Temporal Control Structure Reference Manual

(MIT System Version 23.6)

Thomas A. Russ[1]

March 22, 1993

[1]This research has been supported by Supported by National Institutes of Health grants R24 RR 01320 from the Division of Research Resources, R01 HL 33041 from the National Heart, Lung and Blood Institute, R01 LM 04493 from the National Library of Medicine, and by a grant from the Hewlett-Packard Corporation.

Abstract

This manual describes the Temporal Control Structure (TCS), a programming system designed to facilitate the construction of Artificial Intelligence (AI) programs which use time dependent data. The TCS sets up a data dependency structure which is used to assure the currency of the system's data base. Specific temporal problems addressed are the ability to 1) correctly use data that arrives out of temporal sequence and 2) correctly incorporate changes to previously reported data. Information updating is a selective process that follows data dependencies and propagates changes until the reasoning system reaches a quiescent state.

Reasoning itself is encoded in Lisp, using TCS constructs to establish the dependency. The manual describes all of functions of the TCS needed by a user of the system. There is also an extended example.

Keywords: Artificial Intelligence, Temporal Reasoning, Expert Systems Tools

Contents

List of Figures

Chapter 1

Introduction

The Temporal Control System (TCS) is a tool to make writing time-dependent programs simpler. It will handle the details of ensuring that the all variable values are updated and code executed when values used in the reasoning change over time. TCS was created to handle problems that arose in designing an expert system for use in an intensive care unit. Specific capabilities that were needed included:

1. The ability to correctly use data that arrives out of temporal sequence. This can occur when the data is not immediately available to the program. For example, this type of delay can be due to communication delays or the need to process raw input. The prototypical example of this is laboratory test data, where the results are available only after time-consuming testing and analysis.

2. The ability to deal with data that is subsequently changed. The prototypical example of this would be input data that has been subsequently edited (*i.e.*, errors in the input data were found and corrected).

These two problems require a system that can update the values of data derived from the time-dependent inputs. In a large system efficiency dictates a selective updating. These goals drove the development of the control system.

1.1 Overview of the TCS

The TCS has reasoning entities (called modules) which operate on data contained in variables. Each module can calculate any function of its inputs desired and post the results to its output variables. Modules are also allowed to have internal state. The control system itself is responsible for the bookkeeping needed to insure that all of the modules are operating on the current data. This allows the data to change without the programmer needing to explicitly take this into account. The control structure also monitors the internal state of modules, so that changes to state can also trigger the reexecution of a reasoning module.

There are three basic user-level entities in the control system: *variables*, *modules* and *systems*. Variables and modules are organized into systems. There is no interaction between different systems. Variables are used to hold information that is time dependent. This could either be point data (individual data points) or interval data (time spans over which values are valid). The reasoning is encoded in modules, which use the system's variables for input and output. Modules interact with other modules only through variables. These links implicitly establish the data dependency structure for the system. Modules can influence their own execution by using internal state. Associated with the modules are processes. A new process is created for each module when it is executed (over a time span).

The control system handles the job of scheduling which modules need to execute. This is determined by watching the input variables to each of the modules and queueing modules for execution whenever the inputs to that module change. When a module is executed, all interval variables are guaranteed to have only one value. They may therefore be treated as constant. Point data is supplied for the interval over which the module is being executed. For each such execution, a *process* is created. For more details on the scheduling algorithm, see page 27.

If variable values change, the data dependency structure assures that all of the affected modules and only the affected modules are executed. The new variable values are checked against the old values to stop propagation of values as soon as no further change is detected.

`time` [*Representation*]

The time representation used in the control structure are integers extended by the inclusion of two distinguished values, `:-infinity` and `:infinity` which represent the beginning and end of the time line, respectively. The special time value manipulation functions described in Chapter 6 work on this representation.

1.2 Use of the System

You will need to load the code into your lisp system. At MIT, TCSis preloaded in the MEDG band on the Symbolics. It can be loaded on the Sun and the Macintosh by connecting to the MEDG file server. The files are found in the directory `/u/tcs/files/` the Sun version and `/u/tcs/mac/` for the Macintosh version. You need to load the file `tcs-make.lisp` and then call the function `load-clos-tcs` with no arguments. This will cause the control system to be loaded into the lisp system. This will also set up the package system as described in the next section. For convenience, the user may wish to use the `TCS-USER` package for programming. This can be done by executing the form `(in-package "TCS-USER")`.

TCS requires Symbolics System Genera 7.2for operation on the Lispm machines, Lucid Common Lisp with CLOS on Sun workstations, or Macintosh Common Lisp 2.0.

1.3 Questions and Maintenance

This system is maintained at MIT by Thomas Russ of the Clinical Decision Making Group. This documentation corresponds to system version 28.0in use at MIT.

Questions and bug reports should be sent to:

Thomas A. Russ
545 Technology Square, Room 411
Cambridge, MA 02139–1986
U.S.A.

Phone: (617) 253–3533
Net: `tar@lcs.mit.edu`

1.4 Other Publications

The initial design of this system was done by William Long and Thomas Russ. Descriptions of the control structure design and examples of using it can also be found in the following publications:

- William J. Long and Thomas A. Russ, "A Control Structure for Time Dependent Reasoning," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 230–232, 1983.
- Thomas A. Russ, "A System for Using Time Dependent Data in Patient Management," in *MEDINFO 86: Proceedings of the Fifth Conference on Medical Informatics*, pages 165–169, 1986.

Chapter 2

Packages and External Interface

Two packages are created by the TCS . The first one (`TCS`) is used by the control structure itself and is considered to be the control structure implementor's package. The second one (`TCS-USER`) can be used by applications that depend on the control structure.

`TCS-USER` imports most (but not all) of the external symbols from `TCS`. If the user desires, the other non-imported symbols can also be imported. This is not done by default to avoid potential name conflicts.

2.1 `TCS` Package

`TCS` *nicknames:* `CONTROL-SYSTEM` [*Package*]

This is the control structure implementor's package. It uses the `SCL` (Symbolics Common Lisp) package.

The TCS package has the following external symbols:

CURRENT-SYSTEM	EXPOSE	SET-TIME
GRAPH-SLEEP-TIME*	FIND-MODULE	SHOW-PROCESS-HISTORY
GRAPHIC-SURFACE-TYPE*	FIND-SYSTEM	SHOW-QUEUE
MODULE-DEFN-FORMS*	FIND-VARIABLE	START
MODVAR-DEFN-FORMS*	FUTURE?	START-TIME
NAME-TRUNCATE-CHARACTER*	GET-CURRENT-VALUE	SYSTEM-DEFINITIONP
TBASE*	GRAPH	SYSTEM-TIME
TPRINT-DEPTH*	GRAPH-TRACE	SYSTEMP
TPRINT-MULTIPLE*	GRAPH-UNTRACE	TIME
TRACE-STREAM*	HOW-MANY-VARIABLES	TIME-ADD
UNDEFINED-VARS*	IMPORT-LIST	TIME-MAX
ASK-VAL	INCREMENT-TIME	TIME-MIN
ASK-VARIABLES	INPUT-OF	TIME-PART
AXIS-LABEL-FUNCTION	INPUTS	TIME-SUB
AXIS-TITLE-FUNCTION	INSTANTIATE-SYSTEM	TIME<
BEGIN	INSTANTIATED-SYSTEMS	TIME<=
BEGIN_TIME	LIST-MODULES	TIME=
CONSTRUCT-SYSTEM-DEFINITION	LIST-MODVARS	TIME>
CREATE-GRAPH	LIST-MODVARS-AND-MODULES	TIME>=
DEEXPOSE	MAKE-PARTIAL-SYSTEM	TIMEP
DEF2POINT	MAKE-POINT	TITLE
DEFCONTEXT-TRANSDUCER	MAP-POINT-VALUES	TRACE-MODULE
DEFFUTURE	MODULES	TRACE-MODULES
DEFMEMORY	NAME	TRACE-QUEUE
DEFMODULE	NAME-EQUAL	TRACE-QUEUE?
DEFMODVAR	NOW	TRACED-MODULES
DEFPATTERN	OUTPUT-OF	TYPE
DEFPERSISTENCE	OUTPUTS	UNTRACE-MODULE
DEFRULE	PARSE-REAL-TCS-DATE	UNTRACE-MODULES
DEFSYSTEM	PAST?	UNTRACE-QUEUE
DEFTRANSDUCER	PATTERN	UPDATE-VALUE
DESTROY	PROCESS-HISTORY	UPDATE-VALUES
DESTROY-GRAPH	PUSH-POINT	VALUE
DISABLE-PROCESS-HISTORY	QUEUE-MODULE	VALUE-AT
DISPLAY	REFRESH	VALUE-PART
DISPLAY-TRANSLATION	REMOVE-VALUE	VAR-LIST
DRAW-GRAPH	RESET	VAR-VALUES
ENABLE-PROCESS-HISTORY	RESTORE-SYSTEM	VARIABLES
END	RUN-QUEUE	WHY
END-TIME	SAVE-SYSTEM	WHY-LIST
END_TIME	SET-GRAPH	WRITE-PS

`import-list` [*Variable*]

Variable used to hold the symbols that are exported to the `TCS-USER` package. For

its value, see below.

2.2 TCS-USER Package

TCS-USER [*Package*]

This is the control structure user's package. It uses the SCL package on the Symbolics.

The TCS-USER package has no external symbols.

The TCS-USER package imports the following symbols from TCS:

GRAPH-SLEEP-TIME	FIND-SYSTEM	SAVE-SYSTEM
TRACE-STREAM*	FIND-VARIABLE	SET-GRAPH
BEGIN	FUTURE?	SET-TIME
BEGIN_TIME	GET-CURRENT-VALUE	SHOW-PROCESS-HISTORY
DEF2POINT	GRAPH-TRACE	SHOW-QUEUE
DEFCONTEXT-TRANSDUCER	GRAPH-UNTRACE	START
DEFFUTURE	INCREMENT-TIME	START-TIME
DEFMEMORY	INPUT-OF	SYSTEMP
DEFMODULE	INPUTS	TIME-PART
DEFMODVAR	INSTANTIATE-SYSTEM	TRACE-MODULE
DEFPATTERN	MAP-POINT-VALUES	TRACE-MODULES
DEFPERSISTENCE	MODULES	TRACE-QUEUE
DEFRULE	NAME	TRACE-QUEUE?
DEFSYSTEM	NOW	UNTRACE-MODULE
DEFTRANSDUCER	OUTPUT-OF	UNTRACE-MODULES
DISABLE-PROCESS-HISTORY	OUTPUTS	UNTRACE-QUEUE
DISPLAY	PARSE-REAL-TCS-DATE	UPDATE-VALUE
DISPLAY-TRANSLATION	PAST?	UPDATE-VALUES
ENABLE-PROCESS-HISTORY	PATTERN	VALUE-AT
END	PROCESS-HISTORY	VALUE-PART
END-TIME	REMOVE-VALUE	VARIABLES
END_TIME	RESET	WHY
FIND-MODULE	RESTORE-SYSTEM	WHY-LIST
	RUN-QUEUE	

Note in particular that neither the time comparison and arithmetic functions nor the graph control functions are imported. This means that the user will either have to use a package prefix (*i.e.*, TCS:TIME=) or else import them himself.

Chapter 3

System

A system is used to organize and link variables and modules together. Data can only be shared among the modules inside of a given system.

Each system maintains an internal `system-time` which is the value of the current time for the executing system. The value of this variable (and derived values) is optionally available to modules of the system by the inclusion of the appropriate variable name in the module input list. These variables are `now`, which is a point variable holding the current time as its value and located at a time point corresponding to the current time. It is always the case that the time and the value part of `now` are the same. There will always be only one point on the time line for `now`. `Past?` and `future?` are interval variables which are true only if the interval they are covering is before or after `now`, respectively. Since these variables are normal system variables as defined by `defmodvar`, any module which depends upon them will be reexecuted any time that they change. If `now` changes a lot, this can have efficiency implications for any module depending on the current value of `now`.

3.1 System Definition

`defsystem` *name var-list mod-list* [*Macro*]

Defines *name* to be a system with variables whose names in *var-list* and modules whose names appear in *mod-list*. The variable and module names should be symbols. They do not need to be defined at system definition time, only when the system is instantiated (see below). The *name* is used to identify a class of systems and is called the *system-type*.

Example:

```
(defsystem example
   (first-data-variable last-data-variable)
   (module-the-first module-the-second module-the-third))
```

Note that none of the entries are quoted. Before instantiating this system, the modules and variables must be defined using `defmodvar` and `defmodule`.

`system-definitionp` *symbol* [*Function*]

Returns `t` if *symbol* has a system definition associated with it, otherwise `nil` is returned. A symbol will have a system definition associated with it if it was used as the *name* argument to a `defsystem` form. It does not need to be instantiated.

`*current-system*` [*Value*: `nil`] [*Variable*]

Holds the current system. This is used to specify a system to use for context when querying the user about variable or module names. It can be bound by the user before a function call. It is set globally whenever `instantiate-system` is invoked to create a new system.

`*undefined-variables*` [*Value*: `nil`] [*Variable*]

A list of undefined variables found during system instantiation. This is provided as a debugging aid.

`instantiate-system` *name* `&optional`*sys-name* [*Function*]

Creates a new instance of the system *name*, assigning it an internal name of *sys-name*, which should be a symbol. If no *sys-name* is given, then a unique name is generated based on *name*. All parts of a system must be defined before it is instantiated. If *name* is not defined as a system, an error is signaled. Variables that are not defined in the system definition, but are referenced by modules in the system are added to the list of undefined variables and an error message is printed.

`instantiated-systems` [*Function*]

Returns a list of all of the currently instantiated systems. A list of all of the instantiated systems of a given type `wanted-type` can be obtained by evaluating the following:

```
(remove-if-not #'(lambda (sys) (eq (system-type sys) wanted-type))
               (instantiated-systems))
```

`system` [*Inherits from* `system-mixin`] [*Flavor*]

Flavor which implements the system of a control structure. the symbol `system` is an internal symbol of the package `CS`. A system has the following instance variables:

name	The name of the system.
system-type	The generic name of the system, the name specified in `defsystem`.
variables	The list of system's variables.
modules	The list of the system's modules.
ask-variables	The list of variables for which ask values are defined. Currently not used.
system-time	The current time for the system.
now	Cache for the point variable holding `now`.
past?	Cache for the interval variable `past?`.
future?	Cache for the interval variable `future?`.
queue	The queue of pending processes.
trace-queue?	Flag for queue tracing.
graph	Holds the graph object for graphic tracing.
gtrace?	Flag for graphic tracing of queue execution.
type	`'SYSTEM`
traced-modules	List of modules currently being traced.
keep-history?	Flag for process history. Default `nil`.
process-history	Process history list.

All variables are readable and initable. *name, variables, modules, ask-variables* and *queue* are writable. It is strongly recommended, however, that users of the control structure refrain from modifying the instance variables except through the methods provided below.

