check to see if the number of parameters passed to an actuator map the number expected. Instead, actuator functions are expected to do their own type checking. The name of an actuator is prefaced with the \texttt{\'} symbol (backwards apostrophe); this allows the translator to distinguish between sensor and function names appearing in the parameter expression from the actuator itself. If the actuator object is defined in
the local context, then Meta is able to deduce that the name refers to an actuator. However, such inferences are not possible for objects in nonlocal contexts. The context may not be currently available when the expression is parsed. Consequently, to avoid this ambiguity actuator names are uniformly prefaced with the ' character.

7.3.1 The NPL Expression Language

The predicates and parameters to actuators are written in the NPL guarded command language as postfix expressions. The expression terms are constants of the base, interval, and set types which are supported by Meta along with references to sensor objects. A full range of type-polymorphic functions are built into Meta, including all the standard arithmetic and relational operators. In addition, the possible modal operator is provided for relational operators, for use with interval values. Unmodified relational operators applied to interval values are interpreted as meaning definitely; the possibly and definitely relational operators are defined in Table 7.1. Note that for relation $\mathcal{R}$, POSSIBLY $\mathcal{R} \equiv \neg$DEFINITELY$\neg \mathcal{R}$. These operators are useful when comparing interval values returned by imprecise sensors.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Symbol</th>
<th>DEFINITELY</th>
<th>POSSIBLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>$&lt;$</td>
<td>$\beta &lt; a$</td>
<td>$\beta &lt; b$</td>
</tr>
<tr>
<td>equal</td>
<td>$=$</td>
<td>$(\alpha = a) \land (\beta = b)$</td>
<td>$(\alpha \leq b) \land (\beta \geq a)$</td>
</tr>
<tr>
<td>greater than</td>
<td>$&gt;$</td>
<td>$a &gt; \beta$</td>
<td>$b &gt; \beta$</td>
</tr>
</tbody>
</table>

The NPL translator parses expressions into an internal form represented as a directed acyclic graph. As a consequence of this translation, duplicate subexpressions are eliminated. The efficient postfix representation of expressions allows for speedy translation.
7.3.2 Example Guarded Commands

We present in this section some simple examples of guarded commands. These examples also serve to illustrate some of additional built-in features of the system.

Suppose an instrumented application contains a sensor percent_used that gives the percent of buffer space used by the application; the application has an actuator more_buffers that allocates more buffers. We might have a rule that says if the percent of buffer utilization goes above ninety percent, then allocate more buffers:

\[ \text{percent\_used > 90} \rightarrow \text{more\_buffers} \]

The guarded command for this rule is

\[ \text{percent\_used} \ 90 \rightarrow \text{TGUARD} \ '\text{more\_buffers}' \]

Note the usage of TGUARD instead of GUARD; this is useful if the buffer allocation routine for some reason can’t finish its work by the time the next state transition occurs.

The following rule, when executed in the context of some machine program, sends mail to user “wood” whenever the load on the associated machine goes above two or more than five people are logged in.

\[ \text{load} \ 2 \ \text{OR Size(login)} \ 5 \rightarrow \]

\[ \text{mail}('\text{wood@cs.cornell.edu}', \ 'Load is ', \ \text{load}, \ ', \ 'users, ', \ \text{login}) \]

This rule is written as the following guarded command:

\[ \text{load} \ 2 > \text{login} \ # \ 5 > \text{TGUARD} \]

\[ \text{"wood@cs.cornell.edu" "Load is " load "users, " login 'mail} \]

The Meta mail actuator concatenates its second through last parameters and sends them to the user whose mail address is specified as the first parameter. The # operator is the size function; it returns the size of its operand. This function is
polymorphic, returning a value and type appropriate for the type of its operand. For sets, it returns the cardinality of the set.

Special actuators are built into all Meta stubs for joining and leaving an aggregate. A context may join an aggregate \( A \) by executing a rule that invokes the JOIN actuator with the aggregate name as a parameter; leaving is accomplished by invoking the LEAVE actuator. The following rule, executed in the machine context, states that the FreeMachine aggregate context should be joined if the load is less than 0.5 and it should be left if the load is greater than 1.5.

\[
\begin{align*}
\langle \text{load} &< 0.5 \rightarrow \text{Join("FreeMachine")} \\
\| & \text{load} > 1.5 \rightarrow \text{Leave("FreeMachine")} \rangle
\end{align*}
\]

The corresponding guarded command is:

\[
\text{load} \ 0.5 < \text{TGUARD "FreeMachine" 'JOIN} \\
\text{ALTERNATE load} \ 1.5 > \text{TGUARD "FreeMachine" 'LEAVE}
\]

One guard may enable another as its action; this allows state machines to be coded as guards.

### 7.3.3 Auxiliary Commands

The Meta client library supports two additional facilities for client use. New sensors may be defined for a given context\(^4\) by writing an NPL expression and giving the expression a name. Once such a sensor has been defined, it then may be referenced like any other sensor, appearing in guarded command expressions or being the object of a subscription request.

Meta clients may also define macro variables in a manner similar to defining sensor expressions. Macro definitions facilitate writing complex guarded commands, particularly those guarded commands whose actions invoke other guarded commands; an example of this is shown in the next section.

\(^4\)The definition logically should be for a context class, but this has not yet been implemented.
7.4 Example

We gave an example in Section 3.4 showing how the behavior of a finite state automaton could be represented as a set of guarded commands; we revisit that example in this section, giving the NPL program for those guarded commands.

The rule in that example states that if the number of processes in the SigPro aggregate goes below two or if the aggregate load remains above five for a minute, then a new SigPro process is to be created by invoking the Create actuator. The Create actuator is assumed to be synchronous, not returning until after a new SigPro process has been created or the creation is aborted due to an error.

The first part of this rule was expressed by the guarded command

\[(\text{size} < 2 \rightarrow \text{Create})\]

This guard may be written in NPL as

\[\text{size} 2 < \text{GUARD} \: \text{Create}\]

The behavior for the second part of the rule was represented by the automaton shown in Figure 3.3; for convenience, this figure is reproduced here as Figure 7.3. From this automaton, we had constructed the following guarded commands. For state $q_0$ of the automaton, we had the alternative command

\[
(\text{state} = 0 \land \text{load} > 5 \rightarrow \text{state} := 1, \text{StartTimer}(60))
\]

and for state $q_1$ we had the alternative command

\[
\begin{align*}
\{ & \quad \text{state} = 1 \land \text{Timer}(60) \rightarrow \text{state} := 0, \text{Create} \\
\text{state} = 1 \land \text{load} \leq 5 \rightarrow \text{state} := 0, \text{CancelTimer} \}
\end{align*}
\]

Note that these guards explicitly encode which state the automaton is in. Since Meta allows alternative commands to be dynamically added and deleted, we can instead simply encode the state of the automaton by which NPL alternative command
is active. The *StartTimer* functionality is provided in Meta by the **TIMER** function. This takes one operand, a time value \( t \); the value of the function becomes true \( t \) milliseconds after the guarded command was received for execution (which is equivalent to becoming active). Since timers are associated with specific alternative commands, it is not necessary to explicitly cancel a timer; when an NPL alternative command exits, any timers set by the command are also automatically cancelled by Meta.

We will represent the two states of the automaton by two alternative commands. Since each state has a transition to the other, we define each alternative command as macro, with the command representing state \( q_0 \) being defined as the macro variable \$Q0\) and the command for state \( q_1 \) being defined as the macro variable \$Q1\). The command for state \( q_0 \) is

\[
\text{load 5 > TGUARD } \$Q1 \ 'LNPL 'EXIT}
\]

and the command for state \( q_1 \) is

\[
60000 \text{ TIMER TGUARD 'Create } \$Q0 \ 'LNPL 'EXIT ALTERNATE load 5 <= TGUARD } \$Q0 \ 'LNPL 'EXIT}
\]

To start the automaton running, we use the following rule:

\[
\text{load 5 > GUARD "state 2\n" 'PRINT } \$Q1 \ 'LNPL 'EXIT}
\]
This rule is identical to the rule for state $q_0$, except that we use \texttt{GUARD} instead of \texttt{TGUARD}. If the condition is initially true, we want to immediately take the action; if we had used \texttt{TGUARD}, it would have required the predicate to have been false before going true.

### 7.5 Summary

This chapter presents the Meta routines for instrumenting an application. A program may be augmented with sensor and actuator procedures and linked in with the Meta library to produce a Meta context. Those programs having the same set of sensor and actuator routines make up a context class. In addition to the contexts made available by instrumenting programs, new contexts may be formed by defining aggregates, made up of members of some context class. Aggregates are implemented by designating some program to serve as the aggregate context; these servers may be replicated for fault-tolerance.

Each instrumented program proceeds through a sequence of states, with each state transition signified to Meta by calling the \texttt{meta\_notify} coroutine. Sensors and actuators are written by the programmer to provide Meta with access to the state. Sensors may be polled by the Meta stub periodically or as directed by the application.

Meta clients may control the instrumented application by writing rules in a guarded command language called NPL. Rules are written with respect to some context, though a rule may reference objects in both that context and other contexts; obviously local references require much less overhead. This guarded command language allows for arbitrarily complex expressions to be written, using an easily translatable postfix language. In addition to remote and local sensor references, expressions may be built using the full range of arithmetic, relational, and boolean operators. Additional functions may be defined by the programmer.
After each state transition, the Meta stub examines its collection of rules for executable guards. A guard may become executable in one of two ways. The GUARD operator tests to see if the state satisfies some predicate; the TGUARD operator tests to see if some event has occurred.

The next chapter presents the implementation of Meta, focusing on the special problems presented by operating in a distributed environment.
Chapter 8

The Implementation of the Meta Toolkit

We now turn our attention to the implementation of the Meta toolkit. Each component of a distributed application that incorporates the Meta library thereby contains a copy of the Meta stub. These stubs, providing support for instrumentation and control, must cooperate together to ensure that each stub carries out the programmed control policies in a consistent manner.

The implementation of this distributed toolkit was significantly facilitated by building it on top of the ISIS toolkit [Bir91,BJ87b]. ISIS is a general purpose toolkit for building distributed systems, that addresses many of the consistency problems raised by Meta. So before discussing the implementation of Meta, we review the facilities provided by ISIS.

8.1 ISIS

The key property provided by ISIS is the concept of virtual synchrony. Programming distributed applications in an asynchronous environment is made difficult largely because of asynchrony between processors. Events may be observed by different
processors at different times and in different orders. ISIS attempts to mask this asynchrony, reordering events if necessary to make the underlying communications system appear synchronous to the application programmer. Specifically, the behavior of the system becomes indistinguishable from a synchronous system. Only by consulting a real-time clock does it become apparent that distributed events are not truly happening at each component at the same time.

ISIS also provides support for fault-tolerance. It transforms an asynchronous system capable of omission failures—failures to send or receive messages—into a fail-stop, virtually synchronous system. Of course, in theory, this conversion is impossible [FLP85]. In particular, it is impossible to distinguish in an asynchronous system between a tardy processor and a failed processor. As mentioned briefly in Chapter 6, ISIS “solves” this problem by using timeouts and declaring a tardy processor to be dead. Consequently, ISIS does assume some degree of synchrony in the underlying system. In the rare case where ISIS makes a mistake, ISIS forces reality to conform to its view of the world: it causes the tardy processor to crash.

