Tools for Constructing Distributed Reactive Systems

Keith Marzullo[*] Mark D. Wood[†]
marzullo@cs.cornell.edu wood@cs.cornell.edu

Cornell University
Department of Computer Science
Ithaca, New York 14853
February 22, 1991

Abstract

Many distributed applications can be cast as a *reactive systems*, where a reactive system consists of an instrumented program that is monitored and controlled by an input-driven control program. Examples of non-real-time reactive systems include monitoring and debugging systems, tool integration services, and network and distributed application managers. There is currently little support for building reactive systems. This paper describes the *Meta* toolkit that provides such support. Using Meta, a distributed system can be *instrumented* with a *sensor* and *actuator* abstraction that exposes the state of the system for purposes of control. Then, a control program can be written that interacts with the instrumented system using *guarded commands*. Of particular concern is the efficiency of control, so Meta allows the control program to be distributed in order to take advantage of locality as much as possible.

[*]This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, Contract N00140-87-C-8904. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision. This work was also partially supported by a grant from Xerox.

[†]This author was also partially supported by a G.T.E. Graduate Student Fellowship.

1 Constructing Reactive Systems

A *reactive system* [HP85] architecture is characterized by a *control program* that interacts with an instrumented process called the *environment*. The control program is input-driven: it monitors the environment and reacts to significant events by sending commands to the environment. The most common use of a reactive system architecture is for process control systems, in which case the environment is an instrumented physical process.

A reactive system architecture, however, can be used for more than process control systems. Monitoring and debugging systems [Fid88,MC88] have an obvious reactive system structure. Similarly, *tool integration services* [GI90,Rei90] can be cast as reactive systems. These services relate actions in one software tool to actions in other tools: for example, selecting text in a text editor and setting a breakpoint in a program debugger. A tool integration service can be built as control program that monitors tool actions and invokes tool operations when some sequence of actions occurs. *Distributed application management* [MCWB90] is a third example of a system with a reactive structure. In this case, the environment consists of the distributed application and its supporting services (for example, the operating system, network servers, and hardware), and the purpose of the control program is to make the application adaptive to changes in the set of operational machines, the input load, important input events, and so on.

This paper describes the Meta system, a toolkit that provides the glue needed to build non-real-time reactive systems. Using Meta, a distributed program can be *instrumented* with *sensors* and *actuators* in order to expose its state for purposes of control. Meta provides mechanisms that allow a control program, written as a set of guarded commands, to query the state of the instrumented application and to respond by invoking actuators when some condition of interest occurs. Meta includes facilities for structuring individual components into collections of components for purposes of fault-tolerance. In addition, Meta guarantees that the monitoring and reaction can be done as locally as possible and that it is done in a *consistent* manner (defined in this paper).

The kinds of systems Meta is intended for are like those mentioned earlier: the environment is a program that can run independently of the control program, and may have been written without much thought for how it might interact with a control program. In contrast, monolithic reactive systems can be built with existing tools. For example, in a factory floor system, the sequencing of parts from one workcell to another is easily cast as a reactive system: the sequencer is a control program that monitors and interacts with the floor transport system. This system can be built with the sequencer being a highly available service explicitly invoked by workcells. If most sequencing decisions are based on localized information, however, Meta would be a useful tool for building even this monolithic reactive system.

Meta itself is built on top of another toolkit, the Isis system [BJ87, BJKS88]. In fact, the Meta project was started when a group of us in the Isis project worked on constructing a large distributed application constructed from off-the-shelf components [MCWB90], and found Isis lacked tools for supplying the distributed control needed to manage this application. Not surprisingly, building on Isis has both simplified and complicated the implementation of Meta, as will be discussed in this paper.

The paper proceeds as follows. Section 2 gives a high-level description of how a set of programs is instrumented and controlled. Section 3 discusses in detail how a component is instrumented and Section 4 presents the interpretation and evaluation of guarded commands. Section 5 extends Section 4 to guarded commands that reference several components. Performance information is presented in Section 6 and we discuss our future plans in Section 7.

2 Architectural Overview

There are two steps to building a reactive system with Meta. First, the environment is instrumented with sensors that read the state of components and with actuators that allow the state to be changed. Then, a control program that references these sensors and actuators is constructed. The control program interacts with the sensors and actuators through guarded commands, described below. Simple control programs can be expressed solely in terms of guarded commands, and more complex control programs can be written as conventional programs that submit guarded commands to Meta.

In many ways, instrumenting a program is similar to exporting a set of remote procedure call entry points. A sensor is a function that returns the relevant state, and an actuator is a procedure that alters the relevant state and returns either "success" or "failure". A program is instrumented by binding a name to each sensor function and actuator procedure. The program itself is also bound to a name, and the program name and the names of its sensors and actuators form a naming context called a *base* *context*. For example, consider a compilation server whose load is defined to be the number of enqueued jobs. The server also has a procedure that cancels the currently active job. If this server runs on a machine named "rocky", we could bind the name "compile(rocky)" to the server, "load' to a function that returns the number of enqueued jobs, and "cancel' to the procedure that cancels the currently active job. In this case, the base context *compile(rocky)* defines the sensor *load* and the actuator *cancel*. All contexts consisting of the same set of sensors and actuators form a *context class*: for example, the context class of compilation servers.

A *Meta stub* is analogous to an RPC server stub (e.g., [BN84]). It exists in the same address space as the instrumented program and services requests made on the base context. In fact, this stub serves as a simple RPC stub, in that a remote client can procedurally call sensor functions and actuator procedures. However, Meta stubs support two additional features not found in RPC stubs:

1. A client can *subscribe* to a sensor or set of sensors. The subscribing client is asynchronously notified whenever the value of any of the sensors in the set change. Issues of sampling are discussed in Section 3.

2. A client can give the stub a guarded command to be executed. The client is asynchronously notified when the command terminates.

