JPL Publication 92-23

Event-Driven Simulation in SELMON: An Overview of EDSE

Nicolas F. Rouquette
Steve A. Chien
Leonard Charest, Jr.

August 1992

NASA
National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 92-23	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Event-Driven Simulation in SELMON: An Overview of EDSE		5. Report Date August 1992 6. Performing Organization Code
7. Author(s) Nicolas F. Rouquette, Steve A. Chien, Leonard Charest, Jr.		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No. 11. Contract or Grant No. NAS7-918 13. Type of Report and Period Covered JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code RE 159 BK-595-12-33-00-00

15. Supplementary Notes

16. Abstract

We describe EDSE, a model-based event-driven simulator implemented for SELMON, a tool for sensor selection and anomaly detection in real-time monitoring. The simulator is used in conjunction with a causal model to predict future behavior of the model from observed data. The behavior of the causal model is interpreted as equivalent to the behavior of the physical system being modeled.

This report provides an overview of the functionality of the simulator and the model-based event-driven simulation paradigm on which it is based. Included are high-level descriptions of the following key properties: event consumption and event creation, iterative simulation, synchronization and filtering of monitoring data from the physical system. Finally, we discuss how EDSE stands with respect to the relevant open issues of discrete-event and model-based simulation.

17. Key Words (Selected by Author(s))	18. Distribution Statement
1) Computer Programming and Software 2) Systems Analysis 3) Physics (General)	Unclassified; unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	18	

JPL Publication 92-23

Event-Driven Simulation in SELMON: An Overview of EDSE

Nicolas F. Rouquette
Steve A. Chien
Leonard Charest, Jr.

August 1992

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Abstract

We describe EDSE, a model-based event-driven simulator implemented for SELMON, a tool for sensor selection and anomaly detection in real-time monitoring. The simulator is used in conjunction with a causal model to predict future behavior of the model from observed data. The behavior of the causal model is interpreted as equivalent to the behavior of the physical system being modeled.

This report provides an overview of the functionality of the simulator and the model-based event-driven simulation paradigm on which it is based. Included are high-level descriptions of the following key properties: event consumption and event creation, iterative simulation, synchronization and filtering of monitoring data from the physical system. Finally, we discuss how EDSE stands with respect to the relevant open issues of discrete-event and model-based simulation.

Contents

1 Introduction

This report describes EDSE (**e**vent-**d**riven **s**imulation **e**ngine), a model-based event-driven simulator implemented for the SELMON (**sel**ective **mon**itoring) tool [Chien et al., 1992; Doyle et al., 1992]. In SELMON, the simulator is used in conjunction with a causal model to predict future behavior of a physical system from observed data. Observations of physical system behavior are possible because the system is monitored by a set of sensors.

These predictions can be used in two ways: First, current data can be used to predict future performance of the system. When predictions about the system turn out to be different from the actual performance of the modeled system, the predictions may be interpreted as discrepancies [Doyle et al., 1989; Dvorak & Kuipers, 1989]. Second, the model can be used to determine whether future performance of the modeled system will exhibit certain critical qualities (e.g., cascading alarms).

The focus of this report is to describe the SELMON simulator EDSE. The core principles of EDSE are inherited from event-driven simulation [Banks & Carson, 1984]; the target application domains originate in model-based reasoning of physical systems [Hayes, 1989]. Applications of model-based simulation are described elsewhere [Chien et al., 1992]. Additionally, there are other model-based simulation paradigms, such as total envisionment (which generates all possible states and state transitions of the system [Forbus, 1984]) and attainable envisionment (which generates the possible behavior histories of a system from a given state [Kuipers, 1986]).

This report provides an overview of the functionality of EDSE and the model-based event-driven simulation paradigm on which the simulator is based. It is intended to describe at a high level the key properties and algorithms of EDSE. Supplementary information about building causal models with the modeling language EDSEL can be found in [Charest, 1992].

