Making Real-Time Reactive Systems Reliable*

Keith Marzullo
marzullo@cs.cornell.edu

Mark Wood
wood@cs.cornell.edu

Cornell University
Department of Computer Science
Ithaca, New York 14853
March 30, 1990

A *reactive system* [3] is characterized by a *control program* that interacts with an *environment* (or *controlled program*). The control program monitors the environment and reacts to significant events by sending commands to the environment. This structure is quite general. Not only are most embedded real-time systems reactive systems, but so are monitoring and debugging systems and distributed application management systems.

Since reactive systems are usually long-running and may control physical equipment, fault-tolerance is vital. Our research tries to understand the principal issues of fault-tolerance in real-time reactive systems and to build tools that allow a programmer to design reliable, real-time reactive systems.

A reactive system has a structure much like that of a client-server system (see Figure 1). Both systems contain components that are input-driven—servers are input-driven by the clients, and the control program is input-driven by its environment. This is an important similarity, since there exist techniques for making deterministic servers fault-tolerant [1,7]. There are, however, two essential differences between reactive systems and client-server systems:

1. Whereas a client explicitly invokes a server, the environment does not explicitly invoke the control program. Instead, the environment is in-

*This position paper was read at the Fourth European SIGOPS Workshop, September 3–5 1990, Bologna, Italy. 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.

strumented with *sensors* and *actuators* that allow the control program to read and change the state of the environment.

2. Since the environment does not explicitly invoke the control program, any synchronization between these programs must be done in ways other than by using rendezvous. One technique is to express the system specification in terms of *time*; that is, to set hard real-time constraints on the execution of the control program. We call a reactive system with real-time constraints a *real-time reactive system.*

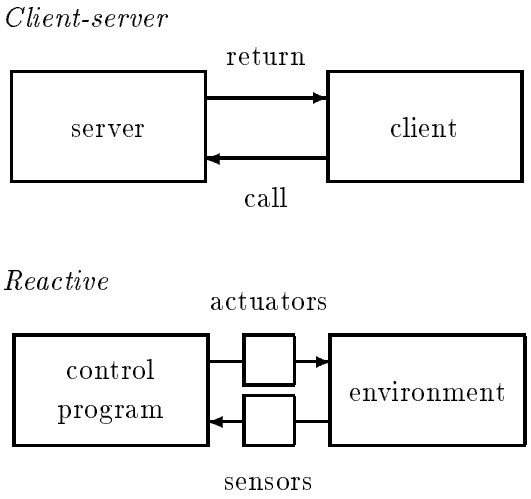

Figure 1: System Structures

In order to make real-time reactive systems reliable, several issues must be addressed:

- How can a control program be built to tolerate failures of sensors and actuators? To achieve this, we have developed a methodology for transforming a control program that references physical values into one that tolerates sensors that can fail and can return inaccurate values.
- How can the real-time reactive system be built to tolerate failures of the control program? Towards this goal, we are investigating whether the techniques presented in [1,7] can be extended to real-time reactive systems.

- How can the environment be specified in a way that is useful for writing a control program? Towards this goal, we are investigating whether a system with real-time constraints can be expressed as an equivalent system without such constraints.

Meta

In the *Meta* project [6,2] we have been developing an architecture based on sensors and actuators that supports the development of reliable reactive systems. This architecture is very general—for example, it supports management of distributed applications as well as process control systems. The major issues in reactive systems for distributed application management are instrumentation of the application, representation of the application, and efficient and fault-tolerant monitoring of the application.

The functional architecture of Meta is shown in Figure 2. A programmer follows two steps using this to develop the control program of a distributed application. First, the programmer instruments the application and its runtime environment with sensors and actuators. A set of sensors and actuators are provided by default, including sensors on machine load, allocated resources, and the global variables of running programs. Second, the programmer describes the application using an object-oriented data model and writes the control program referencing this data model. The control program can both make direct calls on the data model and register a set of *policy rules* that Meta will monitor and enforce.

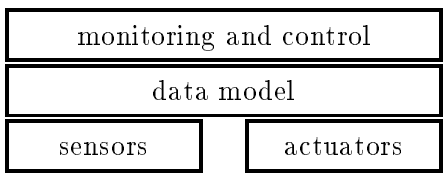

Figure 2: *Meta* Functional Architecture

Sensors are functions that return typed values of the application state or the environment state. Sensors may be *polled* for the current value of the function, and a *watch* may be set on a sensor, instructing Meta to notify the client when the sensor value satisfies some user-defined predicate. Sensors and actuators are implemented by stubs that run on machines supporting the

application. In order to instrument a process with a new sensor or actuator, the process being instrumented registers a procedure with the sensor stub. The stub is responsible for responding to poll requests and for periodically checking to see if any outstanding watches are satisfied. Additionally, an instrumented process can notify the stub that a specified sensor's value has changed, allowing the stub to reevaluate the affected outstanding watches.

