Common Lisp Instrumentation Package: User Manual

David L. Westbrook, Scott D. Anderson, David M. Hart and Paul R. Cohen

Computer Science Technical Report 94-26

Experimental Knowledge Systems Laboratory
Department of Computer Science, Box 34610
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003-4610

This work was supported by a University Research Initiative grant, ONR N00014-86-K-0764, and by ARPA/Rome Laboratory under contracts F30602-C-91-0076 and F30602-C-93-0100.

Common Lisp Instrumentation Package: User Manual

Experimental Knowledge Systems Laboratory
Computer Science Department, LGRC
University of Massachusetts
Box 34610
Amherst, Massachusetts
01003-4610

David L. Westbrook
Scott D. Anderson
David M. Hart
Paul R. Cohen

CLIP Version 1.4
January 31, 1994

 This work was supported by a University Research Initiative grant, ONR N00014-86-K-0764, and by ARPA/Rome Laboratory under contracts F30602-91-C-0076 and F30602-93-C-0100.

Abstract

This document is the user manual for the Common Lisp Instrumentation Package (CLIP), a tool for automating experimentation and data collection. CLIP was designed to be used in combination with the Common Lisp Analytical Statistics Package (CLASP), though it can be used as a standalone instrumentation package for Common Lisp applications.

CLIP is available via anonymous ftp from *ftp.cs.umass.edu* in the directory `pub/eksl/clip`. CLIP requires Common Lisp and the Common Lisp Object System, CLOS. CLIP runs on a number of platforms under a variety of lisp implementations. Check the CLIP release notes in `pub/eksl/clip` for a detailed list of currently supported platforms.

If you have any problems, questions, or suggestions, contact us at:

`clasp-support@cs.umass.edu`

Contents

List of Figures

Chapter 1

CLIP

1.1 Motivation

We collect information from software systems for many reasons. Sometimes, the very purpose of the system is to produce information. Other times, we collect data for debugging, feedback to the user—in general, for understanding the system's behavior. Unfortunately, there seem to be few general tools available for this task. Instead, we often find ad hoc, domain-specific code, spread throughout a system, varying from component to component, despite strong similarity in the requirements for any data collection code.

CLIP, the Common Lisp Instrumentation Package, standardizes data collection. It is named by analogy with the "alligator clips" that connect diagnostic equipment to electrical components. By standardizing data collection, we can more easily add and delete instrumentation from our systems. We can reuse a piece of data collection code, called a "clip," in other systems we are studying. We can use general tools to analyze the collected data. Instrumentation in a system becomes more easily used and understood, because the basic functionality of the system is separated from the functionality of the instrumentation.

We designed CLIP to be used for scientific experiments, in which a system is run under a number of different conditions to see whether and how the behavior changes as a function of the condition. Consequently, we view an experiment as a series of trials, varying the settings of experiment variables and collecting information. Generally speaking, an experiment comprises the following steps:

1. Creating clips and inserting them into the system to be studied. Generally, clip measures one particular aspect of a system, but you can also define clips that combine many measurements, including other clips. Often, the clips you need will already be defined.

2. Define the experiment, in terms of what system is to be run, any necessary initialization code, and the conditions under which it is to run (different input parameters or environmental conditions).

3. Run a series of trials, saving the output into a CLASPformat data file.* This format is described in the CLASP manual section Data Manipulation Functions, although it isn't necessary to understand the file format. CLASP can read what CLIP writes.

*CLIP can also write data files in standard tab or space delimited format used by most other statistical packages, databases and spreadsheets.

4. Analyze the data using CLASP.

1.2 Implementation

CLIP provides five major forms to support experiments:

- `define-simulator`,
- `defclip`,
- `define-experiment`,
- `write-current-experiment-data`, and
- `run-experiment`

Each is described in detail below.

1.2.1 Define-simulator Macro

`define-simulator` *(name &key system-name system-version reset-system start-system stop-system schedule-function deactivate-scheduled-function seconds-per-time-unit timestamp)* [*Macro*]

Define the interface to a simulation. The following options are recognized:

`:system-name`	string naming system	[2]
`:system-version`	function or form which handles the arguments of the experiment and returns a string which denotes the version of the system [a]	[1]
`:start-system`	function or form that handles the experiment variables and arguments of the experiment; this function is called during the experiment loop to begin execution of the system; when it returns the trial is considered to be completed	[2]
`:reset-system`	same as :start-system; this function will be called during the experiment loop before the system is started;	[1]
`:stop-system`	same arg handling as :start-system; this function can be used to execute code when a trial is shutdown; it is most useful when instrumenting multiprocessing systems where other processes need to be terminated	[1]
`:schedule-function`	function or form that handles the lambda list *(function time period name)* and optionally returns a data-structure which represents the event; used to provide access to the simulator's event-scheduling mechanism	[3]
`:deactivate-scheduled-function`	function or form that handles one arg, namely the data structure returned by the `schedule-function` function; used to provide access to the simulator's event-scheduling mechanism	[1]
`:timestamp`	a function or a list of (<function-name> <clip-name>) where function should return the current time in units specified by `:seconds-per-time-unit`	[3]
`:seconds-per-time-unit`	the number of seconds in 1 simulator time quantum (default is 1)	[3]

Keywords marked [1] are optional; Keywords marked [2] are required, and keywords marked [3] are required when using time-series clips

[a] Functions must accept the formal arguments specified in `define-experiment` and the actual arguments specified in a call to `run-experiment`. Forms can refer to the arguments lexically.

The `define-simulator` form is used to eliminate the need to specify redundant information when multiple experiments are to be defined for the same system. All of the options specified in this form can be overridden in a `define-experiment` form.

1.2.2 Define-experiment Macro

`define-experiment` *(name arguments &body body &aux documentation)* [*Macro*]

A `define-experiment` form sets up the environment in which the system is to be run. Options for `define-experiment` support the sequential nature of experimentation. That

is, for the purposes of alligator clips, we view an experiment as a series of trials, varying the settings of experiment variables and collecting information. The `define-experiment` macro allows code to be run at the beginning and end of the experiment, usually to set up data-recording or to do data analysis. It also allows code to be run at the beginning and end of each trial; often this is used to reset counters, clear data structures, or otherwise set up the data collection apparatus. Through the `:script` keyword a user may add specific code to run at particular times during the experiment. A user can also specify experiment variables and their associated sets of values; the experiment code will ensure that all combinations of the experiment variables are run. For example, specifying:

```
:variables ((rating-scheme '(:OPPORTUNISTIC :ORDER-BY-ORDER))
      (percentage from 100 downto 50 by 10))
```

describes an experiment with twelve conditions (six values of percentage times two kinds of rating schemes), and so the number of trials will be a multiple of twelve, with an equal number of trials being executed under each condition. The options accepted by `define-experiment` are:

`:simulator`	specify the name of the simulator definition to use
`:before-trial`	called with the current values of the experiment variables and arguments to specify operations performed before each trial (such as initializations)
`:after-trial`	similar to `:before-trial`, specifies operations to be performed *after* each trial. Summary data files for each trial can be written at this time using `write-current-experiment-data`.
`:before-experiment`	called with the arguments. The value should be a form that refers to the arguments or a function that handles the correct number of arguments. Used to initialize system and define experiment environment.
`:after-experiment`	Similar to `:before-experiment`, called with just the arguments. Used to close files, write any experiment summarization output and clean up after the experiment.
`:instrumentation`	is a list of names defined with `defclip` which will be enabled during the experiment. Use `write-current-experiment-data` in the after-trial code to write the data to the output-file.
`:variables`	({(var exp) | (var {loop-for-clause})}) - define experiment variables
`:locals`	({(var exp) | var}*) - bind variables in the context of the experiment
`:script`	({(descriptive-name initial-time [next-time] form) — script-element-name}*) *initial-time* can be a string, a number or form to evaluate. *next-time* is a time interval and should be a fixnum or form.

In addition, `define-experiment` accepts all the options accepted by `define-simulator`.

The following steps are generated from the `define-experiment` specification to run a typical experiment. After initializing the system using the `:before-experiment` form, it loops over the cross-product of the specified experiment variables, running one trial for each element in the cross-product. This is repeated for as many times as is specified in the `:repetitions` option. After the last trial, the `:after-experiment` form is run.

```
Run :BEFORE-EXPERIMENT code with args
LOOP
  Update experiment-variables for trial
  Run :BEFORE-TRIAL code with experiment-variables and args
  Run :RESET-SYSTEM code with experiment-variables and args
  Instantiate Script Events
  Reset and Enable all the instrumentation
  Run :START-SYSTEM code with experiment-variables and args
  <system runs (possible periodic and event based collection done)>
  Run :AFTER-TRIAL code (possible post-hoc collection done)
END LOOP when all trials completed
Run :AFTER-EXPERIMENT code with args
```

1.2.3 Write-current-experiment-data Function

`write-current-experiment-data` *(&key separator format instrumentation stream)* [*Function*]

Causes each experiment instrumentation to write its data to *stream* or the output-files specified in the `run-experiment` call or in a `defclip`. *separator* should be a character which will be used to separate fields. It defaults to the value of `*data-separator-character*`. *format* should be one of `:CLASP` which means write a clasp native format data file, `:ASCII` which means write a standard separator delimited data file including column names or `:DATA-ONLY` which is the same as `:ASCII` except no column names are included. *format* defaults to the value of `*output-format*`. *instrumentation* can be used to specify a subset of the experiment's instrumentation to write to the data file.