A system has a printed representation of `#<SYSTEM` *name*`>`.

3.2 General System Functions

`save-system` *system filename* `:`*overwrite :inputs-only :force-time* [*Function*]

This function writes a machine readable dump of *system* to the file *filename*. This dump can later be restored by the function `restore-system` described below. `T` is returned. If `:overwrite` is non-`nil`, then an existing file with the name *filename* will have a new version written. If `nil` (the default), then a continuable error is signaled. If `:inputs-only` is non-`nil`, then only input variable values (variables that are not the output of any module) will be saved. If `nil` (the default), all variable values will be saved. If `:force-time` is specified, it must be a valid TCS time. Before saving *system,* the system time is temporarily set to `:force-time`. If `nil` (the default), then the current system time is used.

Important note: In order for the save and restore to work properly, all of the values associated with variables *must* have a printed form that allows the lisp reader to reconstruct the object in question. This is the default for most lisp structures and also for defstructs. It is *not* the case for flavors, however. If this condition is not met, save and restore will *fail.*

`restore-system` *filename* [*Function*]

Uses a dump written by `save-system` (described above) in *filename* to restore the system to the state it was when the dump was made. The system type must already be defined in the lisp code, and any packages referenced must already exist. It is not necessary, however, to be in the proper package to read in the file, because that information is cached. There is no defaulting of filenames. The newly restored system is returned. If a system with the same name already exists, then the existing system is overwritten by the restored system. This function returns two values: the restored system and a string of the date and time the dump was written.

See also the important note above concerning the types of values that can successfully be saved and restored.

`reset` **system** [*Generic Function*]

Resets the system to its initial state. The id counter for process numbers is reset to `0`, the current system time is set to `0`, the queue and process history are cleared, and all modules and variables are re-initialized. All data currently present in the system is irrecoverably lost. The values of `now`, `past?` and `future?` will be reset to their proper values. The *process-history* is cleared (set to `nil`).

`find-system` *name* [*Function*]

Returns a system with the name *name*. *Name* must be the specific individual name of the system, not its system type. If no such system exists, then `nil` is returned.

`find-module` **system** *module-name* [*Generic Function*]
`find-variable` **system** *var-name* [*Generic Function*]

Finds a module or variable with the specified name in the system *system*. The value of *system* must be of type system, not just the name of the system. If no such module or variable exists, then `nil` is returned.

`display` **system** `&optional`*stream* [*Generic Function*]

Prints a description of the system on *stream*. The default value of *stream* is `t`, the standard I/O stream. The format of the display shows the name of the system, and the names of the modules, point and interval variables for the system.

3.3 The System Queue

`show-queue` **system** `&optional`*stream* [*Generic Function*]

Prints a listing of the system queue to *stream*. The default value of *stream* is `t`, the standard I/O stream. The listing shows the processes currently on the queue. Sample output follows:

```
QUEUE:
     PROCESS   1 [PENDING] for K+_DATA from 10 to INFINITY
     PROCESS   2 [PENDING] for HEART-FAILURE from 10 to INFINITY
     PROCESS   3 [PENDING] for ACUTE-MI-DATA from 10 to INFINITY
```

Each line is a separate process, printed using the process `display` function (see page 37). The name of the process, its associated module and the time over which it is scheduled to run is printed. If the queue is empty a message to that effect is printed.

`run-queue` **system** `&optional`*print-stats? limit* [*Generic Function*]

Causes the system to loop through the queue, executing processes until the queue is empty. Errors are trapped by an unwind-protect form and a process requeued for the interval in which the error occurred before the function will return. Unless the optional argument *print-stats?* is `nil`, statistics about the running of the queue will also be printed. The number of processes queued and run will be printed, along with the percentage of queued processes actually executed. The default is to print statistics. The optional argument *limit* can be used to set a limit on the number of processes that will be run in this invocation of `run-queue`. The queue will then be run until either it is empty or *limit* processes have been executed. The default is not to have a limit. `Run-queue` returns three values: the number of processes remaining on the queue (0 unless an error occurs or *limit* is specified), the number of processes executed and the number of processes queued.

`queue-module` **system** *module-name start end clip?* [*Generic Function*]
`&optional`*add-up-front?*

Adds the module *module-name* to the queue over the period from *start* to *end*. Processes are automatically created as necessary to maintain the policy of a single interval variable value for each process interval. *end* must be strictly greater than *start* or else errors will occur (this is not checked by the code). A point impetus can be signaled by having *end* be `nil`. If the *clip?* flag is not `nil`, then *start* and *end* are taken to be strict boundaries for the time interval over which modules are to be spawned. This affects only modules which have no interval variables in their input lists. If `nil`, then such processes may extend outside the bounds of *start* and *end*, if no other appropriate interval can be determined.

The optional argument *add-up-front?* allows processes to be added to the front as well as the end of the queue. If `nil` (the default), new entries are added at the end of the queue. If `t`, then the processes "jump" to the head of the queue. The jumping feature is available for giving priority to certain types of process queueing.[1] It is not anticipated that the TCS user will need to invoke this function.

[1] This is used internally by the TCS for efficiency reasons, for example, to complete the time interval if the process just run did not cover the entire interval over which it was originally scheduled. Since the module can affect the interval over which it executes by changing the values of its `begin_time` and `end_time`, it may not run for the full time for which it was scheduled when added to the queue.

`dequeue-module` **system** *module-name start end* [*Generic Function*]

Remove *all* processes for *module-name* which overlap the period from *start* to *end*. The function requeues as necessary so as not to shorten the periods covered. That is, if the removal of an overlapping process leaves a gap between the removed process' start time and *start*, a new process is queued to cover that interval.

`enable-process-history` **system** [*Generic Function*]
`disable-process-history` **system** [*Generic Function*]

These functions control whether a full process history is kept or not. If enabled, a full process history is kept. If disabled (the default) then no process history is kept in the system. Keeping a process history will allow one to retrace the time course of the execution of the system and generate explanations which can explain why a certain decision was made at a given time. Note, however, that this will greatly increase the amount of storage that is used by the system, since all of the historic information will be remembered. The current status of history keeping can be determined by using the accessor function `keep-history?`.

`show-process-history` **system** [*Generic Function*]

Prints the process history on the default output stream. If a full system process history is being kept (See `add-to-process-history` below), then that is used as the source. Otherwise, the history is derived from the process data stored in the individual modules, and so will be a reflection of the processes that make up the current view of the state of the system.

`add-to-process-history` **system** *process* [*Generic Function*]

Adds *process* to the process history of the system. This is a list of all the processes that were ever run in this instance of the system (subject to `reset`). See also the processes list in the individual modules. This function is called internally by the system code when the process history is enabled. This history is only kept if the `enable-process-history` function has been called. By default it is not kept. Keeping a process history will greatly increase the memory requirements of the system.

3.4 System Time

`increment-time` **system** `&optional`*amount* [*Generic Function*]

Increments the system time by *amount*. If *amount* is not specified, it defaults to `1`. This affects the value of the TCS variables `now`, `past?` and `future?`. If a negative value for the increment is specified, then a continuable error is signaled. If continued, the negative value is accepted.

`set-time` **system** *value* `&optional`*silent* [*Generic Function*]

Sets the system time value to be *value*. This affects the value of the variable `now`, which is accessible in modules. Any legal time value is acceptable. No check is made to see whether the time is before or after the current time. If the optional *silent* argument is non-`nil`, then printing of the new time value is suppressed. By default *silent* is `nil`. If the variable `*tbase*` has a numeric value, then a real world time is printed, with the system time in parentheses. Otherwise just *value* is printed.

3.5 Debugging Aids

`trace-queue` **system** [*Generic Function*]
`untrace-queue` **system** [*Generic Function*]

Enables or disables the tracing of the queue as processes are added or deleted from the queue. If tracing is enabled (the default state), then each process added to the queue is identified as it is added and each process executed by the the system is also logged. Output goes to the stream `*trace-stream*`, which defaults to be the standard output. An example of the output is in Figure 3.1.

`trace-module` **system** *module* [*Generic Function*]
`untrace-module` **system** *module* [*Generic Function*]
`trace-modules` **system** *module-list* [*Generic Function*]
`untrace-modules` **system** *module-list* [*Generic Function*]

Enables or disables tracing of individual modules as they execute. This trace will show the value of the input and output variables as well as the contents of the memory variables for the processes of the identified modules. The argument *module* (or the elements of *module-list*) are the names of the modules rather than module objects themselves. Output goes to `*trace-stream*`, which defaults to `t`, the standard output. A list of the currently traced modules is held in the instance variable *traced-modules* (accessible from the `system` via the accessor function `traced-modules`).

For `trace-module` and `untrace-module`, the value `t` has special meaning. If *module* is `t`, then all modules in the system are traced (or untraced).

The display format is a right arrow followed by the module name for an executing process, and then the time period of expected execution. Input variables and the initial value of history and oracle variables are then printed. Upon completion, a left arrow followed by the module name and the time over which the process actually executed is printed. Then the values of output variables and the final value of history and oracle variables is printed. An example of the output is shown in Figure 3.2.

`*graph-sleep-time*` [*Default*: `3`] [*Variable*]

Time in seconds to sleep between graph updates. This controls the length of time

```
Running PROCESS   4 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
  Queue [V] PROCESS   5 [PENDING] for FEVER_DETECT from -INFINITY to INFINITY
Running PROCESS   5 [PENDING] for FEVER_DETECT from -INFINITY to INFINITY
    Ran process from -INFINITY to 0
  Jump  [E] PROCESS   6 [PENDING] for FEVER_DETECT from 0 to INFINITY
Running PROCESS   6 [PENDING] for FEVER_DETECT from 0 to INFINITY
    Ran process from 0 to 30
  Jump  [E] PROCESS   7 [PENDING] for FEVER_DETECT from 30 to INFINITY
  Queue [V] PROCESS   8 [PENDING] for TREAT_FEVER from -INFINITY to 0
  Queue [V] PROCESS   9 [PENDING] for TREAT_FEVER from 0 to 30
  Queue [V] PROCESS  10 [PENDING] for TREAT_FEVER from 30 to INFINITY
Running PROCESS   7 [PENDING] for FEVER_DETECT from 30 to INFINITY
    Ran process from 30 to 60
      (...)
```

The entries identify when a process is queued and ran. The module for which the process is created and the time over which the process runs is also identified. The status will always be "`PENDING`," since the trace prints before the process is executed. The letter in square brackets identifies the reason for queueing a module. For example, "`V`" means a change in variable value, and "`E`" is a change in endpoint. The system identifies processes which run for a shorter time period than originally specified [preceding the "`E`" queueing announcement.] There are two types of queueing entries: "Queue" and "Jump." The latter is used when the new process is added to the front rather than the back of the queue. In effect, it is "jumping" the queue. A full listing of queueing reasons can be found under the description of `help-trace` on page 38.

Figure 3.1: Example of Queue Trace Output

```
==> ACUTE-MI [100 to INFINITY]
    ACUTE-MI-DATA = ((100 . PRESENT))
    ACUTE-MI-STATE = (10 . PRESENT)
 <= ACUTE-MI [100 to 600]
    ACUTE-MI = PRESENT
    ACUTE-MI-STATE = (100 . PRESENT)
--------------------------------------
==> ACUTE-MI [600 to INFINITY]
    ACUTE-MI-DATA = NIL
    ACUTE-MI-STATE = (100 . PRESENT)
 <= ACUTE-MI [600 to INFINITY]
    ACUTE-MI = UNKNOWN
    ACUTE-MI-STATE = (100 . PRESENT)
--------------------------------------
```

Example of output for a TCS module named `acute-mi`, which has an input (point) variable `acute-mi-data` and an output (interval) variable `acute-mi`. `Acute-mi-state` is a history variable.
The first process actually runs from `100` to `600` although scheduled to run to `:infinity`. The second process was queued to take up the remaining time. Observe that the initial history value (`acute-mi-state`) of the second process is the same as the final value of the first process. The value of interval values is printed directly. The internal representation of point variables is printed. This is currently a cons pair of time and value. Users are cautioned not to directly access this representation because it may be changed in the future. The value of a point variable is a list of point value representations.

Figure 3.2: Example of Module Trace Output

that a graph will remain stable for viewing purposes when graphic tracing is enabled. (See graphic tracing below.)

`set-graph` **system** `&rest`*var-names* [*Generic Function*]

Sets up a graph for displaying the values of the variables specified as the `&rest` arguments. The variables are displayed in the graph in the order that they are specified in the argument list. Graphic tracing displays the results of a change in variable values after each module which could affect any of the variables is run. The system then suspends processing for the number of seconds specified in the variable `*graph-sleep-time*`, to give the user time to look at the graph before it is redrawn by the next change. Because of this large delay, the use of graphic tracing will greatly slow down system execution! (You have been warned!) It does, however, give a good picture of how values of variables are changing during the execution of the processes on the queue.

Note that the name of the variables must be quoted. The graph is made large enough to hold all of these variables (if possible given screen limitations) and uses the full screen width.

`graph-trace` **system** `&optional`*start end* [*Generic Function*]

Turns on the graphic tracing. The graphic tracing is only enabled for changes during the time period from *start* to *end*, although the system still executes over all necessary time periods. If not specified, *start* defaults to 0 and *end* to 100. Before using this function, `set-graph` must have been evaluated to set up a graph.

`graph-untrace` **system** [*Generic Function*]

Turns off the graphic tracing.

Chapter 4

Variables

Variables are the data carriers that implement the connections between the different modules. By monitoring the variables, the control structure as a whole is able to keep the input consistency of the system intact. Variables come in two varieties: *point* and *interval*.

Point variables are used to represent individual data points. They have a value and a specific time point associated with them (hence the name). The value is valid only at the particular time point in question. Such variables provide a natural representation for sample data.

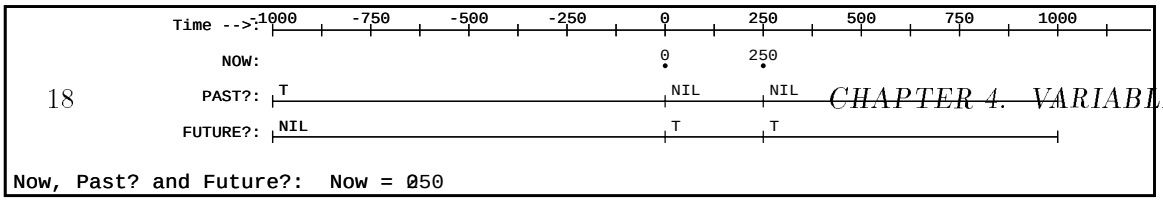
Interval variables, on the other hand, associate values with an interval of time. They have a begin and and end time associated with them. By convention in this system, they have a value that stretches over the entire time line from `:-infinity` to `:infinity`, although most of that period may be occupied with the default value `:unknown`. Interval variables are a natural representation for states or other values that have a duration.

