Regulated Isomorphic Temporal Architecture

January 26, 2000

Version 2.0

Author Information:

Richard Wallace
INFORMATION CENTERED NETWORK SOLUTIONS

RECORD OF CHANGES

Change Number	Date	Brief Description	Entered By
1	11/12/1999	Document creation.	RMW
2	12/28/1999	Added detail for algorithms from Section 2.3 onward. Issued as V1.0.	RMW
3	01/26/2000	Corrected Figure 2, added detail to text after Figure 11, added Section 3 and its subsections. Issued as V2.0.	RMW

1 ABSTRACT

In an event driven system, there are two kinds of data. First is "information," as it labels the data as having relevant importance to a computational process. Second is "superfluous," as it labels the data as having no value to a computational process. The importance of *information* to a computational process is expressed as time– summation–, or delta–critical. Time–critical information must reach a computational process at fixed delta times. Summation–critical data is summed to form a value that triggers events. Delta–critical data must be significantly different from its previous value in order to trigger an event. By use of the conditional matrix vector evaluations, and the guard matrix, it is possible to control events by time, summation, and delta significance without generating superfluous data transmissions. This "data driving" event service can drop network loading by order of magnitude by reducing superfluous data transmissions.

2 ARCHITECTURE

2.1 Event Theory

Events occurring in any system follow the canonical event forms in Figure 1.

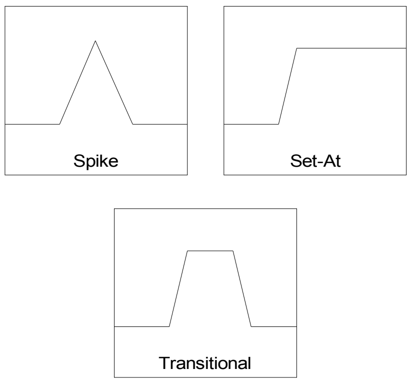


Figure 1 Canonical Event Forms

The "spike", "set-at", and "transitional" forms are the canonical event forms. Each form occurs under some change in time (Δt) applicable to the event space in which it occurs. The use of Δt by these forms assumes that time is infinitely divisible, countable, and continuously and monotonically increasing. Thus, any incremental units of measurement are only sample points of the time continuum. Events may or may not have a "lifetime" associated with their occurrence. An event's lifetime is defined by the semantic meaning of its latency of occurrence and its subsequent observance. All events shall be discrete events.

The spike event form is a singular event that occurs at a Δt of asymptotic zero. Events of this form are considered periodic "heartbeat" events. This event form can be used for counted threshold limits, keep-alive notification, or request for service. This event form can not be queued and has no lifetime.

The set-at event form is a permanent state change that has a long Δt in the new state. This event form can be used for one-time events. This event form has an infinite lifetime and can be queued.

The transitional event form occurs over some period of measurable Δt and an event sensitive entity can perceive state change. This event form is semantically dense as each sampling of Δt can result in a different state. The semantic meaning is compounded based on the frequency of state change and for how many sampling intervals of Δt the state remains constant. This event form has a limited lifetime. Queuing of this event form requires prioritized time-based queuing that can be dynamically edited as higher priority events occur, or as queued events expire, or both.

These three event forms shall be used to describe how events are produced and consumed.

2.2 Event Model

All producers and consumers of events in a data driven event service will follow the following event model for the canonical event forms.

For all events, given event ***E***, condition ***C***, evaluation matrices, guard ***G***, and action ***A*,** Figure 2 illustrates the general stimulus flow model. The model has as input one to **N** inputs to a precondition-event matrix (see Equation 1) that causes an **Action**. The result of this action is input to a postcondition-event matrix (see Equation 1) that may chain to a precondition-event matrix as the output of the postcondition-event matrix may be zero to **N** events. Events leave the general stimulus flow model when they are fully consumed. At the point of consumption, an event becomes a datum germane to the application controlled by events. Each condition–event matrix has one to many resultants. The number of resultants is determined by the number of condition vectors (see Equation 3). Resultants are allowed to be evaluated by conjunction (see Equation 4). This conjunction allows consolidation of many input events to one event. For each of the canonical event forms a specific precondition and postcondition equation is used to evaluate events in fidelity to their canonical form.

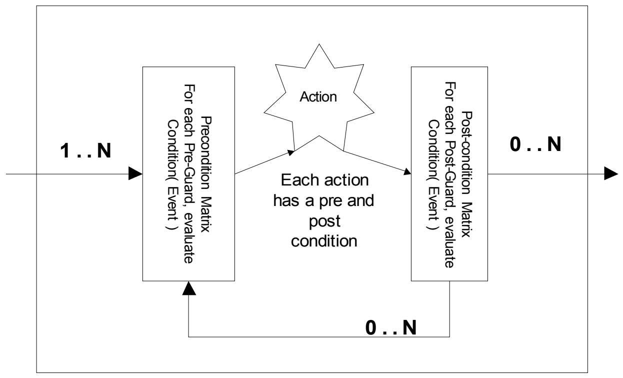


Figure 2 Event Stimulus Flow

All events in a system are known events and are thus named by a universal unique identifier[1] (UUID) used to vector an event to the proper condition–evaluation matrix. Each vector (see Equation 3) is associated with a UUID. Each condition–event matrix has the genotype shown in Equation 1, but each condition for each vector.

All events have the following genotype as shown in Figure 3 where the UUID identifies the event mapping to a condition. The Variable Data part of the event genotype is defined by the application domain and is mapped to the specific condition(s) that via the event UUID. The Variable Data value has a known format and is known prior to construction of the condition(s) mapped to an event UUID. For spike events, no data is required, as the event itself is the data.

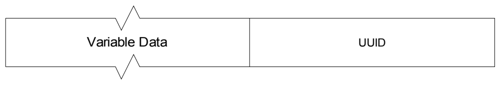


Figure 3 Event Genotype

[1] A universal unique identifier (UUID) is a concept of using a large binary value (128 bits is typical) which combines the hardware address, machine time, and a naming convention intended to ensure the UUID will be unique across all instances, systems, and geographic locations. This is a common technique used in industry.