1.2.4 Run-experiment Function

`run-experiment` *(experiment-name &key args repetitions number-of-trials length-of-trial output-file error-file extra-header starting-trial-number)* [*Function*]

Run-experiment starts execution of the experiment named *experiment-name*. The *args* are passed on to the experiment. *output-file* is optional, but must be specified if `write-current-experiment-data` is called from within your experiment. *number-of-trials* can be used to specify an exact number of trials to run. If *number-of-trials* is not specified it will be calculated so as to vary all the experiment variables across all their values *repetitions* (default 1) times. *starting-trial-number* can be used to change this value to something other than one (1) which is useful for continuing partially completed experiments. *length-of-trial* can be specified for time-based simulators to put a limit on the maximum trial length. *error-file* can also be used to direct the error/debug output to a file. *extra-header* is written at the end of the header of the output file.

1.2.5 Explicitly Stopping Trials

The following functions can be called from user code to explicitly terminate a CLIP trial or experiment. Usually this is done from within the code of a script or as part of an error handler. They all take no arguments and assume that an experiment is in progress.

`shutdown-and-rerun-trial` *()* [*Function*]

This will cause the current trial to be aborted (no data written) and restarted. This is a function that users should call when they have detected an error condition of some sort that renders the trial worthless, but rerunning the trial may work.

`shutdown-and-run-next-trial` *()* [*Function*]

This will cause the current trial to be stopped (data will be written) and the next trial started. This is a function that users should call when want to normally shutdown a trial and collect and report the data from that trial.

`shutdown-experiment` *()* [*Function*]

This will cause the current trial to be aborted (no data will be written) and will return the system to the state it was before the experiment began (by running the after-experiment code).

1.3 Clip Definition

There are basically only a small number of ways to instrument a software system. These are: adding special purpose code to collect data, interrogating global (usually objects with state) data structures or using advice.

The first way is to build in special purpose code to collect data about the state of the system while the system is running. We used this technique in the Phoenix¡ testbed to keep information about the execution time of timeline entries and also message traffic patterns. The Transsim simulator also uses this technique when it keeps track of daily costs, ship locations, demon firing intervals. The key point here is that the only reason for adding the piece of code was to allow an experimenter to analyze the behavior of the system. The simulation itself does not use the information. This method increases the complexity and reduces the readability and maintainability of the software system. In code that is highly instrumented it is often difficult to determine what is intrinsic to the running of the system and what is used to instrument.

Another method involves interrogating global data structures to determine the state *post hoc*. Examples include most everything we collected about Phoenix (ie., fireline built, first plan tried, bulldozer digging times, etc.). This technique is fine if the information is hanging around after you have finished running the simulation, but this is not always the case. Also, there is some information that is time-sensitive, and therefore must be collected on a periodic or event-driven basis. Collecting this type of information often involves resorting to the first method – altering the code itself.

Alternatively, one can use the `advise` facility available in many lisp environments to non-intrusively collect information when functions are called. This is a nice modular approach, but requires a good deal of knowledge about the advise facility and the software system itself. Unfortunately, the `advise` facility is not standardized across Lisp implementations.

The `defclip` macro encapsulates the code necessary to instrument a software system and separates it from the system itself. It avoids the pitfalls of adding special purpose code

directly to the software system while at the same time allowing periodic and event-driven data collection. It also allows collection to be done by perusing global data structures.

1.3.1 Simple Clips

Simple clips have no components and collect data immediately prior to reporting it to the output file at `:after-trial` time. If they are defined with a `:schedule` or `:trigger-event` `defclip` option their default behavior is store all of the data collected during a trial and report a single value which is the mean of all the values collected.[†]

1.3.2 Clips with Components

Clips with components, as specified by the `:components` keyword, generate multiple columns in a data file each time they are reported. Depending on other options they may produce one column per component (*composite clips*) or multiple columns per component (*mapping clips*). Mapping clips are specified using the `:map-function` option to `defclip`. Clips with components are sometimes referred to as *super clips*. For a good example of clips with components and further discussion of their use, see Appendix A.1.1.

1.3.3 Time-series Clips

Clips that have `:components` and either the `:schedule` or `:trigger-event` option are *time-series clips*. They generate multiple data columns in the manner of component clips (which they are) and also multiple data rows. Each row corresponds to a single collection and is either triggered by a particular event or activated periodically. Since time-series clips generate multiple rows, they are generally written to a data file that is separate from the main experiment (summary) data file. The name of the data file associated with a time-series clip is specified using the `:output-file` option to `defclip`.

The `:schedule-function`, `:seconds-per-time-unit`, and `:timestamp` keywords to `define-simulator` must be specified for periodic time-series clips. Event-based time-series clips require only that the Common Lisp implementation provide some mechanism similar to the `advise` function.[‡]

1.3.4 Defclip Macro

`defclip` *(name args (options) &rest body)* [*Macro*]

A `defclip` form defines and names a particular data-collection instrument. Each clip collects a single variable which, together with other variables, make up a dataset. The dataset can then be analyzed by CLASP. The form may contain arbitrary code to gather data for the clip: access to global variables, function calls, and so forth. Naming the clips makes it easy to turn them on and off programatically or with menu selections.

options is a list of keywords and values which modify the main form implemented by the clip. *options* can be any of the following, all of which are optional:

[†]The default behavior if the collected values are non-numeric is to generate an error.

[‡]Actually, one can explicitly call the undocumented (until now), unexported 'clip::collect' function on a particular instrumentation and achieve the same effect as an event-based clip, but requires modifying the code.

`:schedule`	a list of keyword/value pairs; currently allowed are `:period` which provides the time interval between collections for time-series data, and `:start-time` which specifies the time of the first collection; if neither `:schedule` nor `:trigger-event` is specified, collection will be done immediately prior to the report being output to the data stream
`:trigger-event`	a function, list of functions or list of trigger specifications which trigger event-driven collection ; each trigger spec should of the form (<fname> [:BEFORE | :AFTER] [:PREDICATE <fname>])
`:components`	a list of clips that are associated with this clip; ie., they are collected and reported as a group by collecting or reporting this clip
`:map-function`	provides a list of items to map the `:components` over ; this function should return the same arguments in the same order each time it is called
`:report-key`	allows overriding of the column header key which is written to the data stream; avoid using this for component clips unless you really know what you are doing as the default key will be generated to correctly differentiate between multiple invocations of the same clip with different parents; this string is used in a call to 'format'; for most component clips this string should handle the same arguments as the clip
`:initial-status`	whether the clip is enabled or disabled by default
`:report-function`	can be used to override the default report function; for expert users only
`:enable-function`	code[a] to set up data structures for the clip; runs once when the clip is turned on
`:disable-function`	code to remove data structures set up for the clip; runs once when the clip is turned off
`:reset-function`	code to reinitialize data structures at the beginning of each trial; for example, setting counters back to zero
`:display-function`	code for graphical display of the information
`:output-file`	used to specify the output file for time-series clips - merged with the pathname specified in the `run-experiment` call; should be a string or a function of one arg; the function is called with the name of the clip and should return a value suitable for use as the first argument to `merge-pathnames`

[a] 'code' at this level is specific to the user's system, and is assumed to be user-supplied Lisp code. For example, enabling a counter of events in a system may require creation and initialization of a counter variable.

Some examples of `defclip` usage:

A simple clip with code used to report a value:

```
(defclip number-of-dead-bulldozers ()
  (length (bds-that-died)))
```

```
An example showing a clip with components:
  (defclip methods-each-bd (bulldozer)
   "Salient information for each instance of applying a recovery method:"
   (:components (trial-number
                 agent-name
                 method-type
                 failure-type
                 calculate-recovery-cost
                 method-succeeded-p
                 order-of-application)
    :map-function (gather-recovery-method-instances (name-of bulldozer)))
   ;; This code executes before the map-function is executed.
   (send (fire-system) :set-frame-system (name-of bulldozer)))

   (defclip ii-projection-attempts-made ()
     "Clip for reporting the number of attempts that were made by
ii-projection during the last projection of the trial."
     (:reset-function (setf *ii-projection-attempts-made* nil))
     (car *ii-projection-attempts-made*))
```

1.4 Examples from a Simple Agent Simulator

The following three examples illustrate experiment control and data collection using CLIP. The agents in this simulator stochastically increment their state until they reach a final state. The first example defines the experimental interface to an agent simulator and two simple clips for collecting data. The second and third examples build on the first, defining more complicated clips for mapping a simple clip onto multiple agents and for collecting time-series data.