The rest of this report is organized as follows: Section 2 describes the overall architecture of the simulator. Section 3 describes synchronization and filtering techniques relevant to systems monitoring. Section 4 outlines how EDSE stands with respect to open issues of discrete event simulation and model-based reasoning. Section 5 summarizes the main features of EDSE.

2 Architecture of EDSE

EDSE allows SELMON to make predictions about the future performance of a monitored system by applying current data values to a model of the system. Specifically, the simulator uses:

- A behavioral model of the monitored system
- The current state of the monitored system at time T
- A future time T'

to determine the predicted state of the monitored system at time T'. More specifically, the state of the monitored system is represented by two types of information:

1. A set of *quantities* representing the current values for physical quantities at various locations in the system.

2. An *agenda* of *events* which represent changes in quantities that are expected to occur in the future.

Briefly, an event is a 5-tuple $e = (m, q, v, \phi, \tau)$ describing that a mechanism m fired at time ϕ to set a quantity q to a value v at time τ. An event is said to be *created* at time ϕ and *matured* at time τ. An agenda is simply a queue ordered by increasing values of τ. Mechanisms are described below.

With respect to the event-driven simulation paradigm [Banks & Carson, 1984], the simulated objects are quantities in EDSE. Message events are restricted to passing from one quantity object to another as long as the latter quantity interacts with the former via a mechanism. The application of EDSE to a specific physical system requires the definition of a causal model – that is, a network of quantities interacting via mechanisms (see [Charest, 1992] for details). The mechanism interactions implicitly define all of the possible events that can occur during simulation.

The simulation process is characterized by two phenomena: *event consumption* and *event creation*. Event consumption occurs when the current time matches the time of one or more events on the agenda. In this case the appropriate quantities are set to the values prescribed by the event(s). Event creation occurs when quantities in the model change. When a quantity in the model changes, all mechanisms associated with the quantity are evaluated and thereby may cause new events to be added to the agenda. Event consumption therefore induces event creation, and vice versa. In general, simulation consists of quantity changes, followed by mechanism evaluation (event creation), followed by more quantity changes (event consumption), and so on.

Interactions among quantities are modeled by *mechanisms*. A mechanism is a 4-tuple $m = (I, \gamma, \delta, q)$ describing how a set of input quantities I determines the value of an output quantity [1] q via the transfer function γ and delay function δ. Thus, the mechanism specifies when (through I and δ) and how (through I and γ) changes will occur for a quantity q.

During the event consumption phase of simulation, if a quantity q changes value then all the mechanisms that use q as an input will be *triggered* for the next event creation phase. That is, each mechanism $m' = (I', \gamma, \delta, q')$, where $q \in I'$, will be marked in such a way that an event will be created for the mechanism m' during the ensuing event creation phase. An event e is created for each triggered mechanism by evaluating or *firing* the functions γ and δ with respect to input quantities I and the simulation time ϕ. When these functions are fired the mechanism itself is also said to be fired. The resulting event e is described by the 5-tuple (m, q, v, ϕ, τ) where v is the result of firing γ and τ is the result of firing δ.

[1] To simplify explanation in this report, we assume a single output quantity for each mechanism. The newest version of EDSE allows multiple output quantities and conditional mechanisms. For practical purposes, providing multiple output quantities on a single mechanism allows the same behavior as would constructing an identical mechanism for each output quantity. Conditional mechanisms specify conditions over the set of input quantities I. These conditions must be satisfied in order for the mechanism to be fired.

2.1 Event Consumption and Event Creation

The simulation is defined by the event consumption and creation processes. We now focus upon the details of these two processes. This section describes when event creation and event consumption occur in the simulation algorithm.

Event consumption

An event $e = (m, q, v_{new}, \phi, \tau)$ is consumed when its maturation time τ coincides with the current simulation time T. The processing of e consists of replacing the old value $v_{old} = \mathcal{V}(q, T-1)$ with the new value v_{new} indicated by the event. If v_{old} is different from v_{new} in the appropriate quantitative or qualitative sense then the event is propagated by triggering every mechanism m such that q is an input quantity of m. Figure 1 is a pseudo-code description of the event consumption algorithm.