4.1 Now, Past? and Future?

There are three variables that are predefined for all systems. These are the variables that provide access to the system time.

`now` [*Point Variable*]
`past?` [*Interval Variable*]
`future?` [*Interval Variable*]

One of them is the point variable `now`, which is always set to be at the current time. The value of the variable will be the current time. (The value-part and the time-part of the variable will be identical). Since this is a point variable, its value will only be available inside the interval that contains the point. For convenience, two additional interval variables, `past?` and `future?`, are also defined. These variables have the value `t` (`nil`) over the interval from `:-infinity` to the current time and `nil` (`t`) from the current time to



In these examples, "•" denotes the location of a data point in time, and the number above the asterisk is the value of the point. Interval values are written within the interval over which they are valid. Intervals are delimited by vertical bars "|".

Figure 4.1: `Now`, `Past?`, and `Future?`

`:infinity`. If these variables are used, they will segment the process for any module so that one interval will always end on the current time and one will begin at the current time (due to the nature of process scheduling with interval values).

Consider the two examples in Figure 4.1, the first one being the time line with a current time of `0` (the start-up default). The second with a current time of `250`.

Access to the system time should be only through these variables, as this is the only way to guarantee that modules which depend on the value of the variables will be updated automatically when the values of these variables change. The value of *now* can be accessed outside of the interval in which it is located only through the use of history or oracle variables inside a particular module. (See the chapter on modules).

These variables are defined as normal control system variables. The names are reserved. A continuable error will be signaled if you attempt to define variables with the same name.

4.2 Code

`defmodvar` *name type* `:` *ask-val :predicate :initial-value :accessor-function :display-translation :format-string :graph-label* [*Macro*]

This is the definition form for system variables. *Name* must be a symbol. *Type* must be one of `:point` or `:interval`. The keyword arguments have the following meaning:

`:ask-val`	A list of legal values for the value part of the variable. This could be used by an interface system to prompt for values. Default `nil`.
`:predicate`	A function of two arguments that is used to compare the value of the variable. It should return non-nil whenever the two values should be considered to be the same. Default `equal`.
`:initial-value`	Defines the initialization value for an interval variable. This is the value used when the system is instantiated or reset. The setting of this initial value **does not** trigger the normal process scheduling mechanism, so it is the user's responsibility to make sure that the values are compatible. This is primarily to simplify the use of special data structures as the values of interval values. The predicate then need not deal with the system default value. The default value is `:unknown` for interval variables. It is an error to specify this option for a point variable. Its default value is always `nil`.
`:accessor-function`	Used to select a part of a larger structure for display on a graph. It is supported mainly to allow selection of part of a structure such as that created with `defstruct`, although other uses are possible. The value returned by this function then undergoes the display translation. Default `identity`.
`:display-translation`	An alist (association list) of values and abbreviations to be used for nice graphics displays. The abbreviations are supported to allow better data display when many elements appear in one graph. Default `nil`.
`:format-string`	A format compatible string that is used to print the value of the variable in a graphic display. It uses the value returned by the display translation. This allows more precise control of graphic presentations, particularly for numeric data. Default `"~A"`.
`:graph-label`	A string that is used as the label for the variable in a graphic display. If this is not given, then the name of the variable is used. This option allows the specification of a different "pretty" name for graphs.

Example:

```
(defmodvar temperature :interval
        :display-translation '((high . hi) (normal . ok))
        :predicate #'eql)
```

This defines an interval variable called `temperature` which has a display translation and chooses to use the `eql` test for efficiency reasons. Note that the variable name is not quoted,

but that values for the keywords are.

`instantiate-variable` *name system* [*Function*]

Makes an instance of variable *name* in system *system*. *Name* must be a symbol and *system* an instantiated system. A variable of the appropriate type (point or interval) is created. At creation time, it is also initialized. Interval variables are initialized to have the value `:unknown` (or the specified initial value) from `:-infinity` to `:infinity`. Point variables are set to have no values at all. This function is called by `instantiate-system`. It is not anticipated that a TCS user will need to call this function.

`value-part` *item* [*Macro*]
`time-part` *item* [*Macro*]

Returns the value or the time part of *item* respectively. These functions are designed to be used with individual point data representations. The value of a point variable inside a module will be a list of such point data representations, each consisting of a value and a time part. These macros are the preferred way of retrieving the value or the time component of such an individual time point.

If the values list of an interval variable is being manipulated outside of a module, then `value-part` could also be applied to the each interval value to get its value. Inside a module body, the interval variable has already been decoded, so it is not necessary to manually extract the value. In fact, this function will fail. `time-part` would return the internal representation of an interval, so it should not be used (see below).

`start-time` *item* [*Macro*]
`end-time` *item* [*Macro*]

Returns the starting (ending) time of an interval value structure *item*. This is only relevant outside of a module body (Inside a module, the interval variable has only a single value, so the structure has already been decoded). This function should be applied to the interval value structure as a whole. In other words, one should not apply `time-part` and then one of these functions. Note that when manipulating interval values outside of a module, they will be in the form of a list of individual interval values, similar to point values everywhere.

`make-point` *value time* [*Macro*]
`make-intval` *value start end* [*Macro*]

Makes a point (interval) structure which is associated with the time *time* (interval from *start* to *end*). These functions can be useful in connection with the `:initial-value` keyword to history variables used in modules. This is the preferred method of creating point and interval value objects. Users are discouraged from creating them on their own because the internal representation may change. Use of these macros will isolate the user from any such changes.

`map-point-values` *function list* [*Function*]

The value returned by this function is a list of point variable values with the same times as the input *list*, but with values created by `funcall`ing the *function* on each value. It is functionally equivalent to the following code:

```
(mapcar #'(lambda (x) (make-point (funcall function (value-part x))
   (time-part x)))
list)
```

`push-point` *value time location* [*Macro*]

The equivalent of `(push (make-point value time) location)`.

`same-point` *item1 item2 predicate* [*Function*]

A predicate that will test two point data structures `item1` and `item2` to see if they represent the same point. *predicate* is used as the test function on the value parts. Time parts are compared internally for equality. Returns `t` if the times match and the values fulfill the test of *predicate*.

`variable` [Inherits from `system-mixin`] [*Flavor*]

Flavor which implements the basic type of variable for the control structure. This flavor is used to hold the common data and method declarations for both point and interval variables. It has the following instance variables:

name	Name of the variable.
system	System to which this variable belongs.
input-of	List of module names that use this variable as an input value. These are the modules that are activated when the value changes.
output-of	List of at most one module name, which that is the source of this variable's value.
ask-val	List of permissible values for prompt. Currently not used by TCS.
predicate	Test for sameness. Default `equal`.
initial-value	Initialization value for the variable. The default is `:unknown` for interval and `nil` for point variables.
display-translation	Display translation. (see `defmodvar`)
graph-label	A pretty name used by the graphing routine for labeling graphs. If `nil` (the default), then the *name* is used instead.
var-values	List of the currently correct values. It is stored in an internal format and should not be accessed by users.

All variables are readable, writable and initable. Users are warned that changing the *values* of a variable directly will prevent the control system from properly scheduling affected modules to run. Value changes should only be made through the use of the `update-value(s)` functions.

```
VARIABLE BODY_TEMP [Point]:
  Input of: TEMP_EVAL
 Output of:
    0 = 98.8      30 = 101.2      60 = 105.1     120 =   102


VARIABLE PAT_STATE [Interval]:
  Input of: TREAT_FEVER
 Output of: FEVER_DETECT
  -INFINITY to        0 = UNKNOWN
           0 to       30 =  NORMAL
          30 to      180 =   FEVER
         180 to INFINITY = UNKNOWN
```

Figure 4.2: Examples of Point and Interval Variable Display Output

`pointv` [Inherits from `variable`] [*Flavor*]

Sub-type of variable used to implement point variables. It has the following instance variable:

type `pointv`

It is readable, writable and initable along with all the inherited variables. *Type* should never be changed, though! The printed representation is `#<PtVar` *name*`>`.

`interval` [Inherits from `variable`] [*Flavor*]

Sub-type of variable used to implement interval variables. It has the following instance variable:

type `interval`

It is readable, writable and initable along with all the inherited variables. *Type* should never be changed, though! The printed representation is `#<IntVar` *name*`>`.

`display` **pointv** `&optional`*format-stream print-timebase* [*Generic Function*]
`display` **interval** `&optional`*format-stream print-timebase* [*Generic Function*]

Prints a description of the variable on *stream*. *stream* defaults to `t`, the standard I/O stream. The information printed is the name of the variable, the modules of which it is an input or output, and a list of the values. For point variables it is a set of "<time>:<value>" pairs, for intervals it is a list of "<start> to <end>:<value>" lines. Sample output can be seen in Figure 4.2.

If *print-timebase* is specified, it must be a universal time that corresponds to the "0" system time. The time part of the the why explanation is then printed using real world times such as months, days and hours. The printing is done down to the depth specified by the variable `*tprint-depth*`. The conversion between the units of system time and the real world time is controlled by the variable `*tprint-multiple*`.

`update-value` **interval** *new-val start end* [*Generic Function*]
`update-value` **pointv** *new-val new-time* [*Generic Function*]
`update-values` **pointv** *new-val-list begin-time end-time* :*open-left* :*open-right* [*Generic Function*]

These methods are used to change the value of variables. For interval variables, the *new-val* is used as the value of the interval from *start* to *end*. It replaces any previous value over that time. The system will automatically merge the value with adjoining values that are the same with respect to the equality predicate defined for the variable in question.

For point variables, `update-value` merges the *new-val* into the list of current values. It is associated with *new-time*. The `update-values` method replaces all the values between *begin-time* and *end-time* with the point values in the list *new-val-list*. These point values must be valid point values created by `make-point`. The handling of the end time points is controlled by the value of the `:open-left` (default `nil`) and `:open-right` (default `t`) keyword arguments. This function uses destructive list operations, so the value of *new-val-list* could be modified.

`update-value` **pointv** *new-val new-time* [*Generic Function*]

Removes the point variable value at time *time*. If no value exists at the specified time point, nothing is done.

`get-current-value` **pointv** *start end* [*Generic Function*]
`get-current-value` **interval** *start end* [*Generic Function*]

Returns the value of the variable over the interval from *start* to *end*. For point variables, this is a list of all of the point values in that interval (`nil` if none exist). For intervals it is the value that the interval has over that period. If more than one value exists for the interval over the requested period, then an error is signaled.

`value-at` **pointv** *time* `&optional` *return-earlier* [*Generic Function*]
`value-at` **interval** *time* `&optional` *return-earlier* [*Generic Function*]

Returns the value of the variable at *time* exactly. For point variables a value will be returned only if there is a point at that time, otherwise `nil` will be returned. For intervals a multiple value return of the value, and the start and end times of its interval is made. If an interval boundary is found right at *time*, then the value returned is determined by examining the optional argument *return-earlier*. If `t` (the default), then the value returned is that of the interval ending at *time*. If `nil`, then the value starting at that time is returned. The optional argument is supported for point variables in order to have a consistent function call interface. It doesn't have any effect on the value returned from point variables.

`why` **pointv** *start end* `&optional` *stream print-timebase* [*Generic Function*]
`why` **interval** *start end* `&optional` *stream print-timebase* [*Generic Function*]

Prints a trace of how the value of the variable was determined over the interval from *start* to *end*. Output is written to *stream*, which defaults to `t`. The format of the output

```
PAT_STATE has values:
NORMAL from 0 to 30 because
  Module FEVER_DETECT ran from 0 to 30 with input
    TEMP_EVAL = 0:NORMAL, 30:FEVER.
  and with history variables
    LAST-TIME-6952 = -INFINITY
    LAST-STATE-6951 = UNKNOWN
FEVER from 30 to 100 because
  Module FEVER_DETECT ran from 30 to 60 with input
    TEMP_EVAL = 30:FEVER, 60:FEVER.
  and with history variables
    LAST-TIME-6952 = 0
    LAST-STATE-6951 = NORMAL
  Module FEVER_DETECT ran from 60 to 120 with input
    TEMP_EVAL = 60:FEVER, 120:FEVER.
  and with history variables
    LAST-TIME-6952 = 30
    LAST-STATE-6951 = FEVER
```

This was produced by calling the `why` function on the variable `pat_state` from 0 to 100. The variable had two different values (normal and fever), and was produced by three processes, running from 0–30, 30–60 and 60–120. The variable and the module that produced its value are defined in Chapter 9. The variable `LAST-TIME-6952` is a history that holds the last time and `LAST-STATE-6951` holds the last state. These variables are generated by a TCS macro.

Figure 4.3: Example of `Why` Function Output

is to indicate the source (either the user or the output from a module) for the values of the variable. If a module is the source, it has the `why` function invoked on it. (See module documentation for details). This message supports more than one value for interval variables in *start* to *end*. In that case, the output is broken into separate sections for each distinct interval variable value period. Sample output is shown in Figure fig:why.

If *print-timebase* is specified, it must be a universal time that corresponds to the "0" system time. The time part of the the why explanation is then printed using real world times such as months, days and hours. The printing is done down to the depth specified by the variable `*tprint-depth*`. The conversion between the units of system time and the real world time is controlled by the variable `*tprint-multiple*`.

`why-list` **pointv** *start end* [*Generic Function*]

Returns a list structure describing how the value of the variable was determined over the interval from *start* to *end*. The format of the output is:

```
(reason-list value-list)
```

where `reason-list` is a list of process instantiations for the module whose output produced the value, or `nil` if the variable value is not the output of a module. (It is then, presumably, set directly by the user). This `reason-list` is produced by invoking the function `why-list` on the module over the same time period. The `value-list` is a list of point values in standard format. The `time-part` and `value-part` accessor macros can be used to access the respective parts of each time-value pair. Note that the association between values and processes is not handled by this routine. (Contrast with the function below).

`why-list` **interval** *start end* [*Generic Function*]

Returns a list structure describing how the value of the variable was determined over the interval from *start* to *end*. The format of the output is:

```
( (value reason-list) ...)
```

where `value` is a single interval value object. This is not just the value, but includes the starting and ending times. The various components can be accessed using the macros `start-time`, `end-time` and `value-part`. The `reason-list` is a **list** of process instantiations for the module whose output produces `value`. It is `nil` if there is no such module. (The value is then, presumably, set directly by the user.) This is a list of process instantiations because it is possible that more than one different, adjacent processes, using different inputs, could produce the same value. Since adjacent identical values are combined, more than one explanation could exist. Each item in this list is the result of invoking the function `why-list` on the producing module.