1.4.1 Using Simple Clips to Collect Trial Summary Data

This example collects trial summary data about overall agent-cost and task completion-time. Collection occurs at the end of each trial and is written out to a summary file in CLASP format. The `define-simulator` form designates methods for starting the simulation at the beginning of the experiment, resetting and reinitializing it if necessary, and stopping it after the final trial. The `define-experiment` form designates the simulator to use, in this the case the one we have defined. It also specifies: a set of experiment variables and the experimental settings for each, a list of the clips we've defined under `:instrumentation`, initializations for global variables before each trial, and a function for writing out a row of data at the end of each trial.

For each trial, the trial number (assigned sequentially) and the value of each experiment variable are written out to the summary file along with the values of the specified clips. An example of the CLASP output file produced by this experiment is shown in Figure 1.1.

In this first example we have defined two simple clips. One collects the combined cost of all agents at the end of each trial and the other records the time at which the trial ends.

```
;;;-------------------------------------------------
```

```
(define-simulator agent-sim-1
  :start-system (run-agent-simulation :reset nil)
  :reset-system (reset-agent-simulation)
  :stop-system stop-simulation)

;;;-------------------------------------------------------
;;; Clip Definitions

(defclip agents-cost ()
  ()
  (reduce #'+ (find-agents) :key #'cost))

(defclip completion-time ()
  ()
  (current-time))

;;; ******************************************************
;;; The Experiment Definition

(define-experiment simple-agent-experiment-1 ()
  "A test experiment."
  :simulator agent-sim-1
  :variables ((transition-probability in '(.01 .1))
              (cost-factor from 1 to 3))
  :instrumentation (agents-cost completion-time)
  :before-trial (setf *transition-probability* transition-probability
                      *relative-cost* cost-factor)
  :after-trial (write-current-experiment-data))

#| Execute this to run the demo experiment.

(run-experiment 'simple-agent-experiment-1
                :output-file
                #+Explorer "ed-buffer:data.clasp")
|#
```

1.4.2 Using a Mapping Clip to Map Simple Clips Over Multiple Agents

This example defines a mapping clip that maps over all the agents at the end of each trial and returns the cost accrued by each. It produces a summary output file like that in Figure 1.2 recording after each trial an entry that includes trial number, each experiment variable, each of three agent's cost, and the completion time of the trial.

```
;;;-------------------------------------------------------
;;; Mapping clip to collect cost of each agent

(defclip all-agents-costs ()
```

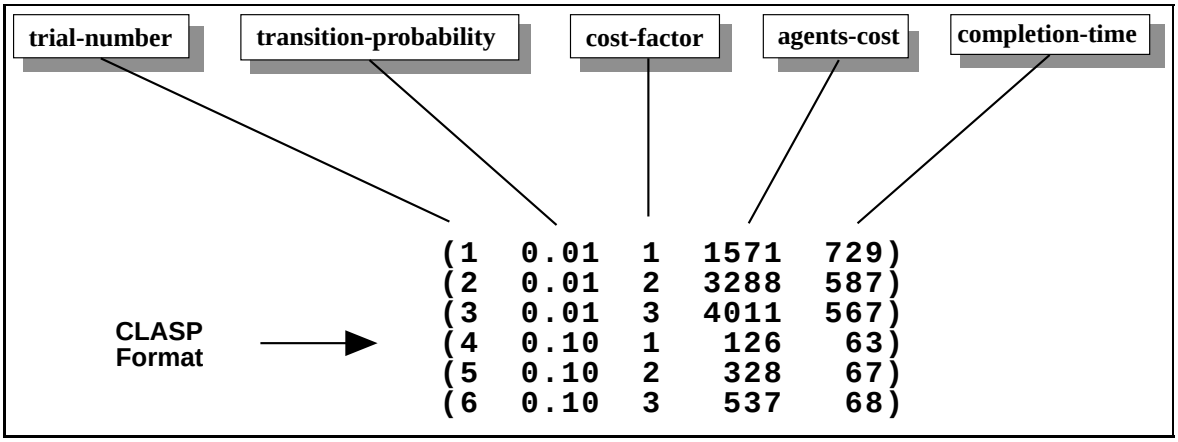

Figure 1.1: Simple Clip Example from the Agent Experiment.

```
  (:map-function (find-agents)
   :components   (each-agent-cost)))

(defclip each-agent-cost (agent)
  ()
  (cost agent))

;;; ***************************************************
;;; The Experiment Definition

(define-experiment simple-agent-experiment-2 ()
  "A test experiment."
  :simulator agent-sim-1
  :variables ((transition-probability in '(.01 .1))
              (cost-factor from 1 to 3))
  :instrumentation (agents-cost all-agents-costs completion-time)
  :before-trial (setf *transition-probability* transition-probability
                      *relative-cost* cost-factor)
  :after-trial (write-current-experiment-data))

#| Execute this to run the experiment.

(run-experiment 'simple-agent-experiment-2
                :output-file
                #+Explorer "ed-buffer:data.clasp")
|#
```

1.4.3 Full Agent Simulator Experiment with Time-Series Clips

This example uses both periodic and event-driven time-series clips to collect data about each agent's state. To use time-series clips the experimentor must supply CLIP with enough information to schedule data collection *during* trials. This is done by specify-

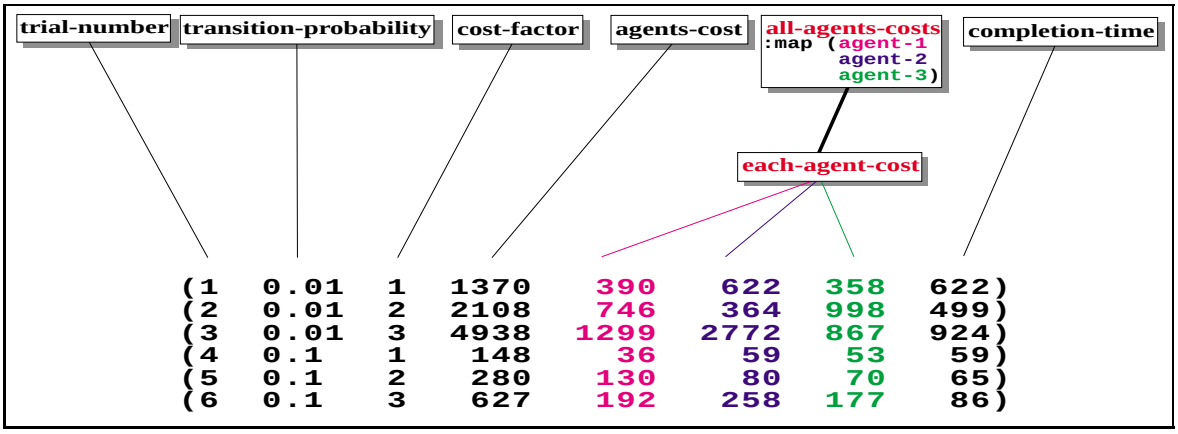

Figure 1.2: Mapping Clip Example from the Agent Experiment. Mapping clips are used to map one or more simple clips over multiple objects.

ing in `define-simulator` a `:schedule-function` that tells `defclip` how to schedule clip execution. The optional `:deactivate-scheduled-function` provides a function for unscheduling clips (if necessary). `:seconds-per-time-unit` tells CLIP how to translate the time units of the simulator into seconds. The value of `:timestamp` is used to automatically record the time of collection in each row of a time-series data file.

This example illustrates three uses of clips: collecting summary data about the highest agent state, gathering periodic snapshots of all agent states, and recording particular aspects of the agents' changes of state. The periodic clip definition specifies an output file and a scheduling interval. Two event-driven clips are triggered in this example by the same function, `change-of-state-event-function`, which looks for any change of agent state. However, since their component clips are not identical, their output is routed to separate files (CLASP file format expects all rows to have the same number of columns and column names).

Four output files are produced by this experiment. As in the previous examples, a summary file records a row of data at the end of each trial. Each change in an agent's state triggers event-driven clips that record a row in a time-series output file, as in Figure 1.3. The event-driven clip definitions `change-of-state` and `change-of-state-pred` produce one output file each. The fourth output file also records time-series data, one row for each periodic snapshot of all agents' states (cf. Figure 1.4).