Event creation

Events are created when the simulator fires the triggered mechanisms. At that time, the transfer function γ and delay function δ of each triggered mechanism $m = (I, \gamma, \delta, q)$ are evaluated to create an event $e = (m, q, v, \phi, \tau)$ where ϕ is the current time, $\tau = \delta(I, \phi)$, and $v = \gamma(I, \phi)$. The maturity of this event is either delayed until τ or immediate, according to the type of delay function δ. Figure 2 is a pseudo-code description of the event creation algorithm.

There are two types of mechanism delay functions: *explicit zero* and *implicit future*. An implicit future delay function means that the evaluation of the delay function will yield a positive event delay. Currently, the event consumption algorithm forces all implicit future delay functions to produce values that are at least one simulation timestep later than the current simulation time.[2] This restriction allows the simulator to determine when event consumption is complete for the current simulation time and therefore when the event creation phase can start.

An explicit zero delay function means that whenever a mechanism is triggered at a given simulation time T the mechanism will also be fired at T. Practically, this means that during the event consumption phase, quantity value changes can lead some mechanisms to be triggered and then immediately fired, thus interleaving event consumption and event creation within a single simulation timestep.

2.2 Event Processing

The simulator is started at a time T_0, with the directive to run iteratively to a new time T_N. The simulator consumes and creates events on a global agenda. Figure 3 is a pseudo-code description of the algorithm for processing one timestep during simulation. For each timestep T such that $T_0 \leq T \leq T_N$, EDSE performs the following actions:

[2]Implementation note: Each simulation timestep corresponds to 1 second of real time for the simulated system.

```
Given   an agenda of events A
        and the current simulation time T

        Let M = ∅
        For each event e ∈ A,
            Let e = (m, q, v_new, φ, τ)
            Let v_old = V(q, T − 1)
            If v_new ≠ v_old then
                Assign V(q, T) the value v_new
                For each mechanism m' = (I', γ, δ, q') where q ∈ I'
                    Trigger m'
                    Add m' to M
        Return the set of triggered mechanisms M
```

Figure 1: Event consumption in EDSE.

1. Consume all events scheduled to mature during the current timestep. This action includes triggering mechanisms whose input quantities have changed.
2. For each explicit zero delay mechanism in the set of triggered mechanisms, create an event that will mature during the current timestep and immediately fire the transfer function to compute the new quantity value.
3. If there are still events scheduled to mature during the current timestep then go to step 1. Otherwise, go to step 4.
4. For each implicit future delay mechanism in the set of triggered mechanisms[3] fire the transfer and delay functions and then create an event that matures according to the result of the delay function.
5. Schedule the new events according to their maturity.
6. Increment the simulation time[4] T.

3 Synchronization and Filtering

It is difficult to ensure that a causal model used with EDSE will predict the behavior of the physical system with perfect accuracy. This lack of model perfection manifests itself

[3]Technically, at this point in the algorithm *all* of the triggered mechanisms must be implicit future delay mechanisms.

[4]The next simulation time is defined by the maturation time of the next event (if any) in the agenda.

```
Given   a set of triggered mechanisms M
        and an agenda of events A
        and the current simulation time T

        For each mechanism m ∈ M
            Let m = (I, γ, δ, q)
            Fire γ and assign the result to v
            Fire δ and assign the result to τ
            Schedule the event e = (m, q, v, T, τ) on A
```

Figure 2: Event creation in EDSE.

```
Given   an agenda of events A
        the current simulation time T

        Let M = ∅
        Let A_T ⊆ A such that e ∈ A_T
            is an event scheduled to mature at T
        While A_T ≠ ∅
            Remove an event e from A_T
            Consume the event e
            Collect the resulting triggered mechanisms into M
            Let M_T ⊆ M such that m ∈ M_T
                is an explicit-zero delay mechanism
            For each mechanism m ∈ M_T
                Create an event e = (m, q, v, T, T)
                Add e to A_T
        For each mechanism m ∈ M − M_T
            Create an event e = (m, q, v, T, τ) where τ > T
```

Figure 3: Processing a single timestep in EDSE.

as discrepancies between actual sensor data and the model predictions. Left unchecked, these discrepancies may grow so large as to make model predictions useless. In order to avoid this phenomenon, the SELMON simulator periodically *synchronizes* the simulation state with observed sensor values.