Figure 4.4 shows the values returned by `why-list` for the same variables used in Figure 4.3.

```
((((0 . 30) . NORMAL)
  ((0 30 ((|TEMP_EVAL| (0 . NORMAL) (30 . FEVER)))
         ((CS::LAST-TIME-6952 . :-INFINITY) (CS:::LAST-STATE-6951 .
:UNKNOWN))
         NIL)))
 (((30 . 100) . FEVER)
  ((30 60 ((|TEMP_EVAL| (30 . FEVER) (60 . FEVER)))
          ((CS::LAST-TIME-6952 . 0) (CS:::LAST-STATE-6951 . NORMAL))
          NIL)
   (60 120 ((|TEMP_EVAL| (60 . FEVER) (120 . FEVER)))
           ((CS::LAST-TIME-6952 . 30) (CS:::LAST-STATE-6951 . FEVER))
           NIL))))
```

This is the program readable version of the information displayed in Figure 4.3.

Figure 4.4: Example of `Why-List` Function Output

Chapter 5

Modules

Modules are the units which hold the executable code for the system. When a module is executed, processes are created, one for each interval over which a module is to be executed. Within this interval, the module code will have access to the following information:

1. The values of the input variables. Any interval variables are guaranteed to have a single value. Access to the system time can be made by including `now`, `past?`, or `future?` in the input list.

2. History variables that contain state information from the previous process. This is data from the past.

3. Oracle variables that contain state information from the next process. This is data from the future.

4. The starting and ending times of the execution interval. These are contained in the variables `begin_time` and `end_time`.

As noted above, interval variables have only one value within each execution interval. The determination of the interval over which to create processes involves segmenting the input variables into intervals over which this condition holds. This determines the intervals over which a module's process will execute. This segmentation is handled automagically by the control structure. Figure fig:segment shows an example of this segmentation for a simple module with three interval variable inputs (A, B and C).

If all of the input variables are point variables, however, this algorithm will not work. In that case, the processes are created to cover the intervals of a previous processes that covered the time of the current change in input values. If no such processes exist, the process will run from `:-infinity` to `:infinity`. This covers the whole time spectrum the first time such a module is invoked and allows future updates to use the first method. It is also possible that the exact method used to queue modules with point data input may change in the future.

Figure 5.1: Process Scheduling for Interval Input Variables

5.1 Point Data, Open and Closed Intervals

A process for a module will have access to all of the point data for the interval in which it is running. This includes points that lie on the end points of the interval. In other words, the data availability is inclusive. If this is not desired, then the module code itself can check for this by comparing the time component of the first and last point data with the `begin_time` and `end_time` of the process. A problem does occur, however, when point variables are the output of modules. It is possible that a value on the boundary of process intervals could be set by either the process ending at that point or the process starting at that point. The default convention is that the value of the temporally later process (along the time line) will have precedence. Operationally, this means that if a process changes a point value that is at the same time that the process ends, the next process will be queued and run. This next process will determine the value of the point variable at the beginning time of the process. If no value is explicitly set, this will *delete* the value from the list. For example, consider two processes P_1 and P_2 for a module with a point variable output. Let P_1 be immediately before P_2. There are four cases that can occur with respect to the output of a point variable value at the boundary time t_b:

1. Neither process produces a value at t_b. No value is produced and there is no conflict.
2. Only P_1 produces a value at t_b. Since P_2 is queued and run, but does not produce a value, P_1's value is deleted. No value is set at t_b.
3. Only P_2 produces a value at t_b. This value stands.
4. Both P_1 and P_2 produce values at t_b. P_2's value takes precedence.

This is the default policy. By using the module options `:open-left` (default `nil`) and `:open-right` (default `t`), the user can exercise control over this process. If intervals are closed on both the right and the left side, then the temporally later process will take precedence as in item 4 above. Note that if an interval is declared to be open at one side, then it will be impossible to set an output variable at the boundary on that side, even if there is no conflict from the adjacent process. In particular this can prevent the system from setting a point value at `:infinity` or `:-infinity`. If the interval is open at both ends, then no value can be set at a boundary point.

5.2 `Defmodule` Macro

`defmodule` *name inputs outputs options* `&rest` *code* [*Macro*]

This is the definition form for describing standard modules. *Name* must be a symbol. *Inputs* is a list of the variable names that provide the inputs to this module. If the input list is empty, however, the module will never be automatically executed by the control structure. It must be added to the queue by the user with the function `queue-module` (see page 11).

Outputs is a list of the variable names that are the outputs of this module. A variable name may legally appear in only one output list in each system. There is no restriction on the number of input lists in which a variable name may occur. All variable names must be symbols. A variable may be both an input and an output variable. The user is responsible for determining that this will not cause infinite looping (*i.e.*, the value of such a variable must eventually stabilize).

Options holds a list of module options as well as the list of internal variables that are used to hold state information for continuing the execution of the module code. A change in this state information will cause an temporally adjacent process in the process list to be reexecuted (if it already exists) or to be created (if no such process exists). There are two basic types of internal state variables: *histories*[1] and *oracles*. Histories are used to remember information and pass it to the next process along the time line. Oracles are used to "see into the future" and pass information to the previous process along the time line. These values must be set in the code that makes up the body of the module (see below).

The format for each variable declaration is:

```
(type variable-name {keyword value}*)
```

The `type` is either `:history` or `:oracle`, indicating which type of internal variable is meant. `Variable-name` is any legal lisp symbol. The following keywords are recognized:

`:initial-value`	initial value for the variable, default `nil`.
`:test`	test function for equality testing, which is used to detect changes for requeueing the next process. The default test is `equal`.

Any number of internal variables may be declared. The values of these variables are available only in the module in which they are declared. When a module's code is executed in a process, the value of the history variables are set to the value they had at the end of the previous time interval's module execution. The value of an oracle variable is set to the value it had at the end of the next time interval's module execution.

The syntax of other options is:

```
(option-keyword argument)
```

The following options are recognized:

[1] The name has been changed from that used in the previous manual. The old name for this type of variable was a em checkpoint. This name is now obsolete, although it will be supported for a while for compatibility reasons. The orginal name is borrowed from the computer operating system concept of checkpoints—the state information needed to continue the operation of an interrupted process.

`:open-left` This option only affects modules which have point variable output. If `t` it means that the output is open at the left (earlier) end of the scale. The default is `nil` meaning that the interval is closed. Note that this does not affect access to input data.

`:open-right` This option only affects modules which have point variable output. If `t` (the default) it means that the output is open at the right (later) end of the time scale. If `nil` the interval is closed. Note that this does not affect access to input data.

`:graph-label` If specified, this should be a string. It will be used as a label if the module appears in a TCS graph. If this option is not specified or is `nil`, the name of the module will be used to construct the label.

`:queue-options` There is currently only one option, `:maximum-intervals`. If specified, then the queueing algorithm will always use maximal intervals for process creation, rather than using incremental process creation. An example of the difference is shown in Figure **??**. There is a tradeoff between the number of processes that will be created and the potential amount of recalculation that will need to be done. If a module has many input values, choosing the maximum intervals scheduling option will result in many processes being run. On the other hand, not using maximal can result in a larger number of processes being created than are strictly necessary. This can complicate the explanation of the reasoning. Also, if much information is being passed via oracle or history varaibles, then using the maximum interval scheduling may be more efficient. There is also the possibility that using the maximum interval scheduling option will make the writing of module code simpler by guaranteeing that each process spans a maximum interval of intersected input variable values.

Code is a series of lisp forms that will be executed. All interaction between this module and other modules is via the side effect of changing the output variables. Inside the body of a module when it is executing, the variable names will be available as local variables. Point input variables will have a list of point variables valid over the period in which a module process is executing. Interval input variables will have the single valid value for that execution period. Point output variables will be initialized to `nil`, interval output variables will be initialized to `:undefined` (unless they are also input variables). Output point variables must be set to be a list of valid point data items (see `make-point` on page 20). Interval variables should be set to their value.

Figure 5.2: Queue Options Keyword Effects

Other variable values accessible inside a module are `begin_time` and `end_time`, which define the limits of the process' execution period. These values can be changed as long as the interval is at least one time unit long and does not extend beyond the initial boundaries. Changing the value of `begin_time` and `end_time` is the only way to affect the extent of the value of output interval variables. The interval output variables take as their interval of validity the interval of the process that produces them. (If the temporal interval over which the process runs shrinks, the control structure automatically queues additional processes to handle the gaps thus created.)

The technique of setting the begin and end times of the module is used most frequently in transforming a stream of point data into a series of intervals. This is known as the abstraction process. What must be done is to aggregate the information from one or more point data values and then set the appropriate bounds for one interval value. This establishes the value and extent of the output interval value.

If access to information about the current time is desired, then the module should include one or more of the variables `now`, `past?` or `future?` in its input list. Details on the information contained in these variables can be found in Section 4.1 on page 17.

Note: The lists *inputs*, *outputs* and *options* are required parameters to this macro. They may not be omitted. They may, however, have the value `nil`.

A module is defined as a new flavor which inherits from the `module` flavor. The new flavor will have the name of the module as its name. Users should note that this will also define a new type with the same name as the module name. Beware of conflicts with other user defined types (such as structures). A new flavor for the processes, inheriting from the process flavor is also defined. The new process flavor will have the module name

concatenated with "`-process`" as its name. The same type definition considerations apply to this flavor.

For each process flavor, TCS defines the generic functions `return-history-values` and `return-oracle-values` (taking no additional arguments), which will return the values of the history or oracle variables for that process using the `values` function of Common Lisp. The order of the values will be the same as their order in the definition form. A list of the history and oracle variable names (in order) can be obtained by invoking the module's generic function `histories` or `oracles` with no additional arguments.

First Example:

```
(defmodule simple
   (a b)          ; inputs – assume both interval variables
   (c)            ; output – assume an interval variable
   ((:history sum :initial-value 0 :test #'=))  ; history

   (setq c (+ a b) sum (+ sum c))              ; body
)
```

This example shows a module `simple` which adds its two inputs and also keeps track of the running sum of all of the additions that were performed. The running sum is only available internally to the routine. It would be possible to add another output variable and set it to the same value if one wished to have access to the running sum outside of the module.

Second Example:

```
(defmodule percent
   (value)      ; An input value (> 0), as an interval variable.
   (percent)    ; Output, also an interval variable.
   ((:history past-max :initial-value -1000)  ; for the past
    (:oracle future-max   :initial-value -1000)) ; for the future

   (setq past-max   (max value past-max))
   (setq future-max (max value future-max))
   (setq percent    (/ value (max future-max past-max)))
 )
```

This example shows a module which sets its output variable `percent` to be the current value of `value` as a fraction of the maximum value of `value` over all time. In order to do this, the largest previous value must be remembered in an internal history variable (called `past-max`) and the largest "future" value in an internal oracle variable (called `future-max`). This allows the information from other time periods to influence the value in this period. It is the programmer's responsibility to see that these variables are properly handled. The history value will initialize `past-max` for the process scheduled to run in the time interval *after* the current interval, and the oracle value will initialize the variable `future-max` for the process scheduled to run in the time interval *before* this one. The initial value of -1000 was chosen arbitrarily. A sample run of this module is shown in Figure 5.3, with the input,

Figure 5.3: Example Module Using Histories and Oracles

output, history and oracle values identified. Note also that to be truly robust, checks for division by zero and similar standard programming conventions would need to be added to this example.

5.3 Other Module Code

`instantiate-module` *name sys* [*Function*]

Instantiates the module *name* of the system *sys*. *Name* is a symbol naming a module defined by `defmodule`. *Sys* must be a flavor instance (*i.e.*, not just the name) of a system. A control system user will probably never need to use this function, since it is called by `instantiate-system` to set up the modules for that particular system. Instantiating the

entire system is the preferred way to set up modules. The function name is an internal symbol of the `CS` package.

`module` [Inherits from `system-mixin`] [*Flavor*]

Flavor which implements a module of the control structure. The symbol `module` is an internal symbol of the package `CS`. A module has the following instance variables:

name	The name of the module.
system	The system to which it belongs. This is a system object, *not just the name!*
inputs	A list of the input variable names.
outputs	A list of the output variable names.
histories	A list of history variable names.
oracles	A list of oracle variable names.
type	`module`
processes	A list of the currently active processes for this module. The current process state.
graph-label	A pretty name used by the graphing routine for labeling graphs. If `nil` (the default), then the *name* is used instead.

All variables are readable. None are writable. *Name*, *system*, *inputs*, *outputs*, *histories* and *oracles* are initable. It is intended that instances of this flavor be created only by the `instantiate-module` function. It is strongly suggested that this only be done internally (*i.e.*, users should only use `instantiate-system`).

A module has a printed representation of `#<MODULE` *name*`>`.

`display` **module** `&optional`*format-stream* [*Generic Function*]

Prints a description of the module on *stream*. The default value of *stream* is `t`, the standard I/O stream. The format of the display shows the name of the module, the input variable names, the output variable names, the history variable names and the oracle variable names.

`why` **module** *start end* `&optional`*stream print-timebase* [*Generic Function*]

Prints an execution trace of the module covering the time from *start* to *end*. Output goes to *stream*, which defaults to `t`. The execution trace identifies the interval over which each of the processes covering the period *start* to *end* executed in, and it identifies the input values and the history or oracle values that were valid at that time. Thus, this functions attempts to explain "why" a module produced the results that it did. Note that the output produced is not printed. This function is called by the variable `why` function (see page 23).

If *print-timebase* is specified, it must be a universal time that corresponds to the "0" system time. The time part of the the why explanation is then printed using real world times such as months, days and hours. The printing is done down to the depth specified by the variable `*tprint-depth*`. The conversion between the units of system time and the real world time is controlled by the variable `*tprint-multiple*`.

`why-list` **module** *start end* [*Generic Function*]

Returns an execution trace of the module covering the time from *start* to *end*. This information is in a form that can easily be parsed in lisp. The execution trace identifies the interval over which each of the processes covering the period *start* to *end* executed in, and it identifies the input, history and oracle values that were valid at that time. Thus, this functions attempts to explain "why" a module produced the results that it did. The format of the list returned is:
((*start-i end-i input-alist history-alist oracle-alist*) ...)

Each element of this list is a subinterval of [*start..end*]. The elements are in temporal order. The individual parts are:

start-i	The starting time of this particular subinterval.
end-i	The ending time of this particular subinterval
input-alist	An association list of the names and values of input variables for the current subinterval.
history-alist	An association list of the names and values of history variables for the current subinterval.
oracle-alist	An association list of the names and values of oracle variables for the current subinterval.

The association lists are contain dotted pairs [`(name . value)`], with `name` being the variable name and `value` being its value. These component can be accessed with the `car` and `cdr` functions. The value of an interval value will be the single value valid over that subinterval. The value for point variables will be a list of point value objects, each consisting of a time and an associated value. Oracles and histories will have user-defined values.

5.4 Processes

`process` [*Flavor*]

Flavor which implements a process of a control structure. The symbol `process` is an internal symbol of the package `CS`. A process has the following instance variables:

module	The module to which this process belongs. It is the actual module, not just the name.
begin	The beginning time for the process' interval.
end	The ending time for the process' interval.
status	Status of this process. One of: `:pending` — Waiting in the queue to execute. `:in-progress` — Being executed. `:done` — Completed with normal return. `:aborted` — Aborted during execution. `:obsolete` — Superseded by more recent one. `:deleted` — Deleted while on queue.
process-id	A running sequence number used to assign a sequence to processes created by the control structure during queue execution.
execution-time	The system time at which the process was executed.
input-values	Value of the module's input variables (in order) at the time this process is executed. This is kept only if the process history for the system is enabled. Otherwise the value is `nil`.
input-histories	As above, but for history values.
input-oracles	As above, but for oracle values.