Collecting time-series data is more time-consuming than collecting summary data. In the first two examples, where only summary data was collected, trials were allowed to run to completion. In the third example periodic clips run every 12 minutes and event-driven clips run as frequently as agent's states change. We can limit the duration of trials by specifying a `:length-of-trial` value to `run-experiment`. This is useful for statistical tests on time-series data, such as cross-correlation, that expect the number of rows for each trial to be the same. Thus, in this example, specifying `:length-of-trial` to be 500 minutes ensures that the periodic data collection will produce the same number of rows for each trial.

```
;;;-----------------------------------------------------

(define-simulator agent-sim
  :start-system (run-agent-simulation :reset nil)
```

```
  :reset-system (reset-agent-simulation)
  :stop-system stop-simulation
  ;; a function that places functions to run on the queue of events.
  :schedule-function (lambda (function time period name &rest options)
                       (declare (ignore name options))
                       (schedule-event function nil time period))
  ;; a function that removes functions from the queue of events.
  :deactivate-scheduled-function unschedule-event
  :seconds-per-time-unit 60
  :timestamp current-time)

;;;-------------------------------------------------
;;; Clip Definitions

;; This post-hoc clip produces two values.
(defclip highest-agent-state ()
  (:components (highest-state highest-agent))

  (loop
    with agents = (find-agents)
    with highest-agent = (first agents)
    with highest-state = (state highest-agent)
    for agent in (rest agents)
    for agent-state = (state agent) do
    (when (state< highest-state agent-state)
      (setf highest-state agent-state
            highest-agent agent))
    finally (return (values highest-state highest-agent))))

;;;-------------------------------------------------
;;; Periodic collection

;; This clip invokes its component every 12 minutes.
(defclip periodic-agent-state-snapshot ()
  (:output-file "snapshot.clasp"
   :schedule (:period "12 minutes")
   :map-function (clip::find-instances 'agent)
   :components   (each-agent-state-snapshot)))

;; Simple clip that returns the state of the agent.
(defclip each-agent-state-snapshot (agent)
  "Record the state at an agent."
  ()
  (state agent))

;;;-------------------------------------------------
;;; Event-driven collection
```

```
;; This clip accepts the arguments passed to `change-of-state-event-function'
;; and simply passes them through.
(defclip change-of-state (agent-name new-state)
  (:output-file "state-change.clasp"
   :trigger-event (change-of-state-event-function :BEFORE)
   :components (new-state agent-name))
  (values new-state agent-name))

;; This clip accepts no arguments and returns two values computed by other
;; functions.
(defclip change-of-state-pred ()
  (:output-file "state-change-pred.clasp"
   :trigger-event (change-of-state-event-function :AFTER)
   :components (fred barney))
  (values (compute-fred) (compute-barney)))

;;; ***********************************************
;;; The Experiment Definition

(define-experiment agent-experiment ()
  "A test experiment."
  :simulator agent-sim
  :variables ((transition-probability in '(.01 .1))
              (cost-factor from 1 to 5 by 2))
  :instrumentation (agents-cost
                    all-agents-costs
                    completion-time
                    highest-agent-state
                    change-of-state-pred
                    change-of-state
                    periodic-agent-state-snapshot)
  :before-trial (setf *transition-probability* transition-probability
                      *relative-cost* cost-factor)
  :after-trial (write-current-experiment-data))

;; Execute this to run the demo experiment.

(defun rexp ()
  (run-experiment 'agent-experiment
                  :output-file "data.clasp"
                  :length-of-trial "500 minutes"))
```

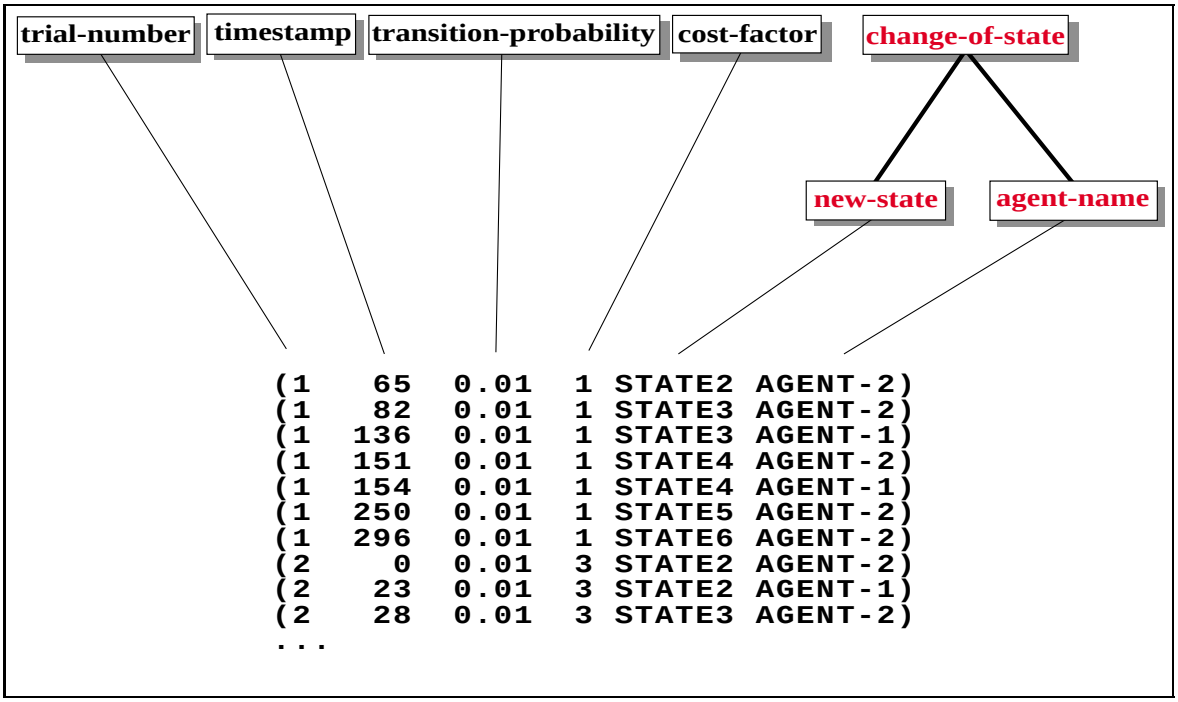

Figure 1.3: Event-Driven Time-Series Clips are used to collect data when an agent's state changes in the Full Agent Experiment.

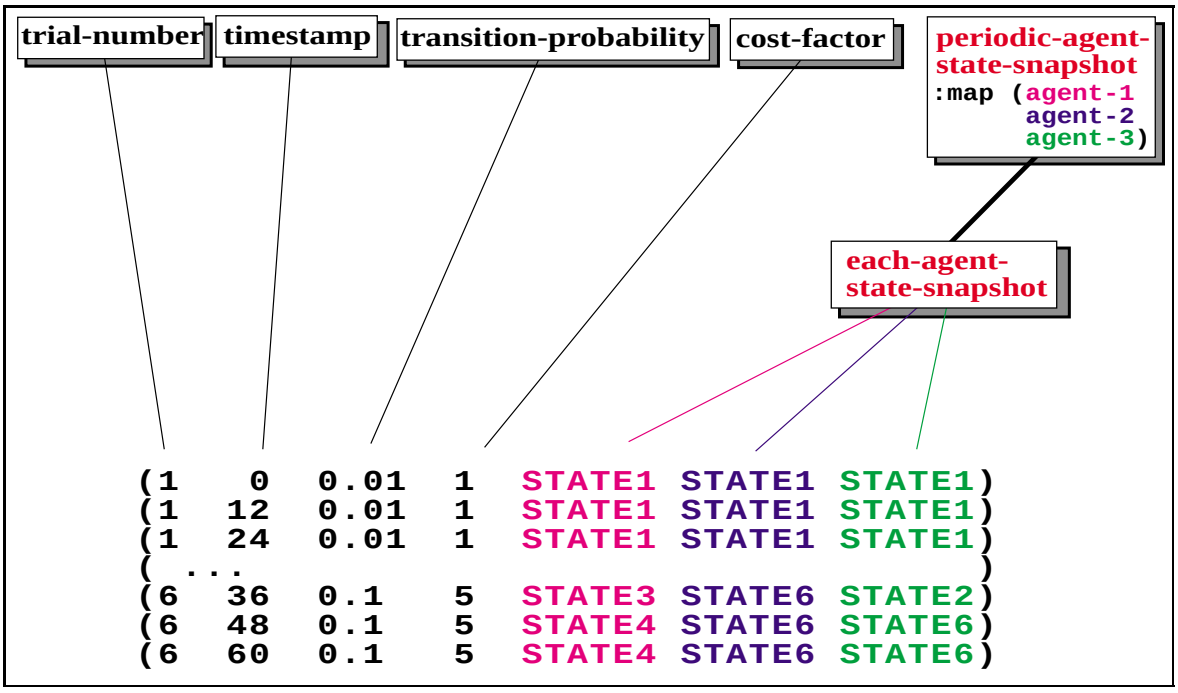

Figure 1.4: Periodic Time-Series Clips collect snapshots of each agent's state at regular intervals during the Full Agent Experiment.

Appendix A

Clip Examples

A.1 More Experiment Definitions Using Clip

The following three examples, taken from different domains, illustrate experiment control and data collection using CLIP. The third example includes output files (in CLASP format) for time-series clips.

A.1.1 Measuring an Airport Gate Maintenance Scheduler's Performance

This extended example of CLIP code was provided by Zachary Rubinstein. The system being studied is an AI scheduler developed by David Hildum for a domain called Airport Resource Management (ARM). The task is to schedule gate maintenance, providing and coordinating vehicles for servicing planes at terminal gates. The experiments shown here were part of a baseline study designed to measure the performance characteristics of several priority rating schemes under resource-constrained scenarios. The dependent variables are delay and fragmentation (how much shuttling between gates occurs). The independent variables are the competing heuristic rating schemes for determining the order in which resource requests are scheduled. For clarity, many of the implementation details have been omitted.

