Event-Driven Architectures for Distributed Crisis Management

K. Mani Chandy, Brian Emre Aydemir, Elliott Michael Karpilovsky and Daniel M. Zimmerman
Computer Science 256-80
California Institute of Technology
Pasadena, California, USA
{*mani, emre, elliottk, dmz*}*@cs.caltech.edu*

ABSTRACT
This paper describes an approach for developing distributed applications that help deal with rapidly changing situations such as terrorist attacks, hurricanes and supply chain disruptions. Important characteristics of such applications are that they must handle unexpected events and that they are often modified on-the-fly, by multiple people who may belong to different organizations, to deal with changing situations.

Abstract and concrete models for specifying, reasoning about, and implementing such systems are presented. In the abstract model, an application is a set of state transition rules over the global state of a distributed system. In the concrete model, an application is a set of message-driven processes and each computation is a sequence of atomic operations in which a process receives a message, changes its state, and sends messages. An implementation of the concrete model using XML as the message and state format, with state transitions specified using XSLT, is briefly described. A key feature of this implementation is that messages and process states are represented using a format that allows applications to be easily observed and modified during their execution.

KEY WORDS
distributed software systems and applications, information systems, distributed agents, crisis management

1 Introduction

Task forces that manage very dynamic situations, such as crises, are fluid collections of individuals and institutions. For example, the task force handling the aftermath of an airplane crash may include the FBI, local police, the Red Cross, the airline, news services and relatives of passengers. Often, the participants in a task force cannot be determined until the crisis occurs. Likewise, the specific components required for a given application needed by the task force may not be predictable and may change as a crisis unfolds. Events that occur in rapidly-changing situations are often unanticipated. Therefore, processes for dealing with such events may need to be created and modified as the situation unfolds.

We first describe some characteristics of applications that support the activities of such task forces, which we call *dynamic applications*, and argue that the differences between dynamic and conventional applications merit using a different programming model and implementation technology for dynamic applications. We then propose both an abstract programming model and a concrete programming model for such applications. Finally, we give a brief overview of our current system implementation and compare our model to some related architectures and concepts.

2 Characteristics of Dynamic Applications

Several characteristics differentiate dynamic applications from conventional applications. In this section we briefly describe the most significant of these characteristics, addressing first those associated with the computational structure of dynamic applications and then those associated with their design and execution.

2.1 Computational Structure

Many conventional distributed applications use a service oriented architecture in which a client requests a service from a server, gets the response from the server and then makes requests to other servers following a sequence of steps often modeled using workflow or business process modeling. In some cases, each server in a business process carries out one step and passes results on to the next server in a predefined flow.

Such request-response protocols are well suited for conventional applications, but are inadequate for dynamic applications. The state of a crisis evolves unpredictably, and task forces typically do not have central authorities that determine how events should be processed. For this reason, it is not possible to specify, implement and test flow graphs for handling events pertinent to a particular situation before it occurs.

To properly deal with rapid deployment and evolution, processes in a dynamic application should be notified automatically when events of interest occur; the processing of these events may, in turn, generate new events. An execution trace of such a dynamic system can produce a causality graph that shows how one event causes others. This graph emerges from the computation, unlike in conventional applications where the flow graph specifies the computation.

2.2 Application Development

Development of a dynamic application is a continuous activity. As a situation unfolds, new components may be required or existing components may need to be modified while the system is executing. Conventional application development, on the other hand, employs rigorous steps with careful testing procedures and sequences of planned releases, and each release consists of a static collection of components. The continuous development process makes runtime monitoring, testing, debugging and administration more important for dynamic applications than for conventional applications. Therefore, the programming model for dynamic applications should allow users to inspect and modify running components, a capability that is not as important for users of conventional applications.

The hardware and operating systems available for use by a dynamic application may not be known until the application is required to deal with an emerging situation. Therefore, the components in a dynamic application should be specified in a notation that can be mapped to a variety of runtime environments. By contrast, releases of conventional applications have the luxury of specifying particular versions of hardware architectures, operating systems, and other supporting software components.

3 Abstract Model

In this section we present an abstract model for the specification and execution of dynamic applications. The abstract model captures the central aspect of dynamic applications: the ability to respond to conditions over global state. Detecting meaningful global snapshots of distributed systems is difficult [2], and the abstract model helps to separate concerns about what conditions need to be detected from details about how the detection occurs.