A *guarded command* is a set of (*predicate, action*) pairs, e.g.,

$$\langle \phi_1 \rightarrow \alpha_1 \ [\!] \ \ldots \ [\!] \ \phi_n \rightarrow \alpha_n \rangle \ \text{ or } \ \langle \phi_1 \rightarrow \alpha_1 \ [\!] \ \ldots \ [\!] \ \phi_n \rightarrow \alpha_n \rangle^*$$

where the asterisk indicates iteration. A predicate ϕ_i is a state predicate, and can reference sensors in any defined context. An action α_i is a sequence of actuator invocations, where each invocation can be from any defined context. The usual meaning of a guarded command (e.g., [CM88]) is that one of its actions α_i is executed in a state that satisfies ϕ_i, the command will eventually execute if continuously enabled, and the command's execution is *atomic*; that is, no intermediate states of α_i are visible. Meta approximates these semantics; in particular, Meta guarantees that ϕ_i is satisfied in a consistent global state, that the parameters to α_i are from the same state in which ϕ_i is satisfied, the command will eventually execute if continuously enabled, and that no intermediate states of α_i are visible in a failure-free execution. Such semantics have proven strong enough for the distributed application management problems that we have considered [MCWB90], but stronger semantics are implementable if necessary.

The reason for choosing a guarded command language is that it allows the the control program to benefit as much from locality as possible. A control program can express a control rule as a finite state automaton whose acceptance state performs a sequence of actuations. This automaton can be broken into guarded commands, which are then distributed to the stubs. Note that a guarded command can be executed by any stub, but the overhead of a stub executing a guarded command is high when the stub does not directly support the referenced contexts. If the guarded command references only a single context, then it should be run by the stub that implements that context. If the guarded command references several contexts, then the best location to run the command depends on several factors. Figure 1 illus-

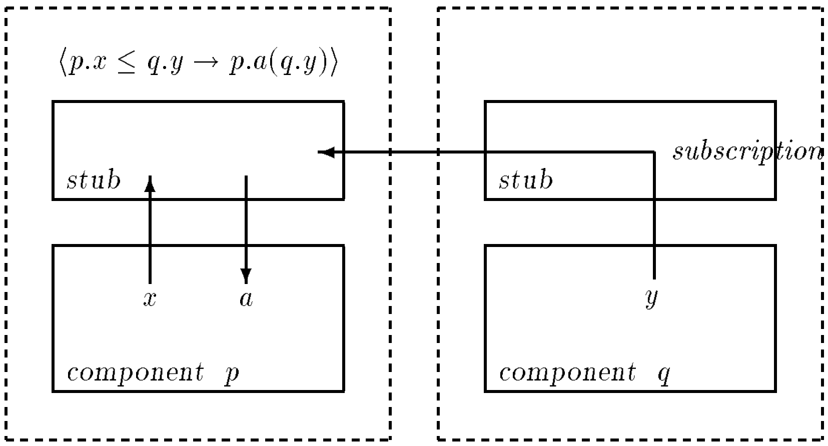

Figure 1: Execution of a Guarded Command, I

trates this issue. In this figure, a guarded command specifies that actuator a in context p is invoked when sensor x in context p becomes less than or equal to sensor y in context q. Having the stub of component p execute this command minimizes the number of remote references; only y is remote and is referenced using subscription. If y changes much more frequently than x, however, it would make more sense to execute the command in the stub of component q (shown in Figure 2).

An *aggregate* is a collection of components, all of the same context class (e.g., compilation servers). An aggregate behaves much like an instrumented program: an aggregate has sensors and actuators as well as a stub that

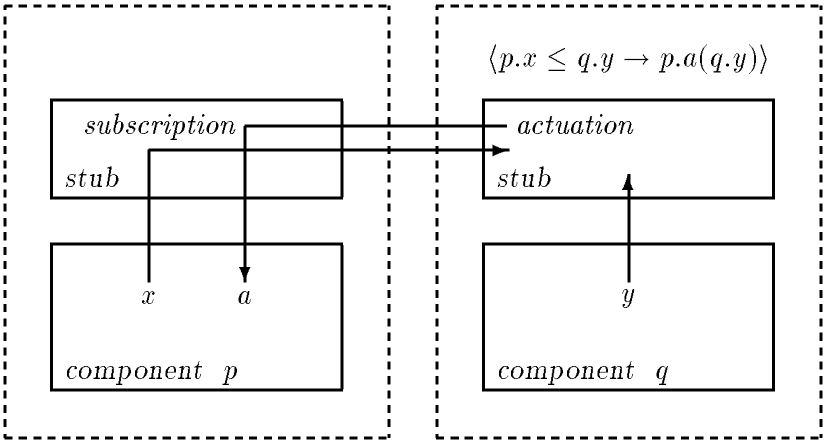

Figure 2: Execution of a Guarded Command, II

implements the aggregate's context. Each sensor in the aggregate's base context class becomes a set-valued sensor in the aggregate's context. For example, an aggregate of compilation servers would have a set-valued "load" sensor, with each member of the aggregate contributing one of the values in the set. Each actuator in the aggregate's base context class becomes an actuator in the aggregate's context. Invoking this actuator invokes in parallel the corresponding actuator in each member of the aggregate. As is discussed in Section 5.2, there is also a method of invoking only a subset of the member's actuators.

In order to create an aggregate, an existing stub must be told the aggregate context name and the aggregate's base context. If more than one stub is instructed to implement an aggregate's context, then one stub will act as a primary and the rest as backups. This, too, is discussed in Section 5.2.

3 Instrumenting the Environment

A program is instrumented by inserting some code and linking the program with the Meta library. When the resulting object is loaded, a Meta stub is created that implements the program's base context. This section describes what instrumentation entails and how the Meta stub and the program interact with each other.

