A Flexible, Extensible Simulation Environment for Testing Real-Time Specifications

Monica Brockmeyer, Farnam Jahanian, Constance Heitmeyer, and Elly Winner

Abstract—This paper describes MTSim, an extensible, customizable simulation platform for the Modechart toolset (MT). MTSim provides support for "plugging in" user-defined viewers useful in simulating system behavior in different ways, including application-specific ways. MTSim also supports full user participation in the generation of simulations by allowing users to inject events into the execution trace. Moreover, MTSim provides monitoring and assertion checking of execution traces and the invocation of user-specified handlers upon assertion violation. This paper also introduces an MTSim component called WebSim, a suite of simulation tools for MT, and an application-specific component of MTSim which displays the cockpit of an F-18 aircraft and which responds to user inputs to model a bomb release function.

Index Terms—Simulation, specification, symbolic execution, monitoring and assertion checking, formal methods.

1 Introduction

Several studies have shown that the cost of detecting and removing software errors increases significantly as the development process moves from requirements specification toward implementation [14]. In fact, the cost of removing an error from a system specification is often an order of magnitude smaller than the cost of removing it from a system that is undergoing integration testing. Other studies have demonstrated that errors in the requirements specification are the most frequent cause of software errors and the most expensive to correct [4].

Because the specifications of practical systems are usually very large, an integrated set of flexible, robust software tools is useful for specifying and analyzing the behavior of such systems [12]. Among the tools effective for testing and debugging specifications early in the design process are *simulators*, which allow the user to generate and examine symbolic executions of the system under development. Simulation based on a formal specification is especially useful because the simulated system behavior conforms to the formally specified behavior. If the user finds problems with the simulated behavior, these problems can be corrected by modifying the formally specified behavior. To ensure that the problems have been corrected, the simulator can be used to symbolically execute the modified specification and the simulated system behavior can be compared to the user's notion of the correct system behavior. The utility of simulation increases as the size and complexity of the specification increases; because large complex specifications may be too large to analyze exhaustively, simulation of the system behavior can complement formal analysis of the system behavior using, e.g., model checking.

A simulator useful in testing and debugging formal specifications should have a number of features:

- **Ability to support plug-in viewers.** The simulator should support alternative *representations* of an execution trace. (In this paper, an execution trace is a finite, timed sequence of events that represent system behavior over time.) That is, users should be able to plug in viewers which are most appropriate for the specification under consideration. Some views of simulated system behavior graphically depict the state of the system at a given time point, while others display the system behavior as a function of time. In some cases, users want a high-level understanding of the system behavior represented by the formal specification. In other cases, users want to focus on the events which represent lower-level specification details. Moreover, certain domains may require application-specific displays to enhance user understanding. For example, application-specific interfaces that mimic the behavior of real-world systems, such as avionics systems, may be useful.
- **Interactive user participation in generating a simulation trace.** The user should be able to control the simulated system behavior. In particular, when the system behavior is nondeterministic, the user should be able to select the desired behavior. One way to support this is to permit the user to inject events into the execution trace.
- **Monitoring/Assertion-checking.** Most monitoring tools only allow the user to set simple breakpoints or to detect the occurrence of simple events. To analyze complex assertions, the user of such tools

- *M. Brockmeyer is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: mab@cs.wayne.edu.*
- *F. Jahanian is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: farnam@eecs.umich.edu.*
- *C. Heitmeyer is with the Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC 20375. E-mail: heitmeyer@itd.nrl.navy.mil.*
- *E. Winner is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891. E-mail: elly@cs.cmu.edu.*

Manuscript received 8 Sept. 1997; accepted 19 Jan. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112757.

must monitor the display during the simulated execution or perform a post-hoc analysis of a log file. Visual inspection is highly error-prone. Moreover, a post-hoc analysis does not permit the user (or the simulator) to respond to an assertion violation during a simulation session. A more powerful option is to design the simulator to support user specification of complex assertions in a high-level language and to automatically monitor the assertions during simulation. Such support for monitoring and assertion checking must be integrated with event injection and flexible displays so that the simulation or the display characteristics can be altered in response to an assertion violation.

This paper has two objectives. First, it describes MTSim, a powerful and flexible approach to simulating system behavior based on formal specifications, which was designed to satisfy the above requirements. Second, the paper demonstrates the power and flexibility of MTSim by describing a suite of simulation and visualization tools. These tools include WebSim, a tool for testing and simulating Modechart specifications, as well as a tool for creating application-specific interfaces for simulation. The latter is demonstrated by an application-specific interface which models the behavior of an F-18 cockpit.

MTSim has been developed within the context of the Modechart toolset (MT) [10], a collection of integrated tools developed by researchers at the Naval Research Laboratory and the University of Texas. MT supports the formal specification of real-time behavior in the graphical Modechart language [26] and analysis via completeness and consistency checking, simulation, and formal verification. However, the MT simulation framework, MTSim, is fairly general and could be applied to other formal simulation tools.

MTSim contains three major components: the server, the client application programmer interface (API), and a collection of viewing tools. The server, which is called the Modechart Simulator Engine, simulates a Modechart specification by constructing execution traces that satisfy Modechart semantics [26]. The MTSim Client API communicates with the MTSim Server in order to specify simulation options, to inject events into the simulation trace, and to register for notification of event occurrences, among other things. Fig. 1 shows the variety of tools that may be incorporated into the MTSim framework.

WebSim, which provides full-featured simulation and testing of Modechart specifications, includes several viewing tools: a general-purpose controller, a log window, an animated display, a time-process display, and a monitoring and assertion checking tool. These prototype tools use the Client API to support various displays, event-injection and monitoring.

A specialized interface for an F-18 cockpit is an example of an application-specific viewer. This viewer was developed to illustrate how an existing interface can be transformed into an MTSim-compliant viewer. The F-18 interface was originally designed to provide a realistic simulation of the behavior visible to the pilot of an F-18 aircraft in setting up and releasing a weapon. The user

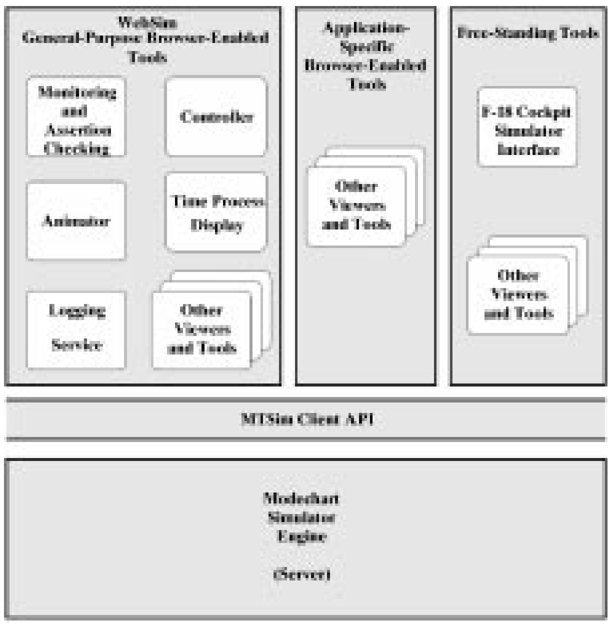


Fig. 1. Simulation tools in the MTSim framework.

interacts with the interface to control the simulated aircraft behavior. The original interface was built by researchers at the Naval Research Laboratories as a customized front end for an SCR [22], [24] requirements specification of a simple weapons system. Only a few hundred lines of code were needed to integrate the F-18 aircraft into MTSim, thus demonstrating the flexibility of the MTSim API.

The remainder of this paper is organized as follows: To put MTSim in context, Section 2 describes other tools developed to support formal methods, emphasizing those designed for real-time systems. Section 3 provides an overview of the Modechart language and describes the Modechart Toolset. Section 4 describes the viewers developed for the MTSim framework. Section 5 describes the MTSim architecture, while Section 6 discusses the different kinds of tools that can be developed and describes the MTSim Client API. Finally, Section 7 concludes by discussing the lessons learned in developing MTSim.

2 Related Work

A comprehensive overview of formal methods for real-time systems appears in [23]. Several researchers of these methods have noted the need for tool support [12], [16], [23]. Tools for analyzing specifications of real-time systems can be organized on a spectrum with model-checking on one end and simulation and monitoring at the other. Partial approaches include examination of a subset of the full computation space, for example, through testing. Like MT, many of the tools described provide both verification through a model-checking technique, as well as simulation, while a few tools are developed to support a single analytic technique. Not all tools which support simulation provide testing or monitoring capabilities for execution traces. Very few support an open framework for attaching user-defined viewers and displays.

A more complete description of the Modechart Toolset is found in [10]; the user manual is [33]. The Modechart Toolset supports specification analysis along the entire spectrum from model checking to simulation and verification. Stuart [34] describes the implementation of the MT Verifier, while Stuart et al. [35] describe a new partial approach, *simulation-verification*. This technique permits the user to simulate a computation prefix for some finite period of time before invoking the MT Verifier to perform model-checking on all computations starting with the given simulation prefix.

The STATEMATE system [19], which has been widely employed in industry, has many capabilities not found in research prototypes, e.g., version management and support for splitting specifications into multiple documents. The STATEMATE user can invoke a variety of static queries about the system specification. STATEMATE also provides a reporting language and a query language based on conjunctions and transitive closures. One of the most powerful capabilities of STATEMATE is the Analyzer tool. This tool provides extensive user control over system simulation, including a simulation control language, attachment of watchdog code for monitoring purposes, and connection of external C code to the symbolic execution. In an approach similar to our monitoring and assertion checking technique, these executions can be monitored via special "watchdog" code, which is defined by the user in the Statechart language. There is no automatic generation of watchdog code. It is also possible to set breakpoints in the Statechart code. In addition, the STATEMATE system provides a simulation control language to provide additional control over the generation of execution traces.

The Cabernet tool [28] provide a formal framework for developing real-time systems based on a temporal and functional extension of Petri Nets. The main components of this tool are a graphical editor, a suite of execution/animation/simulation tools which provide symbolic execution of the Petri net with an animated graphical display in real-time, and a reachability graph builder, a tool for managing hierarchical specification and decomposition of specifications. The semantics of Cabernet are defined by TRIO [17], a first order temporal logic, which was specifically designed for executability. Felder and Morzenti [13] provide history-checking of TRIO specifications by applying a tableaux-based algorithm to a history (execution trace) of a TRIO specification.

The StateTime Tool [31] is a visual toolset for design and analysis of real-time systems. It provided graphical editing, simulation, and verification in its BUILD, VERIFY, and DEVELOP tools. In addition, it supports the semi-automated system development of real-time systems with the DEVELOP tool. The graphical editor (the BUILD tool) supports simulation of system behavior. Variables and states are displayed in a trace window. This tool does not support assertion-checking in simulation, although the VERIFY tool supports checking of specialized assertions through the use of an observer module. This observer module is similar to our monitoring fragment.