2.2.1 Condition–Event Matrix Relationship

Each event matrix *M* has the following specification:

$$\begin{bmatrix} G(V)_1 \\ \vdots \\ G(V)_n \end{bmatrix} \bullet \begin{bmatrix} C_1(E_1) & \cdots & C_k(E_1) \\ \vdots & & \vdots \\ C_n(E_q) & \cdots & C_{n,k}(E_q) \end{bmatrix} \bullet \begin{bmatrix} op_{1,1} & \cdots & op_{1,k-1} \\ \vdots & & \vdots \\ op_{n,1} & \cdots & op_{n,k-1} \end{bmatrix} \rightarrow \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix}$$

Equation 1 Condition-Event Matrix

$$G : G \in C$$

Equation 2 Guard Elements

$$V : (C(E)_{1\ldots k})$$

Equation 3 Condition Vector

$$[R_1 \quad \cdots \quad R_n] : R_1 \wedge R_2 \wedge \ldots R_n \rightarrow True|$$

Equation 4 Resultant Conjunction

Each event matrix *M* is a matrix of event conditions and operations that are applied to those conditions. Each vector *V* of conditions is gated by a guard *G* such that the vector is evaluated if and only if the guard evaluates to *True*. Each evaluated vector has a resultant *R* that can be further reduced by conjunction to *True*. The guard is evaluated first and if *True* then the vector is evaluated. Optionally the series of resultants are evaluated. If *G*, *V*, and optionally *R*, evaluate to *True*, the action is initiated. The matrix can be sparse or dense depending on the condition interactions in the event matrix. Conditions can be compounded by applying a Boolean operation matrix comprehended across the event matrix. Evaluation is by vector with the result being true or false as illustrated thus:

$$\underline{C_1(E_1) \quad op_{1,1} \quad C_2(E_1) \quad op_{1,2} \quad C_3(E_1) \quad \cdots \quad op_{1,k-1} \quad C_k(E_1) \quad \rightarrow True}$$

Equation 5 Realization of Conditions and Operands

Where each *op* can have the logical meaning of **AND**, **OR**, **XOR**, or **NOT**.

In a sparse matrix only cells having conditions are evaluated, else the cell is considered logically *NULL* for evaluation. If any *op* occurs between or adjacent to a logically *NULL* cell; the operation is considered *NULL* as well.

2.2.2 Spike-Event Consumers

For consumers of spike event forms the precondition matrix specification is:

$$\exists E : \forall C_{pre}(E) \rightarrow True$$

Equation 6 Spike-Event Precondition

There exists some event *E* where all pre-conditions *C* on event *E* result in *true*.

The post-condition matrix specification is:

$$\exists C_{pre}(E) \rightarrow True \therefore \exists A(E) \Rightarrow \exists E' \ni C_{post}(E') \rightarrow True$$

Equation 7 Spike-Event Postcondition

There exists some pre-condition *C* on event *E* that results in *true*; Therefore there exits an action *A* for event *E* which implies that there exists some event *E'* such that the post-condition on event *E'* is *true*.

2.2.3 Set-At Event Consumers

For consumers of set-at event forms the precondition matrix specification is:

$$\exists E : C_{pre}(E) \rightarrow True\langle \forall \Delta t \rangle$$

Equation 8 Set-At Event Precondition

There exists some event *E* where the pre-condition *C* on event *E* results in true over the sequence of all increments of time after some initial time. The pre-condition is *False* during the initial time until the time at which the condition is evaluated for the event *E*.

The post-condition matrix specification is ∅ (Null). Event reset by an external source is required to change the state of this event.

2.2.4 Transitional Event Consumers

For consumers of transitional event forms the precondition matrix specification is:

$$\exists E : C_{pre}(E) \rightarrow True\left(\langle \Delta t_1 \ldots \Delta t_n \rangle \in \forall \Delta t\right)$$

Equation 9 Transitional Event Precondition

There exists some event *E* where the pre-condition *C* on event *E* results in true for a sequence of time that is a subset of all time.

The post-condition matrix specification is:

$$\exists C_{pre}(E) \rightarrow True \therefore \exists A(E) \bullet \langle \Delta t_1 \ldots \Delta t_n \rangle \rightarrow True \Rightarrow \exists E' \ni C_{post}(E') \rightarrow True$$

Equation 10 Transitional Event Postcondition

There exists some pre-condition *C* on event *E* that results in *True*; Therefore there exists an action *A* for event *E* resulting in *True* for a sequence of time implying the existence of an event *E'* such that the post-condition on event *E'* is true.

2.2.5 Event producers

Each post-condition specification in Sections 2.2.2, 2.2.3, 2.2.4 for each of the event forms are the specifications for event producers respective to each event form.

2.3 Event Engine Components

The event engine is comprised of the following software modules:

- Event Receipt/Dispatch Object Modules
- Condition-Event Object Modules

The event receipt and dispatch modules are available from a series of message queue vendors and the internal detail is not included in the event engine component design. Figure 4 is the top-level software block diagram showing the user inputs, object module creation and eventual event engine executable. The shaded symbols in Figure 4 are the modules directly used by the event engine.