As part of its start-up sequence, an instrumented program calls the Meta library routine

meta_stub_init(char * base_context_class, char * instance, int ISIS_port)

which names the base context provided by the program (the ISIS_port parameter is used in initializing the underlying Isis system). For example, if we assume that no more than one compilation server runs on a machine, then the server's base context could be named with the following library call:

meta_stub_init("compiler", gethostbyname(), 0)

A function is bound to a sensor with the library routine meta_new_sensor. For example, if svr_q_length is a integer-valued function that returns the number of entries on the server's work queue, then an integer-valued load sensor could be declared with the following library call:

meta_new_sensor(svr_q_length, "load", TYPE_INTEGER, min_period)

The fourth parameter is a lower bound on time between two significant changes of the sensor's value, and can be zero. Meta uses this value to time out cached values of the sensor (explained below). As with sensors, a procedure is bound to an actuator with a library call. The following binds the procedure job_cancel to the name "cancel":

meta_new_actuator(job_cancel, "cancel")

An actuator procedure has a fixed type: it takes a single parameter that points to an internal Meta object and returns a flag indicating success or failure. Other parameters can be passed from the actuator invocation to the actuator procedure, which can be retrieved using other Meta library routines.

The stub maintains a cached value of each sensor that is either subscribed to or is referenced in a locally-executed guarded command. These cached values represent an atomic state of the program as seen by Meta guarded commands.

There is a sampling problem, however: not all states of the instrumented program will comprise a legal atomic state as seen by Meta. For example, the load of a compilation server might be computed by traversing a linked list, and if the sensor function is called while the linked list is being updated, then the associated function may not compute the correct value or may cause the program to crash. A similar problem exists with invoking actuator procedures at arbitrary times. The simplest approach is have the the stub and the instrumented program run as coroutines implemented using procedure calls.

Figure 3 illustrates the flow of control between the instrumented program

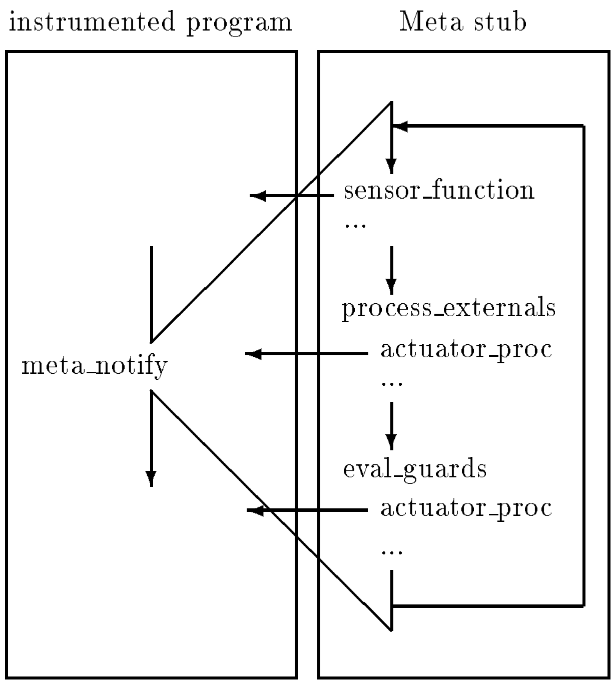

Figure 3: Instrumented Program—Stub Interaction

and its associated Meta stub. The instrumented program periodically calls the Meta library routine meta_notify. This routine transfers control to the stub, which then performs the following operations:

1. Fresh values are obtained for sensors whose cached values have timed out. The instrumented program can explicitly time out cached values of sensors by naming these sensors in a parameter to meta_notify.

2. External requests are processed. Such requests include setting up new

subscriptions, notifying subscribers of changed sensor values, receiving updated values of sensors to which the stub has subscribed, and performing actuations requested from remote sites.

3. Locally-executed guarded commands are evaluated and executed.

Executing a guarded command may recursively call meta_notify, which will only set a global flag in the stub and return. Rather than returning to the instrumented program, the original invocation then resets this flag and iterates again through the above three steps. Doing so can lead to an infinite loop in the stub when guarded commands continue to become enabled. For example, the guarded command "when x changes, increment x" will result in an infinite loop in the stub. This should not be surprising, since the guarded command specifies an infinite loop[1].

If appropriate, the instrumented program can create a separate thread for the Meta stub by using the meta_mainloop library routine. This is appropriate when the instrumented program is multithreaded and so can provide the necessary synchronization. It is also appropriate when the instrumented program contains no thread. For example, the current Meta release includes a program that instruments a machine with sensors for its load, its clock value, the currently logged-on users, etc., and an actuator that executes a shell command. This program includes only instrumentation code, and so invokes meta_mainloop as the last step of initialization.

An aggregate context is defined by specifying the base context class over which the aggregate is defined. The aggregate comes into existence by *endowing* some stub with knowledge of the aggregate, which is done with the meta_endow_aggregate function:

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

We defer until Section 5 the discussion on how aggregates are implemented.

[1] One way to think of Meta is that it allows a programmer to add extra state transitions to an existing automaton. In this model, the infinite loop mentioned above is an example of an unfair execution. It would be easy to eliminate this unfairness by having the stub iterate only a bounded number of times, but the resulting execution could still be unfair; i.e., if the instrumented program stops calling meta_notify.

4 Controlling the Environment

In addition to providing instrumentation support, each Meta stub contains an interpreter for a guarded command language. This language, called NPL[2], is a simple postfix language that has been designed to impose little overhead in parsing and interpretation. Since each stub contains an NPL interpreter, guarded commands governing a program's behavior can be executed at the location that gives the best locality. This is important since nonlocal references are several orders of magnitude slower than local references.

An NPL program is assigned to a specific context for evaluation via the meta_npl call:

```
int meta_npl(char * context, char * program,
    void (*call_back)(int handle, int num_vals, int type[], PTR value[]),
    int user_handle)
```

The call-back routine is invoked when the guard terminates.

4.1 The NPL guarded command

The basic guarded command has the following form:

predicate GUARD *actions* [ALTERNATE *predicate* GUARD *actions*]*

That is, it consists of one or more clauses, where each clause consists of a guard expression followed by the keyword GUARD, followed by an actions. Clauses are separated by the keyword ALTERNATE. A guard expression must evaluate to an integer value, where the value *zero* represents *false* and any other value represents *true*. An action consists of a sequence of actuator invocations, with each invocation consisting of a postfix expression evaluating to the actuator's parameters followed by the name of the actuator.