A dynamic application consists of a set of *variables* and a set of ***when-then** rules*. The **when-then** rules are the sole means of specifying the application's behavior.

A variable can be either *external* or *internal*, and must have a unique name. There is a special variable called *time*, which is neither internal nor external; its value changes continuously to accurately reflect the progress of time in the physical world.

External variables represent current values of real-world quantities, such as the temperature at Los Angeles International Airport or the Dow Jones Industrial Average. Each external variable is associated with a sensor; if multiple sensors report the same real-world quantity, such as temperature sensors located on different runways of the airport, each sensor is associated with a different external variable.

Internal variables are the results of computations within the application, and their values are based on the values of other (external and internal) variables. An example of an internal variable is the five-day moving-point average reading of a particular temperature sensor at Los Angeles International Airport.

The complete set of variables and their values at any point of an execution are collectively called the *state* of the application at that point. The sequence of all prior application states at any point in the execution is called its *history* at that point.

The set of variables can change during the execution of an application. For example, at some point during the execution, an external variable reflecting the volume of blood plasma available at a specific hospital might exist. At a later point, a new external variable such as the number of patients with typhoid admitted to the same hospital could be added. Still later, a new internal variable that quantifies whether the volume of plasma at the hospital is dangerously low given the history of patient admittances could be computed.

All changes to the values of internal and external variables are discrete. For instance, even though the temperature of a boiler may change continuously over time, the temperature variable changes only in discrete increments based on updates from a temperature sensor. A change in the value of a variable is called an *event*.

The execution of the application is governed by a set of **when-then** rules. Each of these rules has the form, "**when** history-predicate **then** state-change-specification." Similar to the set of variables, the set of **when-then** rules can change during the execution of an application.

The history-predicate is a Boolean condition on the history of the dynamic application, such as, "the one-week moving-point average of blood plasma usage in Los Angeles County exceeds three-quarters of the one-week moving-point average of blood plasma inventory available in Los Angeles County." The **when** clause can specify an arbitrary predicate on the history of (global) states. For instance, the predicate can refer to the blood supplies of all hospitals around the world. The abstract model does not deal with how the predicate is evaluated instantaneously.

The state-change-specification defines changes to the internal state variables. For instance, the **then** part of the rule may specify that an alert message be sent to the public health director for Los Angeles County; in this case, the variable is the sequence of messages sent to the public health director and the state change is to append a message to this sequence.

A **when-then** rule is executed as follows. The predicate in the **when** clause is evaluated. If it is *false*, nothing further happens. If it is *true*, the state variables are changed as specified in the corresponding **then** clause. We require that the **then** clauses of all rules with **when** clauses that are simultaneously *true* do not interfere with each other; for example, it is not allowed for one **then** clause to set a Boolean variable to *true* while another clause simultaneously sets the same variable to *false*.

The execution of a dynamic application is event-driven. Whenever an event occurs, all the **when-then** rules in the application are executed concurrently and instantaneously. This may cause a number of other events to occur, as internal variables are changed by the rule executions. As a result of the concurrent and instantaneous execution of rules, all computations in the abstract model are deterministic. Clearly, this type of execution cannot be implemented using real machines.

4 Concrete Model

In this section, we describe a concrete model that approximates applications described with the abstract model as implementable, message-based distributed systems. The concrete model hides implementation details to simplify reasoning about system behavior.

An implementation cannot be faithful to the abstract model because the **when-then** rules cannot all be executed concurrently and instantaneously: time is required to evaluate the predicates on, and make changes to, the global state. The concrete model assumes neither the instantaneous execution of rules nor simultaneous access to the entire global state. It is possible for an implementation to be faithful to the concrete model.

Applications in the concrete model, like those in the abstract model, are formulated with **when-then** rules; however, in the concrete model, the **when** clauses and the **then** clauses name only the local variables of a process. Unlike the abstract model, the concrete model allows nondeterministic computations.

Starting a design with the abstract model allows designers to restrict attention to internal and external variables and rule sets. At a later design step, designers develop a concrete model that approximates the execution of instantaneous global **when-then** rules by specifying processes, messages and communication. At this design step, designers focus their attention on mapping global **when-then** rules to local process operations.

4.1 Sensors, Actuators, and Event Processors

A dynamic application system consists of three types of component: sensors, which generate streams of events based on properties external to the application; actuators, which listen to streams of events and perform actions external to the application; and event processors, which listen to streams of events and generate new event streams of their own. A sensor that monitors the state of a boiler may generate events containing the temperature and pressure of the boiler. An actuator associated with a furnace may receive events that instruct it to change the rate of gas flow and actually perform that action in the physical world. An event processor might receive events about the temperature of a boiler and calculate a running average, generating events when the running average exceeds a particular threshold.