Synchronization is a method of aligning the values of quantities in the model with observed sensor values. That is, we explicitly assign the observed sensor values to the appropriate quantities. This method assumes that we have reasonable confidence in the sensor data. As an alternative, arbitrary user-specified functions can be used to access past sensor values and values of other sensors to determine the synchronized value. This synchronization occurs immediately before we begin the event processing algorithm described in Figure 3.

Unfortunately, synchronization in SELMON is incomplete in the sense that the model state is not completely synchronized. Only the quantity values are set to their corresponding sensor values. Ideally, events in the agenda would also be recomputed to reflect the new sensor readings.

Another difficulty with synchronization is sensor noise. If sensor noise causes a sensor to incorrectly report an anomalous value, synchronization may cause the model to predict discrepant behaviors. In order to minimize this phenomenon, the simulator provides a *filtering* protocol to decrease the impact of spurious sensor readings for particularly noisy sensors.

The filtering protocol allows each sensor s that is associated with a quantity q in the model to have its data values explicitly filtered as they are observed (i.e., before synchronization). Filtering is accomplished by means of a function which transforms a "raw" data value to the appropriate "filtered" value. Filter functions may perform arbitrary transformations of raw data values, using the entire history of raw values for a particular sensor as input. Typically, filtered values are the result of calculating a weighted average of the last few raw data values.

The filtering protocol implemented in EDSE is subject to the following caveats:

- If a sensor does not report a raw value, the sensor filter will generate a filtered value for the sensor anyway. It has been noted that only those sensors which have reported a value should be used to synchronize the model.

- Consider a model where a quantity q_1 is associated with a sensor s_1 and q_2 is associated with a sensor s_2. If there is a zero-delay mechanism $m_{1,2}$ from q_1 to q_2, it has been noted that the mechanism $m_{1,2}$ should not be used to infer a new value for q_2 but rather the value of s_2 should be used. This is consistent with the principle of "trusting" the data as much as possible.

4 Open Issues

EDSE is subject to the open issues typical of discrete event simulation in general and model-based simulation in particular. This section describes what these issues mean for EDSE and, when applicable, what steps can be taken to address them.

Race conditions

A race condition occurs when two events e_1 and e_2 occur at the same simulation time T and they both change the same quantity q. Which of the two event values (v_1 from e_1 or

v_2 from e_2) should be used to determine the value of the quantity $\mathcal{V}(q,T)$? Alternatively, should e_1 and e_2 be fused to form a new event $e_{1,2}$ such that $e_{1,2}$ would be used instead of e_1 and e_2 to change q? Such race conditions can potentially occur with implicit future and explicit zero delays in any combination.

Currently, events are processed iteratively, one at a time (i.e., the simulation is sequential). This means that $\mathcal{V}(q,T)$ will be the value of the last event occurring at T that affects q. This is a source of indeterminism in the simulation. In some cases of zero-delay events, the race condition can be avoided by imposing constraints on the model (see below).

A better solution would fuse all of the events that occur at the same time and affect the same quantity. This poses two problems. First, race conditions are generally unpredictable; therefore, when one is detected, it may be necessary to "undo" some computations, solve the race condition, and proceed again. Research in operating systems addressed this problem by introducing the concept of virtual time [Jefferson, 1985; Jefferson et al., 1987], where the operating system allows independent processes to pursue their computations, even though enforcing global consistency may force them to backtrack and resume from an earlier state.