NPL contains the expected set of arithmetic, relational and set operators. NPL is also extensible, in that additional operators can be defined and implemented as C routines. Such extensibility allows complex functions to be implemented in an efficient manner rather than as NPL interpreted code.

An NPL expression is evaluated with respect to the context specified in the meta_npl call. Sensors and actuators implemented in other contexts

[2]NPL stands for **N**ancy's **P**ostfix **L**anguage in honor of the implementor of the first prototype.

are specified by prefacing the sensor name with the name of the context. When a guarded command references a sensor in another context, the stub executing the command subscribes to the referenced sensor. Issues arising from referencing remote sensors are discussed in Section 5.

Two additional NPL facilities are relevant to this paper. First, new sensors can be defined with NPL expressions. Such sensors can be used like any other sensor: they can be subscribed to and referenced in guarded commands. Second, clients can define NPL macros. This facility is necessary when writing guarded commands that can invoke each other.

4.2 Executing Guards

Let C be the set of NPL guarded commands a stub is executing at some time. Each guarded command $c \in C$ has a *read set* $R(c)$, which is the set of sensors referenced in c. Recall that the stub maintains a cache of values for each sensor in any read set of the commands in C, where the values in this cache represent an atomic state of the environment. This cache is updated (perhaps several times) when the instrumented program calls meta_notify. Let $\Sigma(i)$ be the i^{th} atomic state; i.e., a set of (*sensor name, value*) pairs where $\Sigma(i).n$ is the set of sensor names and $\Sigma(i).v$ is the set of values.

The *trigger set* $T(c) \subseteq R(c)$ of a guarded command is the set of sensors referenced in the guards of c (some sensors might only be referenced as parameters to actuators). In the i^{th} atomic state, some commands c may have their trigger sets change, i.e., $T(c) \cap (\Sigma(i) - \Sigma(i-1)).n \neq \emptyset$. Each such command c has its guards re-evaluated. If c has a guard ϕ_i that was *false* in $\Sigma(i-1)$ and *true* in $\Sigma(i)$, then its associated action α_i is marked *executable*[3].

The stub marks actions in this manner for all commands whose trigger set was changed. The stub then selects one command that has an executable action, and then selects and executes one executable action of this command. If the command does not terminate (via EXIT, defined below), then all of its executable actions are marked *not executable*. An action remains *not executable* until its guard becomes *false*, after which it can become *executable*

[3]Note that this definition assumes $T(c)$ is defined in $\Sigma(i-1)$, which may not hold if c was submitted after the $(i-1)^{th}$ atomic state. Depending on the condition being detected, it may or may not be important to detect a guard becoming true versus it being initially true. The GUARD command will make an action α_i executable when its guard ϕ_i is initially true, and the command TGUARD will only make α_i executable when its guard ϕ_i becomes true.

again. Fair execution of actions requires that if an action becomes executable for some bounded number of times, it will be executed. Meta stubs guarantee this by using an LRU (least recently used) selection policy for each action of a guarded command.

The stub also selects the guarded command for execution by using an LRU policy. Note that executing an action may change sensor values, thereby causing some executable actions to no longer be executable. Conversely, new guarded commands may become executable as a result of executing the action. The stub continues to select commands and actions as long as there are executable commands.

4.3 Built-in Actuators

Several important actuators are built into the Meta library. The EXIT actuator terminates execution of a guarded command, removing the command from the set of guards maintained by the stub. This actuator may take optional parameters which are passed to the call-back routine specified to the meta_npl library routine[4]. For example, the following simple NPL expression alerts the client when the load goes above five. Additionally, the current value of the load is returned to the client:

```
load 5 > GUARD load EXIT
```

The library also includes the JOIN and LEAVE actuators, by which a component may join or leave an aggregate. The NPL actuator takes a context and an NPL program as its parameters, and executes the program in the specified context. By using this actuator in combination with the NPL macro facility, one can express finite state automata machines as guarded commands, thereby allowing ordering constraints to be expressed.

For example, suppose we wanted to take some corrective action if the load remains above five for more than 20 seconds. This behavior is expressed as a finite state automaton in Figure 4, and as an NPL program in Figure 5 (we assume that the program is run in a context named "server"). TIMER is a built-in function that takes a numeric parameter n; it returns true after n milliseconds has elapsed since the guarded command was submitted to the stub.

[4]Strictly speaking, EXIT is not an actuator since it does not alter the state of the instrumented environment. From the point of view of NPL, however, any operator that alters the environment or changes the way it is monitored is an actuator.

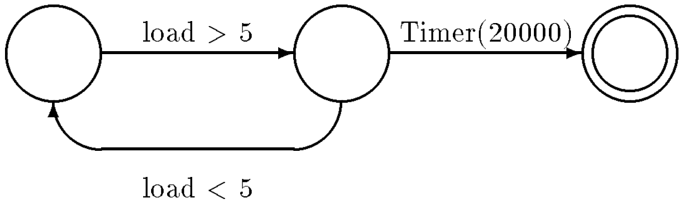

Figure 4: Simple Finite State Automaton

```
load 5 > GUARD
    "server"
    "load 5 < GUARD EXIT ALTERNATE 20000 TIMER GUARD "Idle-Server" LEAVE"
    NPL
```

Figure 5: Expressing Temporal Constraints in NPL

5 Non-Local References

Recall that the operational semantics of guarded commands $\langle \phi \to \alpha \rangle$ requires atomic execution of α in a state satisfying ϕ. Since such a state can be global, Meta must address the problems of detecting a global state predicate and reacting in a consistent manner.