The Software Cost Reduction toolset, SCR*, supports specification and analysis of requirements in the tabular SCR notation [20]. The tools provide completeness and consistency checking [21], specification editing, simulation, and mechanical verification. The SCR notation may be used to specify the required system functions and some time-dependent system behavior, i.e., behavior that can be described using time-outs.

The HyTech system provides tool support for specification and analysis of hybrid systems, systems which contain both continuous and discrete components. This tool performs symbolic model checking on systems modeled as linear hybrid automata [2] which have both finite control mechanisms, as well as real-valued variables modeling continuous environmental quantities. A more efficient version has been completed in C++ and many example systems have been tested on it. The tool accepts textual input of system specifications and analysis commands. The analysis approaches supported include forward and backward reachability analysis, as well as parametric analysis.

The SMV symbolic model-checker supports automated verification of real-time systems modeled as labeled state-transition graphs, where each path corresponds to an execution trace of the program. A state transition graph is represented internally using binary decision diagrams (BDD) which generally provide a more compact representation of the system behavior. The SMV symbolic model checker has been shown to handle very large state-spaces efficiently. Campos et al. [7] describe how algorithms for computing quantitative information about finite-state real-time systems have been incorporated into the SMV model checker.

The Concurrency Workbench [9] is an automated tool for analyzing networks of finite-state processes expressed in Milner's Calculus of Communicating Systems [29], [30]. The tool is designed in a modular fashion in order to accommodate a variety of specification transformations and analytical techniques. Equivalence, pre-order, and model checking, as well as state-space exploration, are supported.

VERSA [8] is another prototype toolkit which has been developed to support reasoning about real-time systems modeled as ACSR processes. ACSR (Algebra of Communicating Shared Resources) is an extension of CCSR (Calculus of Communicating Shared Resources) [15], which was developed to support the notions of both resources and priorities. In particular, the VERSA toolkit has three major functions: rewriting, equivalence testing, and interactive execution. To facilitate the use of ACSR by novice programmers, the graphical specification language GCSR (Graphical Communicating Shared Resources) [3], with semantics defined in ACSR, has been developed and incorporated into the VERSA toolkit.

3 Background on Modechart

This section provides a brief overview of the Modechart language and toolset using the example of a robot controller for a manufacturing assembly line. This example is used for purposes of illustration throughout the paper.

3.1 Modechart

The Modechart Toolset (MT) [10], a collection of integrated tools developed by the Naval Research Laboratory together with researchers from the University of Texas, supports the formal specification of real-time behavior for distributed systems in the Modechart language and formal analysis via formal verification, simulation, and completeness and consistency checking. The toolset includes facilities for graphically creating and editing Modechart specifications.

MT users may perform a variety of static consistency and completeness checks on their specifications. For example, they can check whether all modes have been specified with appropriate types or whether any mode transition expression refers to an event that has not been defined. They may also invoke a verifier that uses model checking to determine whether a Modechart specification satisfies any of a broad class of safety assertions. The MTSim tool complements these tools by symbolically executing Modechart specifications within a complete debugging and testing environment.

The basic construct generated, examined, analyzed, and displayed by the MTSim tool is a *simulation prefix*. The prefix is an initial, finite portion of a potentially infinite simulation trace. The simulation trace is a sequence of sets of event occurrences, where each set contains event occurrences which took place at a single moment in time. This sequence is ordered by the occurrence time.

Modechart [26] is a graphical specification language based on concurrent finite state diagrams. It provides a compact and structured way to represent real-time systems. Although similar to Harel's Statecharts [18], Modechart is specifically designed for the specification of real-time systems. It allows for the specification of modes which represent control information and thus impose structure on the operation of a system.

Modechart is extended from Statecharts with constructs for expressing timing constraints. It has a visual hierarchical structure and a small set of well-defined constructs for the definition of event-driven real-time systems. These constructs include *modes*, *mode-transitions*, *events*, and *timing constraints*. During a nonzero time interval, a mode can be either active or inactive. Informally, the state of a real-time system is described by the collection of modes which are active. Modes are hierarchically arranged in a tree structure; there is a top level mode from which all modes are descended and each mode has only one parent. The children of each mode can execute serially or in parallel, allowing Modechart to capture both concurrent and sequential behavior.

The behavior of a real-time system is captured by *mode transitions*, expressions which describe exit from one mode and entry into another mode. When a mode becomes active (inactive), a mode entry (exit) event corresponding to that mode occurs. These transitions can be specified by timing constraints or can be triggered by events in the system or by predicates on the behavior of modes in the specification. Events in Modechart include *mode-entry*, *mode-exit* events, transition events, and external events. When a mode becomes active, it is said that a mode entry event corresponding to that mode occurs. When a mode becomes inactive, it is said that a mode exit event occurs. Mode transition events occur when one mode is exited and another is entered. Finally, external events represent something happening in the environment which can affect the behavior of the system.

If $\mathtt{M}_1$, and $\mathtt{M}_2$ are modes, the exit of $\mathtt{M}_1$ is indicated by $\mathtt{M}_1 \rightarrow$, the entry of $\mathtt{M}_2$ is indicated by $-\mathtt{M}_2$, and the transition from $\mathtt{M}_1$ to $\mathtt{M}_2$ by $\mathtt{M}_1 \rightarrow \mathtt{M}_2$. External events are represented by single letters, e.g., E. These form the atomic units of a system; assertions are described in terms of timing and ordering relationships between instances of these type of events. Transitions can also be controlled by predicates on modes. For example, the predicate (M) indicates that the mode M is active, while $<$ M) indicates that the mode M has been active for at least one time unit. Further discussion of the Modechart language can be found in [33], [26]. Finally, more complex mode transition expressions can be formed from triggers and timing constraints. More elaborate triggers can be composed by taking the conjuncts of trigger expressions and these conjuncts can be disjuncted together with timing expressions.

3.2 A Robot Controller Example

Consider the robot controller for a manufacturing assembly line illustrated in Fig. 2. A producer process controls a conveyer belt carrying items to be processed by a robot. The producer process is responsible for moving items from position 1 to position 2. When the item is in position 2, the robot picks up the item, rotates away from the belt, and processes the item. (If the robot fails to pick the item up in a timely fashion and the conveyer belt is still moving, the item may move off the end of the conveyer belt and fall onto the floor.) Next, the robot attempts to place the item on another conveyer belt (position 3) where it is removed by a consumer process which moves it to position 4.

The Modehart specification depicted in Fig. 3 is a simplification of the robot specification described in [5]. The full specification has 18 root-level modes, representing 18 parallel components of the system.

The specification describes three processes (Producer, Consumer, and Robot) which are physically distributed and communicate with each other (as well as with the environment) through sensors. These sensors are also specified in Modechart. Finally, Modechart is also used to model certain aspects of the environment in which the system operates. This permits expression of various natural constraints on the behavior of the system.

The producer process, indicated by the *serial mode Producer_Belt*, has two major constraints on its behavior. If the belt has been stopped for one time unit and there is an item in position 1 and no item in position 2, the producer process starts the producer conveyer belt. If the belt has been moving and there is an item in position 2, the producer process stops the conveyer belt. The producer belt observes the environment through sensors which indicate whether there is an item in position 1 or position 2. The consumer process operates in a similar manner, starting and stopping the conveyer belt to move items from position 3 to position 4.

The robot controller moves items from position 2 to position 3, as well as processing them. The constraints on

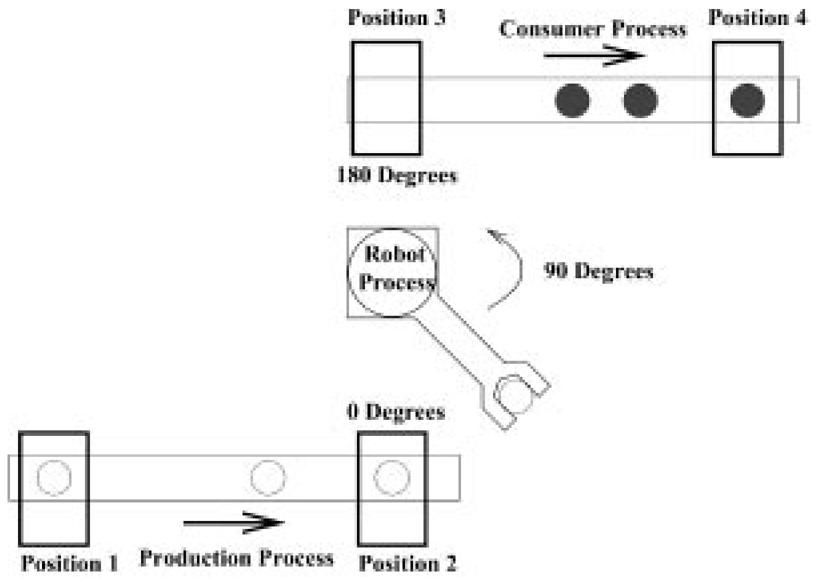


Fig. 2. Diagram of a robot controller for a manufacturing assembly line.