Virtual synchrony extends even to the treatment of failures: the failure of a process is seen as a virtually synchronous event. This means that other processes will see the failure ordered consistently with respect to other events. Consequently, there is no ambiguity as to whether a multicast was seen at all destinations or at only a subset; either all received the message or none received the message.

ISIS does have limits on how it provides virtual synchrony, though. Virtual synchrony is provided only within the framework of two mechanisms: the process group and the ordered broadcast. Since these mechanisms are fundamental to ISIS, we consider each of them briefly.
8.1.1 The ISIS Process Group

A process group is a set of processes operating in a fully-connected network, with the members of the group seeing events pertaining to the group in a virtually synchronous manner.

The process group provides a convenient way to implement a distributed object. The ordered broadcast protocols accept a group address as the destination, and guarantee that delivery is atomic. Since members see group membership changes synchronously, all group members have identical views of the group membership upon receipt of a message. By using the appropriate broadcast protocol, it can also be guaranteed that all group members receive all messages in the same order. The effect of providing these properties is virtual synchrony. These properties enable the members of a group to agree on actions without requiring explicit communication.

ISIS does not cause failures to be seen as virtually synchronous events globally. However, since members of a group see group membership changes in a virtually synchronous manner, a failure by a member of a group is seen by the other members as a synchronous event.

Providing virtual synchrony is expensive. Each change of group membership requires ISIS to run an agreement protocol, to ensure that all the members witness the change in the same order. Large groups are not recommended; groups with less than eight or so members work best. Instead of outright joining a group, a process may become a client of a group. Clients do not see other clients joining and leaving and so are cheaper to maintain; a group may have hundreds of clients.

The group structure has evolved in ISIS to become closely connected with the broadcast protocols. Perhaps most importantly to the user of ISIS, the fastest inter-process communication in ISIS occurs in the context of a group. We will revisit this point after discussing the ordered broadcast protocols.
8.1.2 Ordered Broadcasts

ISIS provides a suite of broadcast protocols. The strongest guarantees are those provided by the ABCAST protocol. For any two messages \( m_1 \) and \( m_2 \) delivered by this protocol, the following two properties from Chapter 4 are satisfied:

**CAUSALITY** if \( \text{send}(m_1) \rightarrow \text{send}(m_2) \) then \( m_1 \) is delivered before \( m_2 \).

**ORDER** if messages \( m_1 \) and \( m_2 \) are delivered to any two components then they are delivered in the same order at both components\(^1\).

The ABCAST protocol also provides a guarantee of atomicity in the presence of failures, satisfying the following property:

**ATOMICITY** If a message \( m \) is sent to some set of processes \( S \) then either all processes in \( S \) surviving the failure receive the message or none of them do.

The ABCAST protocol is fundamental to ISIS’s provision of virtually synchronous behavior. However, ABCAST is a multi-round protocol, and is therefore somewhat expensive. ISIS also provides a cheaper CBCAST protocol, satisfying only the ATOMICITY and CAUSALITY properties, but not the ORDER property. If it is necessary for message delivery to be virtually synchronous, then the ABCAST protocol must be used. But if a message is destined for a single process, or if the delivery order of concurrently-sent messages is not important, then the cheaper CBCAST protocol can be used. CBCAST typically delivers messages using a single round of communication, with the agreement needed to ensure atomicity implemented via clever piggybacking.

8.1.3 The ISIS Architecture

The ISIS library, like Meta, functions by adding to applications a coresident stub. These stubs communicate with each other via one of two ways. In the general case,

\(^1\)As noted shortly, ISIS has two modes of communication: a normal mode and a bypass mode. Messages routed via the bypass mode satisfy the ORDER property only when they are to the same group.
stubs communicate indirectly via a set of separate protocol servers. A typical configuration might have one protocol server running per machine, with each server servicing all the ISIS applications running on the same machine as the server. The protocol servers are bypassed, however, for communication between group members; each group member’s ISIS stub talks directly to the other group members’ stubs. Communication between a group and clients of the group also goes via the bypass mode instead of going through the protocol servers. Bypass communication is significantly faster than communication routed through the protocol servers, approximately three times faster for the ABCAST protocol [BJ87a,BSS91,Ste91].

A process may receive two types of events from ISIS: message deliveries and announcements of membership changes. An application that uses ISIS declares to ISIS which routines to invoke for which event. When an event occurs, the ISIS stub creates a light-weight thread within the process, which calls the appropriate routine. In a Unix environment, the ISIS tasking system is non-preemptive, running each task as essentially a coroutine. Other tasks are allowed to run only when the active thread blocks. Calls by a thread into most of the ISIS library routines, such as the broadcast primitives, will result in the thread blocking; a thread may also explicitly relinquish control. A routine that blocks while processing a given event must be reentrant.

The ISIS toolkit includes facilities for certain styles of computation. In particular, the coordinator-cohort method of replication discussed in Chapter 6 is supported. To use this facility, all replicas call the coordinator/cohort tool upon the occurrence of some event. The tool picks one of the replicas to be the coordinator by using a programmer-specified function. The cohorts block within the coordinator/cohort tool until they are informed that the coordinator has completed the action. Updates to the state may be passed from the coordinator to the cohorts. Should the selected coordinator fail, the tool repeats the process, iterating until the computation has
8.1.4 Alternatives to ISIS

The virtual synchrony abstraction provided by ISIS greatly facilitates implementing a system such as Meta. However, the cornerstone behind virtually synchronous behavior is the group-based ordered broadcast, and ISIS is not alone in providing protocols with the atomicity, causality and order properties.

Another such protocol is the work of [KTFHB89], which is being incorporated into the Amoeba distributed operating system [MvRT+90]. This protocol uses a hardware multicast to achieve high performance. The suite of $\delta$-t broadcast protocols of the Highly Available Systems project [CDSA90] also provide the atomicity and order properties. Moreover, the $\delta$-t protocols are real-time broadcast protocols, guaranteeing delivery within a certain time. This feature, lacking in ISIS, would be necessary for a real-time version of Meta.

The low-level Psync transport protocol [PBS89] respects causality and atomicity though not order. Information about causal dependencies is available to higher-level protocols, which can use that information to provide stronger guarantees such as order. The primary advantage offered by Psync is that by being a low-level, low-cost mechanism, all interprocess communication may be reasonably routed through it, thereby preserving causality across all communication.

Some higher-level research has focused on the problem of constructing replicated services. The lazy replication method of [LLSG90] is intended for the construction of replicated services supporting query and update operations. This approach includes protocols satisfying the causality, order, and atomicity properties.
8.2 The Meta Implementation

The ISIS toolkit, by providing virtually synchronous behavior, greatly facilitated the development of the Meta toolkit. Meta relies upon the virtual synchrony provided by ISIS to ensure that each control component sees a consistent view of the global history. Virtual synchrony is also necessary for Meta’s replication scheme to work. At the same time, the implementation was constrained by the need to make efficient use of ISIS; the performance of Meta is largely determined by the performance of ISIS.

This section presents some of the key aspects of the Meta implementation. We first discuss how different control components are named and how they communicate with each other. Then we explore in some detail the implementation of fault-tolerant, replicated components.

8.2.1 Naming

Recall from Chapter 7 that each Meta stub implements one or more contexts: the base context defined by the program the stub is coresident with, plus any aggregate contexts that the stub may have been endowed with. Each stub has its own set of control programs that it executes. Those programs may reference remote objects, requiring the stub to communicate with other stubs. A remote context reference has the form of context_class(instance).

For each context context_class(instance) implemented by a stub, the stub joins an ISIS process group named Meta/context_class(instance). Remote references are resolved by using ISIS to map the name context_class(instance) to the group address. The cost of this mechanism is the cost of the call to the ISIS routine pg_lookup, which maps a symbolic group name to a group address. The pg_lookup routine functions by having the local ISIS stub communicate with its protocol server. This server, if ignorant of the group address, must broadcast a message to all the other
protocol servers to resolve the address.

An alternative approach would have been to provide an explicit naming service, rather than using the ISIS \texttt{pg\_lookup} routine to provide naming. Such a service could provide address resolution at significantly lower cost with far less message traffic. Replicating the service would be straightforward; a read-one approach could be used to map a name to an address and a write-all approach could be used to register names. However the actual cost of using ISIS to resolve context names is not that high in practice, since names do not need to be resolved frequently. When one Meta stub initiates communicates with another, they typically establish a \textit{connection}. Each stub caches the address of the other, eliminating the need to invoke \texttt{pg\_lookup} for subsequent references by either stub to the other.

\subsection{Establishing Connections Between Stubs}

The exact procedure to establish a connection with a remote context is as follows. The (local) stub first determines the address of the group by calling the ISIS \texttt{pg\_lookup} routine. Then the stub becomes an ISIS client of the group. Finally the stub sends its initial request to the remote stub, identifying itself by its base context. If the message is for a sensor subscription or for a transaction, then upon receipt of the message, the remote stub creates an entry for the sending context, becomes a client of it, and caches the sending context’s address.

By establishing a client relationship, the stubs ensure that communication between the two stubs goes via the bypass mode of ISIS. The sender, by becoming a client of the remote context, also ensures that ISIS will deliver all messages sent from the remote context before reporting a failure of the stub implementing that context. The reciprocal client relationship is mainly a matter of convenience. For persistent connections such as sensor subscriptions, the remote stub implementing the object being subscribed to would like to be informed if the subscribing stub fails.
The remote stub, by establishing a connection, can therefore use a single mechanism to learn of failures. Moreover, the remote stub, by caching the address, will have that address available for subsequent use without having to call `pg_lookup`.

Suppose a stub seeks to establish a connection—for example, for the purpose of establishing a sensor subscription—with a remote context \( C \) that is not available at the time. This connection should be made active when the remote connection does come on line. Meta provides this behavior by having the stub join the process group `MetaSubs/C`. When the context comes on line, the stub implementing the context will broadcast a message to the `MetaSubs/C` group. All stubs with pending connections to this group can establish them at that time. (A better approach would be to only join the `MetaSubs/C` group if the context is not available, but then care would be required to avoid any potential race conditions between the address lookup and failures and restarts.)

Establishing a client relationship is necessary for a system built using ISIS both to obtain fast communication and to ensure failures are reported correctly. A system built without using ISIS would probably find it convenient to combine the failure detection and reporting facility with a naming service.

### 8.2.3 Handling Subscriptions

To subscribe to some remote sensor, a stub establishes a connection as just discussed, requesting a sensor subscription. The receiving stub adds the subscriber to the list of subscribers for the specified sensor and sends to the new subscriber the current value of the sensor. Subsequent changes in sensor value are reported to all the subscribers.