5.1 Consistency

Before obtaining a globally consistent view, the Meta stub must first have a consistent view of the local state. Meta assumes that the local state is consistent when the instrumented program calls meta_notify. Since the instrumented program is blocked while it makes this call, any reaction α_i to the detection of a local predicate ϕ_i will be done in the same state that satisfied ϕ_i. In addition, if α_i only references local actuators, then the execution of α_i will also be atomic in that no intermediate states will be visible to other guarded commands. This is because the stub serializes access to all sensors that can be changed by α_i.

Obtaining a virtually simultaneous view of the global state is more difficult. We address next the three different properties guarded commands provide: detection of a global property, the state in which the reaction occurs, and atomicity of reaction. Our implementation depends strongly on what Isis provides, so we first review the features of Isis that Meta uses.

5.1.1 Isis

The Isis system provides a set of communication protocols and tools for developing fault-tolerant distributed applications. We only describe here the most important features of Isis that were used in the design of Meta. A more comprehensive description of Isis and its underlying principles may be found in [BJ87] and [BJKS88].

- *Fault-Tolerant Process Groups.* Isis provides location transparent communication with a process group and among its members. When a member of a group crashes, the Isis failure detection mechanism first completes all pending broadcasts and then informs the other group members of the failure.

- *Group Broadcasts.* Isis supports a set of broadcast primitives for sending messages to all members of a group. Depending on consistency requirements, the programmer may choose between atomic broadcast primitives (abcast and gbcast) that guarantee a global order on all messages and the more efficient causal broadcast protocol (cbcast) that guarantees only that the delivery order of messages is consistent with causality.

- *Coordinator-Cohort.* Some actions need to be performed by a single process with the guarantee that the action is completed even if that process fails. Isis provides a tool for structuring a computation like this. One member of a process group (the *coordinator*) is chosen to execute the action, while the other members (the *cohorts*) monitor its progress and are prepared to take over for the coordinator should it crash. This tool can be generalized so that a subset of the process group executes the action.

5.1.2 Detection

The problem of detecting whether a global state satisfies a state predicate is fundamental to distributed systems. For example, [CL85] constructs a consistent global state—a consistent cut—by using a flooding protocol, while [Tay89] does the same by blocking the application. Consistent cuts are useful in that they represent a physically-realizable, though not necessarily actual, system state. In particular, if S^* is a consistent cut and ϕ is true along S^*, then there is a valid execution of the system in which ϕ held.

Suppose a stub p_0 is monitoring to see when ϕ becomes true. Let Σ denote the last atomic state seen by p_0. For each nonlocal sensor σ_i referenced in ϕ, p_0 will subscribe to the stub p_i that implements σ_i, and p_i will send the sequence of sensor values $\sigma_i^1, \sigma_i^2, \ldots, \sigma_i^k, \ldots$ to p_0. Whenever p_0 receives a new value of σ_i, it overwrites the old cached value for σ_i, and so Σ will contain the latest values p_0 has received for each remote sensor. We define Σ to be *consistent* if there is a consistent cut of the instrumented system where each σ_i actually equals the value in Σ.

If the following ordering condition is met, then Σ will be consistent:

$$(\sigma_i^k \rightarrow \sigma_j^\ell) \Rightarrow (\sigma_i^k \text{ is delivered at } p_0 \text{ before } \sigma_j^\ell) \qquad (1)$$

The relation $\sigma_i^k \rightarrow \sigma_j^\ell$ means that σ_i had the value σ_i^k before σ_j had the value σ_j^ℓ as defined by causality. To see why Condition 1 guarantees Σ is consistent, suppose Σ were *not* consistent. Then, there must exist sensor values σ_i^k and σ_j^ℓ such that $\sigma_i^k \rightarrow \sigma_j^\ell$, Σ contains σ_j^ℓ, and Σ contains a value of σ_i preceding σ_i^k. This execution is shown in Figure 6, and violates Condition 1. If Condition 1 is met, then Σ must contain a value of σ_i no earlier than σ_i^k since p_0 has received σ_j^ℓ.

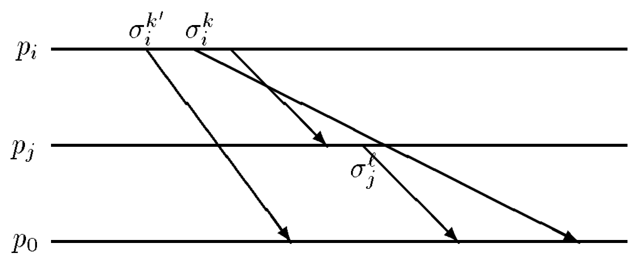

Figure 6: Constructing an Inconsistent Global State

Condition 1 is not strong enough when separate stubs monitor for the same condition ϕ. To see this, consider the sensor values $\sigma_i^k, \sigma_j^\ell$ and $\sigma_j^{\ell+1}$. If the sensor value from p_i are not causally ordered with the values from p_j, then one stub might receive the sensor values in the order $\sigma_i^k; \sigma_j^\ell; \sigma_j^{\ell+1}$ and construct the global state $\Sigma = (\sigma_i^k, \sigma_j^\ell)$ while the other might receive the sensor values in the order $\sigma_j^\ell; \sigma_j^{\ell+1}; \sigma_i^k$ and construct the global state $\Sigma = (\sigma_i^k, \sigma_j^{\ell+1})$. If only one of these two global states satisfies ϕ, then one stub will detect ϕ and the other will not. We can avoid this anomalous behavior by observing the following condition:

If stubs p_0 and p_1 subscribe to sensors σ_i and σ_j, then they receive values from these sensors in a total order consistent with Condition 1. (2)

It would appear that Meta could meet Condition 1 by using the Isis cbcast protocol and Condition 2 by using the more expensive Isis abcast protocol. Ideally, Meta could use cbcast to disseminate sensor values to nonreplicated stubs and abcast to disseminate sensor values to replicated stubs (discussed below). Unfortunately, the current version of Isis (V2.2) does not order messages sent using cbcast with messages sent using abcast. Furthermore, if the application uses cbcast for communication and Meta uses abcast to disseminate sensor values, then Condition 1 is not met[5]. Another Isis protocol, gbcast, does provide consistent ordering to multiple destinations while respecting the causality of messages sent via cbcast, but its cost is prohibitive and would still only guarantee Σ were consistent if the application program used Isis-based protocols. So, Meta currently uses abcast for all sensor dissemination. Doing so guarantees that the constructed global state is consistent as long as sensor dissemination is relatively faster or more frequent than the interprocess communication among the instrumented components.