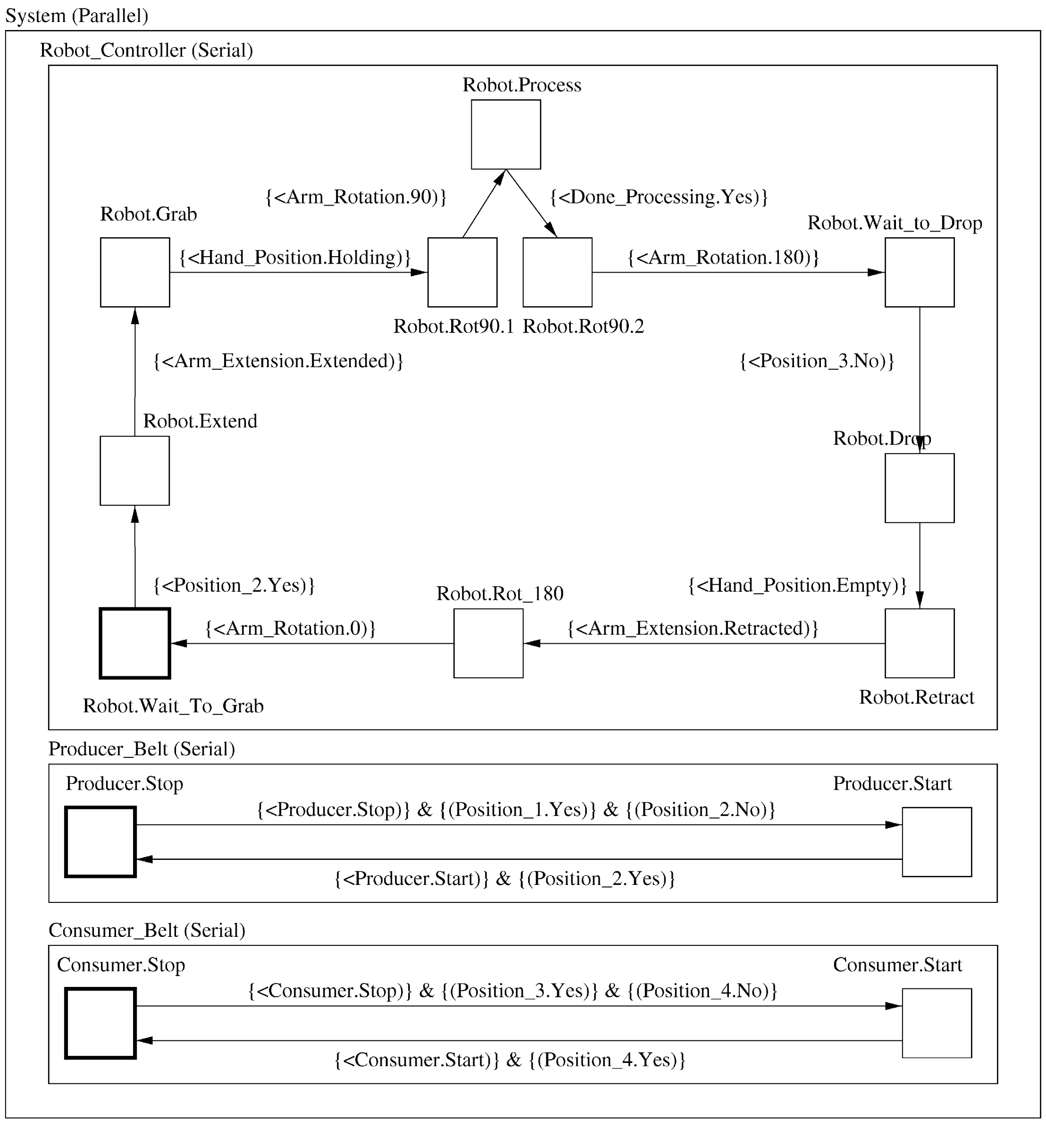


Fig. 3. Modechart specification of processes for a robot controller for a manufacturing assembly line.

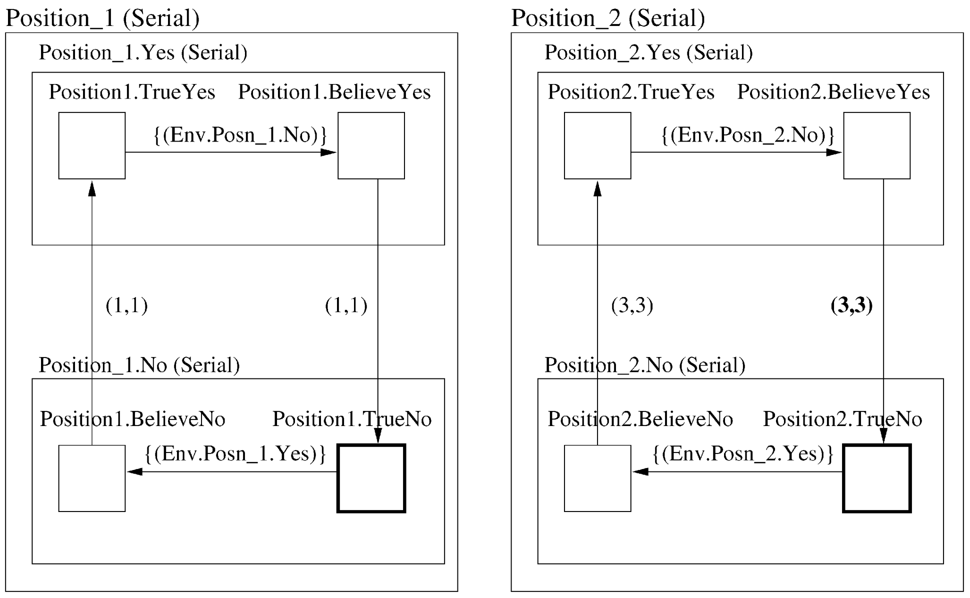


Fig. 4. Modechart specification of sensors for a robot controller for a manufacturing assembly line.

the robot controller's behavior are stated informally as follows: If an item is in position 2 and the robot hand is empty, the robot extends its arm, grabs the item, rotates 90 degrees, processes the item, then rotates 90 degrees. If there is no item in position 3, the robot arm drops the item, then retracts its arm, then rotates -180 degrees. The robot communicates with the producer belt through the sensor indicating whether there is an item in position 2. It communicates with the consumer belt via the sensor indicating whether there is an item in position 3. This control cycle is represented by the cycle of modes, *Robot.Wait_To_Grab*, *Robot.Extend*, *Robot.Grab*, etc., depicted in the figure. The entry of each of these modes sends a signal through a mode transition event to the remainder of the specification that may trigger events in the environment. Similarly, each transition is controlled by events that occur in the environment, some of which are reported by the sensors.

Fig. 4 displays a Modechart specification for some typical sensors. This figure depicts only two of the four sensors required by the system.

The sensors detect the state of the environment and relay this information to the robot controller. Since sensors test the environment only periodically, there is generally some delay in relaying a state change in the environment back to the processes relying on the sensors. As a consequence, it is possible that a sensor will indicate a stale value to the controllers. In Fig. 4, mode *Position1.BelieveNo* indicates a situation where there is an item in position 1, but this fact has not yet been detected by the sensor. After some delay, the sensor tests the environment again and moves to mode *Position1.TrueYes*, which indicates that the sensor now records the correct state of the environment. The mode *Position_1.Yes* indicates the state where the sensor records that there is an item in position 1, while the mode *Position_1.No* indicates the state where the the sensor records that there is no item in position 1. These modes are used by the processes to determine mode transitions.

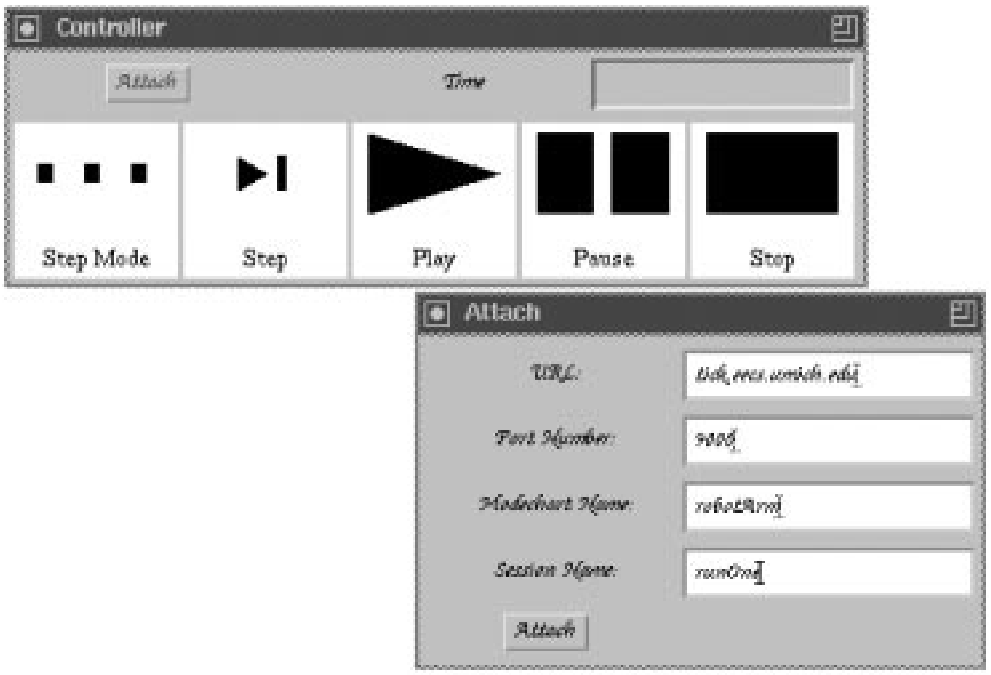
Note that the sensor for position 2 is slower than the one for position 1. This models a communications path which is faulty in some way, leading to a delay in transmission of information to the robot controller. As a consequence, the controller does not learn about changes in the environment at position 2 until three time units after they occur. The MTSim simulation tools, including the MTSim Monitor, are used to explore the effect of this faulty sensor on the behavior of the robot.

As noted above, because sensors sample the environment periodically, there is a delay between the occurrence of an event in the environment and the time which that event is visible to the controller process. In this specification, the sensor at position 2 is slightly slower than than the other sensors, requiring three time units to detect a state change in the environment. The MTSim Tool suite is used to examine the circumstances under which this delay might lead to items falling off the end of the producer belt onto the floor before the controllers have time to take the item for processing or stop the conveyer belt. Although not all details are presented, this example is used for the purposes of illustrating the tools in the remainder of the paper.

4 Simulation Using MTSim

This section describes several prototype MTSim viewers. Each client demonstrates how the MTSim API supports one of the main tasks which are required of an industrial-strength specification simulation environment, namely flexible displays, event-injection, and monitoring and assertion checking. In addition, the tools demonstrate that the MTSim framework is suitable for the development of both general-purpose and application-specific simulation viewers and display tools. These client tools include:

- **WebSim.** WebSim, a suite of graphical, web-centric simulation tools implemented as Java applets on the MTSim Client API. This collection of tools permits a specification designer to view, manipulate, and



URL: tick.eecs.umich.edu
Port Number: 9000
Modechart Name: robotArm
Session Name: runOne

Fig. 5. WebSim controller.

analyze a specification from a variety of viewpoints. Section 4.1 provides an overview of the WebSim Controller. Section 4.2 discusses the support that the MTSim API provides for monitoring and assertion checking in the WebSim Monitor. Section 4.3 describes three types of general-purpose displays provided by WebSim and explains how the MTSim Client API supports flexible displays.

- **F-18 cockpit simulation interface.** Section 4.4 describes an application-specific simulation interface, based on a representation of an F-18 cockpit. This tool was developed to demonstrate the ease with which an existing interface can be integrated into the MTSim framework. It also illustrates the use of the MTSim framework to support injection of events into the simulation trace.