Second, race conditions are typically an artifact of the modelling language: some events are defined to occur simultaneously because they are at the smallest granularity of the time scale. A smaller set of race conditions comes from events occurring at different instants but processed at the same time because of discretization. In either case, the resolution of race conditions ultimately depends on a physical interpretation of the underlying phenomena. In practice, we expect that prioritizing events (see below) will be sufficient in most cases.

Future events

When the delay function of an implicit future delay mechanism evaluates to zero, the mechanism should be treated as an explicit zero delay mechanism. Currently, this issue is avoided since implicit future delays are forced by the simulator to be at least as large as one timestep. For the simulation of physical systems, this is not an issue as long as a minimum time scale can be attributed to the underlying continuous physical phenomena being modeled.

Zero-delay events

From a modeling standpoint, there are roughly two classes of mechanisms used to model a physical system. The first class encodes the dynamics of the continuous physical phenomena occurring in the system. The second class encodes the discrete control changes that occur in the system. The former class of mechanisms typically has implicit future delays since continuous physical phenomena take some time to propagate or produce any effect. The latter class of mechanisms typically has explicit zero delays since control actions are discrete and instantaneous relative to the continuous physical phenomena. However, there is an implicit sequentiality in the simulation of explicit zero events due to the actual propagation of these events in the network of mechanisms. This sequential propagation of zero-delay events defines an artificial time scale that has no physical support; it is an example of "mythical time"[de Kleer & Brown, 1984] in qualitative simulation.

The simulator introduces mythical time as it processes all the zero delay events in

some order before processing any future delay event. Currently, there is no provision in the simulator to provide the user with control over the ordering of zero-delay events; nevertheless, they are processed on a first-come first-served basis. To avoid race condition problems, the modeling language may be augmented to force an *a-priori* order of processing for mythical time. One approach would be to construct a list of zero-delay event types defining a partial ordering in mythical time. This partial ordering would be enforced by the simulator such that all zero-delay events of higher priority would be processed before zero-delay events of lower priority until quiescence.[5] This partial ordering of zero-delay events does not solve the problem entirely: race conditions can still occur with zero-delay events of the same priority. However, it does prevent race conditions across priority levels. To that extent, it may be sufficient for a modeler to use this partial order to avoid race conditions.

Causal consistency

To obtain accurate model predictions, it is necessary that the model reflect in its state information the actual physical processes occurring in the physical system. Noisy data and numerical errors, in particular, are typical factors leading to errors in state information. Since the local scope of the models is narrowed down to components, such errors in state information can result in globally inconsistent states.

For example, a fluid pipe is not always modelled as one component. Take the case when a membrane pressure sensor is mounted on such a pipe. Then, the global model of the pipe consists of three pipe models, and the middle one doubles as a pressure sensor model. In these circumstances, a flow of small magnitude through the global pipe model can well be tainted with round-off errors such that the flows in each of the three pipes are not in the same global direction; this is a clear case of global model inconsistency despite locally consistent models of the constituent components.

Recently, Forbus and Falkenhainer [1992] addressed this issue by adding a self-monitoring module to their numerical simulators. The task of the self-monitoring module is to detect and steer the simulation away from globally inconsistent states.

5 Summary

The SELMON simulator EDSE uses an event-driven paradigm to generate predictions of future physical system performance from a model of the physical system and information about the current system state. These predictions can be used to detect potential discrepancies or to determine if future system behavior will exhibit critical qualities.

Model drift can cause a reasonably faithful model to eventually diverge from actual system behavior over time. In order to deal with this issue, EDSE synchronizes the model with observed sensor data. Also, filtering is used to deal with noisy data in some cases. Several problem issues remain open in the current EDSE implementation.

[5]Here, quiescence at a given priority means all zero-delay events of that priority have been processed and that all the zero-delay events created in the meantime have a lower priority.