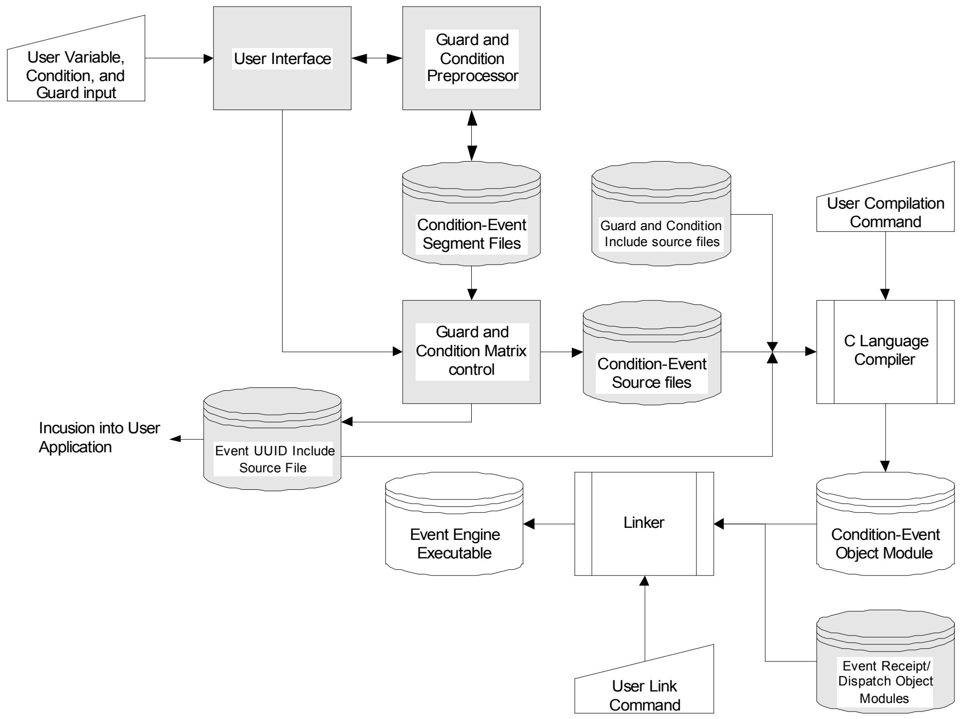
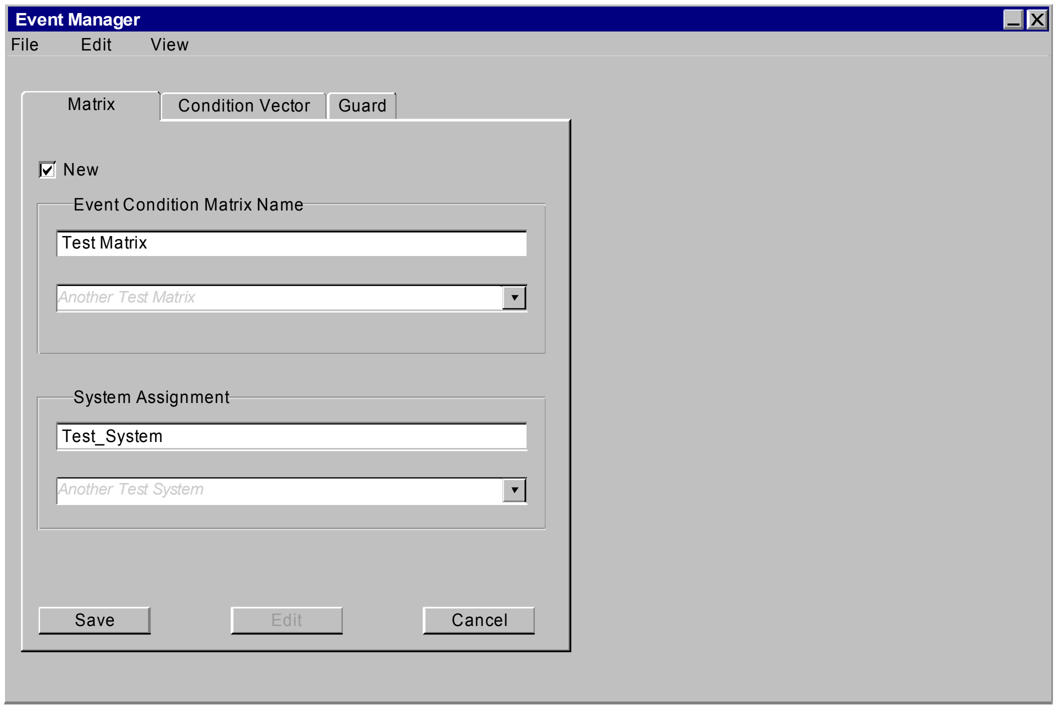

User Variable, Condition, and Guard input
User Interface
Guard and Condition Preprocessor
Condition-Event Segment Files
Guard and Condition Include source files
User Compilation Command
Guard and Condition Matrix control
Condition-Event Source files
C Language Compiler
Incusion into User Application
Event UUID Include Source File
Event Engine Executable
Linker
Condition-Event Object Module
Event Receipt/ Dispatch Object Modules
User Link Command

Figure 4 Event Engine Top Level Software Block Diagram

2.3.1 User Interface

In order to build-up the condition-event matrix shown in Equation 1 the following screen-shots detail the how Equation 1 is constructed.



[✓] New

Event Condition Matrix Name: Test Matrix

Another Test Matrix

System Assignment: Test_System

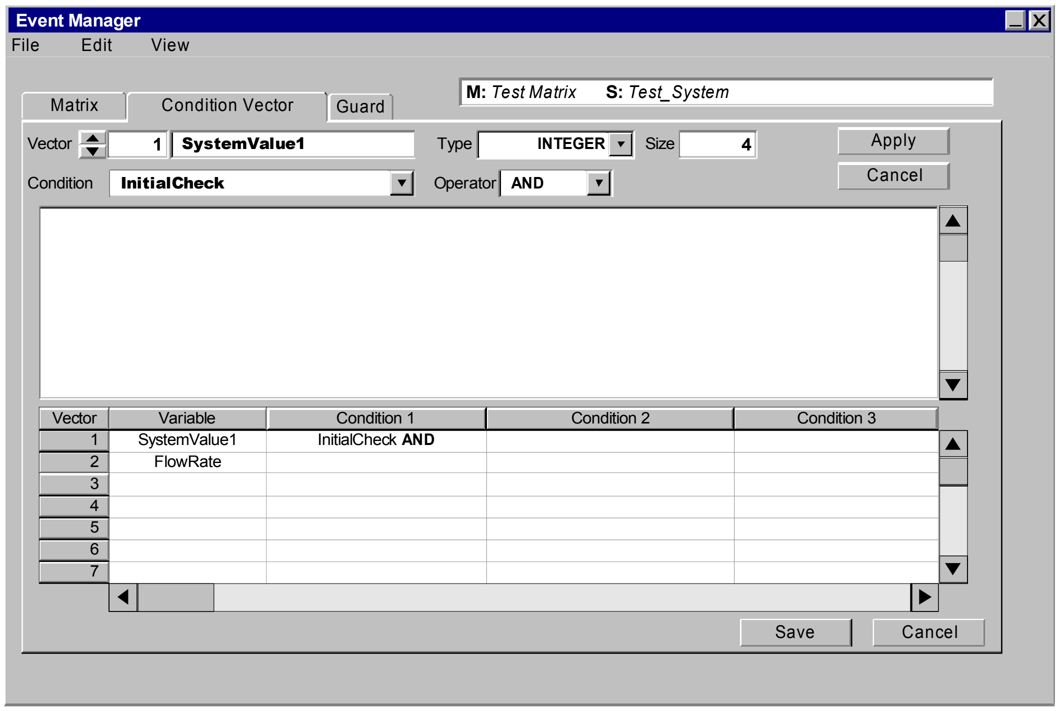
Another Test System

Figure 5 Event Manager Main Window

The Event Manager main window in Figure 5 shows the user interface after start-up for a new condition-event matrix. The user checks the "New" box and only the Matrix Name and System Assignment name fields are active. All other controls for the Matrix tab control are inactive. The user can choose between saving this assignment or canceling with the "Save" or "Cancel" buttons respectively.