4.1 The WebSim Controller

The first WebSim client, the WebSim Controller, is depicted in Fig. 5. The figure shows the starting of a simulation session for the robot controller described previously. The controller client initiates a simulation session, loads a Modechart specification, and starts and stops simulations. The controller also sets simulation parameters, permitting the client to indicate values when the specification is incomplete or to instruct the simulator about how to handle nondeterminism. For example, the controller may indicate that a timing transition should be made as soon as it is eligible, as late as possible, or at some fixed time point. Alternately, the controller might register with the server to be informed when a transition is eligible to be taken. The controller might then indicate when the transition should actually occur.

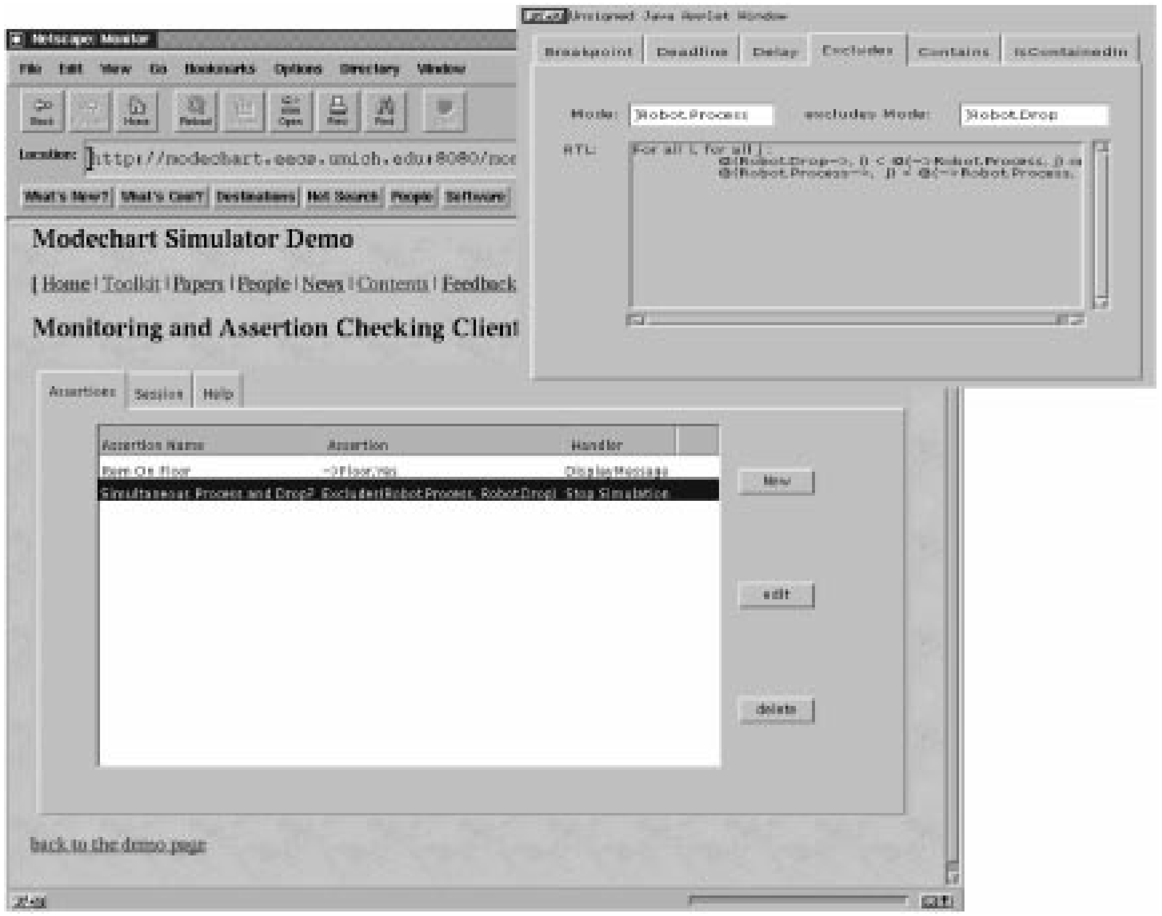
4.2 The WebSim Monitor

The WebSim Monitor (Fig. 6) performs monitoring and assertion-checking capabilities. The basic framework for performing monitoring and assertion-checking on Modechart specifications is found in [6].

Monitoring and assertion-checking can be used to detect an undesirable behavior or violation of a design assumption as an execution trace is generated. By combining evaluation of a property with a graphical representation of the execution trace, the monitoring and assertion-checking tool can work together with other simulation displays to conveniently provide better assurance of the user's intent during the specification and design phase. Monitoring and assertion-checking complement the existing tools in the Modechart Toolset, allowing testing to take place for program specifications and assertions for which formal verification may be impractical. This provides valuable feedback to system designers during the initial design process. A monitoring and assertion-checking tool can also be used to invoke user-defined handlers upon detection of certain properties. The handlers can be used to change the simulator execution profile or even the system state before the computation resumes.

The monitoring and assertion-checking tool, implemented as an MTSim client, meets two requirements. First, the WebSim Monitor should be flexible in how assertions are specified. Two approaches to assertion specification are supported:

- The user can use a subset of Real-Time Logic [27] to specify assertions descriptively. These assertions are specified via a graphical, form-based interface. The user fills the relevant information into the RTL formula, but is not required to write RTL formulae from scratch.


Netscape: Monitor
File Edit View Go Bookmarks Options Directory Window
Back Home Reload Open Print Find
Location: http://modechart.eecs.umich.edu:8080/mo
What's New? What's Cool? Destinations Net Search People Software
Modechart Simulator Demo
[Home | Toolkit | Papers | People | News | Contents | Feedback
Monitoring and Assertion Checking Client
Assertions Session Help
New
edit
delete
back to the demo page
Unsigned Java Applet Window
Breakpoint Deadline Delay Excludes Contains IsContainedIn

Assertion Name	Assertion	Handler
Item On floor	~?Floor.Yes	DisplayMessage
Simultaneous Process and Drop?	Excludes(Robot.Process, Robot.Drop)	Stop Simulation

Mode: Robot.Process excludes Mode: Robot.Drop

```
RTL: For all i, for all j :
         @(Robot.Drop-->, i) < @(-->Robot.Process, j) o
         @(Robot.Process-->, j) < @(-->Robot.Process,
```

Fig. 6. WebSim monitor.

The WebSim Monitor supports flexible specification of assertions by automatically translating RTL assertions into Modechart *monitoring fragments* in order to provide a specification for the monitoring process. The monitoring fragments, expressed as Modechart specifications, are used to represent the assertions of interest. The MTSim Server produces an execution trace of the augmented specification. The monitoring fragment is symbolically executed together with the original specification generating an execution trace that highlights the violation of the original assertion. As a result, detection of the violation of a potentially complex assertion is reduced to detection of a simple event or set of events in the simulation of the monitoring fragment.

These formulae include the following types of assertions, which describe relationships between pairs of modes:

- Excludes
- Contains
- During
- Starts
- Finishes
- Is Started By
- Is Finished By
- After
- Before
- Equals
- Meets
- Met By
- Overlaps
- Overlapped By

These assertions describe possible temporal relationships between intervals of time, first described by Allen [1]. For example, "A excludes B" means that mode A is never active at the same time that mode B is active. "A during B" means that A is always active during some interval of time when mode B is active.

These assertions are complemented by two timing assertions:

- Delay
- Deadline

These assertions are described more fully in [5], which also describes how they can be composed into more complex assertions.

- The user can also specify assertions directly by creating the monitoring fragments in Modechart. This allows maximum flexibility in terms of what kind of properties can be monitored. The monitoring fragments can be developed in the specification editor in much the same manner as are ordinary system specifications. The only real difference is that the user must identify to the WebSim Monitor which events represent violations of the assertions.

The second requirement for the WebSim Monitor is that the user is offered flexibility in terms of what action is to take place upon violation of an assertion. The WebSim Monitoring client offers built-in support for pausing or stopping the simulation, displaying custom messages, event logging, injecting an event into the simulation, and changing a simulation option (e.g., the time between occurrences for an event). User-defined handlers, which allow users to provide custom handlers which launch specialized displays (MTSim viewers), perform calculations or take other actions.

Fig. 6 shows the monitoring of two assertions on the robot controller example. The two assertions are displayed in a list on the main window. The first assertion states that a message is to be displayed if an item falls on the floor ($\rightarrow$ *Floor*). The second assertion indicates the simulation is to be halted if the modes *Robot.Process and Robot.Drop* are active simultaneously (*Robot.Process Excludes Robot.Drop*). The smaller window shows the assertion editor which permits common assertion types to be specified by filling in the relevant modes. The corresponding RTL is displayed below. During the simulation, additional windows may be displayed to indicate the possible violation of these assertions and to show what action has taken place.

4.3 Generic Viewers: WebSim Displays

WebSim contains three types of displays. The first, a simple logging facility, is depicted in Fig. 7 [20]. At the beginning of the simulation of the robot controller, all modes which are initially active are listed. Following, each mode transition event is displayed for the subsequent time points. The user may also choose to display only a subset of transitions, to display mode entry and exit events, or to display transitions which are eligible to be taken but are not taken at a particular time point. The window permits scrolling in order to view the entire simulation trace.

The second display provides an "animated" view of the simulation behavior. This Animator client is depicted in Fig. 8, which depicts an animation of the robot controller. (A similar tool is found in the STATEMATE [25] product.) It displays the execution of a specification graphically by displaying the original Modechart specification. As the simulation progresses, the active modes are displayed as shaded boxes, while modes that are not active are not shaded. This approach is good for displaying the state of the whole system at given points of time. In the figure, the large window depicts the top level view of the robot controller and its environment. Internal details of some modes are not depicted and, by default, mode transition labels are hidden as well. The user can click on a mode or drag the mouse on an area of the screen in order to view more detail. This will cause a smaller window to open, displaying the specified mode or region in more detail. (In the figure, two sensors are displayed with additional detail.) This permits the user to focus on the portions of the specification which are of interest.

Finally, a Time-Process client displays the simulation as a series of horizontal bars. This tool is depicted in Fig. 9. (This tool is based on the interface to the original Modechart Toolset Simulator.) Each bar indicates the behavior of one mode, transition, or external event over time. The figure shows bars representing modes from the robot controller example. Time is indicated by the horizontal axis. Thick lines indicate the periods of time during which each mode is active. For example, the user can examine the display to determine that the mode Hand_Position.Grabbing exits at the same moment that the mode RC.Process is entered. This type of display gives a good summary of the behavior of the system over time, but does not provide a "snapshot" of the system. As a result, the Animator and the Time-Process client are highly complementary.