All variables are readable. *Status* is writable. *Module*, *begin* and *end* are initable.

A process has a printed representation of `#<PROCESS-`*n*`>`, where *n* is the `time-stamp`.

`display` **process** `&optional`*format-stream* [*Generic Function*]

Prints a description of the process on *stream*, which defaults to `t`. The format of the description is a line of

`PROCESS` *n* `[`*status*`] for` *module* `from` *begin* `to` *end*.

where *n* is the process id.

Chapter 6

Utilities

This chapter describes the general purpose code for the control structure. It has routines that can not be naturally associated with systems, variables and modules. The major items are the routines for doing comparisons and arithmetic with time values. There is also a section on tools to aid in the construction of systems and user interfaces with the Symbolics Lisp Machines.

6.1 General Utilities

`*trace-stream*` [*Default*: `t`] [*Variable*]

Stream to print traces to. This can be reset to divert printing into a file or other stream for debugging convenience.

`help-trace` [*Function*]

Prints a help message about the meaning of trace codes for the trace queue function (see system documentation). The trace codes are used to identify the reason a module was added to the queue. They appear in square brackets "[...]" in the trace output. Output goes to the default output stream. The meaning of the codes are:

A Process aborted. A process was aborted and the current process is added to take its place. This is done to maintain system consistency in the face of errors. If the queue is restarted, then the gap caused by the aborted process must be filled.

E Endpoint changed. The `begin_time` or the `end_time` of a process was changed (or both). A new process must be created to cover the gap(s) in the time sequence.

N Next process. If there is no following process already, then one is automatically started. This also is used when the queueing is due to a change in a history variable. This makes sure that the entire time line is covered.

P Previous process. If an oracle value changes, then the previous process is queued.

Q Queue maintenance. It is possible that a new process to added to the queue will overlap with an existing process. Unless the overlap is total, gaps may occur. This entry identifies processes queued to fill those deletion gaps.

U User initiated action. Caused by the user invoking the `queue-module` function directly. [Default reason]

V Variable value changed. A change in an input variable is the impetus.

`time` *point-variable* [*Macro*]
`value` *point-variable* [*Macro*]

Returns the time or the value part of the *point-variable*, respectively. These functions are the same as the `time-part` and `value-part` functions defined in the section on Variables. These functions assume the input of a single point variable representation, *i.e.*, a time-value pair. Note that in point variable values will almost always be available in the form of a list of such pairs.

`time` [*Representation*]

The time representation used in the control structure are the integers extended by the inclusion of two distinguished values, `:-infinity` and `:infinity` which represent the beginning and end of the time line, respectively. The time value manipulation functions which follow all work on this representation. Since times can legally include the infinity keywords, programmers should not use normal arithmetic functions for manipulating times values.

`timep` *x* [*Function*]

Returns `nil` if *x* is not a valid time. A valid time is either an integer or `:infinity` or `:-infinity`. If *x* is valid a non-`nil` value is returned.

`time=` *x y* [*Function*]
`time<` *x y* [*Function*]
`time>` *x y* [*Function*]
`time<=` *x y* [*Function*]
`time>=` *x y* [*Function*]

Comparison functions which work on the time representation.

`time-add` *x y* [*Function*]
`time-sub` *x y* [*Function*]

Implements addition and subtraction for the time representation. If *x* is `:infinity` or `:-infinity`, then the value of *x* is returned. Otherwise if *y* is `:infinity` or `:-infinity`,

then the value of y (negative y for `time-sub`) is returned. Otherwise, normal addition or subtraction occur.

`time-max` x y [*Function*]
`time-min` x y [*Function*]

Returns the maximum or minimum value. Works with the time representation.

`system-mixin` [*Flavor*]

Flavor for holding data and code that is common to all control system flavors. Currently this includes the `system`, `variable` and `module` flavors (but not `processes`). The symbol `system-mixin` is an internal symbol of the `CS` package. It has the following instance variable:

name The name of the flavor.

This variable is readable and initable. The default printed representation is used.

`name-equal` **system-mixin** *test-name* [*Generic Function*]
`flavor-name-equal` **system-mixin** *test-flavor* [*Generic Function*]

Returns `t` if the current flavor has the same name as *test-name* or the same name as the flavor object *test-flavor*, respectively. This test checks for name equality and will return `t` even if the names are interned in different packages.

6.2 Development Tools

This section contains tools to make the bookkeeping easier for systems under development.

`list-modvars` *filename* `&optional` *other-modvar-defn-forms* [*Function*]
`list-modules` *filename* `&optional` *other-module-defn-forms* [*Function*]

Returns a list of the variable or module names (respectively) contained in *filename*. The names are collected from the standard TCS definition forms for variables and various kinds of modules. The user can specify additional forms that are used for definitions. The only restriction on user-specified forms is that the name of the variable or module be the `cadr` of the form. The user specified form names (which must be the `car` of the form) must be a list that is passed as the optional argument *other-modvar-defn-forms* (or *other-module-defn-forms*)

`list-modvars-and-modules` *filename* `&optional` *other-modvar-defn-forms* *other-module-defn-forms* [*Function*]

Returns a multiple value of variable names and module names. This is functionally equivalent to using both forms above sequentially, but is more efficient. As above, additional definition forms may be specified as optional arguments.

`construct-system-definition` *sysname input-filenames* : *variable-forms* *:module-forms* [*Macro*]

Constructs a `defsystem` form for a system with the name *sysname*, and variables and modules determined by reading the forms in the files in the list *input-filenames*. The reading is done using the function `list-modvars-and-modules`. Additional definition forms beyond what the TCS itself provides can be specified in the keyword arguments `:variable-forms` and `:module-forms` for variable and module definers, respectively. The restrictions listed above also apply to the definition forms specified for this routine. All of the arguments to this routine are evaluated except for *sysname*.

A typical use of this macro would be to include it as a form in a set of files comprising a TCS system. It will then automatically generate the system definition based on the contents of the files specified in the filelists. It will therefore always be up-to-date when changes are made to the file.

Example:

```
(construct-system-definition my-system '("SystemFile1" "SystemFile2")
            :module-forms '(my-module-definer another-module-definer))
```

6.3 Interaction Tools

`*current-system*` [*Value*: `nil`] [*Variable*]

Holds the current system. This is used to specify a system to use for context when querying the user about variable or module names. It can be bound by the user before a function call. It is set globally whenever `instantiate-system` is invoked to create a new system.

`*tprint-depth*` [*Value*: `:sec`] [*Variable*]

Specifies the smallest time unit printed by the variable query options (below) or the `why` functions (further below). Allowable values are `:sec, :min, :hour, :day, :month` and `:year`.

`*tbase*` [*Value*: `nil`] [*Variable*]

Holds the base time for using real dates (rather than the integers used inside TCS) for input queries. The format is that used by the common lisp universal dates. By default it will be nil, which means that the current time will be used for input routines.

`*tprint-multiple*` [*Value*: `1`] [*Variable*]

Holds the number of seconds that each TCS time value represents. This is used in the translation of times from real world format into the TCS internal representation. The default

value of 1 means that each TCS time unit is one second. A value of 60 would correspond to each time unit representing a minute, a value of 1/1000 would mean milliseconds, etc. This is used only in the output from the `why` functions.

New keywords have been defined for use with the `tv:choose-variable-values` function of the Symbolics Lisp Machines. They extend the keywords already available with that function. Each of the keywords exists in a mandatory input and an optional input form. The optional input forms have the suffix "`-or-nil`" attached. Some input forms rely on the value of `cs:*current-system*` being bound to an instantiated system. The real world date and past date functions use the value of the variable `cs:*tbase*` being set to a time value for base date calculations.

`:tcs-time`	Accepts only a valid time specification: a number, `:-infinity` or `:infinity`.
`:tcs-time-or-nil`	As above, but an empty (null) answer is accepted as well.
`:tcs-system`	Requires the name of an instantiated system.
`:tcs-system-or-nil`	As above, but an empty (null) answer is accepted as well.
`:tcs-variable`	Requires the name of a variable belonging to the system in `cs:*current-system*`. If this variable does not contain a valid system, then `nil` is returned.
`:tcs-variable-or-nil`	As above, but an empty (null) answer is accepted as well.
`:tcs-module`	Requires the name of a module belonging to the system in `cs:*current-system*`. If this variable does not contain a valid system, then `nil` is returned.
`:tcs-module-or-nil`	As above, but an empty (null) answer is accepted as well.
`:real-tcs-date`	Accepts a real date. Any date parsed by the Symbolics software, as well as the special values "infinity", "-infinity", "negative infinity" and "minus infinity" will be accepted. Note that these are text strings instead of keyword values (in contrast to `:tcs-time`)! The value will be translated to a system time integer value. Incompletely specified times will be taken to be in the future. Ignores `cs:*tbase*`.
`:real-tcs-past-date`	As above, but incompletely specified times will be taken to be in the past.
`:finite-tcs-date`	Accepts a finite real data. Ignores `cs:*tbase*`. Incompletely specified times will be taken to be in the future.
`:finite-tcs-past-date`	As above, but incompletely specified times will be taken to be in the past.
`:based-tcs-date`	Like `:real-tcs-date`, but uses `cs:*tbase*`.
`:based-tcs-past-date`	Like `:real-past-tcs-date`, but uses `cs:*tbase*`.
`:based-finite-tcs-date`	Like `:finite-tcs-date`, but uses `cs:*tbase*`.
`:based-finite-tcs-past-date`	Like `:finite-tcs-past-date`, but uses `cs:*tbase*`.

The value returned by the system, module or variable keywords will be the system, module or variable object, respectively. Note that it will be the object and not the name of the object that is returned. The time options return a TCS time value (integer). In contrast, the date options return a universal time.

Example:

```
  (let ((*current-system* (find-system "my-system"))
variable begin end)
    (declare (special variable begin end))
```

```
    (tv:choose-variable-values
      '((variable "Variable to Graph" :tcs-variable)
(begin "Start of Data" :tcs-time)
(end "End of Data" :based-tcs-date)))
    (list variable begin end))
```

The above program will accept integers and the values `:infinity` and `:-infinity` for the start of data and regular date expressions such as "2/22/89 12:31" for the end of data. The starting time will be a TCS time whereas the ending time will be a Common Lisp universal time.

`parse-real-tcs-date` *string* `&optional` *past-p infinite-p base-time* [*Function*]

Parses *string* and returns a universal time, `:infinity` or `:-infinity`. If *past-p* is `nil` (the default), then all ambiguous times will be in the future. Otherwise they will be in the past. If *infinite-p* is `t` (the default), then infinity values are allowed. In this case, the forms recognized by the Symbolics time string parser are extended by *infinity*, *-infinity*, *negative infinity* and *minus infinity*. If *base-* is specified, it must be a universal time. Times and *past-p* are then interpreted relative to this base time rather than relative to the current time. More details of the parsing algorithm can be found in the Symbolics documentation.

Chapter 7

Support Modules

This chapter describes support routines that simplify the process of using the control structure to do certain simple but often needed tasks. Additional definition macro forms were created which provide a simpler interface for these functions. Currently five basic macro families are supported.

The first is for the definition of a class of modules known as *transducers*. These modules take a single point variable input and have a single point variable output. User specified processing is used to modify the value part of the point variable. No time processing is involved.

The second is for the definition of simple *rules*. These are modules that take any number of interval variables as inputs and have any number of interval variables as outputs. The user specified processing transforms the input values into the output values. Again, no explicitly time dependent reasoning is involved.

Both of the above forms are mappings from a single type of variable to the same type. There is no transformation between point and interval variables.

The third form provides for system-wide *memory* of events. There are two dimensions along which memory is classified. The first describes whether all past values or only the most recent value is remembered. This is the distinction between full or not full memory. The other dimension distinguishes time dependent from eternal memory. Time dependent (decaying) memory sets a time limit on the length of time any part of its memory will retain information. This time is currently fixed for the type of memory and cannot be made data dependent. These options can be specified by the use of keyword arguments.

The fourth form provides support for the definition of *persistence* modules which take point data as input and produce interval variables as output. They allow the user to specify how long after a data point is seen that belief in the presence of that state will persist. A variation of this type of module is a *2-point abstractor*, which applies a functions to consecutive pairs of points.

The fifth form provides support for *pattern matching* among interval variable values. The temporal patterns supported are of fixed length and can specify values as well as duration constraints.

7.1 Transducer Modules

A transducer changes only the *value* of a point variable, leaving its time part unchanged. The function that does this can either take only the point variable input, or it can have a *context* established by additional interval variables.

`deftransducer` *name input output function* :*context :open-left :open-right* [*Macro*]

Defines a transducer from input to output. *Name* is the name of the transducer module. It must be a symbol. *Input* and *output* are the names of the input and output variables. They must be declared with `defmodvar` and must be of type `:point`. *Function* should be the name of a function or a lambda expression. It will be `funcall`ed, so it should be quoted with #'. If no context is specified, this function must take one argument and should return only one value (additional values will be discarded). The `:context` argument is an unquoted list of the names of the input interval variables that establish the context for the transducer function. By default it has the value `nil`, which means no context variables are used. If a context is specified, then *function* must take enough arguments to account for the point data value and the value of each context variable. The point data value will come first, followed by the context variables in the order given in the definition. The function should return only one value (additional values will be discarded). Note that in either case the function receives only a single point variable value datum, not a list of point variable structures nor a complete structure. The time part of the point variable is not accessible to the function.

The keyword arguments `:open-left` and `:open-right` are used to set the options for the module. Their effect is described on page 30. This controls which context is used when a point variable occurs at the boundary between two intervals in the context. The default setting (`:open-left nil, :open-right t`) means that the temporally later value is used (*i.e.*, the evaluation is open on the right, but not on the left).

`defcontext-transducer` *name input context output function* :*open-left* :*open-right* [*Macro*]

This function is now obsolete since its function has been included in `deftransducer`. It will continue to be supported for a while for compatibility reasons, but will eventually disappear.