This fairly weak condition has proven sufficient for distributed application management [MCWB90] since the properties tested have a long persistence. For other applications, such as tool integration and debugging, Meta will have to supply stronger guarantees about the consistency of Σ. There are several techniques that could be used. The most basic, and probably the best, approach would have the underlying communication facility carry message ordering information, e.g. [PBS89]. If all interprocess communication were built on this facility, then Condition 1 could be implemented. A short-term solution is to use a separate event ordering protocol like those

[5] Version 3.0 of ISIS does order messages sent using cbcast with messages sent using abcast, and also delivers messages much more quickly than Version 2.2. Unfortunately, the communication pattern must fit into a very restricted form that Meta does not currently adhere to. We are investigating changing Meta to take advantage of Version 3.0, but we have concerns about the limited communication pattern.

described in [Spe89]. These protocols, however, can generate many additional messages: if a stub subscribes to n other stubs, then each new sensor value can generate up to $2n$ messages. We are currently investigating the practicality of these protocols.

5.1.3 Reaction

Unless the predicate ϕ_i is stable [CL85] or the application can be blocked, it is impossible to guarantee that the action α_i is performed in the same global state in which ϕ_i was detected. For example, consider the situation shown in Figure 2. Unless component p blocks after sending out the value of sensor x, there is no guarantee that p will not change x by the time that actuator a is invoked.

There is a deeper issue, however: at best, Meta constructs consistent cuts S^* in which to evaluate ϕ_i, yet there is no guarantee that the state S^* ever existed. If Meta cannot guarantee that the state that satisfied ϕ_i ever existed, then it cannot guarantee that the action α_i took place in that state.

We have examined several different semantics for detection and reaction and have developed protocols that support these semantics [MN91]. The strongest semantics is that Meta detects ϕ_i if and only if ϕ_i held in some realizable execution of the instrumented system, and the system blocks long enough to guarantee that the reaction occurs in a state satisfying ϕ_i. Weaker semantics include detecting ϕ_i if and only if ϕ_i held in any realizable execution of the system but the reaction occurs in a later state. The weakest semantics is that if Meta finds a consistent cut in which ϕ_i held—either by chance or because ϕ_i held in any realizable execution of the system—then the reaction occurs in a later state. In all cases, any sensors referenced in α_i are given values from a state in which ϕ_i held. All but the last protocols are expensive and are not practical unless communication delays can be bounded.

The current implementation of Meta uses the weakest semantics. For the problems we have considered in distributed application management, these semantics have been sufficient. For example, consider a guarded command that migrates a waiting job from a busy server to an idle server. This command can be run in the busy server's stub, guaranteeing that the job is idle when migrated, and aggregates (described below) can be used to guarantee that one of a set of idle server atomically gets the job.

Nonetheless, not all reactive systems can use the weakest semantics. We plan to incorporate the protocols of [MN91] when we have a version of Isis that runs on a real-time system.

5.1.4 Atomicity

Unlike detection and reaction, it is relatively easy to supply atomicity when there are no failures. If the action consists of a single actuator call to several sites, then atomicity is supplied by using atomic broadcast. If the action consists of more than one actuator call, then an atomic commit protocol is used.

In the current implementation of Meta, however, failures may lead to nonatomic behavior. For example, consider the guarded command $\langle p.x < q.y \rightarrow p.a; q.b \rangle$. If the actuator call $q.b$ fails (perhaps because q has crashed), then the effect of $p.a$ would have to be undone (*backwards recovery*) or p would have to be crashed (*forward recovery*) in order to guarantee atomicity. Instead, Meta would allow the result of $p.a$ to become visible.

While it would be relatively easy to incorporate backwards or forward recovery into Meta, we believe that more experience is needed to understand when each mechanism should be used. We hope to gain this experience with further application of Meta.

5.2 Aggregation

This section discusses a number of additional issues that arise when implementing aggregates.

5.2.1 Membership

Joining and leaving an aggregate is implemented by joining and leaving an appropriately named Isis process group. We use this mechanism because it provides a ready-made membership service, and while the current Isis process group mechanism does not scale to groups containing a large number of machines, support for large process groups should be available in a future version of Isis.

One advantage of using Isis groups is that a stub implementing an aggregate context will be informed of membership changes, and this notification will be causally ordered with respect to any messages sent by the members. Consequently, the stub will never receive a value from a member believed to have departed. Moreover, since the stub is informed when a new member joins the group, it can immediately subscribe to the new member's sensors as needed to implement the corresponding aggregate sensors.

5.2.2 Sensing

Aggregate sensors raise an additional consistency problem. Consider an aggregate of compilation servers and a sensor that is defined to be the maximum load of the aggregate's members. A guarded command may reference both an individual server's load and the maximum load. A change in the server's load can change the maximum load, and the guarded command should see these two changes atomically. Meta does not currently enforce this condition: the guarded command may be evaluated in a state where the individual load is larger than the maximum load. Though we have not yet needed such consistency, we can envision its necessity and are considering mechanisms for supplying it.

References to aggregate sensors are by their nature expensive, and so should be used carefully. To improve locality, as much filtering as possible should be done by the aggregate members. For example, suppose a client is interested in knowing when the server aggregate is moderately loaded. A simple approach would be to have a guarded command that is enabled when the average load goes above some constant value L. Unfortunately, every change in any member's load will cause a message to be sent to the aggregate's stub, which will then recompute the average and test the condition. A more efficient approach would be to test for the median load to go above L. In this case, the aggregate can define, in each member's stub, a Boolean sensor that is true when that member's load is above L. A message is sent to the aggregate's stub only when the load goes above or below L, and the guarded command is enabled when a majority of these sensors are true.