Ideally, sending out subscriptions should be atomic. Each time a stub observes a state transition, it should send out all the changed sensor values to all the subscribers atomically. However, this property is difficult to provide using ISIS. The problem arises in connection with replicated stubs. Since we will discuss the implementation
of replicated stubs in the next section, we touch here only on those aspects relevant to subscriptions. A set of replicated stubs is identified by its base context, say $C_1$. Suppose these replicas are executing a control program calling for a subscription to a sensor in context $C_2$. Then each replica becomes a client of the group $\text{Meta}/C_2$. To inform the replicas of $C_1$ of new sensor values, a stub implementing $C_2$ will send a message to the group $\text{Meta}/C_1$ using the ABCAST protocol, thereby ensuring that all replicas receive the message atomically.

![Diagram](image)

Figure 8.1: Group structure for handling subscriptions

Figure 8.1 illustrates the group structure. Three replicas implement the context $C_1$. These replicas have a subscription for an object in the context $C_2$, which is implemented by two replicas. Each replica in $C_1$ becomes a client of the group for $C_2$ and vice versa; the client-group relationship is indicated in Figure 8.1 by lines going from each group to its clients.

Sending out subscriptions to a set of different subscribers atomically would require broadcasting a message to a set of groups; this style of communication is not supported by ISIS. However, all changes constituting a state change are seen by individual subscribers, replicated or not, as atomic events. This is trivially ensured by sending out all the new values in a single message. Meta's failure to provide atomicity of delivery means that a failure of the sender while sending out sub-
criptions could result in only a subset of the subscribers receiving the last value. Moreover, different subscribers can receive concurrent subscriptions in different orders. As a consequence, Meta is not able to ensure that all stubs see the same linearization of the global history. However, each stub is guaranteed to see a causally-consistent linearization—but only if all interprocess communication is also via the ISIS causality-preserving protocols. This is a very severe restriction. An existing distributed application is quite likely to be built upon an interprocess communication mechanism other than ISIS. If such an application is instrumented using Meta, the messages received by the Meta stubs may not completely reflect the actual causal dependencies. Figure 8.2 illustrates this problem. The control component at q sends a set of sensor values to p, which is represented in the figure by a solid line from q to p. Process q then performs some additional communication and sends a message to r using the TCP protocol, which is denoted by the dashed line from q to r. The receipt of the message from q causes r to change state, resulting in new sensor values being sent to p. Since the communication between q and r was outside of ISIS, ISIS is unaware of the casual dependency and could deliver, as shown, r's later sensor values before q's earlier sensor values.

Figure 8.2: A breakdown of causal dependencies
8.2.4 Implementing Aggregates

A stub servicing an aggregate context $C_A$ joins a special group named $\text{MetaAggs}/C_A$. To become a member of the aggregate, a stub becomes an ISIS client of the group $\text{MetaAggs}/C_A$; to leave the aggregate a stub simply leaves the group.

If all (non-client) members of an ISIS group fail, then the group ceases to exist, with all former clients of the group losing their client status. Consequently this implementation of aggregates is dependent upon the persistency of the aggregate group. Should the group die, the aggregate ceases to exist\(^2\). The probability of the group dying can be diminished by endowing multiple stubs with the aggregate definition.

8.2.5 Replicating Stubs

Fault-tolerance is achieved in Meta by the use of active replication\(^3\), optimized by performing some computations using the coordinator-cohort methodology. The reader is referred back to Chapter 6 for a general discussion of fault-tolerance in a distributed system.

As noted in Chapter 7, Meta only supports the replication of aggregate contexts. Active replication requires each replica to behave as a state machine, progressing through the same sequence of states. Base contexts—contexts defined by instru-

\(^2\)Earlier implementations of Meta got around this problem by having members of an aggregate $A$ outright join the group $\text{MetaAggs}/C_A$ instead of just becoming clients of this group. This approach preserved the membership of the aggregate even if the stub implementing the aggregate failed. However, limitations with ISIS prevented this approach from being used when support for replicated stubs was added. The exact problem is explained in Section 8.2.5.

\(^3\)The current implementation of Meta, Meta 2.1, does not allow new replicas to be added into an active set of replicas: all replicas must be started before any replica receives any external communication. Adding new replicas requires a way to atomically transfer the state from the existing replicas to the new replicas. The state of each replica is rather complex, and the necessary state transfer routines have not yet been written.
mented programs—experience state transitions defined by local events. To replicate a base context, Meta would have to ensure that local events are ordered consistently at all replicas with respect to nonlocal events. This effect can be achieved in Meta by the use of aggregation. To make some base context $C$ fault-tolerant, the context $C$ is transformed to a context class $C'$, with each copy of the process that previously implemented the original context $C$ now being a member of the context class $C'$. An aggregate is formed over $C'$, and all replicas are programmed to join that aggregate. The sensors and actuators of the original context $C$ are now available as fault-tolerant objects in the aggregate context.

If a replicated stub ran as a coroutine to an instrumented application, then there would be local events generated by the application which would have to be consistently ordered at all replicas with respect to external events. Consequently, replicated contexts are typically implemented by stubs attached to programs whose only purpose is to host the stub; the Meta server program of Figure 7.2 provides an example of such a program.

However, there are still some local events whose order relative to non-local events must be agreed upon by all the replicas. For example, Section 7.4 discusses the built-in TIMER function, which becomes true a specified number of milliseconds after the alternative command containing the function was received for execution. Each replica executing an alternative command containing a TIMER reference sets a timer. Since the replicas do not execute in a completely synchronous environment, the timers may trigger at different points in the execution history at each replica. In particular, one replica may have received more external events than another. To ensure that each replica orders the triggering of the timer in the same way with respect to other events, each replica uses the ABCAST protocol to broadcast to the group of replicas the event of the timer going off. Only when the broadcast is received, does each replica, including the one sending the broadcast, consider the timer to have actually
gone off. (This causes a delay in the firing of timer, a delay which is unavoidable in an asynchronous, non-realtime environment.)

Since replicated stubs run as independent processes, they have an empty base context—one without sensors or actuators. Each program implementing a replicated service $S$ calls `meta_stub_init` with $S$ as the context and so each replica joins the group `Meta/S`. The service is named by its base context name $S$. If the `meta_endow_aggregate` command is used to assign to the replicated server $S$ the task of implementing aggregate context $C_A$, then these stubs all join the group `Meta/C_A`. As with any communication connection in Meta, stubs wishing to communicate with this context become clients of the group `Meta/C_A` and send their messages using the ABCAST protocol. This ensures that all replicas see the same sequence of messages, necessary to ensure that each replica behaves as a state machine.

When a replicated set of stubs is endowed with an aggregate definition, the act of endowing the stubs should be seen as a single, atomic event by all replicas. To implement aggregate context $C_A$, the replicas must all join `Meta/C_A` and `MetaAggs/C_A`. If each join is not atomic, then one stub may receive messages that another does not receive, which consequently would have to be forwarded on to the other members. To avoid this problem, the ISIS `pg_subgroup` routine is used. This routine takes as a parameter a subset of processes belonging to some group and causes the subset to atomically join a new, independent group. However, the subgroup must not have already been in existence—which is why the members of an aggregate do not outright join the group `MetaAggs/C_A`. To use the `pg_subgroup` mechanism, the oldest replica calls `pg_subgroup` as part of a coordinator-cohort computation.

The coordinator-cohort mechanism is also used whenever the replicas must all agree on some value. One example of where this is necessary is the `pg_lookup` performed to get the address of a remote context. Suppose two replicas independently call `pg_lookup` to obtain the address for context $C$. Let $C$ initially have address $G_1$, 

but it then crashes and recovers having address $G_2$. One replica might be running
to be faster than the other, and get first the address $G_1$ for $C$ before witnessing $C$’s crash.
The slower replica might only see the address $G_2$. In order for these two replicas
to behave as state machines, they each need to see the same sequence of events.
Meta accomplishes this by having one replica, the oldest, call `pg_lookup` and then
broadcast to the other members of the group the resulting address. This solution
makes the address resolution deterministic. Non-deterministic events can generally
be handled in the state machine approach through the use of coordinator-cohort
computations.

The coordinator-cohort mechanism is also used to carry out remote actuations.
Only the coordinator sends an actuation message to the group corresponding to the
remote context. If this context is replicated, then all replicas perform the action,
and by virtue of being state machines, come up with the same termination status.
Each replica replies to the original coordinator, who waits only for the first reply.
Note that if a coordinator should fail while waiting for a reply, the new coordinator
might end up repeating an action. This is handled by the recipients checking for
redundant actions through the use of message sequence numbers. In the case where
the action has already been performed, the previous termination status is returned.
The state machine approach generally requires voting on the outputs of each replica.
However, the coordinator-cohort approach provides a simpler solution when the only
admissible failures are crash failures, and the external action taken is idempotent.
Remote actuations are made to be idempotent in Meta through the use of duplicate
suppression at the remote end.

Meta always uses the oldest group member of a replicated set of stubs as the
coordinator. For each process group that a process belongs to, ISIS maintains a
view of the current membership of the group. Each process in a group progresses
through the same sequence of group views, observing group view changes as virtually
synchronous events. In theory, after each event, each member of the group will have the same view as to who is the oldest process in the group, and so all processes can agree implicitly on who the coordinator should be. However, Meta generally does not process a new event presented to it by ISIS until it has finished processing the previous event. This property is necessary to ensure that each stub sees a serial sequence of state transitions. As a consequence of this serializing of events, two different replicas might process the same event while having different views of the group membership, since group views naturally are updated by ISIS as soon as they are received. In particular, a tardy processor $P_i$ might believe that some process has died which a faster processor saw as still being alive. On the surface, it might seem that this would cause problems in the selection of the oldest processor to perform a coordinator-cohort computation. However, such is not the case. The potential problem is when the oldest process $P_0$ fails. If $P_0$ or any subsequent successor older than the tardy processor $P_i$ had completed the coordinator-cohort computation, then $P_i$ will have received word of that before receiving the failure event, and will not even attempt to pick a coordinator. Alternatively, if the computation has not yet been finished, $P_i$ will see itself as being the current oldest only if it indeed is. We never have the case where two processes concurrently operate as coordinators.

8.3 Performance of the Meta Toolkit

We listed in Chapter 1 some goals a reactive monitoring and control system should satisfy; minimal system interference and quick reaction were two of the goals mentioned. These goals are related. A system for monitoring and controlling should be transparent to the users of the system—and as such, its operation should not use a noticeable amount of computational or I/O resources. Moreover, not only should reaction take a minimal amount of system resources, the controller should be able to react to events in a minimal amount of time after they occur. The performance
figures in presented in this section quantify the current status of Meta in meeting these objectives.

8.3.1 Experimental Setup

The performance figures presented in this chapter were obtained by running a suite of test programs on Sun 4/60's running Version 4.1.1 of the Sun operating system. The Sun computers were connected via a 10 Mbps Ethernet. The test programs were built using version 3.0.4 of ISIS and version 2.1 of Meta, using the versions as present on November 18, 1991; the Meta and ISIS libraries along with the test programs were all compiled using the Sun C compiler at optimization level two. Although the test machines were part of the general computing resources at Cornell, the tests were run at times when the machines were otherwise idle, and the overall network load was low. Furthermore, each process was run on a separate machine.