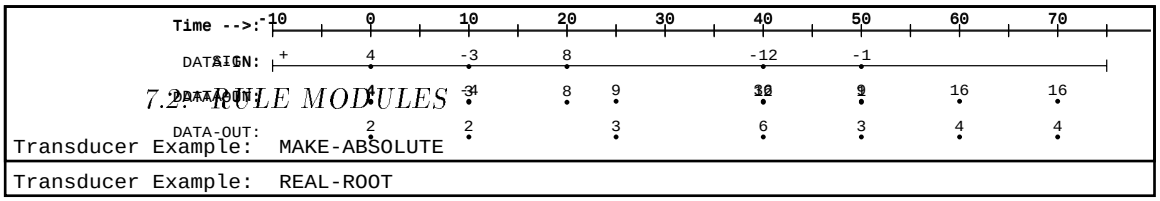


Figure 7.1: Output from Module Make-Absolute

Figure 7.2: Output from Module Real-Root

Examples:

```
(deftransducer make-absolute data-in data-out #'abs)
  ;; Defines a transducer which makes the output variable
  ;; data-out have the absolute value of the values of
  ;; the input variable data-in.

(deftransducer real-root data-in data-out
        #'(lambda (x sn)
            (if (eq sn '+)
                (sqrt (abs x))
                (- (sqrt (abs x)))))
        :context (sign))
  ;;  Data-out will hold the real square root of
  ;;  data-in.  The sign is determined by the value of
  ;; the context variable  sign
```

Note that none of the arguments to the macro are quoted except for the function. The output of these two modules for sample input is shown in Figures 7.1 and 7.2.

7.2 Rule Modules

A rule is a simple module that transforms interval values into other interval values.

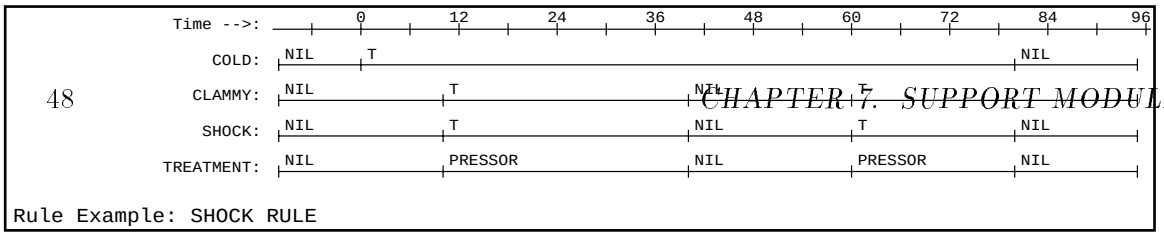


Figure 7.3: Output from Module Shock-Rule

`defrule` *name input-list output-list function* [*Macro*]

Defines a rule from input-list to output-list. *Name* is a symbol which will be the name of the module. *input-list* is a list of the input variables used by the rule. *output-list* is a list of the output variables whose value is determined by the rule. All such variables must be defined to be interval variables using `defmodvar`. *Function* is the name of a function (or a lambda expression itself) which implements the mapping from the inputs to the outputs.

The function will be invoked (using `funcall`) with the values of the variables in the *input-list* as arguments. The order of the arguments will be the same as the order in *input-list*. The function must return a value for each variable in *output-list*. If there is only a single variable in *output-list*, then a regular return can be used. Otherwise, the `values` form must be used to return multiple-values. The order of the values returned must be the same as the order of the variable names in *output-list*.

Example: (assumes the variables are already declared):

```
(defun rule-function (cold clammy)
  ;;  Implement rule: If cold and clammy, then shock is
  ;;  present and pressor should be given.
  ;;
  (if (and cold clammy)
      (values t 'pressors)
      (values nil nil)))

(defrule shock-rule (cold clammy) (shock treatment)
        #'rule-function)
```

Note that none of the arguments to the macro call are quoted, and that the function takes 2 arguments and returns 2 values as required by the rule definition. Sample output is shown in Figure 7.3. By associating a function with the inputs, this syntax allows a more versatile rule than the standard rule-based system, since arbitrary processing is possible. Another difference is that since each variable can be the output of at most one module, multiple rules that all affect the same output are not permitted. They must all be collapsed into one rule function.

7.3 Memory Modules

Memory modules are used to provide a system-wide accessible memory function for point data. The variations are controlled by options given to the macro. They are explained below. Future modules are analogous to the memory modules, except that they look into the future rather than remember the past. These differ from persistence modules (explained later) in that they remember the full contents of the point variable (including the time part), whereas the persistence modules retain only the value portion.

`defmemory` *mod-name input output* :*equal-test :decay-time :full-memory :earliest* [*Macro*]
`deffuture` *mod-name input output* :*equal-test :decay-time :full-future :latest* [*Macro*]

Defines a memory or future module. *Mod-name* is the name of the module that is to be defined. *Input* is the name of a point variable (defined with `defmodvar`) that contains the events that are to be remembered. The memory variable works only with point variable input, not with intervals! *Output* is the name of an interval variable that will hold the memory contents. The memory contents will be the value of the point variable and the time at which that value occurred. In essence, a simple memory copies the point variable into the value field of an interval variable. There are several keyword options. The options listed below are for the memory form. The `deffuture` form has the analogous operation performed:

`:equal-test`	A function for testing for value equivalence. Used in conjunction with `:earliest`. It only affects simple (not full) memories. Default value is `eql`.
`:decay-time`	The duration of a memory event. If specified, this should be an integer. All events will be kept in the memory variable only for this duration. Older events will be eliminated. If specified with the value `nil` [the default], then there is no decay (events are remembered forever).
`:full-memory`	When `t`, then full memory is enabled. This means that all of the events are remembered, subject to `:decay-time`. If `nil`, then only the most recent event is remembered, again subject to modification by `:decay-time`. For future modules, this is called `:full-future`.
`:earliest`	For simple (not full) memories, if T, then the earliest time is remembered for point variable data. In other words, if a series of point variable forms are processed, and all of them have the same value part according to the test function, then the earliest time remains stored in the memory variable. If decay-time is specified, the most recent time is still used for the decay calculation, even though the earliest time is stored in the output. If `nil`, then only the most recent time is placed in the output variable. The choice is between having the first sample value out of a run remembered or just the most recent. In full memories, the point is moot. For future modules, this is called `:latest`. The default value is `t`.

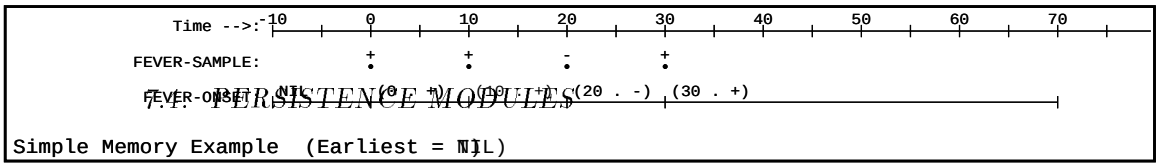
The options combine to give the following general types of memory modules:

Simple: Only the latest value is kept. Depending on the value of `:earliest`, either the first time (`t`) or the last time (`nil`)is also retained. For future modules, this behavior is reversed: `t` remembers the last (most future event) is remembered. This distinction is only important when the same values appear consecutively.

Simple Decay: Like simple, but the memory is reset to `nil` after the decay time. If `:earliest` (`:latest`)is `t`, the reset is still based on the most recent (oldest) data, even though the earliest (latest) time is carried forward. It can thus be the case that the time part of the retained value will be further away than the decay time of the memory.

Full: Remembers the entire list of point variable values to the current time.

Full Decay: Remembers all of the point variable values that are not older than the decay time.



(defmemory fever-onset fever-sample f-onset :earliest <*variable, see below*>)

;; *Remembers the onset of the current fever state,*
;; *either present (+) or absent (-). Assumes that*
;; *the input and output variables are properly declared.*

This figure shows the time line for `fever-onset`, assuming two different values for the `:earliest` keyword argument. The "•" indicates a data point location. The value is above the data point. Vertical bars (`|`) separate intervals. The value is written during the interval itself. Thus `F-ONSET` has the value "`(0 . +)`" from time `0` to time `10`.

Figure 7.4: Simple Memory Module

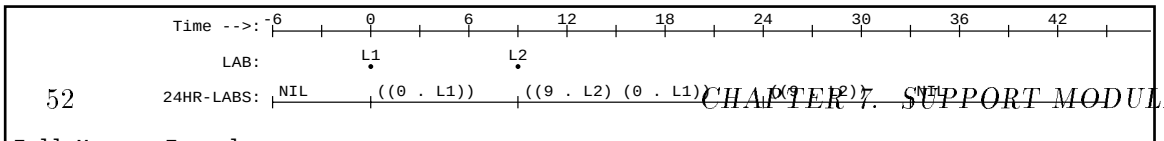
Examples of memory modules can be found in Figures 7.4 and 7.5.

7.4 Persistence Modules

This type of module takes point data and transforms it into intervals. The exact form of the transformation is governed by the persistence or anticipation (persistence into the past) of the data.

`defpersistence` *name input output :persistence :2nd-persistence :anticipation :2nd-anticipation :default :context :persist-use-oldest-context :anticipate-use-newest-context :transform :test :filter-in :filter-out* [*Macro*]

A reasoning module named *name* is defined that takes the point variable *input* and extends the value part of each entry. The results are placed in the interval variable *output*. All of the required arguments must be symbols. The variables for the input and output must be defined to be of the proper type using a `defmodvar` form. The value extension is similar



Full Memory Example

```
(defmemory day-lab lab 24hr-labs
          :full-memory T  :decay-time  24)
  ;;  24hr-labs  will hold all  lab values for the previous
  ;;  24  time units.  We assume a time unit of one hour.
```

This figure shows a full decay memory. The "•" indicates a data point location. "L1" and "L2" are different lab reports. The value of 24HR-LABS is shown in intervals separated by "|". For example, its value from 9 to 24 is ((9 . L2) (0 . L1)).

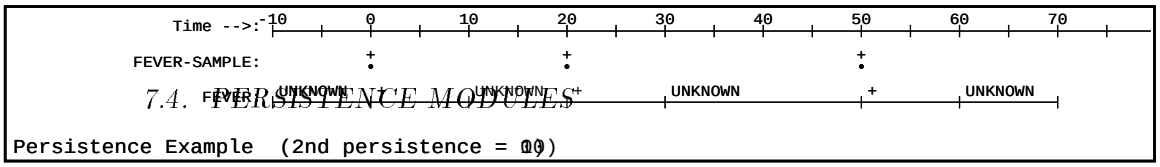
Figure 7.5: Full Memory Module

to that of a simple decay memory, except that only the value is written forward, not the time. Furthermore, it is possible to have the value be written backward in time, in effect, anticipating the data value. Intervals that are not covered by the specified persistence take on the value of the :default argument, whose own default value is :unknown.

The temporal extent of the interval from a point is determined by the keyword arguments :persistence, which defaults to :infinity and :anticipation, which defaults to 0. The arguments :2nd-persistence and :2nd-anticipation only make sense when :persistence or anticipation, respectively, is set to some finite value. By default, each is set to 0. The effect of the 2nd-options is to provide a secondary persistence (anticipation) to a value when there is confirmation of the value. The way this works is as follows: A point value in the absence of any further (later) data is converted into an interval that extends :persistence time units beyond the time of the datum. If, however, a later data point has the same value, then the interval will extend up to :persistence plus :2nd-persistence time units in order to bridge a potential gap between them. This assumes that the extended persistence is sufficient to bridge the gap. If it is not, then only the :persistence value will be used. See the example in Figure 7.6. Anticipation works analogously. If both are present, the the values used for determining whether gaps are bridged is the sum.

When both :anticipation and :persistence are specified, it may be necessary to split the time between input data points because the forward and backward values more than cover the gap. If this is the case, then the difference in time is split, with the ratio being the same as that between the anticipation and persistence values. This will be rounded to the nearest integer. If either value (but not both) is :infinity, then that value will take all of the time between values. If both persistence and anticipation are :infinity, then the difference is split evenly.

The :context argument allows the specification of a context which can be used in




```
(defpersistence fever fever-sample fever :persistence 10
        :2nd-persistence <variable, see below>)
 ;;  Turns  fever-sample into the fever interval.
 ;;  It is either present ( +) or absent ( -). Assumes that
 ;;  the input and output variables are properly declared.
```

This figure shows the time lines for `fever`, assuming two different values (`0` and `10`) for the `:2nd-persistence` argument. The "•" indicates a data point location. The value is above the data point. Vertical bars "`|`" separate intervals. The value is written during the interval itself. "`??`" denotes the value `:UNKNOWN`.

Figure 7.6: Persistence Module

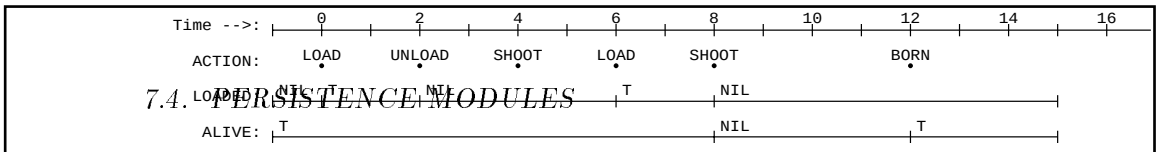
conjunction with the `:filter-in, :filter-out`, and `:transform` arguments. The value of `:context` is an unquoted list of the names of interval variables that are used for establishing a context for making decisions about the persistence of point variables. The context can affect the values of the point variable considered for persistence (via the filter functions) and it can affect the value transformation that is performed (via the transform function). Note that any filter or transform function specified must accept the context variables as arguments, even if they are ignored in the processing. There is also an ambiguity in the choice of context for point values which are at the same time as the endpoints of a context interval. By default, the context present for the output interval is used to determine the action taken. This will be the context following the point value for persistence (the newest context) and the context preceding the point value for anticipation (the oldest context). It is, however, possible to specify that the oldest (for persistence values) or newest (for anticipation) context should be used by specifying `t` for the keywords `:persist-use-oldest-context` and `:anticipate-use-newest-context`. By default these arguments have the value `nil`. The use of context and the choice of which context to use is illustrated using the standard gun firing example from non-monotonic reasoning work (see Figure 7.7).

The `:transform` argument must be a function that returns the value that the interval associated with that point datum value should have. (This allows what is essentially an in-line transducer.) If a function is specified, then it will be called with the value part of the *input* point variable as its first argument. If a context argument is specified, then the context variable values will follow in the order specified in the macro invocation. Any function specified must accept the number of arguments implied by the input and context values. The default value is `nil` (no transform), in which case the interval has the same value as the value part of the point datum from which it is derived.

The `:test` argument is used to specify the test function used to compare transformed data values for equality. It is applied after the transform and is never affected by context variables. It defaults to `eql`.

The `:filter-in` and `:filter-out` arguments allow functions to be specified that filter the raw input data. If specified, the functions will be called with the value part of the *input* point variable as its first argument. If a context argument is specified, then the context variable values will follow in the order specified in the macro invocation. Any function specified must accept the number of arguments implied by the input and context values. The default is `nil` (no filtering), in which case all of the input point data is used. The action taken depends on which filter keyword is used to specify the function. If `:filter-in` is used, then all point values for which a non-`nil` value is returned are used (the filter admits points for consideration). If `:filter-out` is used, then all point values for which a non-`nil` value is returned are discarded (the filter blocks points from consideration). Filtering occurs before the value transformation (by `:transform`) is done. By default, all point values are used. This option can be used, for example, to consider only certain values of the point data for persistence. One possible application would be in generating different persistent intervals from one stream of input data using selected portions of the data.