5.2.3 Actuation

Invoking an aggregate actuator causes all members of the aggregate to execute the action, and if any individual actuator fails, then the aggregate actuator also fails. More fault-tolerant actuation can be accomplished by supplying two additional parameters: m, the number of members to do the action, and a preference list. This list specifies a total order from which members are picked to perform the action, and need not contain all of the aggregate members. The composite action proceeds by invoking the actuator on the first m elements of the preference list. If any of these should fail or return *failure*, then additional members are chosen from the preference list. The composite action fails if the preference list is exhausted before m aggregate members successfully complete the actuation.

An aggregate actuator is implemented with the Isis *coordinator-cohort* facility. The action and preference list is atomically broadcast to all members in the preference list. The first m members of the list are coordinators and perform the action, while the remaining members are cohorts and monitor the coordinators. If a coordinator fails before broadcasting that it has completed the action, then the first cohort takes over. Again, there is an atomicity issue when $m > 1$ and only $m' < m$ members successfully complete the actuation. The current implementation will allow the surviving m' members to continue without recovery from the actuation.

5.2.4 Stub Failure

An instrumented program and its Meta stub fail as a unit, so there is no benefit in making the stub resilient to crash failures. A stub that implements an aggregate context, however, may crash while the members of the aggregate remain operational. Replicating the stub removes this single point of failure.

An aggregate context is replicated by endowing multiple stubs with the same context. All such stubs join an Isis group, with the oldest member serving as the primary and the others serving as backups. The backups must have sufficient state information so that any one of them can take over should the primary fail.

The current implementation of replicated stubs is simple. The following program is called a *Meta server*. Its only purpose is to create a stub that can later be instructed to implement an aggregate context.

```
#include "meta.h"
main()
{   meta_stub_init("meta_server", gethostbyname(), 0);
    meta_main_loop; }
```

This stub has an empty base context, so all sensor values are referenced through subscription. When replicated, all Meta servers will receive sensor values in the same order. The stubs will also receive all client operations in the same order, since such operations are also atomically broadcast to the aggregate stub group.

The primary acts like any other stub, except that it includes the backups as recipients of all actuations. A backup maintains its sensor cache, and since the values are sent using atomic broadcast, when a backup detects that the primary has failed, its sensor cache is in the same state as the primary's when it failed. A backup uses the actuation messages sent by the primary to track the progress of evaluation of a guarded command. In particular, a backup can determine when to discard a guarded command because it has been executed and how far in the actuation sequence the primary has reached when it fails.

When the primary fails, the next oldest member of the process group becomes the new primary (since all processes see the same sequence of process joins and leaves of a process group, each backup can determine locally which is the new primary). The new primary starts where the old primary left off; it completes any ongoing actuation sequence, evaluates the other outstanding guarded commands, and continues.

This scheme is simple, but all referenced sensor values are remote and it relies on the more expensive abcast protocol. We have designed another primary-backup scheme that uses the cheaper cbcast protocol and can reference local sensors (and hence need not run on a Meta server), but it has not yet been incorporated into Meta.

6 Performance

The following performance figures were obtained by running Meta on Sun 4/60's with interprocess communication handled by ISIS over a 10 Mbps Ethernet.

The time to execute a simple guarded command of the form A GUARD B with trivial local sensor A and trivial local actuator B is 84.1 ± 0.1 microseconds (at a confidence of 0.95). This implies approximately 12,000 simple guarded commands can be executed a second.

The bulk of the time for remote actions is, of course, in the message delivery. Since Meta relies upon ISIS for message delivery, the ISIS architecture largely determines Meta's performance. The ISIS architecture is currently being redesigned, and we expect to benefit from some of the new developments. Currently, however, communication is hindered by having all messages funneled through *protocol servers* running on each machine. A process sends an ISIS message via local IPC to the protocol server running on the same machine. This protocol server communicates with the other protocol servers in order to maintain the ordering and delivery constraints. The protocol server on the destination machine then delivers the message to the destination process. Although the cbcast protocol in theory has a lower message complexity than the abcast protocol, the overhead imposed by channeling messages through protocol servers significantly reduces the benefit of using the cbcast over abcast.

The version of ISIS now under development allows for restricted types of broadcasts to be sent directly to the intended recipients. This results in considerable savings; a cbcast of this form costs approximately 5.7 ± 0.2 milliseconds compared to 14.4 ± 0.2 milliseconds in nonbypass mode. However, the new mode of communication requires the sender and receivers to be members of the same group, which is not typically the case in Meta. Meta *does* put members of aggregates in the same group, which will allow us to use the new mode of communication. However, unless Isis group operations (joining, leaving and changes in group communication patterns) become significantly cheaper, we don't see much promise in adapting Meta further.

Running the previous simple guarded command at a remote interpreter takes 32.6 ± 0.6 milliseconds. This figure includes one cbcast to the interpreter that reports the value and a cbcast from the interpreter to effect the actuation. Using abcast does not effect performance in the case where there is only one destination, and only increases the time by about five percent when there are two subscribers.

Aggregate-based operations show a difference in performance between using cbcast and abcast. A Meta server was endowed with an aggregate context containing two members and all three processes were run on separate machines. The guarded command A GUARD B was rewritten so that the guard referenced sensors from both aggregate members and actions were performed on both aggregate members. The time for both members to send their values to the stub plus the time for the stub to invoke actuators on both was measured to be 48.6 ± 0.2 milliseconds using cbcast for all communication, and 55.4 ± 0.3 milliseconds using abcast for all communication. Note that choosing the abcast protocol slows down only the actuation, as that is the only broadcast to a group. In using the new version of Isis, we anticipate a considerable speedup in aggregate-based communication. Furthermore, using the new mode of communication should significantly underscore the differences between using cbcast and abcast protocols.