8.3.2 Cost of Subscriptions

The primary cost in Meta is the cost of remote references, and a fundamental remote reference is the subscription mechanism. Table 8.1 gives the time required to transmit a subscription value, as a function of the number of concurrent subscribers. This time was measured by writing a simple sensor function that returned the value of an integer-typed variable. The sensor was embedded in a program that continually alternated between incrementing the variable and calling meta_notify. The table summarizes the inter-arrival times for two hundred sensor subscriptions sent one after the other.

In the current implementation of Meta and ISIS, the stub sending out the sensor value blocks until it receives an acknowledgement from the ISIS stub at the receiver. As a consequence, sensor values are sent synchronously. However, the intended design of Meta is that the sender not block after sending out a sensor value; this behavior
Table 8.1: Time to receive subscriptions

<table>
<thead>
<tr>
<th>Number of Concurrent Subscribers</th>
<th>Time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>8.7</td>
</tr>
<tr>
<td>2</td>
<td>14.7</td>
</tr>
<tr>
<td>3</td>
<td>16.4</td>
</tr>
<tr>
<td>4</td>
<td>14.9</td>
</tr>
<tr>
<td>5</td>
<td>23.5</td>
</tr>
</tbody>
</table>

will be realized in a planned new implementation of the ISIS system. To determine how this change will effect performance, the Meta code was modified slightly in a way that resulted in ISIS sending out the sensor values asynchronously. (Unfortunately, this modification is not general; it prevents Meta from supporting replicated stubs.) The time required to transmit a subscription value was remeasured using the modified code; the performance figures for the modified, asynchronous version are shown in Table 8.2. The significant increase in performance is not surprising. In the asynchronous benchmark, sensor values were sent out one after the other, as fast as possible. Rather than sending each sensor value in a separate message, the ISIS system bundles up groups of adjacent messages and transmits them as a single packet. Given the low median delay between receiving sensor values, 0.6 milliseconds, we can conclude that most sensor values were received as part of a larger bundle of sensor values. The 0.6 millisecond number represents the time it takes the Meta and ISIS stubs to extract adjacent messages from the same network packet.
Table 8.2: Time to receive subscriptions sent asynchronously

<table>
<thead>
<tr>
<th>Number of Concurrent Subscribers</th>
<th>Time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

8.3.3 Cost of Executing Guards

The cost of executing a guarded command was determined by executing the simple guarded command

\[
(S \rightarrow \text{ToggleS} \parallel \neg S \rightarrow \text{ToggleS})
\]  

(8.1)

where S is a boolean-valued variable and \text{ToggleS} is an actuator that toggles the value of S. The actual code for the sensor and the actuator is shown in Figure 8.3.

```c
int s_sensor(value)
long *value;
{
    *value = s_value;
    return 0;
}
```

```c
int ToggleS(ap)
ACT_NODE *ap;
{
    s_value = !s_value;
    meta_notify(1,&s_handle);
    return META_SUCCESS;
}
```

Figure 8.3: The S sensor and ToggleS actuator

The cost of executing an individual guard was determined by measuring the time between the first call to the \text{ToggleS} actuator and the one hundred thousandth call,
and dividing by 100,000. This test was repeated ten times. The average time per call was 127.1 microseconds. The Sun 4/60 is approximately a 10 MIPS machine, and so this corresponds to approximately thirteen hundred instructions per guard execution.

Profiling the code revealed that each call to \texttt{meta\_notify} resulted in a call to the \texttt{malloc} library routine. Recall that the Meta library gains control when the application program calls \texttt{meta\_notify}. Consequently, when Meta invokes the \texttt{ToggleS} actuator, the call by the actuator to \texttt{meta\_notify} is a recursive invocation. The recursion is handled in Meta by using \texttt{malloc} to save the parameters for subsequent processing by the high-level invocation of \texttt{meta\_notify}. Since \texttt{malloc} is known to be expensive, the \texttt{ToggleS} actuator was rewritten to pass along an empty set of sensor handles, eliminating the need to call \texttt{malloc}. In the version of the program that gave the 127.1 microsecond performance number, the S sensor was sampled only when the stub was explicitly directed to by a \texttt{meta\_notify} call. Since in the revised version of the program the \texttt{ToggleS} actuator called \texttt{meta\_notify} without passing any sensor handles, the S sensor was redefined to be sampled each time \texttt{meta\_notify} was called. The cost of executing an individual guard was measured for the revised program as being 78.9 microseconds, considerably less than the value for the version of the test program with the recursive \texttt{malloc}.

Some of the cost in executing Guard 8.1 is in updating the sensor cache and in reevaluating the guard predicate. To factor out this cost, the S sensor was removed from the program and the following guarded command was run:

$$\langle \text{true } \rightarrow \text{ToggleS} \rangle \quad (8.2)$$

The time to execute this guard was measured and found to be 36.1 microseconds. Comparing this number with the time required to execute Guard 8.1, we can conclude that around 40 microseconds was required to update the sensor cache and reevaluate the guard predicates.
Table 8.3: Time to execute a guard locally

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Average value (µsecs)</th>
<th>Std. Dev. (µsecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>With malloc</td>
<td>127.1</td>
<td>1.6</td>
</tr>
<tr>
<td>Without malloc</td>
<td>78.9</td>
<td>1.4</td>
</tr>
<tr>
<td>With vacuously true guard</td>
<td>36.1</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 8.3.3 summarizes the local execution benchmarks. These performance figures are specific to the particular test programs used. Since the code for the S sensor and the ToggleS actuator is so trivial, these figures can be viewed as representing the cost of a call to meta_notify under different conditions. In general, the running time of the meta_notify routine depends on many factors, including whether or not any sensor handles were passed as parameters, how many sensors need to be polled, and how many guards need to be reevaluated. If the meta_notify routine also needs to process external events, then that too will add to its running time.

### 8.3.4 Executing a Guard Remotely

The cost of remote communication dwarfs the cost of local computation when guards containing remote references are executed. To measure the cost of remote references, the previous alternative command (8.2) was executed again, but this time by a remote Meta server stub. Table 8.3.4 summarizes the observed execution times. Both the best observed execution time and the median execution time for 200 trials are reported, where each time value is the elapsed time between two successive invocations of the actuator. Most samples were close to the lowest observed value, but a scattering of higher values were also observed. One would expect values to be near the lowest value, but (apparently) random delays were sometimes introduced. However, particularly as the number of processes involved increased, the observed...
Table 8.4: Time to execute a guard with remote references

<table>
<thead>
<tr>
<th>Number of Replicas</th>
<th>Time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>23.4</td>
</tr>
<tr>
<td>2</td>
<td>31.9</td>
</tr>
<tr>
<td>3</td>
<td>38.8</td>
</tr>
<tr>
<td>4</td>
<td>41.5</td>
</tr>
<tr>
<td>5</td>
<td>44.1</td>
</tr>
</tbody>
</table>

distribution tended to have a more L-shaped pattern, descending quickly but with a long tail. The spread of the data points is quantified in the table by the columns giving the value of the 90th percentile and the standard deviation.

These guard execution times count the cost from the time a new sensor value is sent out to the time the actuation is performed. In total, three messages are sent by Meta: the message from the remote stub to the server containing the new sensor value, the message from the server to the remote stub requesting that the actuation be performed, and finally the remote stub’s response. The Meta messages sent are all typically small, containing less than a hundred data bytes for the actuation reply, approximately 350 data bytes for the actuation invocation, and approximately 500 data bytes for the sensor value. Much of this data is information passed along by the ISIS system.

We note that running with multiple replicas present significantly increases the execution time. If two or more replicas are present, then the actuation is carried out as a coordinator-cohort computation, necessitating an additional message being sent from the coordinator to the cohort(s). Additionally, each new sensor value must be sent to each replica; these messages are sent sequentially since the communications
system does not provide a hardware multicast. Increasing the number of replicas slows down the expected execution time proportionally, but also results in considerably more variance. This additional variance most likely stems from complexities within ISIS.

Figure 8.4 displays the median execution time versus the number of replicas.

![Graph](image)

Figure 8.4: Median execution time to execute a guard with remote references

### 8.3.5 Executing a Single Remote Action

If the action of a guard references only local actuators, then the action can be performed quite cheaply without any need for interprocess communication. Alterna-
Table 8.5: Time to execute a remote action

<table>
<thead>
<tr>
<th>Number of Replicas</th>
<th>Coordinator-Cohort (time in milliseconds)</th>
<th>Active Replication (time in milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.4</td>
<td>16.5</td>
</tr>
<tr>
<td>2</td>
<td>17.5</td>
<td>18.6</td>
</tr>
<tr>
<td>3</td>
<td>26.6</td>
<td>28.8</td>
</tr>
<tr>
<td>4</td>
<td>28.5</td>
<td>31.1</td>
</tr>
<tr>
<td>5</td>
<td>30.5</td>
<td>33.3</td>
</tr>
</tbody>
</table>


tively, if the action references multiple stubs, then the action must be performed as a transaction. However, the case where the action invokes only a single remote actuator may be handled more simply. Meta carries out such actions by the use of a synchronous ABCAST to the remote context; the reply is the return status of the actuator. Table 8.5 summarizes the statistics observed when executing Guard 8.2 remotely. These numbers were obtained by invoking⁴ the remote actuator 201 times, and recording the time between invocations, for a total of 200 data points.

Meta normally uses the coordinator-cohort method when executing the action of a guard that invokes a single actuator in a remote context. To carry out an action using this approach, the coordinator for the replicated server sends a synchronous

⁴Like many of the guards used for these benchmarks, the guard predicate of Guard 8.2 is continuously true. Since the action of the guard is on a remote context, executing the guard does not directly trigger a state transition. Consequently, the stub could loop infinitely often, executing this guard, without encountering a new state transition. The only way for such an infinite loop to be broken is by the receipt of some external event that negates some guard. Under normal execution, the Meta stub will periodically block and wait for such an external event whenever the interpreter finds itself looping through a set of guards without generating a state transition. This pausing for an external event was disabled for the sake of obtaining these benchmarks.
ABCAST to the group representing the context that is to the perform the action; the reply message is the return status of the action. The coordinator then sends an asynchronous CBCAST message to the cohorts informing them that the action has been carried out. This termination message can be delivered lazily, in parallel with carrying out the next invocation. The performance of this method is summarized in Table 8.5 under the columns labelled "Coordinator-cohort."

![Graph](image)

**Figure 8.5:** Median time to execute a single remote action

For the sake of comparison, the Meta library was modified slightly to use active replication, where all the replicas independently perform the action. The remote actuation test was then repeated with the modified version of the library, with
the observed execution times summarized in Table 8.5 under the columns labelled "Active replication." The median costs for the two methods are shown plotted in Figure 8.5. The active replication method does not scale as well as the coordinator-cohort method for this benchmark.