Once instrumented, the programmer specifies the structure of the application by using an object-oriented data modeling language called *Lomita*. Through Lomita, the application is cast as a temporal object-oriented database. Entity and relationship sets have attributes that fall into one of three categories:

1. *Properties*, whose values are static for a given entity. For example, the type of a machine is a property. The attributes comprising a key must be properties.

2. *Sensors*, whose values are dynamic for a given entity. A sensor attribute can be defined as one of the sensors supplied by the application, or its value can be derived from the other sensors.

3. *Actuators*, which are invoked on an entity. An actuator attribute is defined as one of the actuators supplied by the application.

The intended behavior of the control program is described as a set of Lomita rules. A rule states the action to be performed when a specified condition of the application is observed. Conditions are expressed over sensor values using a real-time extension of temporal interval logic [8]. For example, a simple rule might be that if the number of processes comprising a replicated service is too low or the average service load is too high, then start a new process on the lightest-loaded machine not yet running the service. Lomita will redundantly monitor this condition and guarantee that exactly one version of the rule will react when the condition is satisfied.

We have implemented a version of Meta that runs on UNIX and have used it to control some simple applications. We are now developing Lomita and are applying Meta to more complex applications, such as a reliable version of parallel make, a seismological monitoring application, and a distributed configuration management system. Meta is built on top of the Isis distributed toolkit [2], making many of the issues of fault-tolerance and agreement simple to address.

We are also extending Meta for process control applications. This is a significant extension, since the control program is monitoring and controlling

a physical system. The next section addresses one of the issues raised by this new setting.

Making Sensors Reliable

One of the fundamental issues of making real-time reactive systems reliable is how sensor failures can be masked [4,5]. We have developed a methodology for transforming a control program that references physical values into one that tolerates sensors that can fail and can return inaccurate values. Our methodology is as follows:

1. A specification of the control program is written in terms of the state variables of the physical system. For example, the specification of a program controlling a chemical reaction vessel would refer to a variable T whose value is assumed to be the temperature of the vessel.

2. Each physical state variable referenced by the specification is replaced with a reference to an *abstract sensor*. An abstract sensor provides a set of values (such as an interval) that contains the physical variable of interest. At this step, uncertainty in sensor values becomes an issue, and the specification must be re-examined and possibly changed to accommodate this uncertainty. Ideally, the specification should be strengthened, but in some cases this may not be possible.

3. A control program is written based on the specification produced by Step 2. This program references abstract sensors that are assumed to always contain the correct value of the physical variables.

4. For each abstract sensor referenced by the program written in Step 3, a set of abstract sensors are written such that they fail independently. Each abstract sensor is implemented using a *concrete sensor*, which is a physical device that "reads" a physical variable, such as a thermometer or a pressure gauge. This step will require some knowledge of the physical process being controlled as well as the specification of the concrete sensor.

5. A *fault–tolerant averaging algorithm* is used with these replicated abstract sensor values in order to calculate another abstract sensor that is correct even if some of the original sensors are incorrect. The averaging algorithm assumes that no more than f out of the n abstract

sensors are incorrect, where f is a parameter. The relation between n and f (outside of $0 \leq f < n$) depends on the way sensors can fail.

Note that programs written in Lomita reference abstract sensors, and Meta provides the mechanisms to implement abstract sensors from concrete sensors assuming a simple failure model. The programmer need implement abstract sensors only for specialized applications; for most applications the Meta system handles Steps 4 and 5.

Several issues with our approach remain open. One interesting problem is that of accommodating abstract sensors and actuators in process control problem specifications. In Step 2 of the methodology, the developer needs to examine the impact of uncertainty on the specification. All process control systems must accommodate such uncertainty; our methodology, however, makes this problem explicitly part of the refinement of the problem. We are now looking at ways to express and reason about such uncertainty in process control systems.

Acknowledgements Ken Birman and Robert Cooper have actively contributed to the design of Meta. Our research into the specification of real-time systems is in collaboration with Fred Schneider.

References

[1] Jacob I. Aizikowitz. *Designing Distributed Services using Refinement Mappings.* PhD thesis, Cornell University, Department of Computer Science, January 1990.

[2] Kenneth Birman and Keith Marzullo. ISIS and the Meta project. *Sun Technology*, 2(3):90–104, Summer 1989.

[3] D. Harel and A. Pneuli. *On the Development of Reactive Systems*, pages 477–498. Springer-Verlag, New York, 1985.

[4] Keith Marzullo. Tolerating failures of continuous–valued sensors. Technical Report TR 90-1156, Cornell University, September 1990.

[5] Keith Marzullo and Paul Chew. Efficient algorithms for masking sensor failures. In preparation, 1990.

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

[7] Fred B. Schneider. The state machine approach: A tutorial. *Computing Surveys*, 22(3), September 1990.

[8] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher–level temporal reasoning. In *Proceedings of the Second Symposium on Principles of Distributed Computing*, pages 173–186. ACM SIGPLAN/SIGOPS, 1983.