Running a guarded command has a start-up and a tear-down cost. In order to measure an upper bound on this time, we measured the time needed to do the following sequence operations: send to the local interpreter a guarded command that is immediately executed and causes the interpreter to subscribe to a remote sensor, get the first value of the sensor, and cancel the subscription. This time was measured to be 72.0 ± 0.6 milliseconds.

This value includes the time to parse the guarded command, but the cost of this should be negligible (less than one percent). Note that the guarded command is sent locally via cbcast rather than via a (faster) direct procedure call because we wish to support replication of interpreters. This cbcast results in communication with the local Isis protocol server.

7 Discussion

7.1 Related Work

Although much work has been done on system monitoring, our work differs in that it combines monitoring with control in order to provide support for constructing reactive systems. A prominent example of a system designed strictly for monitoring is the work of Snodgrass [Sno88] in which the system state is cast as a temporal database. One could imagine implementing control by using database triggers, but doing so would most likely be overkill. Databases are designed to maintain large amounts of instance data, thereby allowing complex queries to be formulated. Instead, Meta is careful on what data it sends and maintains in order to keep the overhead small.

Systems for debugging, especially those for debugging distributed systems, are a specialization of general monitoring systems. These systems provide a way to access the system state and to watch for certain predicates to be satisfied through the use of breakpoints [MH89,Bat88]. Of particular interest is the system IDD [HHK85] that permits interval logic expressions in specifying breakpoints.

7.2 Status and Further Directions

At the writing of this paper (February 1991), Meta supports the function described here but has not yet been thoroughly tested or tuned. In particular, there remain some open implementation details about replicated aggregate stubs and the use of preference lists in aggregate actuation. We have a simple suite of tests for Meta, and are currently developing a demonstration program. This program consists of a model server that can be replicated and have jobs migrated using Meta commands. The demonstration program also contains a graphical service monitor that obtains the system information through Meta sensors. This version of Meta, complete with the test suite and demonstration program, will be available in the next release of Meta planned for August 1991. An earlier release of Meta, planned for April

1991, will contain the function described in this paper except for replicated aggregate stubs, as this code currently exists only in prototype form.

Previous released versions of Meta did not support the complete NPL language, but instead had the notion of a `watch`, in which a Meta stub could be instructed to wait for the value of some sensor to satisfy some relation. This earlier work has emphasized the benefit of locality of detection and reaction.

Throughout the paper, we have mentioned several directions that we are currently pursuing. These are: addition of a recovery mechanism, implementation of a set of detection and reaction protocols that supply stronger semantics, supplying atomic detection of sensors with aggregate functions over those sensors, and a more efficient primary-backup algorithm for replicated stubs. The most critical feature is a sensor ordering protocol that will guarantee Meta constructs only consistent cuts. We hope to include this feature in the June release.

The current Meta toolkit is adequate for use in systems in which timing is not crucial. Although guarded commands can make temporal assertions, given the potentially unbounded latencies in the underlying UNIX and Isis platforms, such assertions can only be viewed as approximate upper bounds. However, the structure that Meta provides is general enough that we should be able to extend it to real-time reactive systems as well.

There are two main obstacles we see to extending Meta to real-time systems. The first has to do with the underlying Isis toolkit; to guarantee bounded reaction time, the underlying causal broadcast and group membership protocols must provide some real-time guarantees. A companion project in the ISIS group is currently looking into structuring Isis under Mach to provide these two protocols. The second obstacle has to do with the semantics of guarded commands. Guarded commands currently have the semantics of atomic actions; if a guarded command is continuously enabled, then it will eventually execute. We need to include an upper bound on how long the command can be enabled without executing, and then build a scheduler that either guarantees the command will be executed within its deadline or aborts the command if it cannot be executed within its deadline.

Acknowledgements Several people have contributed to the Meta project. Nancy Thoman designed and wrote the first version of the guarded command language, and Wanda Chiu designed a reactive relational database that provided a testbed for earlier versions of Meta. Kenneth Birman and Robert

Cooper have contributed much to the design of the overall system. We would also like to thank Robert Cooper, Aleta Ricciardi, and Laura Sabel for their helpful comments on earlier drafts of this paper.

References

[Bat88] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*. ACM, 1988.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 123–138. ACM SIGOPS, 1987.

[BJKS88] Kenneth P. Birman, Thomas A. Joseph, Kenneth Kane, and Frank Schmuck. *ISIS — A Distributed Programming Environment User's Guide and Reference Manual*. Department of Computer Science, Cornell University, second edition, June 1988.

[BN84] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison Wesley (Reading, Mass.), 1988.

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194. ACM, 1988.

[GI90] David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environments. In *Proceedings of the Fourth Symposium on Software Development Environments*, pages 1–10. ACM SIGSOFT, 1990.

[HHK85] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: An interactive distributed debugger. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 498–506, 1985.

[HP85] David Harel and Amir Pnueli. *On the Development of Reactive Systems*, pages 477–498. Springer-Verlag, New York, 1985.

[MC88] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 316–323. IEEE, 1988.

[MCWB90] Keith Marzullo, Robert Cooper, Mark Wood, and Kenneth P. Birman. Tools for distributed application management. Technical Report TR 90-1136, Cornell University, June 1990. Submitted for publication.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4), December 1989.

[MN91] Keith Marzullo and Gil Neiger. Detection of global state predicates. Extended abstract submitted for presentation, February 1991.

[PBS89] L. Peterson, N. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the FIELD program development environment. *IEEE Software*, 7(4), July 1990.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[Spe89] Madelene Spezialetti. *A Generalized Approach to Monitoring Distributed Computations for Event Occurences.* PhD thesis, University of Pittsburgh, 1989.

[Tay89] Kim Taylor. The role of inhibition in asynchronous consistent–cut protocols. In J.-C. Bermond and M. Raynal, editors, *Proceedings of the Third International Workshop on Distributed Algorithms, Nice, France, September 1989*, volume 392 of *Lecture Notes on Computer Science*, pages 280–291. Springer-Verlag, 1989.