The example uses several super clips to organize the collection of similar kinds of data, such as all measures of delay. All data collection occurs *post hoc* and is written to a single summary file. Note that two experiments are defined, each of which uses the same clips. The function `delays-information`, called by the clip `delay`, provides an interesting example of the interface between clips and the system being instrumentated.

The `delay` and `fragmentation` clips are examples of *composite* clips. The values returned by the body of a composite clip are one-to-one mapped onto the components of the clip. Each component is itself a clip which accepts an argument and has a body that returns a value. This value is what is output to the data file. The number of values a composite clip produces is equal to the number of its components.

The components of a composite clip have a clip definition automatically generated for them. The default action for this automatically generated component clip is to return its single argument as a single value. For example, the `delay clip` below generates a clip for each of its components automatically. The clip designer can also override this default behavior by writing a clip for that component that does something with the argument before returning it.

```
(define-simulator arm :start-system arm:run-arm)

;;; -----------------------------------------------------------------
;;; CLIP Definitions

(defclip airport ()
  "Return the Currently Loaded Airport"
  ()
  (arm::get-loaded-airport))

(defclip timetable ()
  "Return the Currently Loaded timetable"
  ()
  (arm::get-loaded-timetable))


(defclip delay ()
  "Composite clip to calculate the various delay clips."
  (:components (total-delay number-of-delays average-delay
                std-dev-delay maximum-delay duration-ratio))

  (delays-information))


(defclip fragmentation ()
  "Composite fragmentation clip to calculate the various fragmentation clips."
  (:components (total-servicing-time
                total-sig-frag-time
                total-setup-time
                total-travel-time
                ratio-of-frag-to-servicing
                ratio-of-setup-to-servicing
                ratio-of-travel-to-servicing))

  (resource-fragmentation-information 'arm:baggage-truck-planner-unit))



;;; -----------------------------------------------------------------
;;; Experiments

(define-experiment limit-resources-numerically-over-rating-schemes
                        (&key (rating-schemes *defined-rating-schemes*)
                              (airport-delimiter 10)
                              (airports user::*10-FTS-AIRPORTS*)
                              (timetable-name :20-flights))
  "Using the various rating schemes, constrain the resources
and report the delays."
  :locals ((report-filename nil))
```

```
  :variables ((airport-name in airports)
              (rating-scheme in rating-schemes))
  :instrumentation (timetable
                    total-delay number-of-delays average-delay
                    std-dev-delay maximum-delay duration-ratio
                    total-servicing-time total-sig-frag-time
                    total-setup-time total-travel-time
                    ratio-of-frag-to-servicing
                    ratio-of-setup-to-servicing
                    ratio-of-travel-to-servicing)
  :before-experiment (progn (arm:zack-set-default-demo-parameters)
                            (arm:execute-load-timetable
                              :LOAD-SPECIFIC-FILE timetable-name))
  :before-trial (setf report-filename
                      (delay-initialize-trial rating-scheme
                         airport-delimiter airport-name))
  :after-trial (delay-report report-filename
                             airport-delimiter rating-scheme))


;;; ---------------------------------------------------------------------
;;; Example Invocation

#+COMMENT
(run-experiment 'dss:limit-resources-numerically-over-rating-schemes
                :args '(:rating-schemes
                         (:zack-1 :default :obo-minest-heuristic)
                         :airport-delimiter :20-ref
                         :airports (:detroit-17-bts :detroit-16-bts
                                    :detroit-15-bts :detroit-14-bts
                                    :detroit-13-bts :detroit-12-bts)
                         :timetable-name :20-flights)
                :output-file "hillary:reports;20-REF-FTS-DELAY-STATS.TEXT")


;;; ---------------------------------------------------------------------
;;; Experiments

(define-experiment run-over-rating-schemes
                       (&key (rating-schemes *defined-rating-schemes*)
                             (airport-delimiter 10))
  "Using the various rating schemes, constrain the resources, introduce
orders, and report the delays."
  :locals ((report-stream nil))
  :variables ((rating-scheme in rating-schemes))
  :instrumentation (airport timetable
                    total-delay number-of-delays
                    average-delay std-dev-delay
```

```
                    maximum-delay duration-ratio
                    total-servicing-time total-sig-frag-time
                    total-setup-time total-travel-time
                    ratio-of-frag-to-servicing
                    ratio-of-setup-to-servicing
                    ratio-of-travel-to-servicing)
  :before-trial (setf report-stream
                      (delay-initialize-trial rating-scheme
                        airport-delimiter))
  :after-trial (write-current-experiment-data ))


;;; ---------------------------------------------------------------------
;;; Example Invocation

#+COMMENT
(run-experiment 'dss:run-over-rating-schemes
                      :ARGS `(:RATING-SCHEMES
                               ,(cons :DEFAULT
                                      (cons :ZACK-1
                                            (remove :OPPORTUNISTIC
                                                    *defined-rating-schemes*
                                                    :TEST #'eq)))
                               :airport-delimiter 10)
                      :OUTPUT-FILE "hillary:reports;HILDUM-EXP-1.TEXT")



;;; ---------------------------------------------------------------------
;;; Delay Information

(defun delays-information ()
  (LOOP WITH local-delay-time = 0
        AND network = nil
        AND release-time = nil
        AND due-date = nil
        AND start-time = nil
        AND finish-time = nil
        AND avg-delay = 0
        AND std-dev-delay = 0
        AND max-delay = 0
        FOR order IN (find-units 'order
                       (make-paths '(scheduler * order)) :ALL)
        DO
        (setf network (order$network order))
        (setf release-time (order$release-time order))
        (setf due-date (order$due-date-time order))
        (setf start-time (task-network$start-time network))
```

```
(setf finish-time (task-network$finish-time network))
(setf local-delay-time (- finish-time due-date))
SUM (- finish-time start-time) INTO total-actual-duration
SUM (- due-date release-time) INTO total-desired-duration
WHEN (not (= local-delay-time 0))
COUNT local-delay-time INTO number-of-delays
AND
SUM local-delay-time INTO total-delay-time
AND
COLLECT local-delay-time INTO delays
FINALLY
(unless (zerop number-of-delays)
  (setf avg-delay (float (/ total-delay-time number-of-delays)))
  (setf std-dev-delay
        (LOOP FOR delay IN delays
              SUM (expt (- delay avg-delay) 2)
              INTO sum-of-squares
              FINALLY (return
                        (sqrt
                          (float
                            (/ sum-of-squares
                              (1- number-of-delays)))))))
  (setf max-delay (apply #'max delays)))
(return (values total-delay-time
                number-of-delays
                avg-delay
                std-dev-delay
                max-delay
                (float (/ total-actual-duration
                          total-desired-duration))))))
```

A.1.2 Phoenix Real-Time-Knob Experiment

This section describes an experiment done in the PHOENIX fire-fighting simulation system. We give some background on the Phoenix system and on the experiment design to help the reader better interpret the experiment definition that follows. For more detail on the experiment and its results see Hart & Cohen, 1992 [1].

The Phoenix System

PHOENIX is a multi-agent planning system that fights simulated forest-fires. The simulation uses terrain, elevation, and feature data from Yellowstone National Park and a model of fire spread from the National Wildlife Coordinating Group Fireline Handbook (National Wildlife Coordinating Group 1985). The spread of fires is influenced by wind and moisture conditions, changes in elevation and ground cover, and is impeded by natural and man-made boundaries such as rivers, roads, and fireline. The Fireline Handbook also prescribes many of the characteristics of our firefighting agents, such as rates of movement and effectiveness of various firefighting techniques. For example, the rate at which bulldozers dig fireline varies with the terrain. PHOENIX is a real-time simulation environment PHOENIX agents must think and act as the fire spreads. Thus, if it takes too long to decide on a course of action, or if the environment changes while a decision is being made, a plan is likely to fail.

One PHOENIX agent, the Fireboss, coordinates the firefighting activities of all field agents, such as bulldozers and watchtowers. The Fireboss is essentially a thinking agent, using reports from field agents to form and maintain a global assessment of the world. Based on these reports (e.g., fire sightings, position updates, task progress), it selects and instantiates fire-fighting plans and directs field agents in the execution of plan subtasks.

A new fire is typically spotted by a watchtower, which reports observed fire size and location to the Fireboss. With this information, the Fireboss selects an appropriate fire-fighting plan from its plan library. Typically these plans dispatch bulldozer agents to the fire to dig fireline. An important first step in each of the three plans in the experiment described below is to decide where fireline should be dug. The Fireboss projects the spread of the fire based on prevailing weather conditions, then considers the number of available bulldozers and the proximity of natural boundaries. It projects a bounding polygon of fireline to be dug and assigns segments to bulldozers based on a periodically updated assessment of which segments will be reached by the spreading fire soonest. Because there are usually many more segments than bulldozers, each bulldozer digs multiple segments. The Fireboss assigns segments to bulldozers one at a time, then waits for each bulldozer to report that it has completed its segment before assigning another. This ensures that segment assignment incorporates the most up-to-date information about overall progress and changes in the prevailing conditions.