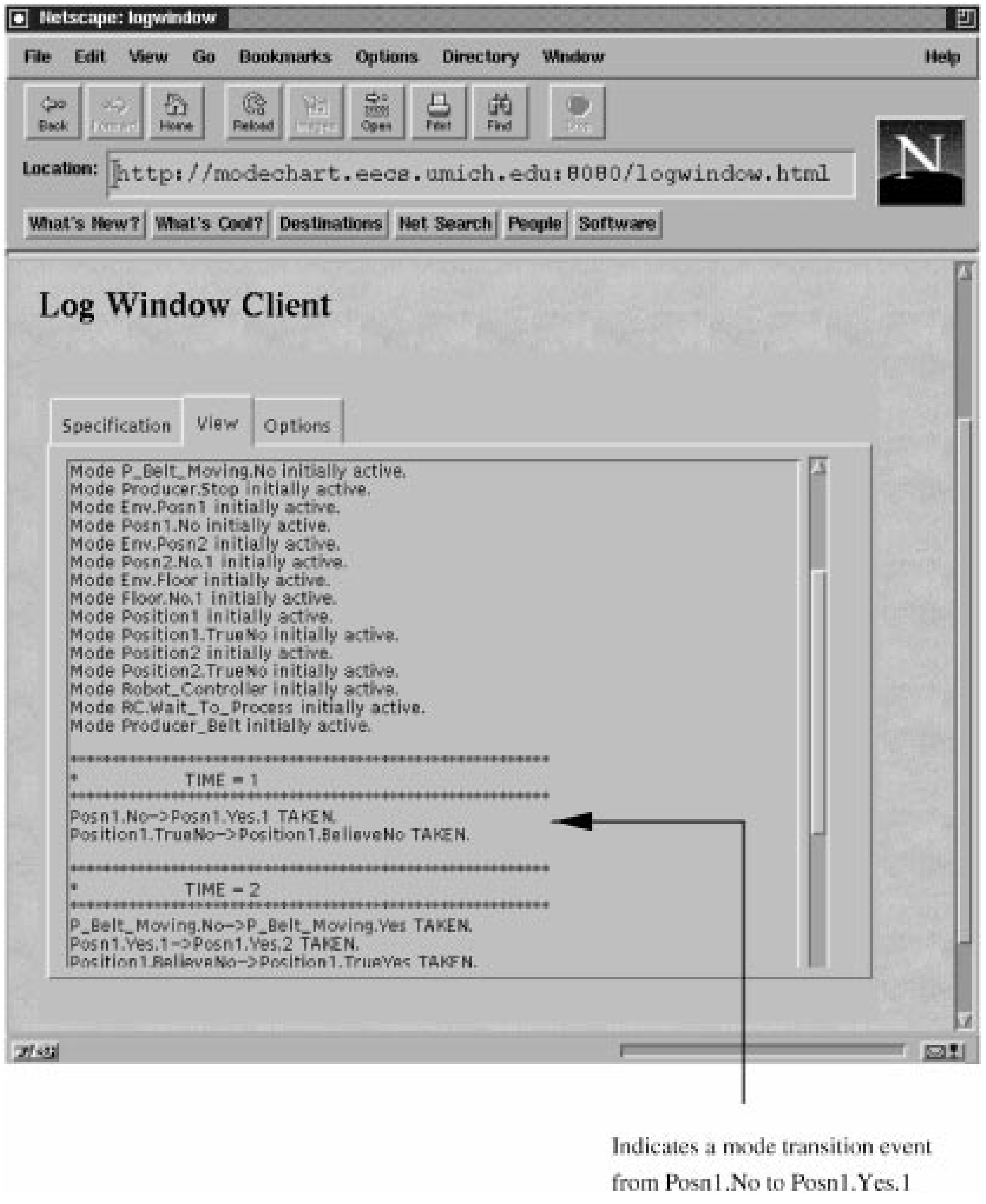
4.4 Application-Specific Tool: An F-18 Cockpit Simulator Interface

To demonstrate the power and flexibility of the MTSim Client API, we integrated the interface for an F-18 cockpit simulator into the MTSim framework. Several goals were achieved in developing this MTSim viewer. First, usefulness of the MTSim framework for implementing application-specific simulation viewers was tested. Because such viewers are used in specialized roles, a rapid and straightforward development process is desirable. Second, it is important to demonstrate an MTSim viewer which participated in the simulation by generating events to be incorporated into the simulation trace. And, finally, it was possible to evaluate how easily existing software could be incorporated into the MTSim framework.

The F-18 cockpit simulation interface was developed by NRL researchers as an interface to the SCR* Toolset [20] and was implemented in Motif using Century Computing's TAE+ interface builder tool [11]. The purpose of this simulation interface was to mimic the physical system represented by the formal specification. In this case, the interface supports simulation of the bomb release mechanism of a navy attack aircraft, such as the F-18. This simulation interface is a visual representation of the interface used by a pilot to interact with an avionics system. The required behavior of the avionics system is represented by a formal specification. In this example, the pilot interacts with the system by using the interface to set up and to release a weapon.

The F-18 cockpit simulation interface, depicted in Fig. 10, is a participant in the simulation process. The user, interacting with the simulation interface, simulates the behavior of the environment, while the MTSim Server simulates the response of the system specification to the events generated by the environment. It is possible to use the F-18 simulation interface together with other display tools to provide additional information to the user. These display tools could include animated displays, log windows, and time-process displays. In addition, a monitoring tool, such as the WebSim Monitor, could be attached in conjunction with these to provide monitoring and assertion-checking.

In Fig. 10, the small picture of an airplane in the upper left corner is a button widget which indicates whether or not the F-18 is airborne. The bullseye in the center of the windshield can be moved to indicate the miss distance (the projected distance from bomb impact point to the target).



```
Mode P_Belt_Moving.No initially active.
Mode Producer.Stop initially active.
Mode Env.Posn1 initially active.
Mode Posn1.No initially active.
Mode Env.Posn2 initially active.
Mode Posn2.No.1 initially active.
Mode Env.Floor initially active.
Mode Floor.No.1 initially active.
Mode Position1 initially active.
Mode Position1.TrueNo initially active.
Mode Position2 initially active.
Mode Position2.TrueNo initially active.
Mode Robot_Controller initially active.
Mode RC.Wait_To_Process initially active.
Mode Producer_Belt initially active.

****************************************************
*           TIME = 1
****************************************************
Posn1.No->Posn1.Yes.1 TAKEN.
Position1.TrueNo->Position1.BelieveNo TAKEN.

****************************************************
*           TIME = 2
****************************************************
P_Belt_Moving.No->P_Belt_Moving.Yes TAKEN.
Posn1.Yes.1->Posn1.Yes.2 TAKEN.
Position1.BelieveNo->Position1.TrueYes TAKEN.
```

Fig. 7. WebSim logwindow.

The Master Function Switch is modeled with a pop-up selector on the left panel, while the right panel displays the stations which contain a bomb. The slider at the bottom left selects a weapon type, while the pushbutton at the bottom is used to advance the clock. On the throttle, there are two pushbuttons. One is used to designate a target. If a target is locked onto, the words "Target Designated" appear under the target. The other pushbutton is used to release the bomb. If all conditions are correct for a bomb to be released (the plane is airborne, the master function switch is set to an appropriate value, a weapon of the correct weapon type is on the selected weapon station, the target is designated, and the miss distance is below a threshold), the bomb is released upon pushing the bomb release button. In this case, the words "Bomb Released" appear on top of the plane icon in the right panel.

Reengineering the SCR version of the simulation interface for MTSim involved two steps. First, the formal SCR specification for the bomb release mechanism was translated into Modechart. (A discussion of this process is beyond the scope of this paper.) Second, after translating the formal specification, the simulation interface was integrated as an MTSim viewer. The F-18 simulator tool has special privileges which permit it to load the specification and start it before the panel is displayed. One change was made to the graphical interface itself, adding a button

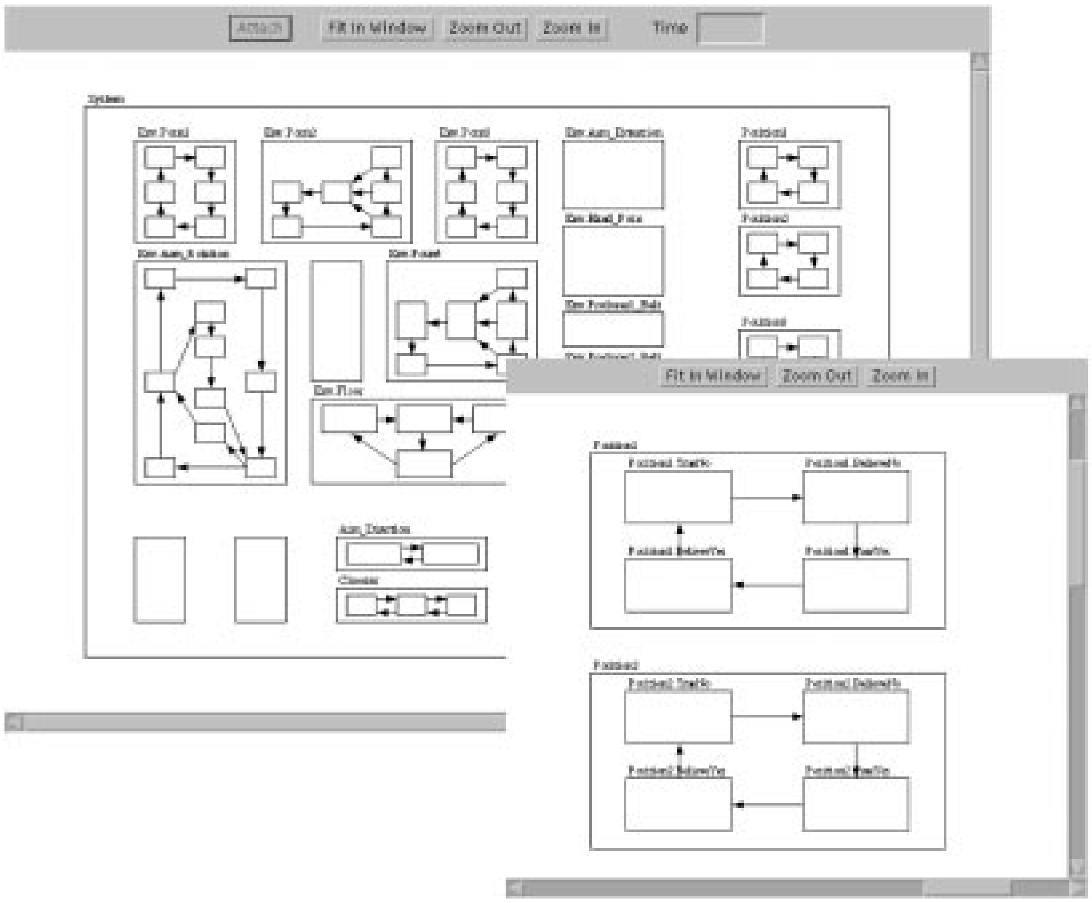


Fig. 8. WebSim animator.