The event engine executable is the name assigned to the system; in Figure 5 the name of the event engine is called "Test_System" and "Test Matrix" is the name of the condition-event matrix. All names are case sensitive. The matrix name and vector variable are used to create the UUID name for events. This name is provides the name-space that is used to route events from publishers to subscribers.

In Figure 6 through Figure 11, the editing function of a created event matrix for a system is accomplished by selecting the name of the event condition matrix and system assignment from the drop-down boxes as shown in Figure 5 and making sure the "New" box is not checked. The "Edit" button will be active and the user will have access to the tab windows as shown in Figure 6 through Figure 11 with the appropriate data filled in from the user's previous session with the Event Manager interface.



M: *Test Matrix* S: *Test_System*

Vector 1 **SystemValue1** Type INTEGER Size 4

Condition **InitialCheck** Operator AND

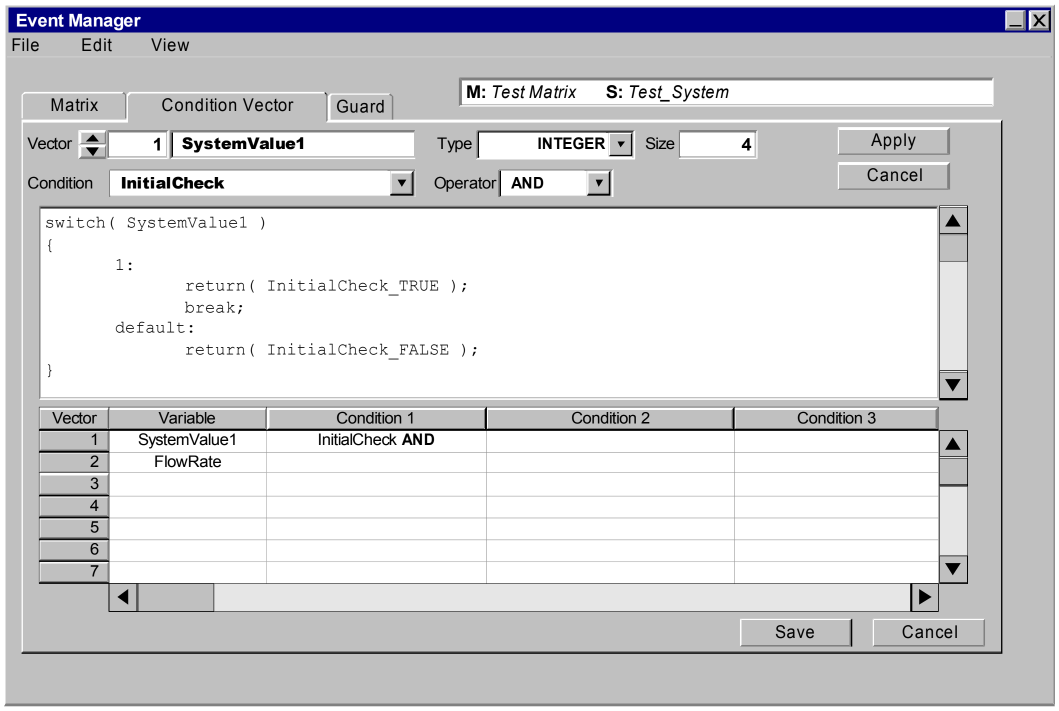
Vector	Variable	Condition 1	Condition 2	Condition 3
1	SystemValue1	InitialCheck **AND**		
2	FlowRate			
3				
4				
5				
6				
7				

Figure 6 Condition Vector Tab

In Figure 6 a great deal of the condition-event matrix is created. Working across the screen from upper-left to lower-right the following fields are shown. All data in the interface is an example only:

- Matrix and System name (i.e. "Test Matrix" and "Test_System")
- Vector and vector variable name (i.e. "1" and "SystemValue1")
- Type and octet[2] size of vector variable (i.e. "INTEGER" and "4")
- The name of the condition and the operator (i.e. "InitialCheck" and "AND")
- The ”Apply" and "Cancel" buttons for the creation of this condition vector
- The coding area for the condition "InitialCheck"
- The tabular interface to show the user what variables and conditions have already been created.
- The "Save" and "Cancel" buttons for the condition matrix "Test Matrix" for system "Test System"

[2] Eight bits. This term is used in networking, in preference to byte, because some systems use the term "byte" for things that are not 8 bits long.



```
switch( SystemValue1 )
{
        1:
                return( InitialCheck_TRUE );
                break;
        default:
                return( InitialCheck_FALSE );
}
```

Vector	Variable	Condition 1	Condition 2	Condition 3
1	SystemValue1	InitialCheck **AND**		
2	FlowRate			
3				
4				
5				
6				
7				

Figure 7 Condition Vector Tab with Condition Code

Figure 7 shows the same control information as does Figure 6 with the addition of the condition logic (expressed in the ANSI C language). The following constructs are allowed:

- IF-THEN
- IF-THEN-ELSE
- CASE-SELECT (i.e. switch). All switch statements must be inclusive of all values passed by the Guard condition.
- The relational operators less than (<), greater than (>), equal(==), not equal (!=), less than or equal to (<=), and greater than or equal to (>=)
- Computational (i.e. stack) variables are allowed for addition (+), sign/subtraction (-), multiplication (*), or division (/)

No pointers or locally declared variables are allowed. Only Boolean values are allowed as return values. All return values must be of the form `<condition_name>_TRUE` or `<condition_name>_FALSE` for conditional computations.