Once a plan is set into motion, any number of problems might arise that require the Fireboss's intervention. The types of problems and mechanisms for handling them are described in Howe & Cohen, 1990 [2], but one is of particular interest here: As bulldozers build fireline, the Fireboss compares their progress to expected progress. If their actual progress falls too far below expectations, a plan failure occurs, and (under the experiment scenario described here) a new plan is generated. The new plan uses the same bulldozers to fight the fire and exploits any fireline that has already been dug. We call this error recovery method vreplanning. PHOENIX is built to be an adaptable planning system that can recover from plan failures. Although it has many failure-recovery methods, replanning is the focus

of the experiment described in the next section.

Identifying the Factors that Affect Performance

We designed an experiment with two purposes. A confirmatory purpose was to test predictions that the planner's performance is sensitive to some environmental conditions but not others. In particular, we expected performance to degrade when we change a fundamental relationship between the planner and its environment—the amount of time the planner is allowed to think relative to the rate at which the environment changes—and not be sensitive to common dynamics in the environment such as weather, and particularly, wind speed. We tested two specific predictions: 1) that performance would not degrade or would degrade gracefully as wind speed increased; and 2) that the planner would not be robust to changes in the Fireboss's thinking speed due to a bottleneck problem described below. An exploratory purpose of the experiment was to identify the factors in the Fireboss architecture and PHOENIX environment that most affected the planner's behavior, leading to a causal model.

The Fireboss must select plans, instantiate them, dispatch agents and monitor their progress, and respond to plan failures as the fire burns. The rate at which the Fireboss thinks is determined by a parameter called the Real Time Knob. By adjusting the Real Time Knob we allow more or less simulation time to elapse per unit CPU time, effectively adjusting the speed at which the Fireboss thinks relative to the rate at which the environment changes.

The Fireboss services bulldozer requests for assignments, providing each bulldozer with a task directive for each new fireline segment it builds. The Fireboss can become a bottleneck when the arrival rate of bulldozer task requests is high or when its thinking speed is slowed by adjusting the Real Time Knob. This bottleneck sometimes causes the overall digging rate to fall below that required to complete the fireline polygon before the fire reaches it, which causes replanning. In the worst case, a Fireboss bottleneck can cause a thrashing effect in which plan failures occur repeatedly because the Fireboss can't assign bulldozers during replanning fast enough to keep the overall digging rate at effective levels. We designed our experiment to explore the effects of this bottleneck on system performance and to confirm our prediction that performance would vary in proportion to the manipulation of thinking speed. Because the current design of the Fireboss is not sensitive to changes in thinking speed, we expect it to take longer to fight fires and to fail more often to contain them as thinking speed slows.

In contrast, we expect PHOENIX to be able to fight fires at different wind speeds. It might take longer and sacrifice more area burned at high wind speeds, but we expect this effect to be proportional as wind speed increases and we expect PHOENIX to succeed equally often at a range of wind speeds, since it was designed to do so.

Experiment Design

We created a straightforward fire fighting scenario that controlled for many of the variables known to affect the planner's performance. In each trial, one fire of a known initial size was set at the same location (an area with no natural boundaries) at the same time (relative to the start of the simulation). Four bulldozers were used to fight it. The wind's speed and direction were set initially and not varied during the trial. Thus, in each trial, the Fireboss receives the same fire report, chooses a fire-fighting plan, and dispatches the bulldozers to implement it. A trial ends when the bulldozers have successfully surrounded the fire or after

120 hours without success. The experiment's first dependent variable then is Success, which is true if the fire is contained, and false otherwise. A second dependent variable is shutdown time (SD), the time at which the trial was stopped. For successful trials, shutdown time tells us how long it took to contain the fire. Two independent variables were wind speed (WS) and the setting of the Fireboss's Real Time Knob (RTK). A third variable, the first plan chosen by the Fireboss in a trial (FPLAN), varied randomly between trials. It was not expected to influence performance, but because it did, we treat it here as an independent variable.

WS The settings of WS in the experiment were 3, 6, and 9 kilometers per hour. As wind speed increases, fire spreads more quickly in all directions, and most quickly downwind. The Fireboss compensates for higher values of wind speed by directing bulldozers to build fireline further from the fire.

RTK The default setting of RTK for PHOENIX agents allows them to execute 1 CPU second of Lisp code for every 5 minutes that elapses in the simulation. We varied the Fireboss's RTK setting in different trials (leaving the settings for all other agents at the default). We started at a ratio of 1 simulation-minute/cpu-second, a thinking speed 5 times as fast as the default, and varied the setting over values of 1, 3, 5, 7, 9, 11, and 15 simulation-minutes/cpu-second. These values range from 5 times the normal speed at a setting of 1 down to one-third the normal speed at 15. The values of RTK reported here are rescaled. The normal thinking speed (5) has been set to RTK=1, and the other settings are relative to normal. The scaled values (in order of increasing thinking speed) are .33, .45, .56, .71, 1, 1.67, and 5. RTK was set at the start of each trial and held constant throughout.

FPLAN The Fireboss randomly selects one of three plans as its first plan in each trial. The plans differ mainly in the way they project fire spread and decide where to dig fireline. SHELL is aggressive, assuming an optimistic combination of low fire spread and fast progress on the part of bulldozers. MODEL is conservative in its expectations, assuming a high rate of spread and a lower rate of progress. The third, MBIA, generally makes an assessment intermediate with respect to the others. When replanning is necessary, the Fireboss again chooses randomly from among the same three plans. We adopted a basic factorial design, systematically varying the values of WS and RTK. Because we had not anticipated a significant effect of FPLAN, we allowed it to vary randomly.