The active replication scheme is computationally more expensive than the cohort mechanism, requiring more messages to be transmitted. Due to the complexity of the ISIS protocols, it is difficult to directly compute the actual number of network packets sent by the different approaches. However, the Meta library, using the coordinator-cohort approach, sends $r+1$ messages for each remote actuation; the active replication approach uses $2r$ messages. (Support for hardware multicast—currently lacking in ISIS—would allow some of the messages in both approaches to be transmitted in parallel.)

Note that with both the coordinator-cohort method and with active replication, a stub may receive duplicate requests to perform the same action. Meta detects duplicate requests by attaching to each message a sequence number. With the coordinator-cohort method, a stub can only receive a duplicate actuation from some remote context for the last actuation the stub carried out on behalf of that remote context. Such a duplicate arises if the coordinator carrying out the action failed before informing its cohorts that it had carried out the action. To provide an at-most-once semantics, each Meta stub remembers the status of the last actuation it performed on behalf of each remote context. When a duplicate request is received by a stub, the stub does not repeat the action but merely sends back the last status value.

However, with the active replication scheme, a stub may receive a duplicate request for any action already performed. To handle such duplicates correctly requires each stub to remember the status of all actuations executed on behalf of remote contexts. Since the history of past actuations is potentially unbounded in length,
stubs would in practice have to periodically perform some sort of garbage collection. For the sake of gathering performance figures, this problem of remembering past actuations was not addressed, but the additional bookkeeping required to correctly implement active replication is not expected to significantly add to the execution time. The performance figures for the active replication case were obtained by repeating the remote actuation loop until the stub carrying out the actions received a duplicate older than the last action performed. The replicas were able to do at least 48 back-to-back actuations before becoming this far out of sync.

8.3.6 Transaction Processing

To carry out a transaction, the stub executing the guard must first acquire locks on all of the participants in the transaction. Then a message containing all the actions to be performed is sent to the first participant. This message is forwarded from participant to participant until all the subactions have been performed, at which point the initiator is informed. The initiator then releases all the locks. To measure the cost of performing a transaction, the following guard was executed (the ALIVE sensor returns true if the corresponding context is available):

\[(C_1.ALIVE \ldots C_n.ALIVE \rightarrow C_1.ToggleS \ldots C_n.ToggleS)\]

Table 8.6 summarizes the observed execution times for \(n = 2\) and \(n = 3\). These statistics are computed over a set of 200 trials, with the length of each trial being the time between two successive invocations of the \(n\)th actuator in the transaction.

To carry out a transaction when operating in the replicated mode, a single coordinator for the transaction acquires the locks. However, to facilitate failure handling, all the replicas are involved in carrying out the transaction once the necessary locks have been obtained. In particular, all replicas send a message to the first participant; duplicates are filtered out by the receiver. All replicas receive word from the last
Table 8.6: Time to execute a transaction

<table>
<thead>
<tr>
<th>Number of Participants</th>
<th>Time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>2, 1 replica</td>
<td>84.8</td>
</tr>
<tr>
<td>2, 2 replicas</td>
<td>102.4</td>
</tr>
<tr>
<td>2, 3 replicas</td>
<td>143.8</td>
</tr>
<tr>
<td>2, 4 replicas</td>
<td>162.1</td>
</tr>
<tr>
<td>3, 1 replica</td>
<td>124.8</td>
</tr>
<tr>
<td>3, 2 replicas</td>
<td>160.3</td>
</tr>
<tr>
<td>3, 3 replicas</td>
<td>205.8</td>
</tr>
<tr>
<td>3, 4 replicas</td>
<td>222.9</td>
</tr>
</tbody>
</table>

participant when the last action in the transaction has been finished and all replicas then send out the same termination message. This hybrid method of active replication with the coordinator-cohort approach simplifies failure handling; if the entire transaction were performed using the coordinator-cohort approach, then failure of the coordinator would result in the transaction being left in any of a large number of possible states.

8.3.7 Aggregate Actuations

To determine the cost of performing aggregate actuations, the following guarded command was run in the aggregate context formed over the context containing the ToggleS actuator.

\[ (true \rightarrow \text{ToggleS}) \]

The actuator was invoked on all active members of the aggregate. Table 8.7 summarizes the cost of performing the actuation as a function of the number of
Table 8.7: Time to execute an aggregate actuator

<table>
<thead>
<tr>
<th>Number of Aggregate Members</th>
<th>One Replica (time in milliseconds)</th>
<th>Two Replicas (time in milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.2</td>
<td>10.6</td>
</tr>
<tr>
<td>2</td>
<td>10.6</td>
<td>12.8</td>
</tr>
<tr>
<td>3</td>
<td>12.3</td>
<td>15.0</td>
</tr>
<tr>
<td>4</td>
<td>10.9</td>
<td>17.0</td>
</tr>
<tr>
<td>5</td>
<td>11.2</td>
<td>19.6</td>
</tr>
<tr>
<td>6</td>
<td>11.6</td>
<td>21.6</td>
</tr>
<tr>
<td>7</td>
<td>11.9</td>
<td>25.3</td>
</tr>
</tbody>
</table>

aggregate members; execution times are shown both for both an unreplicated server and for a server consisting of two replicas. Two hundred measurements were obtained for each case.

The median costs are plotted in Figure 8.6. Note that cost of performing an aggregate actuation is linear in the number of aggregate members, which is as expected since ISIS transmits the message to each member separately. Hardware multicast support would allow the action to be transmitted to multiple members in a single message, which should result in a considerable improvement in performance.

8.4 Summary

This chapter presents the implementation of the Meta toolkit. The implementation makes extensive use of the ISIS toolkit, so key aspects of ISIS are presented before discussing their use in the Meta implementation. ISIS groups are used as the naming structure for Meta contexts. Communication between Meta stubs is typically of the
Figure 8.6: Median aggregate execution time

form client to group. This mode enables communication to be routed via the "fast" bypass mode of ISIS.

Each state transition at a stub potentially results in subscription messages being sent to interested subscribers. Ideally these values should be sent out using a single, atomic broadcast, but because of a limitation in the ISIS system, subscriptions are sent to each subscriber individually. Consequently, Meta does not guarantee that each stub sees the same linearization of the global history; however, each stub will observe a causally consistent linearization.

Replicated server stubs operate fundamentally as state machines, although the the coordinator-cohort mechanism is employed to reduce the amount of overhead and to facilitate synchronization.

Finally, performance figures for certain operations in Meta are presented. Remote
references are roughly three orders of magnitude more expensive than local references. Replicated servers perform somewhat slower than nonreplicated stubs. Running additional replicas causes the performance to decrease in a proportional manner. The Meta transaction facility is somewhat expensive, particularly when running in the replicated mode. Aggregate actuations, however, are not terribly expensive, even when run in the replicated mode. Optimal performance in Meta is achieved by diminishing the number of remote references. Remote actuations should involve only a single participant as much as possible.
Chapter 9

Conclusion

This thesis has presented a methodology for controlling distributed applications. Distributed computing has become increasingly popular as local area networks have become more widespread. The resource servers and daemons present on a network form one type of distributed application. Alternatively, many high-level applications are being now constructed by integrating special-purpose, independent server programs. Application management enables anomalies in system operation to be detected and corrected; a well-managed system is capable of operating more efficiently and providing better service to users than a system lacking provision for high-level control. Distributed applications, however, by virtue of being distributed are particularly difficult to manage.

The methodology developed in this thesis controls distributed applications by applying a reactive system structure. A reactive system consists of two layers: a control layer which continuously senses and acts upon changes in an environment layer. This architecture separates the problems of application instrumentation from control. Any application may be instrumented by a uniform method of embedding sensors and actuators. The instrumented application components become the environment upon which the control layer operates. The control layer executes rules
written on the state of the application, reading the application state through the sensors and changing the application state through actuators.

The Meta toolkit—also presented in this thesis—contains the tools needed to manage distributed applications according to the reactive system methodology. Meta provides the necessary infrastructure to instrument the programs making up a distributed application with sensors and actuators. Once instrumented, the behavior of these programs may be controlled through rules, which are expressed as guarded commands that reference the sensors and actuators. Both instrumentation and control are carried out through the use of stubs that are embedded in the programs making up the distributed application. These stubs support the distributed execution of guarded commands, with each stub capable of executing rules operating on the global state of the system. This method of distributed control provides for increased responsiveness by exploiting locality; it also provides for increased robustness by not pinning the entire control system at any single point.

Chapter 1 of this thesis enumerated a number of goals for distributed application management. One goal was that the layer of control, though distributed, should behave in a logically-centralized manner. Without this property, different control components could take inconsistent and even conflicting actions upon the system state. The Meta toolkit ensures that all distributed control components see the same partial order of events, with ordering based upon the “happens-before” causality relationship. This property is preserved to the extent that interprocess communication is routed via a causality-preserving protocol. Since Meta is built upon the ISIS toolkit, which provides this property, all communication generated by Meta automatically satisfies this property; other application communication will also satisfy this property if routed via ISIS. Limitations within the ISIS toolkit made it infeasible to ensure that all control components see the same total ordering of system events and states. This deficiency is not expected to be problematic in practice, since
different control components often operate upon disjoint parts of the system state.

Another important goal is that any system for management should impose minimal overhead on the application being monitored. At the same time, the manager should react to significant events within the application in a timely manner. Calls made by the application into the Meta library to report state changes are quite cheap, as evidenced by the performance figures. The performance numbers also demonstrate that a guard dependent only upon local aspects of the system state is executed very quickly. However, as would be expected, remote references require considerably more time to handle, diminishing responsiveness.

Since control within Meta is distributed amongst different stubs, care must be exercised to ensure that these stubs operate in a consistent fashion. Consistency is achieved by introducing delays, with higher levels of consistency requiring higher amounts of blocking. The degree of consistency provided must therefore be traded off with the amount of application blocking that is considered acceptable. The choice made within the Meta toolkit is to only block application components to the extent needed to ensure that each control component observes a causally-consistent view of the system state. Meta does not ensure that a detected condition holds at the time of reaction; to do so would require a large degree of blocking and greatly diminish the application’s performance. Nor is Meta able to bound reaction time, operating as it does in a non-realtime environment. The speed of interprocess communication within Meta is largely determined by the performance of the ISIS system. Despite these limitations, the reaction time offered by Meta will be quite acceptable for many conditions of interest in application management.

Robustness of operation is another important property required of a distributed control system. Meta provides fault-tolerant execution of guarded commands via the use of special replicated servers. Additionally, fault-tolerant sensors and actuators may be constructed through use of the aggregation mechanism.
A control system should also be able to scale well with the size of the distributed application being managed. The Meta architecture, with its support for local execution as much as possible, should scale quite well. However, in the current implementation, performance degrades at least linearly in the number of active processes, with the variance becoming quite large as the number of processes is increased. Another goal stated in Chapter 1 is the dynamic specification of rules. Meta is capable of dynamically adding and deleting rules, though currently does not give the user the ability to cancel an active rule; to provide this property is a trivial extension.