Vector variable data that represents time values relies on three special time functions that ensure that the condition-event matrix does not have to rely on user manipulated time functions. The evaluate_time() function can be used in the Guard or Condition code. The create_time(), and get_time() functions can only be used in code external to the event engine. The event engine system time is constant throughout evaluation of the guard and its condition vector.

```
int evaluate_time ( <vector variable>, { BEFORE | AFTER | NOW }, <delta> )
```

Figure 8 Function evaluate_time()

The evaluate_time() function compares the vector variable, a UCT[3] time value with millisecond precision, with the UCT system time with the same precision from the system executing the event engine. The resultant of the function is a Boolean value of *TRUE* or *FALSE*. The special enumeration values of **BEFORE**, **AFTER**, and **NOW** are used to determine the comparison of the system time with the vector variable time. The **NOW** enumeration must have a delta of zero (0.0). This ensures that the comparison is explicit. The delta value is an numerical constant that is a positive floating point value with a precision of 1 millisecond for the number of seconds to apply to the vector variable for the comparison. Table 1 illustrates the usage of the evalueate_time() function.

[3] Coordinated Universal Time (UTC) is the international time standard. It is the current term for what was commonly referred to as Greenwich Meridian Time (GMT). Zero (0) hours UTC is midnight in Greenwich England, which lies on the zero longitudinal meridian.

Vector Variable Value (as an ASCII string)	Comparison	System Time Value (as an ASCII string)	Delta Literal Value (seconds)	Resultant
10:10:00.020	BEFORE	12:00:01.001	10.300	TRUE
10:10:00.020	AFTER	12:00:01.001	10.300	FALSE
10:10:00.020	NOW	10:10:00.020	0	TRUE
10:10:00.020	NOW	10:10:00.021	0	FALSE

Table 1 Example of evaluate_time()

```
void create_time ( *<vector variable> )
```

Figure 9 Function create_time()

The create_time() function is passed a pointer to the user supplied variable to hold the UCT time.

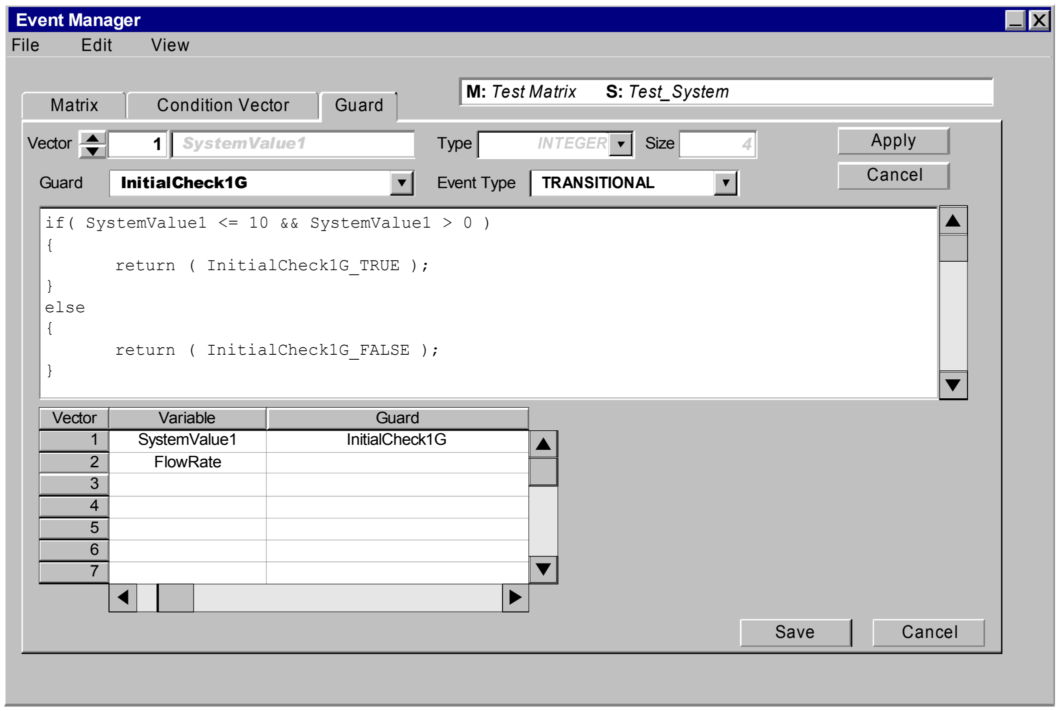
```
char* get_time ( <vector variable> )
```

Figure 10 Function get_time()

The get_time() function is used to examine the time value set in the create_time() function. The format of the character string returned is "HHHH:MM:SS.DDD" where:

HHHH	are hours
MM	are minutes
SS	are seconds
DDD	are milliseconds

Assignments to vector variables are allowed. The user is restricted to one and only one condition assignment in a vector. That assignment occurs after all conditions are evaluated in the vector and if and only if the vector evaluates to *TRUE*. Each vector variable can be used once and only once in a single event matrix. Vector variables can be used in multiple event matrices. Chaining of event matrices ensures that the event value is modified in a deterministic method, without race conditions, and that the event and its value is propagated through-out the system of condition-event matrices resulting in a final event consumption. If a user causes race conditions by having multiple matrices modify a vector variable in parallel, an error window will list the conflict. These restrictions ensure that each condition-event matrix will execute in the same sequence given the same event sequence and event values.



M: *Test Matrix* S: *Test_System*

Vector 1 *SystemValue1* Type *INTEGER* Size *4*

Guard **InitialCheck1G** Event Type **TRANSITIONAL**

```
if( SystemValue1 <= 10 && SystemValue1 > 0 )
{
        return ( InitialCheck1G_TRUE );
}
else
{
        return ( InitialCheck1G_FALSE );
}
```

Vector	Variable	Guard
1	SystemValue1	InitialCheck1G
2	FlowRate	
3		
4		
5		
6		
7		

Figure 11 Guard Tab

Figure 11 shows the logic for the vector Guard creation. A guard may be a condition previously designed and entered into the user interface. The name of the condition is the value used to identify it. In Figure 11, the guard is uniquely named for clarity of the example. Note that the system will check that the guard variable matches the vector variable to ensure consistency.