6 Acknowledgments

The authors, members of the Artificial Intelligence Group in the Advanced Information Systems Section, Information Systems Division, Jet Propulsion Laboratory, would like to thank Richard Doyle, Harry Porta, and Matthew Presley for useful comments on an early draft. The work described in this report was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

Banks, J., & Carson, J. [1984]. *Discrete-event system simulation.* Prentice Hall.

Charest, L. Jr [1992]. Building causal models for SELMON: A user's manual for the EDSEL modeling language. JPL D-10423 (internal document), Jet Propulsion Laboratory, California Institute of Technology. Forthcoming.

Chien, S. A., Doyle, R. J., & Fayyad, U. M. [1992]. Focusing attention in real-time systems monitoring. In *AAAI Spring Symposium on Selective Perception*, Stanford, CA.

de Kleer, J., & Brown, J. S. [1984]. A qualitative physics based on confluences. *Artificial Intelligence, 24*, 7–83.

Doyle, R., Berleant, D., Charest, L. Jr, Fayyad, U., Homem de Mello, L., Porta, H., & Wiesmeyer, M. [1992]. Sensor selection in complex system monitoring using information quantification and causal reasoning. In Faltings, B., & Struss, P. (Eds.), *Recent Advances in Qualitative Physics*, pp. 229–244. MIT Press.

Doyle, R. J., Sellers, S. M., & Atkinson, D. J. [1989]. A focussed context-sensitive approach to monitoring. In *Proceedings of the Eleventh Joint Conference on Artificial Intelligence*, pp. 1231–1237.

Dvorak, D., & Kuipers, B. [1989]. Model-based monitoring of dynamic systems. In *International Joint Conference on Artificial Intelligence*, pp. 1238–1243. Morgan Kaufmann.

Forbus, K. D. [1984]. Qualitative process theory. *Artificial Intelligence, 24*, 85–168.

Forbus, K. D., & Falkenhainer, B. [1992]. Self-explanatory simulations: Scaling up to large models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*. Morgan Kaufmann.

Hayes, P. [1989]. The second naive physics manifesto. In Weld, D. S., & de Kleer, J. (Eds.), *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann. Originally appeared in J. Hobbs and R. Moore (Eds.), *Formal Theories of the Commonsense World*, Ablex, 1985.

Jefferson, D., et al. [1987]. Distributed simulation and the time-warp operating system. In *Proc. of 11th Annual Symposium on Operating Systems Principles*, pp. 77–93.

Jefferson, D. [1985]. Virtual time. *ACM Topics on Operating Systems and Languages, 7*(3), 404–425.

Kuipers, B. J. [1986]. Qualitative simulation. *Artificial Intelligence, 29*, 289–338.