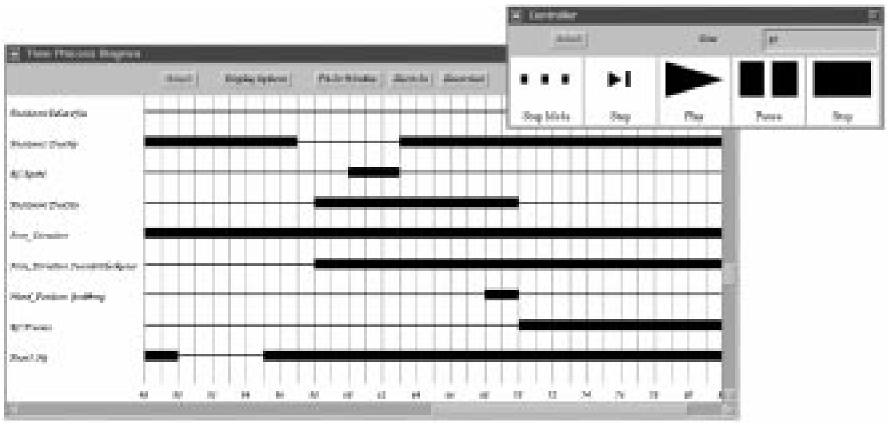


Fig. 9. TimeProcess display.

and a text display at the bottom of the screen to display the time and to permit the user to advance the time.

Because the TAE+ toolkit does not generate Java code, a Java wrapper was used to integrate the generated C code into the MTSim framework. The generated C code was invoked as native methods. The C code was reengineered to invoke the standard MTSim Client API.

The goals set for the development of the F-18 cockpit simulation interface viewer were met. The reengineering of the original interface was straightforward; the total effort required fewer than 150 lines of Java code and about the same number of lines of C code. This illustrates the usefulness and power of the MTSim Client API with regard to building application-specific front-ends to MTSim.

5 MTSim Architecture and Design

The original Modechart Simulator [33] does not support the attachment of specialized viewers. To achieve this capability, the implementation of MTSim separates the simulation mechanism from the simulation policy and display. This implementation supports an extensible environment within which the user would be able to view the behavior of the specification from various perspectives, to have flexible

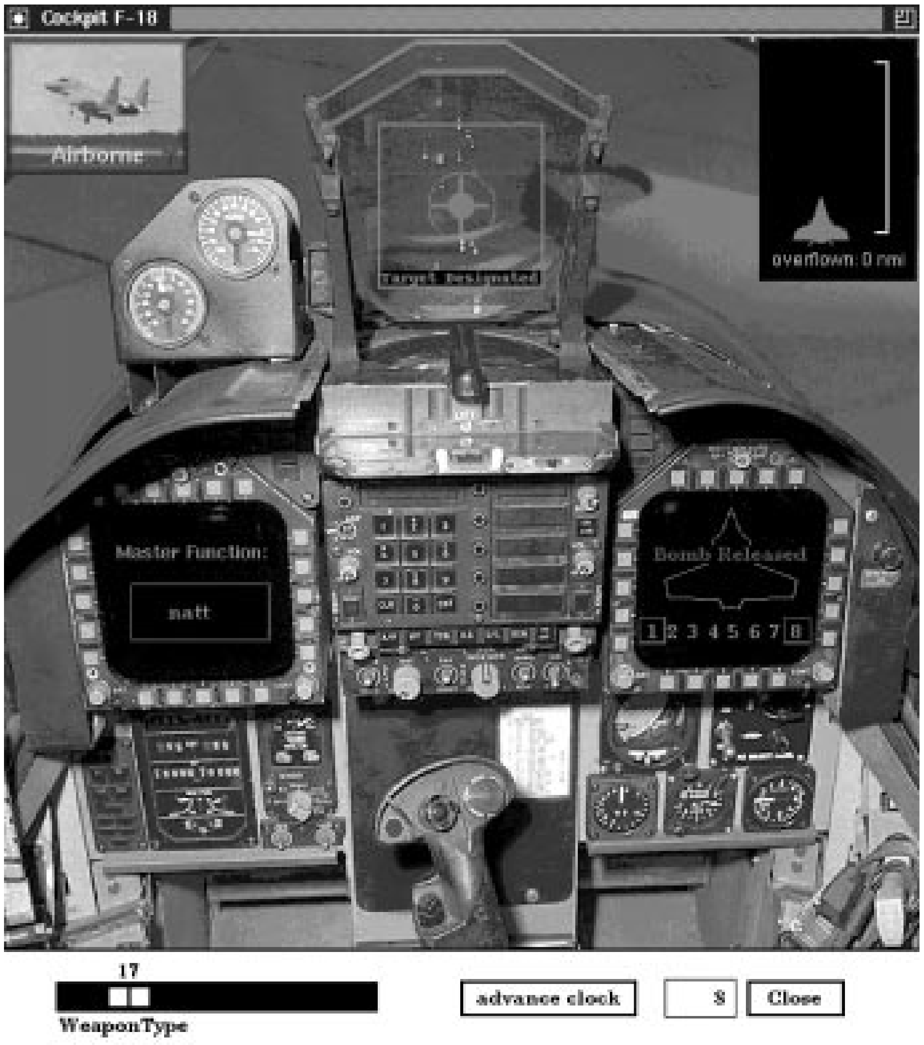


Fig. 10. F-18 cockpit simulation interface.

and powerful control over simulation behavior, to be able to use monitoring and assertion-checking tools, and to be able to incorporate additional tools, if necessary. As a result, MTSim provides a powerful testing and debugging environment for Modechart specifications.

Fig. 11 illustrates MTSim's architecture. The boxes indicate the major modules in the system. In this figure, the dotted line indicates the client API, described in the next section. Plain arrows indicate method invocation, while the dotted arrows indicate network messages. A thick arrow indicates that a module creates instances of another module. The MTSim framework is composed of three major parts: the MTSim Server, the MTSim Client Stub, and MTSim clients. The MTSim Server is responsible for generating execution traces consistent with the Modechart specification. It also accepts connections from clients, satisfies requests from clients, and reports the simulation events back to the clients. The MTSim Client Stub provides an easy-to-use programming interface to facilitate development of a variety of simulator clients. This interface permits a client to attach and detach from a simulation session, set simulation options, inject simulation events into the event stream, synchronize with the MTSim Server, and request notification of event occurrences. Both general-purpose and application-specific clients can be built on the MTSim framework.

The MTSim Server is implemented in a combination of Java and C. Simulation databases and code performing the actual generation of Modechart execution traces are written in C. This code, from the original version of the Modechart Toolset simulator, is combined with Java code, which performs communication, event notification, event scheduling, and other services described below. The MTSim Client Stub is written entirely in Java. This permits it to be easily incorporated into Java client code producing platform independent clients.

5.1 Communication

The communications system is based on remote method invocation and relies on an RMI protocol developed for this application. The underlying transport protocol is TCP/IP.

5.2 The MTSim Server

The major components of the MTSim Server are the Session Manager, the Client Service Managers, the Privilege Manager, the Synchronization Manager, the Event Matcher, and the SimKernel. The Session Manager establishes new

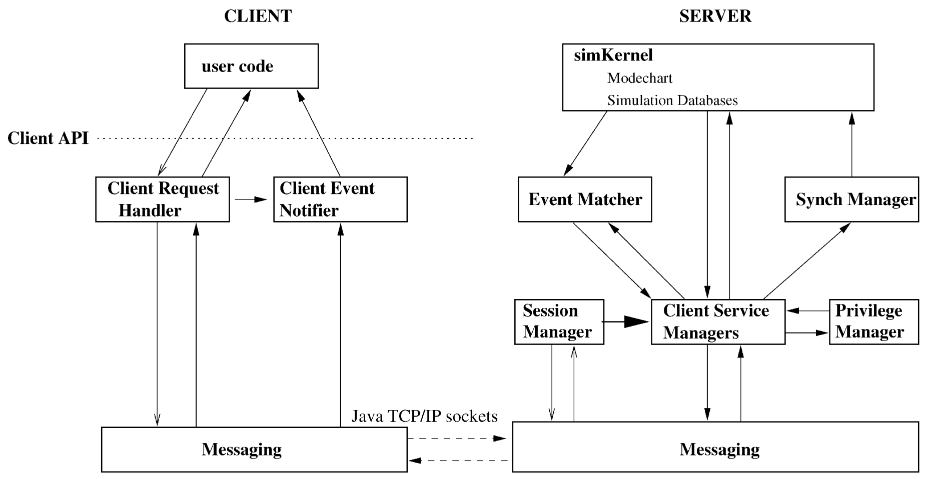


Fig. 11. Overall architecture of MTSim.

simulation sessions and attaches clients to each session as they present the appropriate session identifier. The Session Manager also instantiates a Client Service Manager for each client as it attaches to a session.

The Client Service Managers coordinate with the Privilege Manager to respond to client operations as they come in. Client operations are generally in correspondence with the methods of the client class provided in the Client API.

When a Client Service Manager is asked to start the actual execution of the Modechart simulation, it creates a Synchronization Manager to do so. The Synchronization Manager invokes the SimKernel to initialize the simulation databases. After each time point is simulated, the Synchronization Manager yields to the Event Matcher, which generates Event Notifications which are sent to the Client Service Managers to send back to the clients. Certain of these Event Notifications may be for event occurrences for which the client has asked to synchronize with the server. The Synchronization Manager waits for Continue messages from each such client before it invokes the SimKernel to simulate another time point.

The Synchronization Manager also schedules events which have been submitted by the clients and forwards them to the SimKernel for inclusion in the execution trace.

All of these components are written in Java, with the exception of the SimKernel. The internal representation of the Modechart specification is stored in a C database. Also written in C is the actual code responsible for generating a Modechart execution trace. These C libraries are wrapped in Java, where they are native functions. This approach maintains compatibility between MTSim and other tools in the Modechart Toolset which have been written in C.

5.3 The MTSim Client Stub

The MTSim client stub is the mechanism through which MTSim clients communicate with the MTSim Server. The client stub has two main components, a Client Request Manager and a Client Event Notifier. The client program invokes the methods included in the API on the client stub. The client stub translates these into messages which are sent to the MTSim server. The client stub reads reply messages from the server to determine the return value and whether an exception needs to be thrown to the user code.

The only state maintained by the client stub is held by the Client Event Notifier. When the user submits a request for an event notification to the MTSim Server, the MTSim client stub maintains a callback object for that request. When an event notification is received by the client stub, the Event Notifier invokes the notify method on the appropriate callback object.