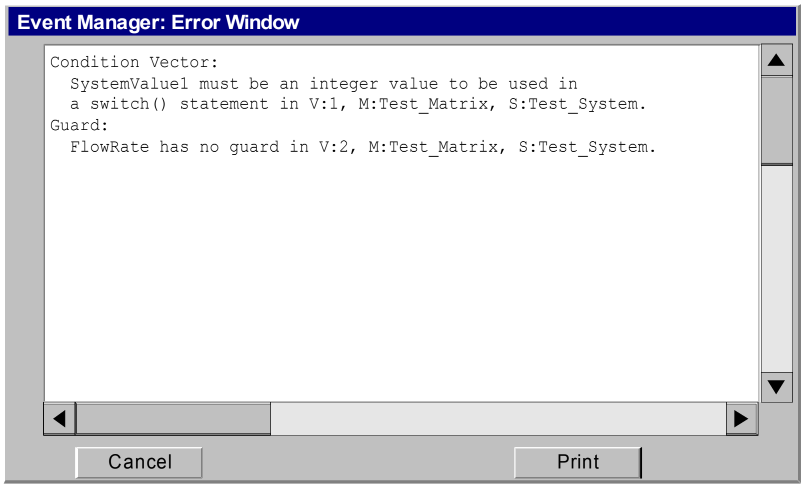
The new fields shown are:

- Guard name and event type (i.e. "InitialCheck1G" and "TRANSITIONAL")

The choice of SPIKE, SET-AT, or TRANSITIONAL event type is done for the guard to allow state memory of the prior event. The state memory of a guard is an internal value, unpublished and unavailable to application programs. The internal state of a guard, if violated invalidates the guard resulting in a FALSE for the vector. This ensures intended event form integrity.

2.3.2 Error Window

If any referential check produces an error, a pop-up text window dialog box appears, as in Figure 12 telling the user of the error and what they should do to correct the error. This error window is active for all functions of the Event Manager. The user can print the error messages to their default printer or cancel the window. A user knows that their condition-event matrix is referentially correct when the Error Window displays "No errors for Matrix or System." The referential check is initiated by any Apply or Save button actions on the part of the user.



```
Condition Vector:
  SystemValue1 must be an integer value to be used in
  a switch() statement in V:1, M:Test_Matrix, S:Test_System.
Guard:
  FlowRate has no guard in V:2, M:Test_Matrix, S:Test_System.
```

Figure 12 Event Manager Error Window

2.3.3 Guard and Condition Preprocessor

The Guard and Condition Preprocessor takes input from the user and compares it to the internal storage to apply object name and data signature referential checking to ensure that the variable, matrix, and system information is consistent with the current context that the user is working in. In Figure 13 the flow of processing is shown.

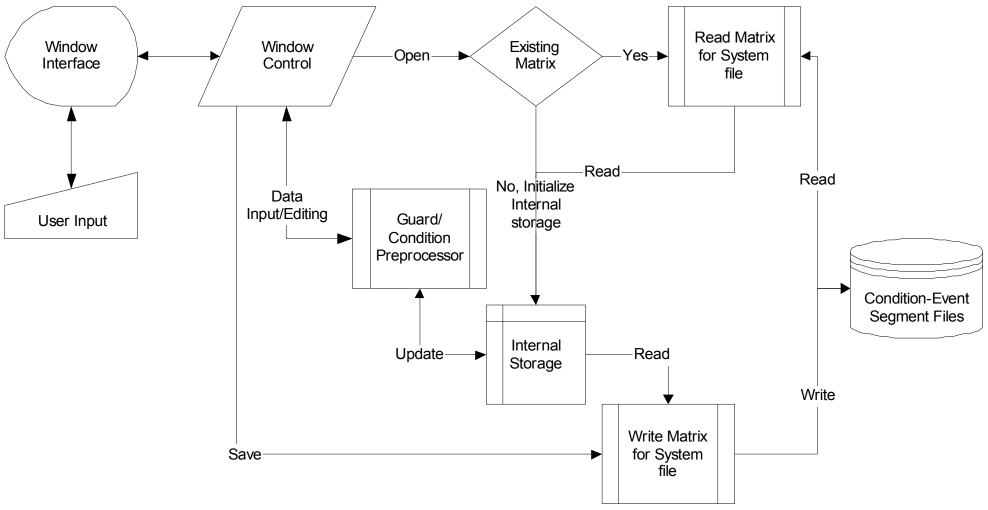

Window Interface
Window Control
Open
Existing Matrix
Yes
Read Matrix for System file
User Input
Data Input/Editing
Guard/ Condition Preprocessor
Read
No, Initialize Internal storage
Read
Condition-Event Segment Files
Update
Internal Storage
Read
Write
Save
Write Matrix for System file

Figure 13 Guard and Condition Preprocessor

The user does manual input using the windows interface. The windows interface control program handles all button, slider, tab control requests. All data input is passed to the vector/guard preprocessor routine that applies referential checks against input data, current working data (held in internal storage) and with data that has previously been input from a prior session. If all data is consistent, as determined by the vector/guard preprocessor, the condition-event segment files are written to the file system. If data is inconsistent, annotated comments are added to the condition-event segment files indicating the inconsistency as shown in the partial file fragment in Figure 14.

```
/*
      **--Condition Vector:
      **--  SystemValue1 must be an integer value to be used in
      **--  a switch() statement in V:1, M:Test_Matrix, S:Test_System.
      **--Guard:
      **--  FlowRate has no guard in V:2, M:Test_Matrix, S:Test_System.
      */
```

Figure 14 Annotated Comments

The special text "`**--`" is a signal to the data store read function that an inconsistency exist from a prior session and that the data contained in the condition-event segment file must be displayed with the error window as shown in Figure 12. The internal storage contains an optimized data structure of the condition/vector and inconsistency data.

A hash[4] value is also written to the file to prevent accidental of intentional modification of the file. This has the form of annotated comments followed by the "Hash:" marker and a hexadecimal value between "<" and ">" characters.

```
/*
**++ Hash: <hexadecimal value>
*/
```

Figure 15 Hash Comments

Hashing functions are commercially available and are used internally to the preprocessor.