All keyword argument values will be evaluated, all regular required arguments will not.



Gun Firing Example

```
(defsystem gun (action loaded alive) (gun-state person-state))

(defmodvar action :point)
(defmodvar loaded :interval :initial-value nil)  ;; When t, the gun is loaded.
(defmodvar alive  :interval :initial-value t)  ;; When t, the person is alive.

  ;; The only actions that affect gun state are load, unload and shoot.
  ;; If the action is load, the gun becomes loaded, otherwise it is unloaded.
(defpersistence gun-state action loaded
:filter-in #'(lambda (x) (member x '(load unload shoot)))
:transform #'(lambda (y) (eq y 'load))
:default nil)

  ;; The only actions that can affect person state are born, die or shoot.
  ;; The context c is used to determine whether shooting will have any effect on life.
  ;; Shooting can only affect life if the gun is loaded.
(defun life-effect-filter (act load)
  (if load
      (member act '(born die shoot))
      (member act '(born die))))

  ;; The effects of the gun being loaded are handled at the filter stage.  A person is
  ;; alive after being born, otherwise any action will kill them.
(defun life-effect-transform (act load)
  (declare (ignore load))
  (eq act 'born))

  ;; The context uses the oldest context at a point because the points represent state
  ;; changes and should not be evaluated using the results of a state change.  For instance,
  ;; after the shoot action, the gun becomes unloaded.  If the oldest state were not
  ;; used, the person would not die, since the evaluation of the shooting would be done
  ;; in the context of an unloaded gun.  If the previous context is used, however, we
  ;; get the results we want.
(defpersistence person-state action alive
:filter-in #'life-effect-filter
:transform #'life-effect-transform
:default t
:context (loaded) :persist-use-oldest-context t)
```

The graph shows the results from running the above code. The "•" indicates a data point location. The value is above the data point. Vertical bars "|" separate intervals. The value is written during the interval itself.

`def2point` *name input output function* :*context :value-only :apply-to-nil :default* [*Macro*]

Defines a module with the name *name*, which takes a point variable *input* and produces values for the interval variable *output*. The output values are calculated by `funcall`ing *function* on consecutive pairs of point values. The duration of the interval value is the distance between the points used to calculate the value. The points will be supplied to *function* as two arguments in chronologic order. The point values will be complete time-value pairs. If *context* is specified, it should be a list of interval variable names. Their values are supplied to *function* in the order given following the two point variable arguments.

The `:value-only` option specifies that *function* is to be called with only the value parts of the point variables. The time part is discarded. By default, this option is `nil`, meaning that the entire time-value point variable structure is passed.

The `:apply-to-nil` option controls the behavior at the ends of the series of point values. Unless there is a point at both `:-infinity` and `:infinity`, there will be an interval which does not have a point value at an end. If this option is `t`, then *function* is called with the non-existent point value represented by `nil`. If this option is `nil`, then *function* is not called and the interval receives the value specified by the `:default` option. If `:apply-to-nil` is not specified by the programmer, then it has the value `nil` if `:value-only` is `t`. Otherwise it has the value `t`. The default value for `:default` is `:unknown`.

Note that if both `:value-only` and `:apply-to-nil` are `t`, *function* will not be able to distinguish between a missing point variable item and a point variable item whose value part is `nil`.

Example:

```
(def2point trend data-in trend-out
     #'(lambda (x y)
           (cond ((> y x) 'increasing)
                 ((< y x) 'decreasing)
                 (t 'steady)))
     :value-only t)
  ;; Calculates the change between two point values.  It uses only
  ;; the value parts and is not applied beyond the last point in the
  ;; input data sequence.
```

7.5 Temporal Patterns

This section presents support programs for the definition of simple temporal pattern matching. The patterns that can be matched are simple fixed-length patterns with optional duration constraints. Pattern matching is only available for interval variables. The fixed-length requirement limits the amount of history information that must be stored in the

pattern-matching module thus improving the updating efficiency of the modules in the face of changing data.

`pattern` *symbol* [*Macro*]

Returns the pattern associated with *symbol*. This pattern is the one specifed in the `defpattern` form for the module with the name *symbol*. The stored form of the pattern is the same as that used to specify patterns to `defpattern` (see below).

`defpattern` *name input output pattern* [*Macro*]

Defines a module with the name *name*, which produces the value `t` for the interval variable *output* covering each interval over which the pattern *pattern* is found to be present in the interval variable *input*. This establishes a record of which intervals satisfy the pattern. The pattern itself will be saved and can be retrieved via the `pattern` macro. A pattern is a list of one or more of the following forms:

(`:value` *value*) Matches if the tested value is `equalp` to *value*.

(`:not` *value*) Matches if the tested value is not `equalp` to *value*.

(`:member` *value-list*) Matches if the tested value is in *value-list* using a test of `equalp`.

(`:none-of` *value-list*) Matches if the tested value is not in it value-list using a test of `equalp`.

(`:any t`) Always matches. The value `t` must be supplied for proper keyword parsing, even though it is ignored in the processing.

(`:predicate` *function*) *Function* must be a function of one argument. It is called with the value for the tested interval and should return a non-`nil` value if the argument is acceptable as a match.

(`:full-predicate` *function*) *Function* must be a function of three arguments. It is called with the value of the tested interval, the starting time of the interval and the ending time of the interval. It should return a non-`nil` value if the argument and the temporal constraints are acceptable as a match. Note that the times can include the special values `:infinity` and `:-infinity`, so the special time arithmetic functions should be used.

In addition to the basic forms listed above, all keywords except `:full-predicate` can also have `:max` and `:min` keywords followed by a non-negative value. These values are used to put lower and upper inclusive bounds on the length of time that an otherwise suitable match must last. If not specified, then the values 0 and `:infinity` are used. Note that even with a 0 lower bound, the pattern element *must* be present (*i.e.*, there is a practical lower bound of 1) A fixed time length pattern can be enforced by making the `:min` and `:max` values the same.

Example: `'((:value b :min 10) (:member (a e i o u) :min 20 :max 50))`

This will match any part of the time line where there is a `b` of at least 10 units duration followed by a single instance of *a, e, i, o, u* of 20 to 50 units duration.

Example: `'((:value a :min 10) (:not a :max 50) (:value a :min 10))`

This will match a pattern of two `a`'s of duration at least 10 separated by something other than an `a` of duration not greater than 50.

Chapter 8

Graphic Output

This module is used to provide a graphic interface for viewing the values of control structure variables. For ease of use, one may wish to import the external symbols for this utility. They are not imported into the `CS-USER` package by default to avoid name conflicts. They consist of:

```
create-graph     end        start
destroy-graph    expose     title
draw-graph       graph      var-list
                 refresh
```

The graphic output will consist of a time line across the top of the screen, with the chosen variables displayed beneath it. On the left side, the variable name will appear. Point variables will have a marker identifying their location on the time line and the value will be printed above the marker. Interval variables will have an interval marked off on a line with the value for each segment above the line. The value of the variable's `display-translation` option can be used to abbreviate the value names and thus make the display more readable.

It is also possible to specify a module as one of the entities to be graphed. In that case, a display similar to that of an interval variable, put labeled with the process numbers of processes which have executed over the graphed time interval will be displayed. This allows one to examine the temporal extent and number of control structure processes are current.

The graph which is created will always appear in the upper left corner of the display. It can be moved around using the Symbolics operating system screen manipulation facilities and the mouse. The priority given to the graph is very low, so that if it becomes partially covered, it will move to the bottom of the window stack. For this reason, the `expose` function will often be needed.

`var-height` [*Value*: `29`] [*Constant*]

Number of vertical pixels per variable entry. The number of variables that can be displayed is the vertical resolution of the display divided by this value, minus 1 (for the

time axis). For a typical display, this means up to about 25 variables can be displayed at once.

`min-graph-width` [*Value*: `500`] [*Constant*]

Minimum width of the graph in pixels. The keyword width argument given to the `create-graph` function must be at least this large.

`*graphic-surface-type*` [*Value*: `'tv:window`] [*Variable*]

Holds the default type of graphics surface. For normal conditions, the default value should be fine. However, it may be necessary to change the default type for use with the Symbolics Frame Up program that uses panes. In this case, a flavor name such as `tv:window-pane` should be used. The graphing package requires only that the `tv:window` functions be supported. If a new value is used, then it should be bound around the invocation of the `create-graph` function (see below).

`*name-truncate-character*` [*Value*: `#\` ←] [*Variable*]

This is the character used to indicate that a name has been truncated in order to fit in a graph label, or in the space available for the printing of an interval variable value.

`graph` [*Flavor*]

Implements the graph object. It is used to hold the graph. It should be created only with the `create-graph` function (see below), which also sets up the window for the display of the graph. The actions of the graph are controlled by sending messages to the graph object. It is important for an application to set the `var-list` to the list of variables (actual control system variables, not just names!), and the `start` and `end` values to the display start and end. If modules are to be displayed, they should also be added to `var-list`. Again, these must be actual modules, not just their names. Variable graphs will be labeled with the variable names. Module graphs will be labeled with a "[P]" followed by the module name. The "[P]" is a reminder that processes for that module will be displayed. Processed are displayed as "`P--<`*process-id*`>`" in the appropriate intervals.

This flavor has the following instance variables:

var-list	List of variables to be displayed. The list is displayed in order from top to bottom. Must be set by the user using `setf` and the `var-list` accessor function in order for any variables to be displayed.. In addition to variables, modules can also be added to this list. Default value is `nil`.
now	If non-nil, this must be a point variable. The times of the point variable will be displayed as points on the time line drawn for the graph. For example, by setting it to the system point variable now, one could have the current time indicated on the time line. Only the time position is marked. The value is not displayed. The default value is `nil`.
start	Start time for the graphic display. Should be set by the user. Default is 0.
end	End time for the graphic display. Should be set by the user. Default is 100.
title	Title for the graph. Must be a string or `nil`. Default is `nil`.
vname-size	Maximum number of variable name characters that can be displayed for this graph size. Excess characters will be truncated and the symbol in `*name-truncate-character*` used.
surface	The drawing surface itself.
axis-title-function	Function that provides the time axis label. If `nil`, then a default value of "Time $\longrightarrow$" is used. The function can be user specified. If a function is given, it must accept three arguments (size of label field in characters, beginning time of graph, end time of graph) and return a string.
axis-format-function	Function that formats the time labels of the graph axis. If `nil`, then a default function that displays the raw system time is used. If a function is given, it must accept one argument (the time value) and return a string. The string will be centered over the tic mark.
vname-start, gstart, gend, za, zb, ts, f	Internal graph-related values.

Var-list, *now*, *start*, *end*, *title* and *surface* are readable, writable and initable. *Axis-title-function* and *axis-format-function* are readable and writable. In addition, *vname-size* is readable. It is strongly recommended that users not set *surface*. *Var-list*, *start* and *end* **should** be set by the user. The *title* is optional. These variables may be set using the standard instance variable setting mechanism: `(setf (cs:var-list` *graph*`)` *list*`)`, for example.

`create-graph` *n-vars* :*width* :*window* [*Function*]

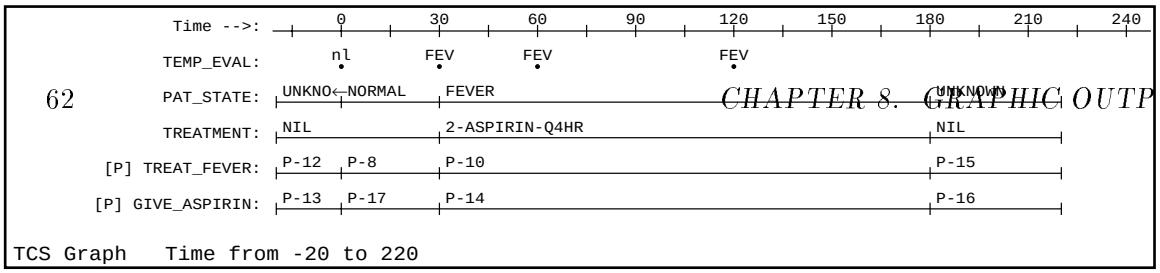


Figure 8.1: An Example of a Control Structure Graph

Returns a graph object with a drawing surface configured for the display of *n-vars* variables. This number is the maximum number of variables which will have space (subject, of course to physical display limitations). Fewer variables may be specified in *var-list*. If more variables are specified, the remaining ones will not have room on the display. The keyword variable `:width` can be used to specify the desired width of the full graph (in pixels). It must be small enough to fit on the screen (otherwise an error is signaled). By default it used the entire width of the screen. The graph will always appear in the upper left corner of the terminal. The width must be at least as large as *min-graph-width*. The basic type of the graph surface is controlled by the variable `*graphic-surface-type*`.

If the `:window` argument is specified, it overrides the *n-vars* and `:width` arguments. The window itself becomes the plotting surface. If this is done, then the number of variables that can be displayed is the height of the window divided by `var-height` minus one (for the time axis). The window type must include `tv:window` and should have the `:save-bits` option set to `t` and the `:blinker-p` set to `nil`.

The amount of space allocated to displaying variable names will be determined by the width of the graph. It will always be at least 10 characters. Excess characters will be truncated and the last character replaced by the "←" symbol. The size for a particular graph can be determined by using the `vname-size` accessor function for the graph object.

An example of such a graph is shown in Figure 8.1. It displays one point variable, two interval variables and two modules' processes. The variables and modules are taken from the example system described in Chapter 9. The start time was set to –20 and the end time to 220.

`how-many-variables` **graph** [*Generic Function*]

Returns the number of variables that can be displayed on the **graph**. This is particularly useful when the graphing surface's size is determined by using the `:window` argument to `create-graph`. If the number of variables to be graphed exceeds this number, then an error will be signaled by `refresh` or `draw-graph`.

`expose` **graph** [*Generic Function*]

The graph surface initially starts off screen. This method causes the graph to become visible, and move to the top of the window stack. It can be used anytime the graph is buried.

`deexpose` **graph** [*Generic Function*]

This moves the graph surface to the bottom of the window stack. This may or may not hide the graph.

`draw-graph` **graph** `&optional`*start-time end-time* [*Generic Function*]

Draws a representation of the variables in the `var-list` on the graphics surface. Optionally *start-time* and *end-time* may be specified. Otherwise the start and end times previously set in the graph flavor are used. Remember that the start and end times are initially unbound. If the times are specified in this function, they replace any currently stored values in the graph's instance variables. If the number of variables to be graphed exceeds the capacity of the graphing surface, then an error is signaled.

`refresh` **graph** [*Generic Function*]

Clears the graph surface and redraws the graph. This should be invoked after any change to the variable values, in order to show the new state of the system. The graphics interface itself has no automatic link to the system that is displays, so the updates must be explicitly called for by the user. If the number of variables to be graphed exceeds the capacity of the graphing surface, then an error is signaled.