6 MTSim Client API

Client tools built upon the MTSim framework are likely to require several key functions. Some clients will only need to register for event notifications in order to display the simulation behavior. Others may want to participate in the simulation behavior by generating events to inject into the simulation trace. Finally, a client may need to start and stop a simulation and to instruct the MTSim Server which specification to simulate. Corresponding to these roles, the Client API supports three types of clients, which are distinguished in terms of the level of privileges granted to them by the MTSim Server. A controller client has the most extensive set of simulator operations and is responsible for initiating a simulation session, indicating to the MTSim server which Modechart is to be simulated and establishing simulation options and defaults. Controller clients may also perform all of the operations which can be performed by the other types of clients.

Once a controller client has initiated a simulation session, other types of clients may attach to that session. These clients, which perform more basic operations, are display clients and participants. Display clients are the most basic type of client. They may attach to a simulation session, register to receive simulation notifications, and detach. Participant clients have the privileges of display clients, but may also generate events which are submitted to the MTSim Server for execution.

6.1 A Typical Client Session

Fig. 12 depicts a typical client session. A typical session for using the API in a client program is as follows:

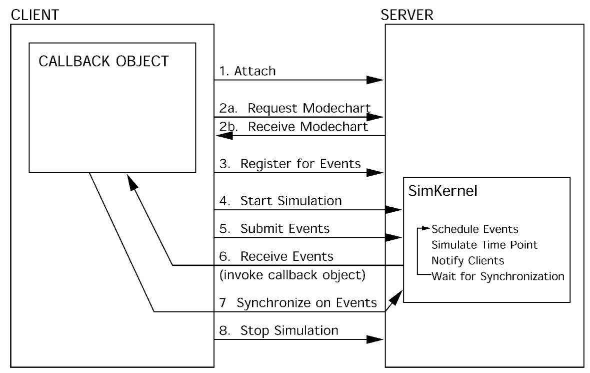


Fig. 12. A typical Client Session.

1. **Attach to a simulation session.** The client program creates a client object of one of the client classes reflecting the desired level of privilege. The client then uses the Attach method of the newly created client object to attach to the MTSim Server, indicating the MTSim server and a session name in the URL string.
2. **Request a Modechart.** Any client may get a static representation of the loaded Modechart by issuing the GetModechart method at any time.
3. **Register for event notifications.** After attaching, the client process will generally register MTEventPatterns with the server using the method RegisterForEventNotification. In doing so, the clients indicate to the server the simulator events for which they would like to receive notification. The MTEventPatterns class provides a powerful approach to specifying individual Modechart events or groups of events. The client program also indicates whether the server should synchronize with the client when the specified event occurs. If synchronization is requested for certain events, upon generation of those events, the server will wait for the client to invoke the Continue method before proceeding with the simulation. Finally, the client provides an object which implements the ClientEventHandler interface. Upon receipt of an event notification which matches the MTEventPattern, the client object will perform a callback to the client by invoking the notify method of the ClientEventHandler object.
4. **Begin a Modechart simulation.** Controller clients will then generally invoke a LoadModechart method and optionally set the simulation options using one of the SetSimOptions methods. They will then start the simulation using the StartSimulation method.
5. **Submit Events.** Clients may submit external events to submit in the simulation any time between the start and the end of the session.
6. **Receive event notifications.** While the simulation is running, the server will generate event notifications in the form of MTEventOccurrence objects and send the notifications to the various clients. The client will be notified as described above and will use the notifications to update its display or perform any relevant computations.

```
class SimpleHandler implements ClientEventHandler {
  ControllerClient sim;
  public SimpleHander(ControllerClient s){
  sim = s;
  }

  public void notify(MTEventOccurrence eo) {
    System.out.println(eo);
    if (eo.event_type == MTEventPattern.TIME)
      && (eo.occurrence_time == 10) {
      sim.StopSimulation();
    }
    if (eo.event_type == MTEventPattern.STOP)
    { // end of simulation
      sim.Detach();
    }
  }
}

public class SimpleClient{
  public static void main(String args[]){
    ControllerClient sim = new ControllerClient();
    sim.Attach(args[0]);
    sim.LoadModechartRemote(args[1]);
    sim.RegisterForNotification(
      new MTEventPattern(MTEventPattern.ANY)
      false, new SimpleHandler(sim));
    sim.StartSimulation();
  }
}
```

Fig. 13. Simple client.

7. **Synchronize on Events.** The clients may choose to synchronize with the server on certain event notifications. In this way, the server will not be able to advance the simulation until all clients have synchronized on all such events received so far.
8. **Stop the simulation.** At some point, the controller client will issue the StopSimulation method and the various clients will detach themselves from the simulation.

A very simple controller which loads a Modechart specification, starts the simulation, and prints out the events as they occur is depicted in Fig. 13. (For simplicity, exception handling is not presented.) Here, the class SimpleHandler provides a basic event handling behavior. The notify method simply prints out each event. If the time is equal to 10, the event handler sends a message to stop the simulation. If the event handler receives notification that the simulation has stopped, the Detach method is invoked. The main class, SimpleClient, sets up the connection to the server, loads a Modechart, registers to receive all events, and starts the simulation.

6.2 Event Notifications and Views

A client may register to receive all simulation events. In this scenario, it is the responsibility of the event handler to distinguish between types of events. A more sophisticated client application, however, may only be interested in certain events or may want to provide different event handlers for different types of events. In particular, the client may want to synchronize on some events and not others. For example, the client may want to synchronize on events which represent the passage of time, but not on other

TABLE 1
Event Types Specified by MTEventPattern

Event Pattern	Mode Descendent Restriction	Name Restriction	When Satisfied	When Scheduled
ALL	YES	YES	YES	YES
MODE_ENTRY	YES	YES	NO	NO
MODE_EXIT	YES	YES	NO	NO
MODE_TRANSITION	YES	YES	YES	YES
EXTERNAL	NO	YES	NO	YES
TIME	NO	NO	NO	NO
TIMEPATTERN	NO	NO	NO	NO
STOP	NO	NO	NO	NO
START	NO	NO	NO	NO
ASSERTION	NO	NO	NO	NO

events. This gives the client time to process all events from each time point before permitting the server to advance to the next time point.

The set of events in which a particular client is interested comprise a user-specific *view*. Although the MTSim Server generates a single computation trace, each user may be interested in a different view or subset of the events in the computation trace. The client can specify the view they want by registering for *Event Patterns*. The event patterns are filtering objects which are registered with the MTSim Server; each such object can be registered with a different client event handler.

The user can restrict the collection of events of interest in four ways.

1. Events can be restricted to those of a particular *type*. These event types include mode entry and mode exit events, mode transitions, external events, and time passage events. Event types are listed in Table 1.
2. Furthermore, some events (Mode Entry, Mode Exit, and Mode Transition events) can be considered to be connected to particular modes. A user can restrict attention to events which are connected to all descendents of particular modes.
3. Events have names. An event pattern can use a name or a regular expression to further restrict the collection of events. For example, the expression $(MTEventPattern.eventType == MODE\_ENTRY)$ $\&\&(Name == "C*")$ will restrict attention to mode entry events for modes having names which start with the letter "C." (Modechart is not case sensitive.)
4. Finally, certain events may become satisfied or be scheduled before they occur. For example, a mode transition may be satisfied without taking place or an external event may be scheduled by the simulator before it occurs. In these cases, the client may register for these special types of notification.

Table 1 also provides a summary of which event types permit these additional restrictions.

7 Discussion

This paper has described MTSim, an extensible simulation framework for formal specifications. This infrastructure supports a customizable environment for simulation, permitting users to plug in specialized viewers and other simulation tools, including monitoring and assertion checking tools. This paper has also presented several tools developed using the MTSim framework, including WebSim, a suite of general-purpose monitoring tools which execute on the World Wide Web in the browser environment, and a free-standing application-specific simulation interface which models the cockpit of an F-18 aircraft.

This section describes the issues raised and the lessons learned in developing the WebSim Tool Suite and the F-18 cockpit simulator tool using the MTSim framework.

7.1 Simulation of Formal Specifications as a Design Tool

Simulation and monitoring of simulation traces provide a *testing* approach to specification validation. In contrast to formal verification, where the entire set of possible system behaviors may be examined, simulation permits examination of a single execution trace. As a consequence, errors present in a specification may not manifest in a single execution trace. These errors could be propagated to an implementation based on a specification.

However, experience has shown that a powerful simulation environment is a useful complement to formal verification because it provides more immediate and more intuitive feedback to the system designer early in the specification process. That is, because generation of a specification trace is computationally less costly than generation of a computation graph, it is more easily used during development to ensure that a specification meets both informal and formal requirements, as well matching a designer's intuition.

An important issue is how users or client tools control the direction of the generation of an execution trace. Since there are generally an infinite number of simulation traces corresponding to a given specification, the problem of how a particular simulation is chosen can be quite complex. Generally, the selection of a simulation trace can be described by a sequence of responses the simulator makes to nondeterministic choices.

Users can exert control over simulation behavior in two ways in the MTSim framework. The first is through the use of an MTSimOptions object which can be submitted to the MTSim by a Controller Client. The MTSimOptions object sets basic defaults, such as the amount of time to take on timing transitions, minimum separation between external events, and initialization options for items which may have been underspecified in the simulated specification. Additional control may be exerted by resubmitting the MTSimOptions object during the execution.

The second way that users interact and direct the generation of an execution trace is through the generation and injection of external events which affect simulation behavior. Client tools which have registered as participants may submit external events to the MTSim Server. These events are included in the execution trace and the simulation is allowed to respond to these events.

Users have a great deal of control over how these events are injected into the simulation trace. One event or a sequence of events may be injected. Injection of events combined with synchronization provides the user with a chance to stop the simulation and examine the system state before injection of additional events. These two approaches permit the user to specify nondeterministic behavior more generally by establishing an overall policy for resolving each type of nondeterminism or at a finer granularity by injection of steering events.

Another lesson learned was the need for constructs to support simulation timing on a "human scale." This means that the simulation is paced in such a way as to improve the user's intuition about simulation behavior. For example, while a simulation may not take place in real-time, delays should be proportionate to the real-time delays being modeled. This can be approximated in our current framework by having a specialized client synchronize on every event and insert the appropriate delays. Of course, communication delays will distort the actual timing. This approach also requires that all other clients synchronize in a timely enough manner so that additional delays do not occur.

The current framework is being extended to permit clients to commit to the amount of time that they require to process a given event notification. Not only will this facilitate insertion of human scale delays, but it will improve performance due to the reduction of synchronization acknowledgments.