9.1 Future Directions

Determining the utility of this toolkit requires additional experience in its use. One research lab is already involved in the development of a resource manager built using Meta\textsuperscript{1}. This group has devoted considerable energy into developing a graphical interface to the Meta system. This type of work is needed in order to make the Meta system more useful to people concerned with application management. Along those same lines, work is under way here at Cornell to develop a high-level language for specifying application behavior [MCWB91]. Rules expressed in this language would be translated into Meta guarded commands.

In addition to building tools and applications on top of Meta, much remains to be done within the toolkit itself. We identify in this section a number of areas requiring additional research.

The Meta toolkit constructs causally-consistent global states against which conditions are evaluated. However, Meta currently is unable to ensure that all control components see causally-consistent states if the components making up an application communicate via a protocol that does not preserve causality. This particular

\textsuperscript{1}The General Electric Corporation's Advanced Technology Lab, Moores-town, N.J.
deficiency could be corrected if the underlying operating system's interprocess communication routines preserved and made available causality relationships. However, constructing merely causally-consistent global states is adequate only detecting for conditions that are (locally) stable, i.e., dependent monotonic. This thesis includes a protocol for constructing valid global states, states which actually occurred in real-time. However, achieving the rigorous semantics offered by this protocol is only accomplished by forcing the different components of the distributed application to behave synchronously. The work of [MN91] examines different semantics for detecting when a nonstable global state predicate is satisfied by the state of a loosely synchronized system. One direction for extending Meta would be to implement these protocols as part of the toolkit.

The Meta toolkit's instrumentation routines are too cumbersome to support fine-granularity monitoring, such as monitoring rapidly changing values in an operating system kernel. New techniques are required to minimize the overhead imposed upon the application by the instrumentation. In a related vein, the Meta approach of manual instrumentation is unsuitable for monitoring a large number of variables within an application. A satisfactory solution to this problem requires automatic generation of instrumentation code.

The current implementation of Meta does not address the problem of recovery from actuator failures. More work is required to determine a suitable framework for the specification and application of recovery methods.

Another promising use of Meta would be industrial process control. That use requires a version of Meta built on top of a real-time operating system. However, a real-time version of Meta would also require many changes to the system itself. In particular, the language for guarded commands would need to be extended to allow timing constraints to be expressed. Moreover, the Meta stub would need a real-time scheduler for executing guarded commands.
Appendix A

The Meta Functional Description

Meta is a toolkit for controlling distributed applications. This fault-tolerant service is directly accessible from all sites in a local-area network via a procedural interface. The toolkit is accessed by linking in the metalib.a library with the client programs.

The distributed application being controlled by Meta is seen through two major abstractions: sensors and actuators. A sensor represents a dynamically changing value, such as the load on a machine or the number of free buffers. An actuator provides a way to change values and/or perform some computation. Examples of actuators include an actuator that executes another program and an actuator that allocates more buffers.

Programs being managed by Meta must first be instrumented by augmenting the code with sensor and actuator functions and then linking in the Meta stub, also found in the metalib.a library. (A separate utility, called spy permits global program variables to be both read and written to without linking in the Meta library.) A given program containing a set of sensors and actuators forms a context. All programs implementing the same set of sensor and actuator objects form a context class. A subset of a context class may be grouped into an aggregate, which is in itself a context. The context class over which the aggregate is formed is called the base context class. An aggregate context automatically inherits the objects of the base context class; each such inherited object becomes a composite-valued type. A composite-value contains the value from each member of the aggregate tagged with that member’s name.

Once a distributed application has been instrumented, it may be controlled via the use of guarded commands. These commands are executed in a distributed fashion by the Meta stubs. There are two parts to the Meta interface: a front-end interface concerned primarily with controlling the application and a back-end interface used for instrumenting programs. The back-end is described first, starting with its type system.
A.1 Types

To instrument a distributed application, the programmer must manually add sensor and actuator functions to the programs constituting the application. Meta interacts with these functions according to a standard typed interface. The supported base types are denoted by the integer constants TYPE_INT, TYPE_REAL, and TYPE_STRING as defined in the file types.h. These are the C language long, double and char * types, respectively. Other types may be constructed from the base types using primitive type constructors. Meta supports a set constructor operating on any of the base types, an interval constructor operating on the two numeric base types, and a composite type constructor used in connection with aggregation.

The numeric constant for set types is the base type constant ORed with the constant TYPE_SET. For intervals the numeric type constant is the base constant ORed with the constant TYPE_INTERVAL. For composites the type constant is the original type (which can be a set or interval type) ORed with the constant TYPE_COMPOSITE. Given a type, the base type may be found by ANDing the type with the constant TYPE_BASE.

The include file types.h defines the type PTR to be a generic pointer to a Meta value.

Special functions are provided to manipulate sets:

\begin{verbatim}
SET set_create()
int set_delete(SET set)
int set_pack(int type, SET set, PTR value)
int set_unpack(int type, SET set, PTR *value)
int set_rewind(SET set)
int cardinality(SET set, int type);
\end{verbatim}

The type SET is defined in the types.h file. The routine set_create creates a set, returning a pointer to a dynamically allocated data structure which contains the set information. Elements are added to the set by calling the routine set_pack. The elements are extracted by repetitively calling the set_unpack function. Each time set_unpack is called, it returns the next value from where it left off the previous time; set_rewind may be called so that the next set_unpack call will return the first value of the set. A set may be deleted by calling set_delete.

The number of elements in a set can be determined by calling the cardinality routine. For example, one of the sensors in the Meta machine utility, the login sensor, returns the set of users logged into a system. The following code fragment could be used to print out the names.

\begin{verbatim}
/* The variable SET value contains the set of names */

for ( num_elmts = cardinality(value,TYPE_STRING | TYPE_SET));
\end{verbatim}
num_elmts;
num_elmts--)
{
    PTR name = NULL;
    set_unpack(TYPE_STRING, value, SM_VALFLD, &name);
    printf("%s
", name);
}
set rewind(value);

Alternatively, one could loop until the set_unpack returns a value less than one. When all the values in a set have been enumerated, the routine set rew ind() can be called to restart the enumeration.

The type modifier TYPE_COMPOSITE is used to denote the types of aggregate values. A value of this type consists of a set of pairs, where each pair is a value and a string denoting the source of that value. The following set of macros may be used to extract values of composites:

    int NUM_COMP_VALS(PTR val)
PTR COMP_VAL(PTR cp, int i)
char *COMP_SRC(PTR cp, int i)

The first function returns the number of values in the composite. The second function returns the ith value and the third function returns the name of the context providing the ith value.

A client can convert between the machine representation of a value and the string representation of a value using the following two procedures:

    char *type_to_ascii(type, PTR value)
void ascii_to_type(type, char *ascii, char **pos, PTR *value)

The first procedure converts the value from its machine representation to a string. The string is stored in a statically-allocated buffer which is rewritten by subsequent invocations of type_to_ascii. The second routine, ascii_to_type, takes a string representation of a value and converts it to its machine representation. This routine takes four parameters. The first is the type to be read from the string, which is the second parameter. The third parameter is a pointer to a string pointer; this pointer is set to point to the next character in the string after the substring that was converted. A string may contain multiple values, where each value is separated by white space; the value returned through the third parameter is useful for iterative processing. However, a null pointer can be passed if the return position is not needed. The last parameter is a pointer to a value pointer; this is set to point to the static buffer containing the result.

The following procedure will print the string representation of the pointed to value on the specified stream; it calls type_to_ascii.
type_fprint(FILE *fh, int type, PTR val)

The next function returns the number of bytes of storage required to store an object of the given type. Note that sets are stored in special dynamically sized objects and so this routine returns only the size of a pointer for any of the set types; the same comment applies to strings.

int type_size(i)

Finally, each type has a textual name; routines are provided to convert between the numeric representation of a type and the textual representation. Set type names are formed by enclosing the base type name in curly braces and interval type names are formed by enclosing the base type name in square braces. The following table summarizes the mapping from type names to constants and includes the C-language native type corresponding to each Meta type.

<table>
<thead>
<tr>
<th>C Base Type</th>
<th>Constant</th>
<th>Name</th>
<th>Interval Name</th>
<th>Set Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>TYPE_INT</td>
<td>integer</td>
<td>[integer]</td>
<td>{integer}</td>
</tr>
<tr>
<td>double</td>
<td>TYPE_REAL</td>
<td>real</td>
<td>[real]</td>
<td>{real}</td>
</tr>
<tr>
<td>string</td>
<td>TYPE_STRING</td>
<td>string</td>
<td>N/A</td>
<td>{string}</td>
</tr>
</tbody>
</table>

The following routines convert between the string name of a type and its numeric identifier.

int name_to_type(char * name)
char * type_to_name(int type)

A.2 Instrumenting a Program

A set of procedures are available in the metalib.a library to be used when instrumenting a program with sensors and actuators. These functions all return a status code indicating their success or failure; a value of zero means the routine succeeded. Error codes may be found in meta_err.h.

Recall that a program instrumented with Meta defines a context. Before sensors or actuators can be declared, the program must initialize the Meta stub, declaring to the stub its context. The initialization routine has the following type signature:
void
meta_stub_init(
    char *context_class,
    char *instance,
    int isisport,
)

To replicate a context, multiple instances of the implementing program may be run. However, the Meta system needs to know that the context is being replicated. The long form of the meta_stub_init routine has an extra parameter for this purpose. Passing a value of TRUE will enable this program to operate in concert with other, optional instances of this program. Replication is transparent in all other respects. The long form of the initialization routine has the following form:

void
meta_stub_init_l(
    char *context_class,
    char *instance,
    int isisport,
    int is_repeated
)

### A.2.1 Sensors

Sensors are the mechanism that exposes parts of the program state to Meta. Sensors are implemented by C functions, which are declared to the Meta stub via the meta_newsensor procedure. This routine takes the name of the sensor as a string along with the address of the function implementing the sensor. Additionally, the type and polling period must be specified. The procedure, whose definition follows, returns a handle for the sensor.

```
int
meta_newsensor(
    int (*sensor)(),
    char *name,
    int type,
    int pollperiod
)
```

<table>
<thead>
<tr>
<th>sensor</th>
<th>Address of sensor function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name function is to be known by.</td>
</tr>
</tbody>
</table>
**type**  
Type of sensor function, e.g., TYPE_INT.

**pollperiod**  
Maximum number of milliseconds that should elapse between two successive polls of the sensor by Meta. If zero is specified, then the sensor is polled each time meta_notify is called. The constant NEVER_POLL can be specified to indicate that the sensor should only be sampled when specifically passed as a parameter to the meta_notify routine.

A long form of this function exists, taking one additional parameter. This parameter, func_arg, is passed as a parameter to the sensor function when it is invoked. This facility allows the same function to implement multiple sensor definitions.

```c
int
meta_newsensor_l(
    int (*sensor_function)(),
    char *name,
    int type,
    int pollperiod,
    int func_arg
)
```

The format for sensor functions is as follows, where type is the native C type.

```c
int
sensor_function(
    type *value,
    int func_arg
)
```

**value**  
Points to address of object to get the sensor value. The sensor function is expected to store at this address the value of the sensor. For sets the value should be of type SET * and for strings this is of type char *.