`destroy` **graph** [*Generic Function*]

This should be called when the graph is no longer needed. It removes the graphics surface and frees up the window resources that were used to create it. This is the method invoked by the `destroy-graph` function. The latter is the preferred method of invocation, as other clean up forms may be present in the `destroy-graph` function.

Chapter 9

Examples

This chapter will present one extensive example of TCS code, a sample use of the system, and some modifications to the original code.

9.1 Fever Example 1 Definition

This section presents an extended example of programming using the control structure. The example problem is the detection and care of fever in a patient. The medical knowledge is simplistic, since this is meant to show in principle how one would go about coding a system.

```
;; File:          /u/tcs/example/example.lisp
;
; This is an example of a simple system that uses the control structure for
; reasoning.  The problem that is being addressed is the recommendation of
; therapy for fever.  Body temperature is the basic input variable that the
; system uses.  It goes through the stages of data transducers, data
; abstractors, rules and generators.  The medical knowledge in the system is
; not very sophisticated.  The example is designed only to demonstrate
; features of the control structure.
;


;; First set up the package to use the control structure.  The CS-USER
;; package imports the basic functions needed for control structure
;; programming.

(in-package "TCS-USER")

;; SYSTEM
;;
;; Set up the system definition.  The system name will be EXAMPLE, and it
```

```
;; will have the the variables BODY_TEMP, TEMP_EVAL, PAT_STATE, TREATMENT
;; and DO_ACTION.  The reasoning modules used will be TEMP_EVAL,
;; FEVER_DETECT, TREAT_FEVER and GIVE_ASPIRIN.
;;

(defsystem example
  (body_temp temp_eval pat_state treatment do_action)
  (temp_eval fever_detect treat_fever give_aspirin))

;; VARIABLES
;;
;; Now the variables will be set up.  None of the special options will be
;; used because this is a fairly simple system.
;;

(defmodvar body_temp :point)    ; This will hold the basic input data.  The
                                ; ... point values will be degrees Fahrenheit.
(defmodvar temp_eval :point    ; Holds the evaluation of the body temperature.
  :display-translation         ; ... This will be either NORMAL or FEVER.
  '((normal . "nl") (fever . "FEV")))

(defmodvar pat_state :interval) ; Holds the patient state.  It will be one of
                                ; ...  :UNKNOWN, NORMAL or FEVER. It is the
                                ; ...  abstraction of TEMP_EVAL.
(defmodvar treatment :interval) ; Holds the therapy recommendation.  In this
                                ; ...  system that will be either NIL or
                                ; ...  2-ASPIRIN-Q4H (2 aspirin every 4 hours).
                                ; ...  This is the result of a rule operating
                                ; ...  on PAT_STATE.
(defmodvar do_action :point)    ; Holds the actual action to be carried out
                                ; ...  such as GIVE-TWO-ASPIRIN.  This is
                                ; ...  generated from TREATMENT.
```

```
;; MODULES
;;
;; This is the heart of the system.  The modules define how the data is
;; transformed from its initial state into the final output form.
;;

(deftransducer temp_eval body_temp temp_eval
   ;; This defines a transducer that will change a numeric input value into
   ;; a categorical point value of FEVER or NORMAL.  The DEFTRANSDUCER form
   ;; applies the lambda function to all elements of TEMP_EVAL.

              (lambda (x) (cond ((> x 99.5) 'fever)
                                (t 'normal))))


;; The abstractor takes the input data points and turns them into states.
;; The approach taken in the following modules is to have intervals start at
;; the same time as the data becomes available which indicates that the new
;; state is reached.  In other words, fever is defined to start at the time
;; that a feverish temperature is recorded.  An alternate approach
;; would be to try to do an interpolation of some sort.
;; A given patient state will persist for 60 minutes beyond the last data
;; point indicating the existence of that state (unless an explicit
;; data-driven change is found sooner).  This is a sufficiently common form
;; of time-dependent reasoning that support from the control structure is
;; provided via a macro.
;;

(defpersistence fever_detect temp_eval pat_state :persistence 60)


(defmodule treat_fever (pat_state) (treatment) nil
 ;; A very simple module.  The rule specifies the therapy to be given
 ;; whenever fever is present.  It could have also been specified with a
 ;; DEFRULE form, in which case the TREATMENT would be the value returned by
 ;; a function which took PAT_STATE as its only argument.

 (if (eq pat_state 'fever)
     (setq treatment '2-aspirin-q4hr)
     (setq treatment nil)))
```

```
(defmodule give_aspirin (treatment) (do_action)
          ((:history last_action :initial nil)) ; Hold last time aspirin given.

           ;; This is a generator.  It takes a state describing a periodic action
           ;; and transforms it into a sequence of points, each one of which
is
           ;; one instance of the action to be taken.  This module, for example,
           ;; takes a prescription and produces action points which direct when
to
           ;; give medication.

 (setq do_action nil)
 (when (eq treatment '2-aspirin-q4hr)
     ;; First set up the next treatment time.  If no past action, then start
     ;; treatment immediately (at BEGIN_TIME).  Otherwise, start at 4 hours
     ;; (240 minutes) after the last treatment (or at BEGIN_TIME, whichever
     ;; is later).  Finally, loop forward through time and generate new
     ;; orders.
     ;;
     ;; This routine does not make use of the system variables NOW or PAST?,
     ;; so it could conceivably produce requests to give treatment in the
     ;; past.  Note further that this routine requires the extent of
     ;; treatment to be bounded, because there is no internal restriction on
     ;; how far into the future the point variables will be created.  If the
     ;; treatment were open-ended, then the generation would go on forever.
     ;; It would, of course, be possible to limit the generation time by
     ;; explicitly checking against the current time by including the built-in
     ;; module variable NOW in the argument list.  It would probably also be
     ;; reasonable to remember the value of NOW in a history variable, at
     ;; least long enough to guarantee that the generation goes sufficiently
far
     ;; into the future.

   (let ((next-time (if (null last_action)
                        begin_time
                        (tcs:time-max (tcs:time-add last_action 240) begin_time))))
     (loop while (tcs:time< next-time end_time)
           do (setq last_action next-time)
              (push (tcs:make-point 'give-2-aspirin next-time) do_action)
              (setq next-time (tcs:time-add next-time 240))))))
```

9.2 Fever Example 1 Execution

This section will give a short trace of the interactions involved in loading and executing the system just defined above. In the following transcript, the user's input will follow the arrow ⇒ prompt and appear in italics. System values returned will be in normal font. The system will be loaded, instantiated, variables set up for communication and then simple data will be entered. At the end, a graph will be created and the results of system execution shown.

⇒ *(in-package "TCS-USER")*
```
#<Package TCS-USER 152203066>
```

⇒ *(load "tcs:tcs;example;example")*
```
Loading TCS:TCS;EXAMPLE;EXAMPLE.BIN.NEWEST into package TCS-USER

#P"Z:>arry>r7>tcs>example>example.bin.16"
```

⇒ *(setq sys (instantiate-system 'example))*
```
#<SYSTEM EXAMPLE-121>
```

⇒ *(setq body-temp (find-variable sys 'body_temp)*
temp-eval (find-variable sys 'temp_eval)
pat-state (find-variable sys 'pat_state)
treatment (find-variable sys 'treatment)
do-action (find-variable sys 'do_action))
```
#<PtVar DO_ACTION>
```

⇒ *(display sys)*
```
SYSTEM EXAMPLE-2:
 Modules: GIVE_ASPIRIN   TREAT_FEVER  FEVER_DETECT     TEMP_EVAL
 Pt Vars:    DO_ACTION     TEMP_EVAL     BODY_TEMP           NOW
 Int Var:    TREATMENT     PAT_STATE       FUTURE?         PAST?
NIL
```

⇒ *(update-value body-temp 98.8 0)*
```
  Queue [V] PROCESS   1 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
98.8
```

⇒ *(update-value body-temp 101.2 30)*
```
  Queue [V] PROCESS   2 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
101.2
```

⇒ *(update-value body-temp 105.1 60)*
```
  Queue [V] PROCESS   3 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
105.1
```

```
⇒  (update-value body-temp 102 120)
  Queue [V] PROCESS   4 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
102

⇒  (show-queue sys)
QUEUE:
     PROCESS   4 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
NIL

⇒  (run-queue sys)
Running PROCESS   4 [PENDING] for TEMP_EVAL from -INFINITY to INFINITY
  Queue [V] PROCESS   5 [PENDING] for FEVER_DETECT from -INFINITY to INFINITY
Running PROCESS   5 [PENDING] for FEVER_DETECT from -INFINITY to INFINITY
    Ran process from -INFINITY to 0
  Jump  [E] PROCESS   6 [PENDING] for FEVER_DETECT from 0 to INFINITY
Running PROCESS   6 [PENDING] for FEVER_DETECT from 0 to INFINITY
    Ran process from 0 to 30
  Jump  [E] PROCESS   7 [PENDING] for FEVER_DETECT from 30 to INFINITY
  Queue [V] PROCESS   8 [PENDING] for TREAT_FEVER from 0 to 30
Running PROCESS   7 [PENDING] for FEVER_DETECT from 30 to INFINITY
    Ran process from 30 to 180
  Jump  [E] PROCESS   9 [PENDING] for FEVER_DETECT from 180 to INFINITY
  Queue [V] PROCESS  10 [PENDING] for TREAT_FEVER from 30 to 180
Running PROCESS   9 [PENDING] for FEVER_DETECT from 180 to INFINITY
Running PROCESS   8 [PENDING] for TREAT_FEVER from 0 to 30
  Queue [V] PROCESS  11 [PENDING] for GIVE_ASPIRIN from 0 to 30
  Jump  [P] PROCESS  12 [PENDING] for TREAT_FEVER from -INFINITY to 0
Running PROCESS  12 [PENDING] for TREAT_FEVER from -INFINITY to 0
  Queue [V] PROCESS  13 [PENDING] for GIVE_ASPIRIN from -INFINITY to 0
Running PROCESS  10 [PENDING] for TREAT_FEVER from 30 to 180
  Queue [V] PROCESS  14 [PENDING] for GIVE_ASPIRIN from 30 to 180
  Jump  [N] PROCESS  15 [PENDING] for TREAT_FEVER from 180 to INFINITY
Running PROCESS  15 [PENDING] for TREAT_FEVER from 180 to INFINITY
  Queue [V] PROCESS  16 [PENDING] for GIVE_ASPIRIN from 180 to INFINITY
Running PROCESS  11 [PENDING] for GIVE_ASPIRIN from 0 to 30
Running PROCESS  13 [PENDING] for GIVE_ASPIRIN from -INFINITY to 0
  Jump  [N] PROCESS  17 [PENDING] for GIVE_ASPIRIN from 0 to 30
Running PROCESS  17 [PENDING] for GIVE_ASPIRIN from 0 to 30
Running PROCESS  14 [PENDING] for GIVE_ASPIRIN from 30 to 180
Running PROCESS  16 [PENDING] for GIVE_ASPIRIN from 180 to INFINITY
14 Processes Ran, 17 Queued.  Run ratio = 82%
0
14
17
```

⇒ *(display body-temp)*
```
VARIABLE BODY_TEMP [Point]:
  Input of: TEMP_EVAL
 Output of:
      0 =  98.8      30 = 101.2      60 = 105.1     120 =   102
NIL
```

⇒ *(display treatment)*
```
VARIABLE TREATMENT [Interval]:
  Input of: GIVE_ASPIRIN
 Output of: TREAT_FEVER
    -INFINITY to       30 =             NIL
           30 to      180 = 2-ASPIRIN-Q4HR
         180 to INFINITY =             NIL
NIL
```

⇒ *(display do-action)*
```
VARIABLE DO_ACTION [Point]:
  Input of:
 Output of: GIVE_ASPIRIN
    30 = GIVE-2-ASPIRIN
NIL
```

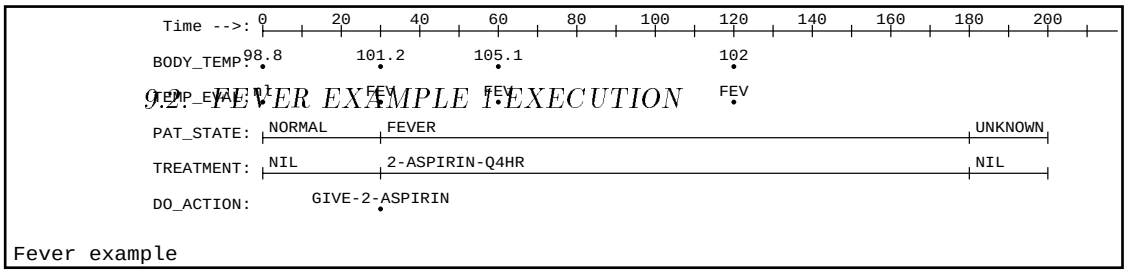
⇒ *(setq graph (tcs:create-graph 5 :width 912))*
```
#<TCS:GRAPH 34012116>
```

⇒ *(setf (tcs:start graph) 0*
(tcs:end graph) 200
(tcs:title graph) "Fever example")
```
"Fever example"
```

⇒ *(setf (tcs:var-list graph)*
(list body-temp temp-eval pat-state treatment do-action))
```
(#<PtVar BODY_TEMP> #<PtVar TEMP_EVAL> #<IntVar PAT_STATE>
 #<IntVar TREATMENT> #<PtVar DO_ACTION>)
```

⇒ *(tcs:draw-graph graph)*
```
NIL
```

⇒ *(tcs:expose graph)*



Fever example

T

9.3 Fever Example 2 Definition

```
;;
;; This is an alternate system which uses many of the same modules and
;; variables as the first example.  The only difference is the addition
;; of a new interval variable SPECIES and a different type of
;; transducer, this one context sensitive.

(defsystem example2
  (body_temp temp_eval pat_state treatment do_action species)
  (temp_eval_s fever_detect treat_fever give_aspirin))

(defmodvar species :interval)   ; Holds the species of the "patient." Currently
                                ; ... only HUMAN and DOG are recognized.

(defmodule temp_eval_s (body_temp species) (temp_eval) nil
 ;; This implements a context-sensitive transducer.  By having the context
 ;; dependencies taken care of here, the abstractor can be made simpler.  It
 ;; also provides a justification for using transducers in the first place,
 ;; since the direct transformation of individual data points is fairly
 ;; simple.  This modules lets the same therapy management code handle
 ;; multiple species.  (Exactly how to give a dog two aspirin every four
 ;; hours is an exercise left to the reader)
 ;;
 ;; This could also have been defined using the :context option to the
 ;; DEFTRANSDUCER form and substituting a function that takes two arguments.
 ;;

 (setq temp_eval
       (tcs:map-point-values
         #'(lambda (temp) (if (> temp (case species
                                        (human 99.5)
                                        (dog 101.5)
                                        (t 100.0)))
                              'fever 'normal))
         body_temp)))
```

Index