7.2 Performance and Implementation Issues

Integration of plug-in viewers has been hampered by performance issues. This is due, in part, to slow execution of the Java interpreter. Just-in-time compilers [32] and similar efforts may be able to compensate. However, for this application, the advantages of execution on multiple platforms, in or out of the browser, were felt to outweigh the drawbacks.

The most serious performance impairment, however, is due to slow communication between the MTSim server and the MTSim client tools. This can be largely attributed to the Java TCP/IP implementation. Efforts are underway to mitigate this slowness through the use of UDP/IP communication and multicast mechanisms.

Client tools may register to *synchronize* on certain events when they register for event notification. Synchronization means that the MTSim Server does not advance the simulation until a synchronization acknowledgment is received from the client. Synchronization is used so that client tools may control the rate of the simulation so that each client may process events without lagging behind other clients or the MTSim Server. Synchronization may also be used to implement breakpoints. When used together with monitoring, the breakpoints can be predicated on the satisfaction or violation of a complex condition.

Performance is also highly sensitive to the synchronization behavior of the client tools. Client tools may be loosely or tightly synchronized. Since the server does not advance the simulation until it has received acknowledgments for all event notifications for which synchronization was requested, all the clients may be delayed until one client synchronizes. As a result, finely grained synchronization can result in significant overhead. Therefore, performance can be improved by eliminating unnecessary client synchronization.

Since synchronization is designed to permit client tools to respond to simulation behavior, the type of synchronization required varies with the particular requirements of each type of client. For example, the Time-Process Display described in Section 4.3 is designed to emphasize a longitudinal view of the simulation. As a result, it need not synchronize at every time point since it may be acceptable for the Time-Process display to fall slightly behind in displaying simulation events. In contrast, the Animator tool is designed to display a sequence of system states. Therefore, it synchronizes on each event notification. Consequently, all client tools which are executing at the same time must wait for the Animator to acknowledge this event before the simulation can continue. Therefore, it is critical that such tools perform calculations, update displays, and send acknowledgments as quickly as possible. (When an incorrect client fails to acknowledge an event notification for which it has synchronized, the effect is even more serious, resulting in the halting of the simulation.)

A final issue involved the MTSim API. The usefulness of higher-level constructs than those provided by the MTSim API became evident during the development of the WebSim Tool Suite. The various tools shared many common elements, such as graphical interfaces for registering with the server and selection of events for event notification, that might well be integrated into the MTSim API.

As described in Section 4.4, the F-18 cockpit simulation tool was easily integrated into the MTSim framework, despite having been developed for another formal specification tool. In fact, translation of the specification was among the more challenging aspects of the task. With regard to integration of the simulation software, the most difficult task was "wrapping" native code generated by the interface builder so that it could interact with the Java API. Invocation of the MTSim API was straightforward; the event notification constructs were a close match for those in the original software. As mentioned previously, the entire effort was completed in a few days and required only a few hundred lines of code.

There are several goals for future work. Among these is support for persistent simulation sessions, which will include development of a model for logging and replay services. In addition, it is important to develop additional WebSim tools, investigate other types of monitoring (such as performance monitoring), and to extend the types of assertions supported by the WebSim monitor. Finally, use of the MTSim framework to support simulation of other formal specification languages, such as SCR, is being evaluated.

Acknowledgments

The authors would like to acknowledge the contributions of Bruce Labaw for his review and suggestions on earlier drafts of this paper, as well as Eylon Caspi and others at the Naval Research Laboratory who developed the graphical F-18 cockpit interface. This work is supported in part by the US Naval Research Laboratory under Grant N00014-94-P2015.

References

[1] J.F. Allen, "Maintaining Knowledge about Temporal Intervals," *Comm. ACM,* vol. 26, no. 11, pp. 832-843, 1983.
[2] R. Alur, T.A. Henzinger, and P.-H. Ho, "Automatic Symbolic Verification of Embedded Systems," *Proc. IEEE Real-Time System Symp.,* p. 2-11, 1993.
[3] H. Ben-Abdallah, I. Lee, and J.-Y. Choi, "A Graphical Language with Formal Semantics for the Specification and Analysis of Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* pp. 276-286, 1995.
[4] B. Boehm, *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice Hall, 1981.
[5] M. Brockmeyer, "Monitoring, Testing, and Abstractions of Real-Time Specifications," PhD thesis, Dept. of Electrical Eng. and Computer Science, Univ. of Michigan, 1999.
[6] M. Brockmeyer, F. Jahanian, C. Heitmeyer, and B. Labaw, "An Approach to Monitoring and Assertion-Checking Distributed Real-Time Systems," *Proc. Workshop Parallel and Distributed Real-Time Systems,* Apr. 1996.
[7] S. Campos, E. Clarke, W. Marrero, and M. Minea, "Computing Quantitive Characteristics of Finite-State Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* pp. 276-286, 1994.
[8] D. Clarke, I. Lee, and H. Xie, "Versa: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems," Technical Report MS-CIS-93-77, Dept. of Computer and Information Science, Univ. of Pennsylvania, Sept. 1993.
[9] R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *ACM Trans. Programming Languages and Systems,* vol. 15, pp. 36-72, 1993.
[10] P.C. Clements, C.L. Heitmeyer, B.G. Labaw, and A.T. Rose, "MT: A Toolset for Specifying and Analyzing Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1993.
[11] Century Computing, *TAE Plus Users Guide, Version 5.3,* Sept. 1993.
[12] D. Craigen, S. Gerhart, and T. Ralston, "An International Survey of Industrial Applications of Formal Methods," Technical Report NRL-9581, Naval Research Laboratory, Washington, D.C., 1993.
[13] M. Felder and A. Morzenti, "Validating Real-Time Systems by History-Checking TRIO Specifications," *ACM Trans. Software Eng. and Methodology,* vol. 3, no. 4, Oct. 1994.
[14] D.A. Gabel, "Technology 1994: Software Engineering," *IEEE Spectrum,* vol. 31, no. 1, pp. 38-41, Jan. 1994.
[15] R. Gerber and I. Lee, "CCSR: A Calculus for Communcating Shared Resources," *Proc. CONCUR '90,* pp. 263-277, 1990.
[16] S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems," *IEEE Software,* vol. 11, no. 1, pp. 21-28, Jan. 1994.
[17] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO, a Logic Language for Executable Specification of Real-Time Software," *J. System Software,* vol. 12, pp. 107-120, 1990.
[18] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, 1987.
[19] D. Harel et al., "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.,* vol. 16, Apr. 1990.
[20] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," *Proc. COMPASS:95,* 1995.
[21] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj, "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications," *IEEE Trans. Software Eng.,* vol. 24, no. 11, Nov. 1998.
[22] C. Heitmeyer, B. Labaw, and D. Kiskis, "Consistency Checking of SCR-Style Requirements Specifications," *Proc. Int'l Symp. Requirements Eng.,* Mar. 1995.
[23] *Formal Methods for Real-Time Computing,* C. Heitmeyer and D. Mandrioli, eds. John Wiley and Sons, 1996.
[24] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 3, pp. 231-261, July 1996.
[25] iLogix Corporation, *The Languages of STATEMATE,* 1991.
[26] F. Jahanian and A.K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 10, Oct. 1994.
[27] F. Jahanian and A.K.-L. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 12, no. 9, pp. 890-904, Sept. 1986.
[28] D. Mandrioli, A. Morzenti, M. Pezze, P. SanPietro, and S. Silva, "A Petri Net and Logic Approach to the Specification and Verification of Real Time Systems," *Formal Methods for Real-Time Computing,* C. Heitmeyer and D. Mandrioli, eds., chapter 6. John Wiley & Sons, 1996.
[29] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.
[30] R. Milner, *A Calculus of Communicating Systems.* Springer-Verlag, 1990.
[31] J.S. Ostroff, "Statetime—A Visual Toolset for the Design and Verification of Real-Time Systems," Technical Report CS-ETR-94-07, York Univ., 1994.
[32] M.P. Plezbert and R.K. Cytron, "Does 'Just in Time' = 'Better Late than Never'?" *Proc. Principles of Programming Languages,* pp. 120-131, Jan. 1997.
[33] A. Rose, M. Perez, and P. Clements, "Modechart Toolset User's Guide," Technical Report NRL/MRL/5540-94-7427, Center for Computer High Assurance Systems, Naval Research Laboratory, Washington, D.C., Feb. 1994.
[34] D. Stuart, "Implementing a Verifier for Real-Time Systems," *Proc. Real-Time Systems Symp.,* pp. 62-71, Dec. 1990.
[35] D. Stuart, M. Brockmeyer, A. Mok, and F. Jahanian, "Simulation-Verification: Biting at the State-Space Explosion Problem," *IEEE Trans. Software Eng.,* to appear.

Monica Brockmeyer received her PhD degree in electrical engineering and computer science from the University of Michigan in 1999. She received her BS and MS degrees from the University of Michigan in 1986 and 1995, respectively. She is an assistant professor at Wayne State University, Detroit, Michigan. Her research interests include formal methods and real-time and fault-tolerant computing.

Farnam Jahanian received the MS and PhD degrees in computer science from the University of Texas at Austin in 1987 and 1989, respectively. He is currently an associate professor of electrical engineering and computer science at the University of Michigan. Prior to joining the faculty at the University of Michigan in 1993, he was a research staff member at the IBM T.J. Watson Research Center, where he led several experimental projects in distributed and fault-tolerant systems. His current research interests include real-time and fault-tolerant computing and network protocols and architectures.

Constance Heitmeyer holds MAs in mathematics and history from the University of Michigan and currently is head of the Software Engineering Section of the Naval Research Laboratory's (NRL) Center for High Assurance Computer Systems. Before assuming her current position at NRL, she was a visiting scientist at the NATO Undersea Research Center in La Spezia, Italy. She served in 1996 as program chair for the 11th Annual COMPASS Conference and in 1997 as general chair for the Third IEEE Symposium on Requirements Engineering. She has also served as an associate editor for the *Journal of Real-Time Systems* and is currently an associate editor of the *Requirements Engineering Journal*. In 1996, she coedited a book on formal methods for real-time systems with D. Mandrioli. She also serves on the steering committee of IFIP Working Group 2.9 on software requirements and the steering committee of the International Symposium on Requirements Engineering. Ms. Heitmeyer and David Parnas are currently guest editors of a special issue on tabular notations of the Kluwer journal *Formal Methods in System Design*. Her research interests are in formal methods, requirements, and real-time computing.

Elly Winner is a PhD student at Carnegie Mellon University, Pittsburgh, Pennsylvania. This research was performed while she was an undergraduate student at the University of Michigan, where she received her BS degree in 1998. Her current research interests are in artifical intelligence, particularly in inexact planning and in multirobot interactions.