2.3.4 Guard and Condition Matrix Control

The condition-event segment files are inserted into standardized event matrix control code. The flow of processing is shown in Figure 16.

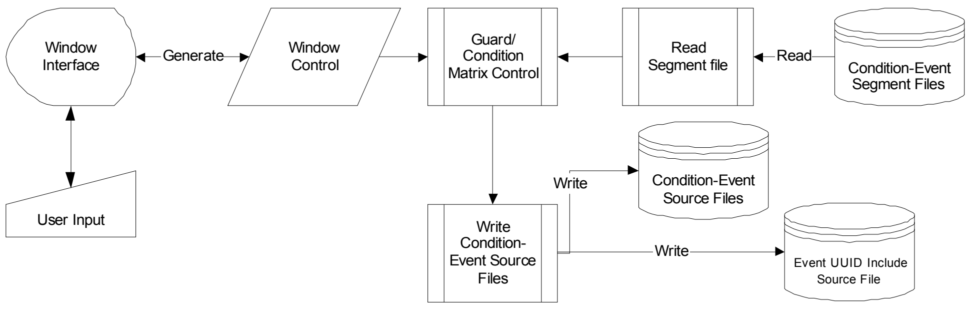
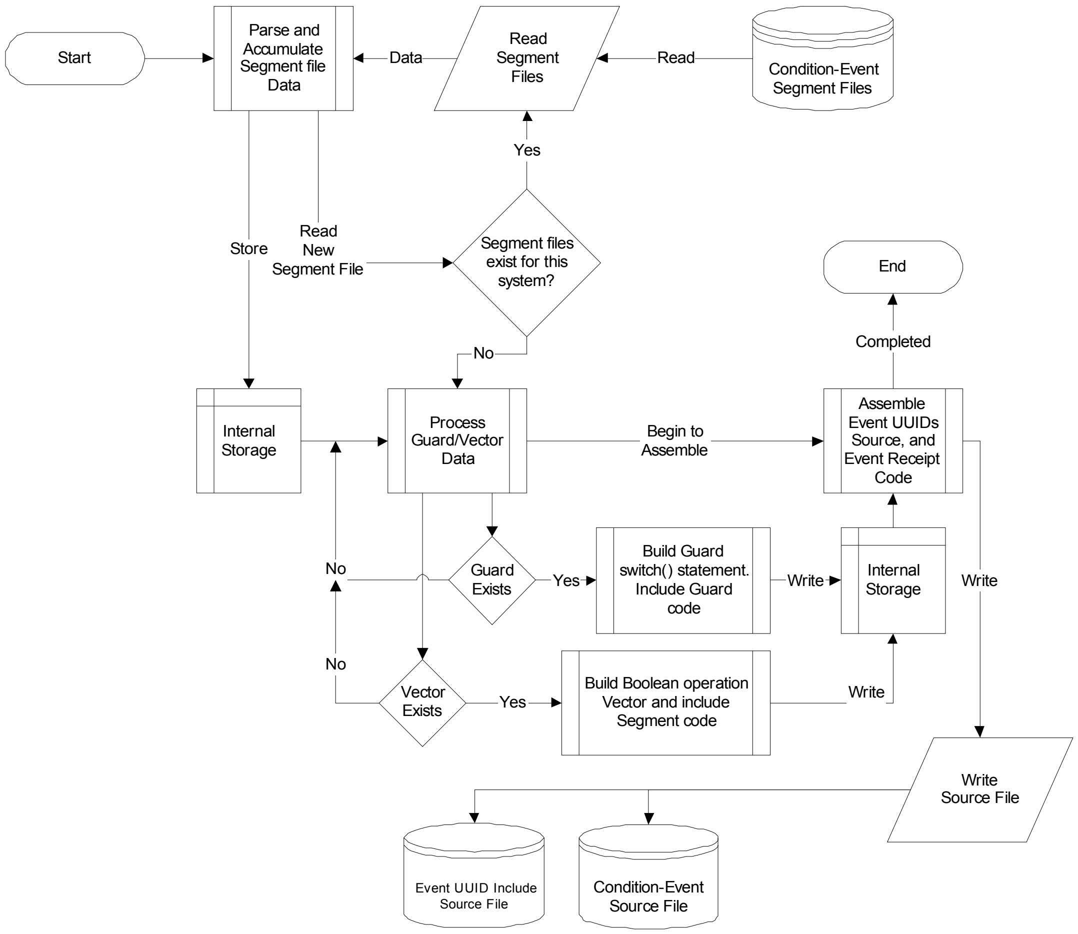

Window Interface
Generate
Window Control
Guard/ Condition Matrix Control
Read Segment file
Read
Condition-Event Segment Files
User Input
Write
Condition-Event Source Files
Write Condition-Event Source Files
Write
Event UUID Include Source File

Figure 16 Guard and Condition Matrix Control

As each of the segments are of a uniform construction the insertion of the segment files into the event-matrix control code is a mechanical process. As the segment files are correct by construction as shown in prior sections of this document, the processing shown in Figure 17 contains only success case logic for processing of the *Write Condition-Event Source Files* module shown in Figure 16.

[4] Hash: Hashing is the transformation of a string of characters into a shorter fixed-length value, or key, that represents the original string.



Start
Parse and Accumulate Segment file Data
Data
Read Segment Files
Read
Condition-Event Segment Files
Yes
Store
Read New Segment File
Segment files exist for this system?
End
Completed
No
Internal Storage
Process Guard/Vector Data
Begin to Assemble
Assemble Event UUIDs Source, and Event Receipt Code
No
Guard Exists
Yes
Build Guard switch() statement. Include Guard code
Write
Internal Storage
Write
No
Vector Exists
Yes
Build Boolean operation Vector and include Segment code
Write
Write Source File
Event UUID Include Source File
Condition-Event Source File

Figure 17 Write Condition-Event Source Files Module

2.3.5 Guard and Condition Include Files

The include files needed by the Guard and Condition entries are used to hold type definitions and structure declarations necessary for the Guard and Vector statements and associated temporary data storage for completion of the source file generation.

These type definitions and structure declarations are supporting of the event engine creation, but can be expressed in many standard forms and are not part of the invention.