Sensors, actuators and event processors are collectively called *processes*. Processes do not share state, and interact only by receiving and generating events. In distributed systems terms, an event is a message and an event stream is the sequence of messages sent along a channel. The streams of events sent by sensors and event processors correspond to changing values of state variables in the abstract model.

4.2 Process Structure

The different types of process in a dynamic application are described in different ways. We do not discuss the sensors or actuators in detail, because they are entirely dependent on factors external to the dynamic application; a temperature sensor, for example, may be a piece of hardware that varies the voltage on its output with the temperature it senses, while an actuator may be a mechanical device that opens or closes a particular valve in a network of pipes. For the purposes of this discussion we concentrate on the event processors, as they are the components that are under the direct control of users of the distributed application; we assume only that sensors will generate appropriate event streams, that actuators will consume appropriate event streams and that sensors and actuators have unique identifiers.

Each type (or class) of event processor in an application is specified by a set of variables, a *state transition function* that maps an incoming event and current state to a next state and an *event generation function* that maps an incoming event and current state to a sequence of generated events.

An event processor instance is specified by an event processor class, an initial state and a unique identifier. The initial state is specified when the instance is constructed. The unique identifier is used for communication with other event processors via an event dissemination mechanism. We do not give a detailed description of the event dissemination mechanism here; all that is important for the purposes of this discussion is that it is capable of communicating events between event processors using their unique identifiers as an addressing mechanism.

The operation of an event processor is as follows. It is inactive until it receives an event. When it receives an event, it changes its state according to its state transition function and generates a sequence of events specified by its event generation function.

An event processor can set a *timeout* value when it makes a state transition. If the event processor does not make another transition within a time duration equal to the timeout value, it receives a timeout message. If it does make a transition within the duration, the timeout is cancelled and it does not receive a timeout message.

Each event processor has a set of input ports and a single output port. Each input port has a unique name. A message (copy of an event) in the system is addressed to a specific port of a specific process. Each state of the process is associated with a nonempty set of input ports and an optional linear priority ordering of these ports. A message arriving on any port in the set causes a state transition, while a message that arrives on any other port does not cause a transition. When a message arrives on any port in the set, the process checks the ports in priority order for messages and accepts the message from the first nonempty port that it finds.

5 Implementation

Our implementation of a system faithful to the concrete model stores the states of event processors as XML documents, specifies their state transitions using XSLT and uses XML messages to encapsulate events. XML and XSLT were chosen for their portability: it is possible to implement a system capable of parsing XML documents and transforming them according to XSLT specifications using any one of several different programming languages and computing platforms, and all such implementations can interoperate as long as they are written to work with the same underlying communication infrastructure. We chose Java as the implementation language, using the Apache Xalan-Java XSLT engine [1] to process the state transitions.

The system implementation represents each event processor as a state-based automaton and is responsible for the execution of large numbers of these state-based automata in a single Java virtual machine. These automata communicate with each other, and with automata in other virtual machines, using an information dissemination network to distribute the previously mentioned XML messages. For the purposes of this discussion we assume that the dissemination network is capable of delivering messages from one automaton to another using some addressing scheme, but we do not describe it in detail; in particular, we do not specify any particular ordering for message delivery, or any other quality of service characteristics except for the fact that all messages sent are eventually delivered to the correct destinations.

The implementation performs several distinct tasks: it receives messages from and submits messages to the dissemination network; it determines the order in which messages will be delivered to automata; it implements the timeout mechanism, which allows each automaton to request a "wakeup call" message if no messages arrive for it from the dissemination network within a specified amount of real time; and it executes state transitions on the automata. In addition, it maintains enough information in persistent, stable storage (such as on a disk array) to ensure that the system can be restarted with minimal lost information in the event of a transient system failure.

Execution of the current implementation proceeds as follows. When a message arrives from the dissemination network for one of the automata in the system, or when a message is generated by a timeout condition, it is placed in a named queue corresponding to the automaton and port to which it is addressed. Automata in the system have multiple named input ports on which messages can be received, and can select in any given state the set of input ports they are listening to and the priority order, if any, assigned to those ports. When there are messages on one or more of the ports that an automaton is listening to, the system selects the message on the highest-priority port (or a random port, chosen fairly from the ports the automaton is listening to, if no priorities have been set). It then loads the automaton into memory, using its state from persistent storage, and executes the appropriate state transition of the automaton based on the type of the message. The state transition may change the state variables of the automaton and may also cause new messages to be generated. The system submits any generated messages to the dissemination network for delivery and saves the new state of the automaton back to persistent storage.