Rtk Experiment Clips

All data collection in this experiment occurs at the end of each trial. Many of these clips invoke system specific calls to the simulator and to agents' state memories, again, as in the first example, illustrating the interface between clips and the system under study. Of particular interest here is the script invoked to recreate the same experimental scenario for each trial (see the `:script` keyword to `define-experiment` below). This script has a set of instructions to the Phoenix simulator's event-scheduler that introduce a fixed set of environmental changes that are part of the experimental control.

```
(define-experiment real-time-knob-experiment (use-exp-style)
  "Simple experiments for exercising and testing the real time knob."
```

```
  :before-experiment (real-time-knob-experiment-init-before-experiment
                       use-exp-style)
  :before-trial (real-time-knob-experiment-init-before-trial)
  :after-trial (real-time-knob-experiment-after-trial)
  :after-experiment (real-time-knob-experiment-reset-after-experiment)
  :variables ((real-time-knob in '(1 3 5 7 9 11))
              (wind-speed in '(3 9 12)))
  :instrumentation (number-of-bulldozers
                    plan-to-contain-fire       ; FPLAN
                    all-plans-to-contain-fire
                    fires-started
                    shutdown-time              ; SD
                    number-of-fires-contained
                    total-fire-line-built
                    r-factor
                    area-burned
                    agents-lost
                    all-agent-instrumentation
                    fireboss-instrumentation
                    bulldozer-instrumentation)

  :script
  ((setup-starting-conditions "12:29"
     (progn
       (send (fire-system)
             :alter-environment-parameter 'wind-direction 315)
       (send (fire-system)
             :alter-environment-parameter 'wind-speed (wind-speed))))
   (start-fire "12:30"
     (send (fire-system) :start-fire 700 (point 50000 40000)))))

;;; ---------------------------------------------------------------------
;;; Utility functions used by the :before- and :after- forms to
;;; initialize and reset the experiment environment

(defun real-time-knob-experiment-init-before-experiment (use-exp-style)
  (when use-exp-style
    (setf (interaction-style t) 'experiment))
  ;; Don't allow fires to skip over fire lines.
  (send (fire-simulation) :set-spotting-scale-factor 0)
  ;; Get rid of flank attack, etc.
  (modify-knowledge-base))

(defun real-time-knob-experiment-init-before-trial ()
  (gc-immediately :silent t))

(defun real-time-knob-experiment-after-trial ()
  ;; This is now done in the `shutdown-trial' method.
```

```
  (write-current-experiment-data))

(defun real-time-knob-experiment-reset-after-experiment ()
  (setf (interaction-style t) 'normal))

;;; ---------------------------------------------------------------------

(defmacro pct (part wh)
  `(if (zerop ,wh) 0 (* 100.0 (/ ,part ,wh))))

(defmacro /-safe (dividend divisor)
  `(if (zerop ,divisor) 0 (/ ,dividend ,divisor)))

;;; ---------------------------------------------------------------------
;;; Instrumentation definitions...

(defclip number-of-bulldozers ()
  (:report-key "Number of bulldozers")
  (length (find-agent-by-name 'bulldozer :multiple-allowed t)))

(defclip area-burned ()
  (:report-key "Area burned (sq. km)")
   (send (real-world-firemap) :fire-state))

(defclip shutdown-time ()
  (:report-key "Shutdown time (hours)")
    (current-time))

;;; ---------------------------------------------------------------------
;;; Fire clips

(defclip fires-started ()
  (:report-key "Fires started")
  (let ((cnt 0))
    (map-over-fires (fire) (:delete-fire-frames nil)
       (incf cnt))
    (values cnt)))

(defclip fires-contained ()
  (:report-key "Fires contained")
  (mapcan #'(lambda (fire)
              (when (eq (f:get-value* fire 'status) 'under-control)
                (list fire)))
          (f:get-values* 'actual-fire 'instance+inv)))

(defclip number-of-fires-contained ()
  (:report-key "Fires extinguished")
  (length (fires-contained)))
```

```
;;; ---------------------------------------------------------------------
;;; Fireline clips

(defun expand-extent-in-all-directions-by (extent distance-in-meters)
  (let ((temp-point (point distance-in-meters distance-in-meters)))
    (extent (point-clip (point-difference (extent-upper-left extent)
                        temp-point))
            (point-clip (point-sum (extent-lower-right extent) temp-point)))))

(defun length-of-built-line-in-extent (extent)
  (let ((line-length 0))
    (dofiremap (point :upper-left (extent-upper-left extent)
                      :lower-right (extent-lower-right extent))
      (do-feature-edges (edge point (real-world-firemap) :edge-type :dynamic)
        (incf line-length (feature-edge-length edge))))
    (values line-length)))

(defclip r-factor ()
  (:report-key "R factor")
    (/-safe (total-fire-line-built) (total-perimeter)))

(defclip total-perimeter ()
  ()
  (reduce #'+ (fires-contained)
          :key #'(lambda (fire)
                   (fast-polyline-length
                     (fire-perimeter-polyline (fire-origin fire)
                                              *fire-perimeter-resolution*)
                     t))))

(defclip total-fire-line-built ()
  (:report-key "Fireline Built (meters)")
  (reduce #'+ (fires-contained)
          :key #'(lambda (fire)
                   (length-of-built-line-in-extent
                     (expand-extent-in-all-directions-by
                       (accurate-fire-extent (fire-center-of-mass fire)
                                             (point-on-polyline-furthest-from
                                               (fire-center-of-mass fire)
                                               (fire-boundary fire)
                                               nil))
                       *fire-sector-extension*)))))

;;; ---------------------------------------------------------------------
;;; Agent clips

(defclip agents-lost ()
```

```
  ()
  (mapcan #'(lambda (agent)
              (when (eq (f:get-value (send agent :self-frame) 'status) :dead)
                (list (name-of agent))))
          (all-agents)))

;;; "all-agent-instrumentation" is a mapping clip that collects information
;;; from its components, all of the Phoenix agents defined in this
;;; scenario.  Each of the :component clips is run for each agent found
;;; by the :mapping function.  Values returned by the :component clips
;;; are written out to the data file in sequence.

(defclip all-agent-instrumentation ()
  "Records the utilization of all the agents."
  (:components (agent-overall-utilization
                agent-cognitive-utilization
                agent-message-handling-time-pct
                agent-action-selection-time-pct
                agent-error-recovery-cost
                agent-error-recovery-percentage-of-cognitive-time
                number-of-frames-on-timeline)
   :map-function (cons
                  (find-agent-by-name 'fireboss)
                  (find-agent-by-name 'bulldozer :multiple-allowed t))))

(defclip agent-overall-utilization (agent)
  (:report-key "~a overall utilization")
  (pct (task-cumulative-cpu-time agent) (current-time)))

(defclip agent-cognitive-utilization (agent)
  (:report-key "~a cognitive utilization")
  (pct (phoenix-agent-cumulative-action-execution-time agent)
       (current-time)))

(defclip agent-message-handling-time-pct (agent)
  (:report-key "~a message handling pct")
  (pct (phoenix-agent-cumulative-message-handling-time agent)
       (current-time)))

(defclip agent-action-selection-time-pct (agent)
  (:report-key "~a action selection pct")
  (pct (phoenix-agent-cumulative-next-action-selection-time agent)
       (current-time)))

(defclip agent-error-recovery-cost (agent)
  (:report-key "~a error recovery cost")

  (f:using-frame-system ((name-of agent))
```

```
    (reduce #'+ (gather-recovery-method-instances (name-of agent))
            :key #'determine-recovery-cost)))

(defclip agent-error-recovery-percentage-of-cognitive-time (agent)
  (:report-key "~a ER % of cognitive time")
  (pct (agent-error-recovery-cost (name-of agent))
       (phoenix-agent-cumulative-action-execution-time agent)))

(defclip number-of-frames-on-timeline (agent)
  (:report-key "~a number of frames on timeline")
  (f:using-frame-system ((name-of agent))
    (unwind-protect
        (let ((cnt 0))
          (labels ((count-frame (frame)
                     (unless (f:get-value frame 'counted)
                       (incf cnt)
                       (f:put-value frame 'counted t)
                       (count-frames-after frame)
                       (count-frames-below frame)))
                   (count-frames-below (start-frame)
                     (dolist (frame (tl-has-components start-frame))
                       (count-frame frame)))
                   (count-frames-after (start-frame)
                     (dolist (frame (tl-next-actions start-frame))
                       (count-frame frame))))

            (dolist (frame (tl-has-start-actions (f:get-value 'initial-timeline
                                                              'instance+inv)))
              (count-frame frame)))
            (values cnt))
      (f:map-frames #'(lambda (frame)
                        (f:delete-all-values frame 'counted))))))

;;; -------------------------------------------------------------------
;;; Fireboss clips

(defclip fireboss-instrumentation ()
  "Instrumentation for the fireboss."
  (:components (agent-total-envelope-time)
   :map-function (list (find-agent-by-name 'fireboss))))

(defclip agent-total-envelope-time (agent)
  (:report-key "~a total envelope time")
  (f:using-frame-system ((name-of agent))
    (reduce #'+
            (f:pattern-match #p(instance {f:value-in-hierarchy-of
                                          '(ako instance) 'plan-envelope})))))
```

```
;;; ------------------------------------------------------------------------
;;; Bulldozer clips

(defclip bulldozer-instrumentation ()
  "Mapping clip mapping reflexes-executed :component over all bulldozers."
  (:components (reflexes-executed)
   :map-function (find-agent-by-name 'bulldozer :multiple-allowed t)))

(defclip reflexes-executed (agent)
  (:report-key "~a reflexes executed")
  (reduce #'+ (standard-agent-model-reflexes agent)
          :key #'reflex-execution-count))

(defclip count-of-deadly-object-in-path-messages ()
  (:enable-function (trace-message-patterns
                     '(:message-type :msg-reflex :type :error
                       :reason :deadly-object-in-path))
   :disable-function (untrace-message-patterns
                      '(:message-type :msg-reflex :type :error
                        :reason :deadly-object-in-path)))
  (message-pattern-count '(:message-type :msg-reflex :type :error
                           :reason :deadly-object-in-path)))

;;; ------------------------------------------------------------------------
;;; Plan clips record which plan(s) was used during a trial to fight the fire.

(defun the-first-fire-started ()
  (first (last (f:get-values* 'actual-fire 'instance+inv))))

(defclip plan-to-contain-fire ()
  (:report-key "Plan to Contain Fire")

  (get-primitive-action
    (tl-entry-of-plan-to-contain-fire)))

(defun tl-entry-of-plan-to-contain-fire ()
  (f:using-frame-system ((name-of (find-agent-by-name 'fireboss)))
    (let (top-level-actions)
      (dolist (possible (f:get-values* 'act-deal-with-new-fire 'instance+inv))
        (when (equal (f:framer (the-first-fire-started))
              (variable-value 'fire :action possible))
          (push possible top-level-actions)))
      (f:get-value*
        (first (sort top-level-actions
                             #'>
                     :key #'(lambda (x)
                              (f:get-value* x 'creation-time))))
                   'has-end-action))))
```

```
(defclip all-plans-to-contain-fire ()
  (:report-key "All plans to Contain Fire")
  (f:using-frame-system ((name-of (find-agent-by-name 'fireboss)))
    (mapcar #'(lambda (act-deal-with-our-fire)
                (get-primitive-action (f:get-value* act-deal-with-our-fire
                                                    'has-end-action)))
            (sort
              (mapcan #'(lambda (act-deal-with-new-fire)
                          (when (equal (f:framer (the-first-fire-started))
                                       (variable-value 'fire :action
                                          act-deal-with-new-fire))
                            (list act-deal-with-new-fire)))
                      (f:get-values* 'act-deal-with-new-fire 'instance+inv))
              #'< :key #'(lambda (x) (f:get-value* x 'creation-time))))))
```