3 Usage

This section gives an example usage of the invention for an organization of event systems from event design through setup of the guard and condition variables. Figure 18 shows the notation legend for Figure 19. The following sections detail the design, analysis and construction of the event matrices.

3.1 Design of an Example Event System

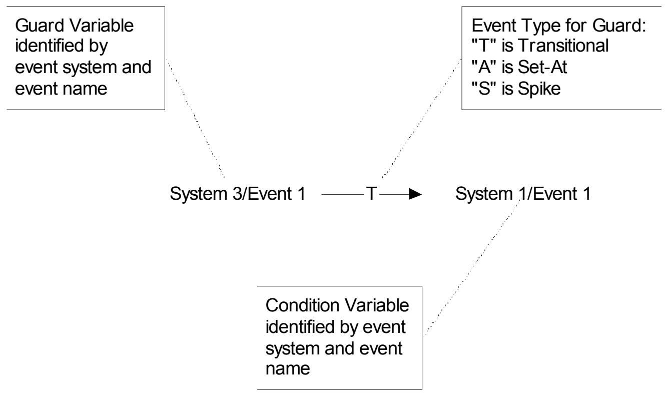

Guard Variable identified by event system and event name
Event Type for Guard: "T" is Transitional "A" is Set-At "S" is Spike
System 3/Event 1
T
System 1/Event 1
Condition Variable identified by event system and event name

Figure 18 Usage Design Notation

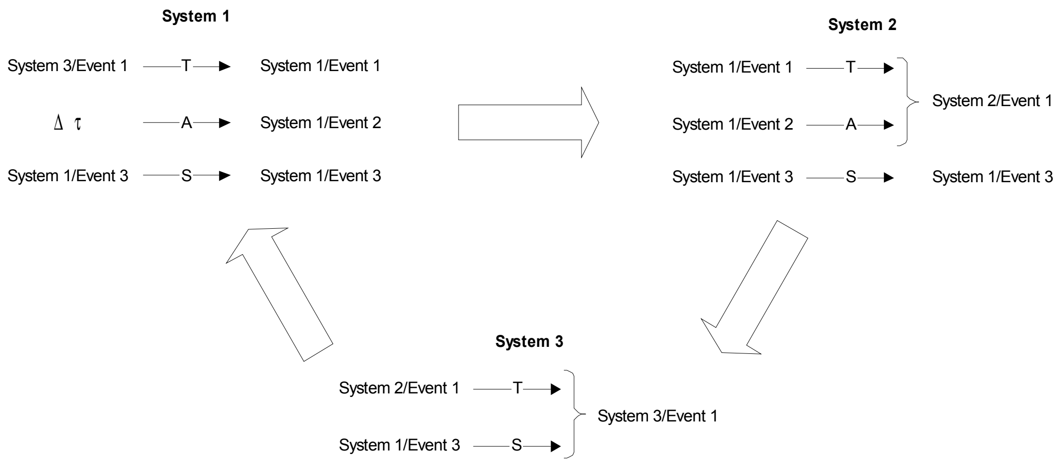


Figure 19 Usage Example Event System

3.2 Analysis of an Example Event System

The example event system has three event systems. The design from Figure 19 has analysis done for correct matrices per system and detection of any race or deadlock conditions. The following tables show the analysis leading to direct data entry in the Event Manager. The event manager would generate a UUID for each of the system plus matrix plus variable names. The names are kept as mnemonics here for clarity and tracking to Figure 19.

From the design, the seminal event system has to be identified by the designer as the event system has to have some initial state prior to entering its steady-state. In this organization of systems, system one is the seminal system.

Event systems can have design errors such that an event sequence is deadlocked (either at initiation or from event interaction), exponential (commonly known as event storms), endless (event consumption error) or quiescent (event starvation) – the latter two may be correct given the parameters of the event system organization. In this organization of systems after an initialization which results in system three, event one being delivered to system one, the system is endless – which is the design intent here.

Vector/ Matrix/ System	Guard Variable	Event Type	Guard Condition	Condition Variable	Event Condition	Boolean
1/1/1	S3/E1	T	G1(S3/E1)	S1/E1	C1(S1/E1)	--
2/1/1	S1/ET	A	G2(S1/ET)	S1/E2	C2(S1/E2)	--
3/1/1	S1/E3	S	C3(S1/E3)	S1/E3	C3(S1/E3)	--

Table 2 Example Event System 1

System 1 has three guard variables. Note guard variable S1/ET is an externally generated time value that is consumed by event system one. The time value is used to generate an event to be consumed by system two. Variable S1/E3 is a self-stimulating spike event so its guard is its condition and thus the generation of system one's event three is delivered to system one as well as system two.

Vector / Matrix/ System	Guard Variable	Event Type	Guard Condition	Condition Variable	Event Condition	Boolean
1/1/2	S1/E1	T	G3(S1/E1)	S2/E1	C4(S2/E1)	AND
2/1/2	S1/E2	A	G4(S1/E2)			--
1/2/2		S		S1/E3	C3(S1/E3)	--

Table 3 Example Event System 2

System two has two matrices. Matrix one has a conjunction of S1/E1 and S1/E2 producing a new event S2/E1. Guard four is used to verify that S1/E2 is set-at a value and that if S1/E2 is violated, S2/E1 is never generated as it would fail the conjunction. This is an example of Equation 4 Resultant Conjunction. Matrix two is an example of a non-self-stimulating spike event that is in fact a pass-through in event system two.

If S2/E1 had been the guard variable for matrix two, vector one, then a cascade of event matrices within an event system would have resulted forcing the generation of the same two events (S2/E1 and S1/E3) but with an explicit order dependency between S2/E1 and S1/E3.

Vector / Matrix/ System	Guard Variable	Event Type	Guard Condition	Condition Variable	Event Condition	Boolean
1/1/3	S2/E1	T	G5(S2/E1)	S3/E1	C4(S3/E1)	AND
2/1/3	S1/E3	S	C3(S1/E3)			--

Table 4 Example Event System 3

System three shows the same evaluation signature of system two, matrix one. The result is the generation of S3/E1, which is sent to system one, thus completing the event cycle. This last event generation marks the end of the initialization of the event system, and the beginning of the steady-state of the event system.

Thus the system is complete, *quod erat demonstrandum*.