The execution of state transitions is carried out using an XSLT engine. The states of the automata are stored in an XML format in persistent storage, and each automaton has two XSLT documents associated with each of the message types it can receive. When a typed message triggers a transition, one XSLT document maps the incoming message and the current state to a new state, analogous to the state transition function in the concrete model, and the other maps the incoming message and the current state to a set of outgoing messages, analogous to the event generation function in the concrete model. The type of a message is a unique identifier used to select the correct XSLT document to use for that message and is not a type in the sense of type theory or the Java type system.

In order to perform all its tasks efficiently, the implementation is designed to operate with a large degree of concurrency. There are two pools of Java threads in the system; one thread pool is responsible for writing received messages to persistent storage and providing delivery notifications, and the other is responsible for executing the XSLT engine on appropriate documents to carry out automaton state transitions. The synchronization mechanisms used allow the simultaneous handling of multiple messages and state transitions, while ensuring that no automaton is placed in an inconsistent state.

6 Usage Example

Consider a rapidly changing situation such as the aftermath of an earthquake. The earthquake causes gas line disruptions as well as a fire near a lab in which scientists work with radioactive material. An officer decides to add a **when-then** rule to the abstract model, which states that if a fire comes within two miles of such a lab then a code red alert is sent to all fire fighting units within ten miles. The alert is to be sent every two minutes until the condition (fire within two miles) ceases to exist. After identifying a set of **when-then** rules, the officer develops a design using the concrete model. Next, we discuss how this is done.

A dynamic application system has two searchable directories. The first, a directory of event streams, serves essentially the same function as UDDI [13] for Web Services: it describes the semantics of the event streams available for use by the dynamic application and contains the unique identifiers of the generators for those streams. For example, the directory has enough information for a human to determine whether the string “China” refers to the country or to a type of porcelain. The second, a directory of event processor classes, contains enough information to allow users to determine whether existing event processors classes are suitable for their needs and, if so, to instantiate those classes appropriately.

The officer searches the event stream directory and finds one or more event streams that identify locations of fires. She finds an event class for sending periodic alerts to all mobile units within a specified distance of a location, as well as a class that sends alerts when an event (such as a fire or chemical spill) is within a specified distance of a location. She creates instances of both classes and connects the instances by feeding the stream of events generated by one instance into the other instance. She specifies the initial states of instances using an XML file, and connects the instances by invoking commands from a command line; she may instead use some other user interface, if available, to create the connections and specify the initial states.

If she doesn’t find the classes she needs, her next step is to create a new class. She creates the XML file that declares the variables and the XSLT files that specify the state transition and message generation functions for her new class; she may instead use an assistive user interface, if one is available, to automatically generate the files that correspond to her specifications. Once this is done, she inserts a description of the class into the class directory and constructs a specific instance of the class with the appropriate parameters and input event streams.

7 Related Work

The idea of starting with an abstract model based on rules (similar to **when-then** rules) and then refining the model to deal with processes and local variables appears in the UNITY formalism [3]. The abstract and concrete models proposed here are heavily influenced by stepwise refinement in UNITY. A dynamic application within the abstract model can be successively refined, where refinements organize **when-then** rules into groups that name variables that either are not modified in other groups or are modified only according to specified protocols. Each group of rules and its associated variables corresponds to a process in the concrete model.

The abstract model differs from UNITY in two significant ways. First, all rules are executed instantaneously and concurrently in the abstract model, whereas the actions in UNITY are executed sequentially in a weakly fair order. Second, the abstract model deals with time explicitly, whereas UNITY has no concept of time other than an ordering of actions.

Reasoning about a temporal logic model such as UNITY or TLA [11] is easier than reasoning about the concrete model proposed here. A UNITY program either satisfies its specification or it does not; there is no concept of inaccuracy, as there is between the abstract and concrete specifications of a dynamic application.

Theories and notations for software control systems consider time explicitly. The Computation and Control Language [10], developed by Klavins, is an elegant notation for control systems that uses UNITY as a basis. Giotto [8], a time-triggered language for embedded programming, deals explicitly with sensors, actuators and mode switches. A difference in emphasis between these works and the models and implementation described here is the focus on very dynamic applications with continuously changing processes.