A.1.3 Example from a Transportation Planning Simulation

This example comes from a baseline experiment in bottleneck prediction at a shipping port in a transportation planning simulator called TransSim. On each day it predicts the occurrence of bottlenecks at a single port in the shipping network, then captures data about actual bottlenecks for later comparison. The experiment collects time-series data, a fragment of which is shown after the example. The summary output file (data collected after each trial) is also shown. Note the use of the keywords to `define-simulator` (`:schedule-function`, `:deactivate-scheduled-function`, `:seconds-per-time-unit` and `:timestamp`) that define the time-series clips.

```
;;; -----------------------------------------------------------------
;;; Clips for the collection of time series data over the course of a trial

(defun current-day ()
  "Return the current day of simulated time from 0."
  (/ (current-time) 24))

(defclip port-state-snapshot ()
  "Record state information for a port at the end of each day."
  (:output-file "port-state"
   :schedule (:period "1 day")
   :map-function (list (port 'port-1))
   :components (ships-en-route ships-queued ships-docked
                expected-ship-arrivals predicted-queue-length)))

(defclip ships-en-route (port)
  "Record the number of ships en route to a port."
  ()
  (length (apply #'append (mapcar 'contents (incoming-channels port)))))

(defclip ships-queued (port)
  "Record the number of ships queued at a port."
  ()
  (length (contents (docking-queue port))))

(defclip ships-docked (port)
  "Record the number of ships docked at a port."
  ()
  (length (apply #'append (mapcar 'contents (docks port)))))

(defclip expected-ship-arrivals (port) ()
  (progn
    (update-prediction)
    (let ((prediction-units (find-units 'prediction '(port-model) :all)))
      (float (/ (round (* 10000
                          (expected-value (first prediction-units)))) 10000)))))
```

```
(defclip port-predicted-value (port) ()
  (let ((prediction-units (find-units 'prediction '(port-model) :all)))
    (generate-prediction (expected-value (first prediction-units)))))

(defclip port-actual-change (port) ()
  (let ((prediction-units (find-units 'prediction '(port-model) :all)))
    (length
      (set-difference
        (ships-previously-in-port port 0)
        (ships-previously-in-port
          port (days-in-future (first prediction-units)))))))

(defclip predicted-queue-length (port) ()
  (let ((prediction-units (find-units 'prediction '(port-model) :all)))
    (pred-queue-length (first prediction-units))))

;;; ---------------------------------------------------------------------
;;; Clips for the collection of data at the end of a trial

(defclip prediction-score ()
  (:output-file "score"
   :components (score-for-day)
   :map-function '(2 5)))

(defclip score-for-day (day)
 ()
 (compute-score-for-day day))

;;; ---------------------------------------------------------------------
;;; Experiment and simulator definitions

(define-simulator transsim
  :system-name "TransSim"
  :start-system (simulate nil)
  :reset-system reset-transsim-experiment
  :schedule-function schedule-function-for-clips
  :deactivate-scheduled-function transsim::reset
  :seconds-per-time-unit 3600
  :timestamp current-day)

(define-experiment test-experiment ()
  :simulator transsim
  :instrumentation (prediction-score port-state-snapshot)
  :variables ((prediction-threshold in '(0.6 0.75 0.9))
              (eta-variance-multiplier in '(0.2 0.4 0.6))
              (prediction-point in '(5)))
  :after-trial
    ;; Other options here include building CLASP datasets,
```

```
    ;; exporting to some database, massaging the data or some
    ;; combination fo these.
    (write-current-experiment-data))

;;; ---------------------------------------------------------------------
;;; Utilities

(defun reset-transsim-experiment (prediction-threshold
                                  eta-variance-multiplier
                                  prediction-point)
  (setf *prediction-threshold* prediction-threshold)
  (format t "~&Prediction threshold = ~a" prediction-threshold)
  (setf *eta-variance-multiplier* eta-variance-multiplier)
  (format t "~&ETA variance multiplier = ~a" eta-variance-multiplier)
  (setf *prediction-point* prediction-point)
  (format t "~&Prediction point = ~a" prediction-point)
  (initialize-simulation))

(defun schedule-function-for-clips (function time period name)
  (if period
    (transsim::schedule-recurring-event (transsim::event-actuator :external)
                                        :function function
                                        :time time
                                        :period period
                                        :type (or name :instrumentation))
    (transsim::schedule-event (transsim::event-actuator :external)
                              :function function
                              :time time
                              :type (or name :instrumentation))))

(defun rexp (&key &optional number-of-trials)
  (run-experiment 'test-experiment :output-file "ed-buffer:out"
                                   :number-of-trials number-of-trials))
```

Shown below is the summary output file produced for 9 trials. This file is in CLASP format. It begins with an informative header string, followed by a series of strings, each of which is a column name in the data table. Rows of the table follow, stored as lists. Each list contains one element per column name.

```
"
**********************************************************************
****
**** Experiment: Test-Experiment
**** Machine: Miles
**** TransSim version: Unknown
**** Date: 10/1/93 11:26
**** Scenario: None
**** Script-name: None
```

```
**** First trial number: 1
**** Last trial number: 9
**** Number of trials: 9
**** Max trial length: Unknown hours
*****************************************************************

The key follows:"
"Trial"
"Prediction-Threshold"
"Eta-Variance-Multiplier"
"Prediction-Point"
"Score-For-Day 2"
"Score-For-Day 5"
(1 0.75 0.2 5 100 100 )
(2 0.75 0.3 5 100 100 )
(3 0.75 0.4 5 100 100 )
(4 0.8 0.2  5 100 100 )
(5 0.8 0.3  5 100 100 )
(6 0.8 0.4  5 100 100 )
(7 0.85 0.2 5 100 100 )
(8 0.85 0.3 5 100 100 )
(9 0.85 0.4 5 100 100 )
```

This is a fragment of the time-series data (from the first trial only) showing the flat file structure produced by component and time-series clip relationships.

```
"
*****************************************************************
****
**** Experiment: Test-Experiment
**** Machine: Miles
**** TransSim version: Unknown
**** Date: 10/5/93 15:17
**** Scenario: None
**** Script-name: None
**** First trial number: 1
**** Last trial number: 12
**** Number of trials: 12
**** Max trial length: Unknown hours
*****************************************************************

The key follows:"
"Trial"
"Timestamp"
"Prediction-Threshold"
"Eta-Variance-Multiplier"
"Prediction-Point"
"Ships-En-Route port-1"
```

```
"Ships-Queued port-1"
"Ships-Docked port-1"
"Expected-Ship-Arrivals port-1"
"Predicted-Queue-Length port-1"

(1 0 0.6 0.2 5 0 0 0 0.0 0 )
(1 1 0.6 0.2 5 0 0 0 0.0 0 )
(1 2 0.6 0.2 5 0 0 0 0.0 0 )
(1 3 0.6 0.2 5 0 0 0 0.0 0 )
(1 4 0.6 0.2 5 0 0 0 0.0 0 )
(1 5 0.6 0.2 5 2 0 0 0.0597 0 )
(1 6 0.6 0.2 5 2 0 0 0.3357 0 )
(1 7 0.6 0.2 5 2 0 0 0.7643 0 )
(1 8 0.6 0.2 5 2 0 0 0.9961 0 )
(1 9 0.6 0.2 5 2 0 0 0.9977 0 )
(1 10 0.6 0.2 5 4 0 0 0.683 0 )
(1 11 0.6 0.2 5 3 0 1 0.2361 0 )
(1 12 0.6 0.2 5 3 0 1 0.6389 1 )
(1 13 0.6 0.2 5 3 0 1 1.4062 1 )
(1 14 0.6 0.2 5 3 0 1 1.9885 1 )
(1 15 0.6 0.2 5 3 0 0 2.071 1 )
(1 16 0.6 0.2 5 3 0 0 1.5729 0 )
(1 17 0.6 0.2 5 1 1 1 0.5053 1 )
(1 18 0.6 0.2 5 1 1 1 0.8974 2 )
(1 19 0.6 0.2 5 1 1 1 0.9996 2 )
(1 20 0.6 0.2 5 1 1 0 1.0 1 )
(1 21 0.6 0.2 5 1 0 1 1.0 1 )
(1 22 0.6 0.2 5 0 1 1 0.0 1 )
(1 23 0.6 0.2 5 0 1 1 0.0 1 )
(1 24 0.6 0.2 5 0 1 0 0.0 0 )
(1 25 0.6 0.2 5 2 0 1 0.1331 0 )
(1 26 0.6 0.2 5 2 0 1 0.355 0 )
(1 27 0.6 0.2 5 2 0 1 0.6276 1 )
(1 28 0.6 0.2 5 2 0 0 0.946 0 )
(1 29 0.6 0.2 5 4 0 0 1.2677 0 )
(1 30 0.6 0.2 5 4 0 0 2.0143 1 )
(1 31 0.6 0.2 5 4 0 0 2.8832 2 )
(1 32 0.6 0.2 5 4 0 0 3.3096 2 )
(1 33 0.6 0.2 5 4 0 0 2.4576 1 )
(1 34 0.6 0.2 5 2 1 1 1.9172 3 )
(1 35 0.6 0.2 5 1 2 1 0.9961 3 )
```

References

[1] David M. Hart and Paul R. Cohen. Predicting and explaining success and task duration in the phoenix planner. In *Proceedings of the First International Conference on AI Planning Systems*, pages 106–115. Morgan Kaufmann, 1992.

[2] Adele E. Howe and Paul R. Cohen. Responding to environmental change. In *DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 85–92. Morgan Kaufmann, 1990.

Index