Appendix: EDSE Source Code

The following program listing presents the Lisp source code for a possible implementation of EDSE. The function `simulate` is the sole entry-point to the simulator. All of the necessary data structure definitions (e.g., `event`, `agenda`) and some of the support function definitions have been omitted in the interest of space.

```
(defun current-event (agenda timestamp)
  "Return the first event on the agenda if it is scheduled
   to happen on or after the given timestamp."
  (declare (type agenda agenda)
           (type fixnum timestamp)
           (values (or event nil)))
  (loop for current-event in (events-of agenda)
        if (null-event-p current-event)
        do (error "The agenda is empty.")
        if (< (maturity current-event) timestamp)
        do
        (cerror "Ignore the mis-scheduled event."
                "Current time is ~a. ~@
                 Event ~a is scheduled to occur at ~a."
                timestamp
                (name current-event)
                (maturity current-event))
        (warn "Skipping event ~a."
              (name current-event))
        (pop (events-of agenda))
        (deallocate-event current-event)
        else return current-event
        finally (error "The agenda is empty.")))

(defun split-agenda (agenda timestamp)
  "Split the agenda with respect to the given timestamp.
   Return two subagendas: events maturing at timestamp
   and events maturing after timestamp."
  (declare (type agenda agenda)
           (type fixnum timestamp))
  (loop with mature-events = (events-of agenda)
        initially
        (unless (equal* (maturity (first (events-of agenda)))
                        timestamp)
          (error "Malformed agenda: the maturity of the ~
                  first event was expected to be ~d."
                 timestamp))
        for split on (events-of agenda) by #'cdr
        as (first second . rest) = split
        while (and second (equal* (maturity second) timestamp))
        finally
```

```
        (shiftf (events-of agenda) (rest split) nil)
        return (values (make-agenda mature-events) agenda)))

(defun schedule (pending-events agenda)
  "Merge the pending-events into the agenda."
  (declare (type agenda pending-events agenda)
           (values agenda))
  (merge 'list
         (events-of agenda)
         (sort pending-events #'< :key #'maturity)
         #'< :key #'maturity)
  agenda)

;;; Pending-mechanisms are passed in as a performance hack:
;;; PUSHNEW can be used to optimally add only those
;;; mechanisms which are not already on the list.
(defun mature (agenda pending-mechanisms timestamp)
  "Pass the value of each event to the appropriate quantities
   and mark as pending all of the output mechanisms from
   those quantities."
  (declare (type agenda agenda)
           (type pending-mechanisms)
           (type fixnum timestamp)
           (values immediate-mechanisms pending-mechanisms))
  (loop with immediate-mechanisms = nil
        for event in (events-of agenda)
        do (loop for quantity in (outputs (mechanism event))
                 as change-p = (not (equal* (value quantity)
                                            (value event)))
                 when change-p
                 do (loop for mechanism in (outputs quantity)
                          if (immediate-mechanism-p mechanism)
                          do (pushnew mechanism
                                      immediate-mechanisms
                                      :test #'eq)
                          else do (pushnew mechanism
                                           pending-mechanisms
                                           :test #'eq))
                 ;;unconditionally
                 do (setf (value quantity) (value event)
                          (timestamp quantity) timestamp))
        finally return (values immediate-mechanisms
                               pending-mechanisms)))

(defun propagate (pending-mechanisms timestamp)
  "Create and return a list of events from the given
   pending mechanisms."
```

```
  (declare (type list pending-mechanisms)
           (type fixnum timestamp)
           (values pending-events))
  (loop for mechanism in pending-mechanisms
        when (compute-context-match mechanism timestamp)
        collect (allocate-event mechanism timestamp)))

(defun simulate-immediate-events (agenda timestamp)
  "Consume all of the events on the agenda which mature at
   the given timestamp."
  (declare (type agenda agenda)
           (values agenda pending-mechanisms))
  (loop with immediate-agenda
        and immediate-mechanisms
        and pending-mechanisms = ()
        and pending-events
        initially
        (multiple-value-setq (immediate-agenda agenda)
          (split-agenda agenda timestamp))
        do
        (multiple-value-setq
            (immediate-mechanisms pending-mechanisms)
          (mature immediate-agenda pending-mechanisms
                  timestamp))
        (deallocate-agenda immediate-agenda)
        while (and immediate-mechanisms
                   (setq pending-events
                         (propagate immediate-mechanisms
                                    timestamp)))
        do (setq immediate-agenda
                 (make-agenda pending-events))
        finally return (values agenda pending-mechanisms)))

(defun simulate (instant limit disposition)
  "Consume events from the global *agenda* until the
   limit is reached."
  (let ((timestamp (time instant))
        (iterations 0))
    (flet ((time-limit ()
             (> timestamp limit))
           (iterations-limit ()
             (> iterations limit)))
      (loop with limit-p = (ecase disposition
                             (:time #'time-limit)
                             (:iterations #'iterations-limit))
            and pending-mechanisms
            as event = (current-event *agenda* timestamp)
```

```
while event
do
(setq timestamp (maturity event))
(incf iterations)
until (funcall limit-p)
do
(multiple-value-setq
    (*agenda* pending-mechanisms)
  (simulate-immediate-events *agenda*
                             timestamp))
(schedule (propagate pending-mechanisms
                     timestamp) *agenda*)
finally return *agenda*))))
```