**func_arg**  
Parameter specified in the meta_newsensor_l call; may be omitted if the sensor was declared to Meta using meta_newsensor.

The following dummy sensor always returns the string “hello, world.”
int hello(value)
char **value;
{
    static char *message = "hello, world."
    *value = message;
    return 0;
}

The following function is the implementation of the localtime sensor, as provided by the machine program.

int mytime(value)
long **value;
{
    extern long time();

    time(value);
    return 0;
}

Meta will call the sensor function at least as often as the pollperiod parameter passed to meta_newsensor, assuming the Meta library gains control at least that often.

The programmer must inform Meta when the program is in a consistent state and that sensors may therefore be sampled. This is accomplished by calling the meta_notify procedure. This routine will cause Meta to poll all sensors whose timers have expired and to process any necessary external events. The routine meta_notify can also be used to indicate that the values of specified sensors have changed; sensors are identified by the handles returned from the meta_newsensor calls. By passing a set of sensor handles, the programmer can ensure that Meta sees changes in different sensor values together.

void
    meta_notify(
        int num_handles,
        int handles[]
    )

num_handles The number of handles being passed.
The `handles` vector of handles, where each handle is an identifier returned by the `meta_newsensor` routine.

If an application consists just of a set of sensors and actuators, with no main thread of control, the following routine may be called to serve as the main thread of control. This routine does not return, so it should be the last routine invoked in the user program.

```c
void meta_mainloop()
```

Sometimes a program may not care when Meta runs; the program is such that Meta may safely run each time the program blocks. In that case, the following routine may be called to create a thread that will call `meta_notify` each time an ISIS event occurs.

```c
void meta_thread()
```

Note that when calling `meta_thread`, the program is still responsible for ensuring that Meta has opportunities to run: either by invoking blocking ISIS calls, by calling `sleep`, or by calling `isis_accept_events`.

The next procedure creates the sensors listed in the following table; these sensors return resource usage statistics for the current process. All of these sensors are of type integer.

```c
void
sn_rusage(
    int pollperiod
)
```

The `pollperiod` polling period to be used by Meta for these sensors, in milliseconds.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>utime</td>
<td>user time in seconds</td>
</tr>
<tr>
<td>stime</td>
<td>system time in seconds</td>
</tr>
<tr>
<td>etime</td>
<td>total elapsed time in seconds</td>
</tr>
<tr>
<td>maxrss</td>
<td>maximum resident set size in bytes</td>
</tr>
<tr>
<td>pfaults</td>
<td>number of page faults</td>
</tr>
<tr>
<td>blockio</td>
<td>number of block input/output operations</td>
</tr>
</tbody>
</table>

### A.2.2 Actuators

Actuators are the mechanism through which Meta controls an application. An actuator is a procedure that takes one parameter, a value of some type. Actuators are
named in the same manner as are sensors, i.e., \textit{context.object}.

Each actuator is declared using the following procedure:

\begin{verbatim}
int
meta_newactuator(
    int (*actuator_function)(),
    char *name
)
\end{verbatim}

\textbf{function}   The address of the function implementing this actuator.

\textbf{name}      The name the actuator is to be known by.

A long form of this function exists, taking one additional parameter. This parameter may be used to pass information to the actuator function, allowing the same function to implement multiple actuator definitions.

\begin{verbatim}
int
meta_newactuator_1(
    int (*actuator_function)(),
    char *name,
    PTR func_arg
)
\end{verbatim}

Actuator functions have the following form:

\begin{verbatim}
int
actuator_function(
    ACT_NODE *ap,
    PTR func_arg
)
\end{verbatim}

\textbf{ap}      Points to the structure containing the parameters for the actuator.

\textbf{func_arg}   Argument passed to the \textit{meta_newactuator_1} call; may be omitted if \textit{meta_newactuator} was used to declare the actuator.

The actuator parameters may be referenced by using the following routines:

\textbf{NUM_ACT_PARMS(ap)}   Returns number of parameters being passed.

\textbf{ACT_TYPE(ap, i)}   Returns the type of the \textit{i}th parameter; the first parameter is numbered zero.

\textbf{ACT_PARM.ISBOT(ap)}   Returns true if the value of the \textit{i}th parameter is undefined.

\textbf{ACT_PARM(ap, i)}   Returns a pointer to the \textit{i}th parameter.
Actuator functions must return either META_SUCCESS or META_FAILURE depending on whether the actuation succeeded or failed.

The following simple actuator takes one parameter, a string, and invokes the Unix system routine with that parameter.

```c
int exec(ap)
ACT_NODE *ap;
{
    if ((NUM_ACT_PARMS(ap) != 1) || (ACT_TYPE(ap,0) != TYPE_STRING))
        return META_FAILURE;
    if (system((char *) ACT_PARM(ap,0)))
        return META_FAILURE;
    else
        return META_SUCCESS;
}
```

## A.3 Front-End

This section describes how sensors and actuators may be accessed. Of especial importance is the NPL guarded command language.

### A.3.1 Initialization

Access to Meta objects is provided by including the `metalib.a` library and referencing the include file `meta.h`. Before using any other library routine, the process must first initialize the client stub which provides access to Meta objects.

```c
void
meta_init(
    long ISISPort
)
```

**ISISPort** The port number to be used. If zero, then the system uses the port number specified in the environmental variable ISISPORT.

Note these routines call `isis_start_done`; consequently if the program also uses ISIS, its ISIS start-up sequence should be done before calling these routines.

### A.3.2 Client Operations

All of the following routines return as a result a status code; the error codes are defined in the file `meta.err.h`. 
The following routine is called to obtain the current value of a specific sensor, as specified by its context and name.

```c
int
meta_current_value(
    char * context,
    char * sensor,
    PTR value,
    int * type,
    message **mval
)
```

- **context**  Context implementing the sensor.
- **sensor**   Name of the sensor.
- **value**    Set to the value of the sensor obtained when the sensor was last polled by the system during a meta_notify call. Note that for the string type, the pointer value is of type char * and so points to the string; for set types, the pointer value is of type SET.
- **type**     Set to the type of the value of the sensor.
- **mval**     Set to point to a message containing the value; the value remains valid until the message is deleted.

An NPL program may be downloaded to a specific context by the following function:

```c
int
meta_npl(
    char * context,
    char * program,
    void (*call_back)(),
    int handle
)
```

- **context**  Context to send program to.
- **program**  String denoting the NPL program.
- **call_back** Specifies a function to call when the NPL program exits; this parameter may be specified as NULL if no call-back is desired.

The call-back function has the following form:
void
call_back(
    int handle,
    int status,
    int num_vals,
    int types[],
    PTR vals[]
)

handle    Handle as provided to the meta_npl command.
status    Status code from the stub which was given the program.
um_vals   Number of values being returned.
types     Vector of integers specifying the type of each value.
vals      Vector of pointers to the actual values. The values pointed to are only valid while the call-back routine is active; they should be copied if they need to persist beyond that. Note that for string types, the pointer value is of type char * and so points to the string; for set types, the pointer value is of type message * and points to the message representing the set.

A macro may be defined using the meta_macro call. The macro may then be referenced in any NPL expression evaluated in that context; this is accomplished by prefacing the macro name with a $ symbol.

int
meta_macro(
    char * context,
    char * macro_name,
    char * macro_def
);

The use of aggregates provides a way to group together members of the same context class. For example, one might define a FreeMachine aggregate consisting of all machines whose load is less than 1.0. To implement an aggregate, a stub must be designated to handle the aggregate sensors and actuators. Stubs are named by their base contexts, i.e., the context name passed to the meta_stub_init routine.

An aggregate context automatically inherits all the sensors and actuators of the base context class, the context class over which the aggregate is formed. The types of the sensors are promoted to be of type composite, where a composite value consists of a list of values from each member of the aggregate with each value tagged with the name of the context providing that value.
The following routine endows a stub with an aggregate context. In effect, this function creates the aggregate. NPL level commands are used to cause interpreters to join or leave the aggregate.

```c
int
meta_endow_aggregate(
    char * endowee,
    char *agg_name,
    char *base_name
)
```

**endowee** Identifies the stub to implement the context; specify the base context of that stub.

**agg_name** Context name the aggregate is to be known by.

**base_name** Context class over which the aggregate is to be formed.

Sensors may be defined by naming an NPL expression; the value of the expression is the value of the sensor. The sensor’s value changes whenever the value of the expression changes; the expression is reevaluated whenever one of the sensors referenced in the expression changes. A sensor defined in this manner may be treated like any other sensor.

```c
int
meta_define_sensor(
    char * context,
    char * sensor_name,
    char * sensor_expr
)
```

A program may subscribe to a remote sensor by calling the following function, passing along as a parameter a call-back routine. This call-back routine will be invoked by the Meta stub with the value of the specified sensor at the time the subscription request is processed by the remote stub. The call-back routine is also invoked to notify the program of any subsequent changes in sensor value.

```c
int
meta_subscribe(
    char *context,
    char *sensor,
    void (*func)(),
    int arg,
    int *handle
)
context  Context containing sensor.
sensor  Name of sensor.
func  Call-back function to be called each time the sensor value changes. It currently is guaranteed that the sensor value has indeed changed each time this routine is called.
arg  User-specified parameter to be passed to the call-back routine.
handle  Set to a handle to identify this subscription; used to subsequently cancel the subscription.

This routine returns appropriate status indication if the sensor is not known in that context. Otherwise, the specified function will be called using the following protocol whenever the value of the named sensor has changed:

```c
void
func(
    char *context,
    char *sensor,
    PTR value,
    int type,
    int arg
)
```

custom  Context containing sensor.
sensor  Name of sensor.
value  Pointer to value of sensor.
type  Type of sensor value. The constant -1 is passed if the sensor is no longer available, i.e., if it was previously available but is no longer. If the context containing the sensor is not available at the time the subscription request is made, then the call-back routine will not be invoked until the sensor value does become available.
arg  User-specified argument, as passed to the meta_subscribe routine.

A subscription may be cancelled using the following routine, passing as a parameter the handle passed back by the meta_subscribe routine.

```c
int
meta_cancel_subscription(
    int handle
)
```
handle  The handle as passed back from the `meta_subscribe` function.

Any type of sensor supported by Meta may be subscribed to: those sensors declared by the `meta_new_sensor` routine, those sensors defined by calling the `meta_define_sensor` routine, and the composite-valued sensors which are automatically inherited by an aggregate.

The following routine takes a Meta error number and prints to `stderr` a textual description of the error.