Luckham proposed the Rapide™ event pattern language [12] for specifying event systems. The iQueue pervasive data composition framework [5], developed at IBM, also has an event specification language. We do not propose a programming language for specifying **when-then** rules or for defining events; our process composition is by name rather than by semantics. We assume that human beings search directories of event streams and event processes, understand the meanings of the results and compose them by naming the streams sent to each process.

Cooper and Marzullo proposed several algorithms for consistent detection of global predicates [6]. Chase and Garg showed that the problem of detecting a generalized global predicate (the **when** part of a **when-then** rule) in a distributed system is NP-complete, and proposed efficient algorithms for detection of specific classes of global predicates [4]. We have not explored using these algorithms for detecting **when** predicates because the predicates of interest in dynamic applications may hold for very short durations. For these applications, rapid and occasionally erroneous predicate detection is preferable to extremely slow but completely accurate detection.

Harel introduced *statecharts* [7], an extension of state machines for dealing with real time. Our state machine model handles time in a more restrictive way than statecharts, by allowing only the specification of timeouts.

A great deal of work has been carried out on command and control systems. Our work differs in that we explore on-the-fly modification of the system whereas conventional systems are statically configured for a particular situation. Research on sensor networks for military, environmental and security applications has been catalyzed by low power, very low cost sensors. Sensor networks deal with the detection of complex events by using fixed sets of sensors and processor types. We instead explore networks with changing sets of processor types and instances.

Sense and respond systems such as the iSpheres Halo™ [9] platform, which has a rich library of components for sensing, actuation, and complex event detection, can be used to address the same problems as dynamic applications. Our work on message-triggered state machines follows work on the iSpheres platform. A point of departure is that our states and state transitions are encoded in formats that facilitate implementations on multiple programming platforms and enable on-the-fly discovery and modification of processes.

8 Conclusion

The individuals and organizations that deal with crises and other rapidly changing situations can benefit significantly from flexible information systems, where applications can be frequently changed and rapidly deployed in a variety of runtime environments. The dynamic application models and implementation presented here can be used to create such systems.

The models we have presented use the structuring mechanism of **when-then** rules to allow the programming of dynamic applications without the need for a complex software engineering process. Our implementation is designed to be as independent of specific hardware and programming environments as possible by using cross-platform standards, with events and states specified as XML documents and state transitions specified with XSLT.

Our dynamic application models are inappropriate for most conventional applications, where the model of choice is a business process represented as some form of flow graph in which nodes represent requests for services and responses from servers. The abstract model of dynamic applications finesses the most important part of conventional applications—the flow of steps in a business process—to an atomic action in the **then** part of a rule. Similarly, the concrete model of dynamic applications does not represent business process flows, whether transactional or not. Business processes represented by causal sequences of steps may be deduced from traces of messages and states in our concrete model, but they are not the means of specification. This is a weakness of the models presented here. A model in which the event-driven aspects are captured by the **when** parts of rules and the service-oriented flow aspects are captured by the (possibly non-atomic) **then** parts of rules could be more useful for such applications.

We are continuing to explore models, tools, and methodologies for developing applications to deal with crises and other evolving situations.

References

[1] The Apache XML Project. Xalan-Java. `http://xml.apache.org/xalan-j/`.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining the global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison–Wesley Publishing Company, Reading, MA, 1988.

[4] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[5] Norman H. Cohen, Apratim Purakayastha, Luke Wong, and Danny L. Yeh. iQueue: A pervasive data composition framework. In *Proceedings of the Third International Conference on Mobile Data Management*, pages 146–153, January 2002.

[6] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, May 1991.

[7] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of EMSOFT 2001*, volume 2211 of *Lecture Notes in Computer Science*. Springer–Verlag, 2001.

[9] iSpheres Corporation. Halo™. `http://www.ispheres.com/`.

[10] Eric Klavins and Richard M. Murray. Distributed computation for cooperative control. Submitted to *IEEE Pervasive Computing*, 2003. `http://www.cds.caltech.edu/~murray/papers/2003m_km03-ieeepc.html`.

[11] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[12] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison–Wesley Publishing Company, Reading, MA, 2002.

[13] The Organization for the Advancement of Structured Information Standards. Universal Description, Discovery and Integration of Web Services. `http://www.uddi.org/`.