```c
void meta_error(
    int errnum
)
```

### A.4 The NPL Programming Language

The basic alternative command—set of guarded commands—takes on the following form:

```
predicate GUARD action sequence
[ALTERNATE predicate GUARD action sequence]
```

that is, it consists of one or more clauses, where each clause consists of an guard expression the operator `GUARD`, and an action expression. Clauses are separated by the keyword `ALTERNATE`.

The predicate is a postfix expression usually referencing sensors and evaluating to an integer value. The value zero is considered false, any other numeric value is considered true.

The action sequence is a list of actuator invocations, with each invocation consisting of a postfix expression followed by the name of the actuator. Actuator names are prefaced with a `` so the system can distinguish an actuator from its parameters. The postfix expression need not evaluate to a single value; this allows multiple parameters to be passed.

Macros may be referenced by prefacing the name of the macro variable with a `$`. A macro variable is bound to a macro definition using the `meta_macro` command described in Section A.3.2

#### A.4.1 Executing Guarded Commands

Each time the `meta_notify` routine is called, all guards whose value has potentially changed are reevaluated. One or more guards may become executable; if more than one guard is executable, then a guard is fairly picked for execution. In particular,
the system chooses guards in a round-robin fashion, searching for executable guards by starting at the guard last executed.

The guard chosen for execution may have multiple clauses that are executable; exactly one is picked for execution, again in a fair fashion. This clause is then executed by invoking in a sequential fashion the actuators listed for the action for the selected guard. Note that the values of all objects are frozen for the duration of the actuation the moment the execution begins. In the current implementation, if one actuator aborts, the rest of the action is aborted, but nothing else happens. In particular, the effects of any actuators invoked before the action was aborted will persist.

### A.4.2 Predefined Sensors and Functions

The base context for each Meta program contains the integer-valued sensor ALIVE; this sensor always returns true. However, if a particular context is not available, then the value of that context’s alive sensor is BOTTOM, i.e., undefined. This sensor is most useful in forming aggregates. It is also used in the predicate of a guard to ensure that all the contexts referenced in the action part of the guard are available.

The following set of functions is predefined in the global context and may be referenced by any program.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>Binary</td>
<td>value of -1, 0, or 1</td>
</tr>
<tr>
<td>&gt;</td>
<td>Binary</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Binary</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Binary</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Binary</td>
<td>less than or equal</td>
</tr>
<tr>
<td>=</td>
<td>Binary</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Binary</td>
<td>not equal</td>
</tr>
<tr>
<td>#</td>
<td>Unary</td>
<td>size</td>
</tr>
<tr>
<td>AVG</td>
<td>Unary Composite</td>
<td>average value</td>
</tr>
<tr>
<td>SORT</td>
<td>Unary Composite</td>
<td>sort by value</td>
</tr>
<tr>
<td>+</td>
<td>Binary integer, real</td>
<td>arithmetic add</td>
</tr>
<tr>
<td>-</td>
<td>Binary integer, real</td>
<td>arithmetic subtract</td>
</tr>
<tr>
<td>*</td>
<td>Binary integer, real</td>
<td>arithmetic multiply</td>
</tr>
<tr>
<td>/</td>
<td>Binary integer, real</td>
<td>arithmetic divide</td>
</tr>
<tr>
<td>AND</td>
<td>Binary boolean</td>
<td>logical and</td>
</tr>
<tr>
<td>OR</td>
<td>Binary boolean</td>
<td>logical or</td>
</tr>
<tr>
<td>NOT</td>
<td>Unary boolean</td>
<td>logical not</td>
</tr>
<tr>
<td>TIMER</td>
<td>Unary integer</td>
<td>returns true after ( \text{operand} ) milliseconds</td>
</tr>
</tbody>
</table>

In an aggregate context, all the sensors in the context class over which the aggregate is defined may be referenced; their type is a composite of the original type.
A.4.3 Expression Evaluation

Postfix expressions are evaluated in the normal manner. Type checking is strict: the operands to a binary function must be of the same type. However, there are some exceptions to this rule. For arithmetic functions, if either operand is a real, then both are promoted to real; the type of the result is also real.

In general, if the operand to a unary function has the value BOTTOM, or if either operand to a binary function has the value BOTTOM, then the resulting value is BOTTOM. However, the boolean operators AND and OR are special:

\[
\text{FALSE and BOTTOM} = \text{BOTTOM and FALSE} = \text{FALSE}
\]

and

\[
\text{TRUE or BOTTOM} = \text{BOTTOM or TRUE} = \text{TRUE}
\]

The constant BOTTOM is predefined and may be used in expressions.

A.4.4 Predefined Actuators

The following set of actuators is predefined in the global context.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOIN</td>
<td>string</td>
<td>Joins the specified aggregate</td>
</tr>
<tr>
<td>LEAVE</td>
<td>string</td>
<td>Leaves the specified aggregate</td>
</tr>
<tr>
<td>PRINT</td>
<td>string</td>
<td>Prints out the string on stdout</td>
</tr>
<tr>
<td>EXIT</td>
<td>variable</td>
<td>Terminates the guarded command</td>
</tr>
<tr>
<td>NPL</td>
<td>string, string2</td>
<td>Executes NPL program string 2 on context string1</td>
</tr>
<tr>
<td>LNPL</td>
<td>string</td>
<td>Executes NPL program, local context</td>
</tr>
</tbody>
</table>

The EXIT actuator takes a variable number of parameters. These values are returned to the client when the call-back routine passed to meta_npl is invoked.

Programs executing in an aggregate context automatically have access to all the objects in the context class over which the aggregate is formed, the base context class. If an object is referenced and is not found in the aggregate context class, it is assumed to be an object in the base context class. The object's type is promoted to TYPE_COMPOSITE.

A.4.5 Aggregate Invocation

Recall that composite actuators are those actuators existing in an aggregate context. When invoking such aggregators, two additional parameters are specified, specifying which members of the aggregate are to perform the action. The first parameter is an integer, denoting the number of aggregate members which are supposed to perform
the action. The second parameter is a composite value. If the first parameter is a non-negative number \( n \), then the aggregate action is performed on the first \( n \) members making up the composite value. (Recall that a composite value is an ordered set of pairs of the form (subvalue, source); the action is carried out at the corresponding source). Should any of the first \( n \) members crash while carrying out the action, then additional members are select to perform the action, until \( n \) members have successfully completed the action. If however, any member of the aggregate returns a failure code, as opposed to crashing, then the actuation is aborted.

If the first parameter has instead the value -1, then the second parameter is ignored and the action is invoked on all active members of the aggregate.

For example, suppose we have a program containing the `exec` actuator defined previously, which takes one parameter and runs it as a Unix shell command. This program belongs to some context class; suppose that a Meta server is endowed with an aggregate formed over that context class. Then the following NPL program causes all active members of that aggregate to execute the Unix "date" command:

1 GUARD -1 0 "date" 'exec 'EXIT

The following program will run the "date" command on five members of the aggregate:

1 GUARD 5 ALIVE "date" 'exec 'EXIT

A typical application of this feature might involve sorting the composite value to determine an ordering preference. For example, the included machine program contains a load sensor and the exec sensor. The following program, when given to an aggregate over the machine class, will run the "cruncher" program on the least loaded machine.

1 GUARD 1 load SORT "echo cruncher" 'EXIT

### A.5 Running Meta Applications

Before a Meta application can be executed, the ISIS system must be running first. ISIS is connected to the standard fashion for ISIS applications. In particular, the environment variable `ISISPORT` may be used to set the ISIS port. If this variable is set to zero, or if it is not defined, then the system services file is consulted to determine the ISIS port number.

Several standard applications are provided with Meta.

**machine** This program instruments a machine, and includes objects such as a load sensor and an exec actuator.
server The Meta NPL server; this program serves only to host a Meta stub and is used for handling aggregate definitions.

query This program provides a simple interface to the client Meta library routines.

A.5.1 The Machine Program

A set of objects of context class machine are installed on a machine by running the client program machine. The sensor polling period may be specified as an optional argument, in the following form:

    machine [polling period]

This context provides the following sensor objects:

archetype A string giving the architecture of the host machine.

boottime A integer number giving the time the machine was last booted, measured in seconds from midnight GMT, January 1, 1970 (the standard Unix time format).

load A real number giving the current load of a machine.

localtime A integer number giving the value of the machine’s time-of-day clock, as returned by the Unix time function.

login A set of strings giving the account names of the current users of the system.

The machine program also provides the following actuators:

exec Executes a Unix command. Takes one string parameter; that parameter is passed to the UNIX system routine.

mail Sends mail. The first parameter must be a string giving the recipient’s user-id. The rest of the parameters are converted to strings and sent as the body of the message.

A.5.2 The Server Program

The server program is invoked by

    server [-r] [optional instance]

The -r option specifies that the server should run in the replicated mode.

The context of the server is simply server if not instance is specified; otherwise the context is server.instance.
A.5.3 The MetaCmd Program

The Meta command program is invoked by

```
metacmd [optional commands]
```

If no command line arguments are present, the program reads from standard input.

The following commands are recognized:

**poll sensor**

Invokes the `meta_current_value` routine to obtain the current value of the specified sensor.

**endow context aggregate base**

Invokes the `meta_endow_aggregate` routine to create the specified aggregate which is formed over the given base context class. The aggregate is implemented by the stub denoted by `context`.

**exit** Causes the program to exit.

**macro context macro_name definition**

Uses the `meta_macro` routine to define the macro `macro_name` as having the specified definition and residing in the specified context.

**npl context program**

Uses the `meta_npl` routine to send the specified NPL program to the specified context for execution.

**sensor context sensor_name expression**

Uses the `meta_define_sensor` routine to define the sensor `sensor_name` to be the given expression and residing in the specified context.

**subscribe subscribe sensor**

Generates a subscription to `sensor` by using the `meta_subscribe` routine. The received sensor values are displayed on the standard output device.

**wait** Causes the MetaCmd program to wait until all previous NPL programs have completed execution before processing additional commands.
A.6 Limitations

The number of actuators invoked in a guard is limited to be no more than the constant `MAX_ACTPARMS`.

The number of members in an aggregate is limited to be `MAX_AGGR_MEMS`.

No limit is imposed by the Meta type system on the length of strings; however, some of the input and output routines do limit the character length of textual representation of a type to be less than the constant `TYPE_MAXSIZE`. This limit includes special characters such as the curly braces and spaces used in set representations.

A subscription request to a non-existent object does not fail but is simply ignored.

If a context is replicated, all replicas must be started up before any attempt is made to communicate with the replicas.

A.7 Constant Definitions

The current definitions of system constants are presented here for your convenience. These numbers may change in future releases of the system.

```c
#define TYPE_INT 0
#define TYPE_REAL 1
#define TYPE_STRING 2

#define TYPE_BASE 15
#define TYPE_SET 16
#define TYPE_INTERVAL 32

#define TYPE_MAXSIZE 256

#define SN_DIVIDEBYZERO -15
#define SN_BADAGGINVOCATION -14
#define SN_MACROOODEEP -13
#define SN_NOMACRO -12
#define SN_NAMEDEFINED -11
#define SN_TYPEMISMATCH -10
#define SN_BADINTERVAL -9
#define SN_NOMEMORY -8
#define SN_SYNTAXERR -7
#define SM_FAILED -6
#define SM_NAMETOOLONG -5
#define SN_NOTKNOWN -4
#define SM_NOTKNOWN -3
#define SM_NOTAVAIL -2
```
#define SM_INTERNAL -1

#define MAX_AGG_MEMS -16
#define MAX_ACTPARMS 8
Bibliography


